Clone Tracker: Tracking Inconsistent Clone Changes in A Clone Group

MD. JUBAIR IBNA MOSTAFA
BSSE 0614

A Thesis
Submitted to the Bachelor 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

BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

CLONE TRACKER
TRACKING INCONSISTENT CLONE CHANGES IN A CLONE GROUP

MD. JUBAIR IBNA MOSTAFA

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

Professor

To
Md. Golam Mostafa, my father
Mossammat Rokshana Tasmim, my mother
Sumaiya Binte Mostafa, my elder sister
Mariam Binte Mostafa, my younger sister
Md. Jonayet Ibna Mostafa, my younger brother

whose enormous love and courage is the living of my life

Abstract

Source code cloning is a common activity in software development as well as a threat to the maintainability of the source code. Because code clones reduce the reliability of source code and increase the maintenance cost. In maintenance, evolving clones require more attention because of clone changing behavior. In a clone group, all clones are similar and require similar change to keep consistency among those clones. In a scenario where one clone is changed and others remain unchanged may lead to inconsistency. To resolve this issue an approach is proposed and a plugin is developed to track changes in clones. It provides change information with similar clones that may need to be updated accordingly.

In the proposed approach, clones are detected and stored in a repository. From that clone groups or classes are created based on those similarities. Clones are mapped into files so that it requires less time to search clones of a fragment. Based on the editor events fragments are queried from the repository. After comparison with an edited fragment, this provides real-time change information.

The plugin is incorporated with the IntelliJ Idea so that it can help developer to maintain cloned codes in maintenance time. In study with three open source software, it performs well in accuracy. Although the source code is changed manually to know how it performs in different scenario like insertion, deletion or modification. It can detect changes in a clone fragment and shows other clones of the clone group which may require change to adapt the functionality. However, some requirements to use the plugin may limit the usability.

Acknowledgments

First of all, I praise Allah, the Almighty for giving me the opportunity to complete my research project. I would like to express my heartiest gratitude to my supervisor Dr. Kazi Muheymin-Us-Sakib for his direction, guidance, motivation, and advice throughout the project. He has been assiduous and fastidious to make the project great and to bring the best out of me.

I would also like to thank my beloved classmates of BSSE 6th Batch. This project would be incomplete without their continuous source of joyousness, support, and motivation. They delighted me and kept me motivated towards my goal throughout the project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Code clones are duplicated code fragments in the source code. Copying and pasting of same code fragments introduce clone in a repository. Usually, copying and pasting fragment of codes is a common practice in software development. A copy is created due to the developer's ignorance about the language and features, having limited time to do it properly, or even not caring about future maintenance problems. In that repository, code clones may change consistently or inconsistently in a clone group. The consistent change means a similar change is applied to all clone instances in a clone group. Whether inconsistent change means a change is applied to one of the clone instances and leave other instances without any change. Code clones are considered harmful to software if those changes are not consistent across the repository. Because, inconsistency within similar clones may produce bugs and, bugs increase maintenance cost [1].

After more than a decade, code clone based research still have a greater attention to researchers for better maintenance of a software. Most of the researchers focused on clone detection, clone evolution, and clone re-factoring. While source codes are increasing with time, the number of cloned codes also increases in the repository. To meet the changing behavior of requirements, sometimes change is needed to one fragment which may have cloned fragments. Changing one of the

cloned fragments and ignore similar one may create inconsistencies [2]. Because it is difficult for a developer to track and change all similar cloned codes to avoid introducing future bugs. This lack of consistencies increases maintenance efforts, which is also shown in the literature that cloned codes require more efforts than non-cloned codes [3].

A clone inconsistency tracker can help a developer to maintain cloned codes and can reduce the chance of introducing clone related bugs. In this chapter, the motivation behind this work and challenges involved in this work have been discussed. The research questions and contribution of the project have included in the chapter. To get an overview of how the report has been organized is also mentioned at the end of this chapter.

1.1 Motivation

Real-time code clone tracking and notifying to the developer about changes is required to effectively manage cloned codes. This reduces the possibility of introducing clone related bugs in the repository. Because inconsistency among cloned codes of a clone class is one of the reasons of bug creation [4]. These are the motivation behind this work. A tool will be developed which can support clone inconsistency tracking in software development.

Jens Krinke [5] conducted a study of consistent and inconsistent changes to code clones. Based on two hypothesis he showed that about half of changes are inconsistent in code clone. To monitor changing pattern of clones, empirical study on inconsistent changes at release level have also been conducted in the literature [5]. To find how often inconsistencies happen in code fragment, Yoshiki et at. have tried to derive modification patterns by mining how the code was changed in the past [6]. To minimize inconsistent changes in a clone, developers should know which clones are related. And all similar clones need to change simultaneously.

Many research has been conducted to detect code clone in software such as [7, 8]. Researchers used different methodologies, techniques, and tools to accurately identify code clones. Text-based (e.g. SDD, NICARD), token-based for example CCFinderX, Gemini, iClones, tree-based, for example, Deckard, CloneDigger, SimScan, metrics-based for example Davey and graph-based, for example, Duplix, Gabel approaches are used to detect clones [9]. Deep learning based clone detection also proposed in the literature [10]. In evolution of code clone, many research haseen conducted to acquire knowledge about how clone evolve in subsequent versions [11, 12]. They tried to find out pattern of clone growth by applying different measurements such as clone ratios [13].

For this reason, detection of clone and changes of clone is necessary to know. It is difficult to detect cloned codes accurately because of its inherent complexities such as language dependability, matching criteria, etc. To find similarities by syntactic analysis, different coding styles of source code require pre-processing and normalization before applying matching algorithms. On the other hand, tracking changes and notifying to the developer about inconsistencies have several problems. Those are (i) cloned fragments instability, (ii) large number of cloned fragments maintainability and (iii) detecting and tracking language independently. A cloned fragment instability may happen for several reasons. Those are the cloned fragment (i) may be deleted, (ii) moved into another file or (iii) contained file may be deleted. After tracking changes, Notify to the developer is another task need to be considered. Due to these reasons, detection of code clones, tracking cloned fragments and notifying about clone changes are difficult tasks to achieve. These challenges guides to research the problem.

1.2 Research Questions

In a software repository, code clone is introduced due to the violation of DRY(Don't Repeat Yourself) principle. Cloned codes increase maintenance costs if clones are not managed properly. Studies have found that almost half of the cloned code changes are inconsistent change [5]. Inconsistent change in a clone group while fixing a bug may produce more bugs or have a chance to occur in future. So, this leads to following research question

- How to resolve inconsistencies among clones groups?

To solve this problem, first, accurately identifying of code clones and categorizing into different clone groups is needed. Second, tracking a cloned fragment of a clone group is required to know the changing information. This research question can be solved by answering two following sub-questions.

1. How to detect code clones accurately based on their similarities?

   To solve this problem, source code parsing and normalizing are required. Then codes need to be transformed into a structure that helps to identify duplicated fragments based on their similarities.

2. How cloned codes can be tracked to detect inconsistencies?

   To solve this problem, each cloned fragment needs to be checked after editing some part of source code. If one clone instance or less than the number of clones of a clone group are changed, this may create inconsistencies. The developer will be notified to update change to other clone instances.

1.3 Contribution and Achievement

The contribution of this work is a technique to track clone inconsistency. This is implemented as a plugin support in IntelliJ Idea to provide real-time change information and notification of inconsistencies. As mentioned in the Section 1.2 this purpose can be achieved by solving two other problems. To solve the first problem text-based clone detection technique has been used. A source code repository is given as input to detect clone. From the repository, method fragments are extracted, normalized and pretty-printed for comparison as a potential clone. Pretty-printed means codes are in same layout and formatting. Based on textual similarity clones are identified and stored in a file. From the clone log clone classes are also created. This will be used to get clone instances of a code fragment and served as consistency checking server.

The second question, that is how a cloned code can be tracked to get change information and detect inconsistency has been addressed by file wise clone mapping and comparing previous and updated code fragment. Mapping is used to reduce the search space for which clone is currently changing in the editor. This helps to get real-time notification of clone change. The comparison is used to get change and inconsistency information.

The proposed approach and implemented plugin was used in three software namely JHotDraw, DNSJAVA and Eclipse JDT Core. In manual changed of some clone fragments, plugin detected changes and showed other clones of the clone group accurately. This change information with plugin support was useful to adopt changes in other clones. So it can be concluded that the proposed approach can be used to effectively support clone management tasks.

1.4 Organization of the Report

In this section, the organization of the report has been shown to provide a roadmap to this document. The organization of the chapters in this report has been mentioned in the followings.

Chapter 2: The definitions and background information of code clones and changes in cloned codes with inconsistent change are described in this chapter.

Chapter 3: In this chapter, the literature studies have been presented categorically to know the existing works and research trends in the literature.

Chapter 4: This chapter demonstrates the proposed methodology to track the inconsistent changes within clone group in real time.

Chapter 5: The experimental setup and implementation of the proposed methodology as a plugin have been provided in this chapter.

Chapter 6: This summarizes the whole report, identifies threads to validate the proposed method and highlights future work.

Chapter 2

Background Study

Usually, in software development, copy and paste technique is used to take advantage of existing implementation. This is because it takes less effort and time for a developer. This copying and pasting of a code fragment and creating duplicated code fragments is called code cloning and duplicated code fragments are called cloned codes. However, in software maintenance, extending one functionality, modifying one feature or fixing a bug in one copy of a clone group requires more effort and time. Because whole clone group inspection is required to create the same impact on other duplicated code fragments. On the other hand, this requires checking all clone instances of the similar type and may require updating all of those clone instances. Changing some of those instances and remaining unchanged similar one may create inconsistency within a clone group. To fix this issue, tracking of similar clone fragments and checking consistency continuously is required. In case of any such inconsistency, notifying to developers may resolve the problem.

To understand the inconsistent change in the clone group and consequences of the change, it is required to know code clone, types of clone and the types of changes in clones. This chapter provides the concepts of code clone, types of clones, and changes to clones in terms of the versions of a system.

```
1.  void sumProd(int n) {
2.      float sum=0.0; //C1
3.      float prod =1.0;
4.      for (int i=1; i<=n; i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.1: Code Fragment

2.1 Concepts of Code Clones

In this section code clones related terms and terminologies like code fragment, clone, clone types etc. are described.

2.1.1 Code Fragment

A consecutive sequence of code lines denotes a Code Fragment (CF) [14]. Comments may or may not included with it. It can be a function definition, begin-end block or sequence of statements. A CF is identified by its source code location which means the file, begin and end line number of corresponding code. If 'f' is a file, 's' and 'e' is the start and end line number respectively, then a fragment is denoted by a triple $CF=(f, s, e)$ [15]. In case of clone evolution, a code fragment is denoted by it's file name, start line number, end line number and version number because without version number it can not be identified correctly [14]. Figure: 2.1 denotes a code fragment.

2.1.2 Code Clone

A code fragment is similar to another code fragment for a given function of similarity is called code clone [9]. If CF_1 and CF_2 are two code fragments and ϕ is the similarity function then triple (CF_1, CF_2, ϕ) is called clone. Figure 2.2 shows

an example of code clone.

```
1.  void sumProd(int n) {
2.      float sum=0.0; //C1
3.      float prod =1.0; //C
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      float sum=0.0;
3.      float prod =1.0;
4.      for (int
    i=1;i<=n;i++)
5.      {
6.          sum=sum + i;
7.          prod = prod * i;
8.          foo(sum, prod);
9.      }
10. }
```

Figure 2.2: Code Clone

2.1.3 Clone Class

Two similar *CF*s those are clone to each other is called clone pair. A set of code fragments syntactically similar or identical to each other is called clone class or clone group. A clone group can be specified by the tuple (CF_1, CF_2, CF_3,..., CF_n, ϕ). Here, each pair of distinct fragment is a clone pair: (CF_i, CF_j, ϕ), i, j ϵ 1...n, i$\neq$j [15] of the clone group.

Clone class identification is important to know similar clone instances. In case of bug fixing, it eases the task. Figure 2.3 shows a clone class that consists of four clone fragments.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(s, p);
8.      }
9.  }
```

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n)
2.  {
3.      int sum=0; //C1
4.      int prod =1;
5.      for (int
    i=1;i<=n;i++)
6.      {
7.          sum=sum + i;
8.          prod = prod * i;
9.          foo(sum, prod);
10.     }
11. }
```

```
1.  void sumProd(int n)
2.  {
3.      float sum=0.0; //C1
4.      float prod =1.0;
5.      for (int
    i=1;i<=n;i++)
6.      {
7.          sum=sum + (i*i);
8.          prod = prod*(i*i);
9.          foo(sum, prod);
10.     }
11. }
```

Figure 2.3: Clone Class

2.1.4 Clone Type

There are two main kinds of similarity between code fragments those are syntactic and semantic similarities. Syntactic similarity means fragments which are similar based on the source code and Semantic similarity means fragments which are

similar based on functionality. Basically, the syntactic similarity is the result of copying a code fragment and pasting into another location of source code. Based on textual similarity code clones are three types such as Type-1, Type-2, and Type-3. The semantically similar clone is the fourth type. Definition of these four types of clones is given below.

Type-1

Two identical code fragments excluding the differences in white-space, layout, and comments is called Type-1 clone [9]. This type of clone often creates by copying one function from one file to another without any changes. Figure 2.4 denotes a Type-1 clone.

```
1.  void sumProd(int n) {
2.      float sum=0.0; //C1
3.      float prod =1.0; //C
4.      for (int i=1;i<=n;
    ++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      float sum=0.0;
3.      float prod =1.0;
4.      for (int
    i=1;i<=n;i++)
5.      {
6.          sum=sum + i;
7.          prod = prod * i;
8.          foo(sum, prod);
9.      }
10. }
```

Figure 2.4: Clone Type 1

Type-2

Tow identical code fragments, except the differences in identifier names, literal values, white-space, layout and comments is called Type-2 clone [9]. Usually, in the source code, this kind of clone is found most frequently than any other types

of the clone. Figure 2.5 shows an example of Type-2 clone where variable names are different from one another.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.5: Clone Type 2

Type-3

Syntactically similar code fragments that are differed at the statement level including Type-1, Type-2 clone differences is called Type-3 clone [9]. Modification in statement level creates this kind of clones. Statements may be added, deleted, or modified with respect to each other. Figure 2.6 shows an example of Type-3 clone where one line is removed from one fragment.

```
1.  void sumProd(int n) {
2.      float sum=0.0; //C1
3.      float prod =1.0;
4.      for (int
    i=1;i<=n;i++)
5.          {sum=sum + i;
6.          prod = prod * i;
7.          foo(prod); }}
```

```
1.  void sumProd(int n) {
2.      float sum=0.0; //C1
3.      float prod =1.0;
4.      for (int
    i=1;i<=n;i++)
5.          {sum=sum + i;
6.          //line deleted
7.          foo(sum, prod); }
```

Figure 2.6: Clone Type 3

Type-4

Two or more code fragments that perform same functionality but are implemented by different syntactic variants is called Type-4 clone [9]. This type of clone refers as semantically similar clone. Figure 2.7 represents a Type-4 clone where both fragment perform same operation.

```
1.  void sumProd(int n) {
2.      float sum=0.0; //C1
3.      float prod =1.0;
4.      for (int
    i=1;i<=n;i++)
5.          {prod = prod * i;
6.          sum=sum + i;
7.          foo(sum, prod); }}
```

```
1.  void sumProd(int n) {
2.      float sum=0.0; //C1
3.      float prod =1.0;
4.      int i=0;
5.      while (i<=n){
6.          sum=sum + i;
7.          prod = prod * i;
8.          foo(sum, prod);
9.          i++ ; }}
```

Figure 2.7: Clone Type 4

Clone types define how similar clone fragments are in a source code. To know the changing characteristics, the severity of clones etc., it is required to define

clones into one of the clone types. Depending on the type, clone codes stability may differ from one another.

2.2 Clone changes

Considering clone evolution, different types of changing patterns are described in the literature. Here, change patterns are defined for clone group since this research focuses on clone group changing patterns. Six changes in clone group such as same, add etc. are mentioned in in the research [11].

Suppose, two consecutive versions of a software system are version i and version *i+1*. Clone group of version i and *i+1* are defined as CG_i and $CG_{(i+1)}$ accordingly. Below, six types of clone changes are described briefly with these notions.

2.2.1 Same

A clone group is not changed in subsequent versions is called same change. All clone instances of a clone group CG_i is not changed in next version clone group $CG_{(i+1)}$. This means that no changes happen in the clone group. So the textual similarity of old and new group is $(CG_i, CG_{(i+1)}) = 1$.

For example, clone group CG_i consists of two code fragments {*CF1*, *CF2*} in version i. In version *(i+1)*, the clone group $CG_{(i+1)}$ remains the same with clone fragments of $CG_{(i+1)}$. Figure 2.8 represents a clone group in version i with its clone instances.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.8: Same change: Clone fragments in version i

Figure 2.9 represents the changed clone group of version i in version $i+1$. Since this is an example of same change between two versions, there is no changes in updated version $i+1$.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.9: Same change: Clone fragments in version $i+1$

2.2.2 Add

If one or more code fragments are added in the clone group in version *(i+1)* from the previous version i of that clone group is called add change. If a clone group CG_i consists of two clone fragments and after changes in the source code a new code fragment is added in that group, then it denotes as an add change. This is happened due to copying one or more code fragment from clone group CG_i. After clone detection in new version *(i+1)*, the newly added code fragments are appeared in $CG_{(i+1)}$.

For example, clone group CG_i consists of CG_i=$\{CF1, CF2\}$ in version i, due to change in version (i+1), clone group $CG_{(i+1)}$ have a newly added clone fragment *CF3* $CG_{(i+1)}$=$\{CF1, CF2, CF3\}$. Figure 2.10 represents a clone group in version i where the clone group consists of two code fragments.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.10: Add change: Clone fragments in version i

Figure 2.11 represents the changed clone group of version i in version *i+1*. Since this is an example of add change between two versions, a new code fragment is added in version *i+1*. Now, the clone group consists of three clone fragments in the version *i+1*.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

```
1.  void totalMultiply(int num) {
2.      float total=0.0; //C1
3.      float multiply =1.0;
4.      for (int i=1;i<=num;i++)
5.          {total=total + i;
6.          foo(total, multiply)
7.          multiply=multiply *
    i;
8.  }}
```

Figure 2.11: Add change: Clone fragments in version *i+1*

2.2.3 Subtract

This type of change ensures that at least one code fragment in CG_i does not appear in $CG_{(i+1)}$. For example developers re-factored or removed a code clone from the group.

Assume, a clone group CG_i consists of three code fragments CG_i={*CF1*, *CF2*, *CF3*} in version i, due to change in version *(i+1)*, clone group $CG_{(i+1)}$ has one less

clone fragment than the number of clone in previous version *CF3*, $CG_{(i+1)}$={*CF1*, *CF2*}. Figure 2.12 represents a clone group in version i where the clone group consists of three code fragments.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

```
1.  void totalMultiply(int num) {
2.      float total=0.0; //C1
3.      float multiply =1.0;
4.      for (int i=1;i<=num;i++)
5.          {total=total + i;
6.          foo(total, multiply)
7.          multiply=multiply *
    i;
8.  }}
```

Figure 2.12: Subtract change: Clone fragments in version i

Figure 2.13 represents the changed clone group of version i in version *i+1*. Since this is an example of subtract change between two versions, a code fragment is removed in version *i+1*. Now the clone group consists of two clone fragments in version *i+1*.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.13: Subtract change: Clone fragments in version *i+1*

2.2.4 Consistent Change

In the consistent change, all clone fragments of CG_i have been changed or updated consistently. So all changes are appeared in new clone group $CG_{(i+1)}$. For example, developers applied same change to all clone instances. This consistent updates or changes to all clone instances within a clone group is called consistent change.

In figure 2.14, the two code fragments represent a clone group in version *i*. In the first code fragment, a method *foo(p, s)* is called at the end of *'for' block*. Similarly in the second code fragment, the method *foo(sum, prod)* is called at the end of *'for' block*.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.14: Consistent change: Clone fragments in version i

In figure 2.15, both code fragments are updated in version *(i+1)* with the method call within an output statement. Since the change occurred in both clone fragments of the previous version i, this change denotes as a consistent change.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p=1.0;
4.      for(int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          print("Value:
    "+foo(p, s));
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          print("Value:
    "+foo(sum, prod));
8.      }
9.  }
```

Figure 2.15: Consistent change: Clone fragments in version *i+1*

2.2.5 Inconsistent Change

In inconsistent change, at least one clone fragments of clone group CG_i changed or updated inconsistently means similar update or change operation was missing to one of the clone instances. So one clone fragment disappears in new clone group $CG_{(i+1)}$ or creates an inconsistency with other clone fragments of the group. For example, developers forgot to change one code fragment of CG_i. This inconsistent updates or changes to any clone within a clone group is called inconsistent change. In figure 2.16, a clone group of version i consists of two code fragments. Each of those have a method call *foo(p, s)* and *foo(sum, prod)* respectively at the end of *'for' block*.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p=1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.16: Inconsistent change: Clone fragments in version i

In figure 2.17, the first code fragment is changed by a standard output with the method call *foo(s, p)* and the second code fragment is remains unchanged in the updated version *i+1*.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for(int
    j=1;j<=n;j++){
5.          print("Value: " +
    foo(p, s));
6.          s=s + j;
7.          p = p * j;
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod=1;
4.      for(int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.          prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.17: Inconsistent change: Clone fragments in version *i+1*

2.2.6 Shift

This type of change ensures that at least one clone instance in new clone group $CG_{(i+1)}$ partially overlaps with at least one clone instance in old clone group CG_i.

For example, a change in clone fragment *CF1* of the clone group CG_i in version i have changed *CF1* so diversely that the *CF1* in version *(i+1)* partially similar to the old *CF1*. In figure 2.18 a clone group is defined by two code fragments.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  void sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      for (int
    i=1;i<=n;i++){
5.          sum=sum + i;
6.‘         prod = prod * i;
7.          foo(sum, prod);
8.      }
9.  }
```

Figure 2.18: Shift change: Clone fragments in version i

In figure 2.19, the second code fragment is changed with return type, code structure and a return statement in version *(i+1)*. This indicates a shift change in the clone group.

```
1.  void sumProd(int n){
2.      float s=0.0; //C1
3.      float p =1.0;
4.      for (int
    j=1;j<=n;j++){
5.          s=s + j;
6.          p = p * j;
7.          foo(p, s);
8.      }
9.  }
```

```
1.  int sumProd(int n) {
2.      int sum=0; //C1
3.      int prod =1;
4.      int i = 1;
5.      while(i<=n){
6.          sum = sum + i;
7.          prod = prod * i;
8.          i = i + 1;
9.      }
10.     return foo(sum,
    prod);
11. }
```

Figure 2.19: Shift change: Clone fragments in version *i+1*

2.3 Summary

To fully understand this research and existing works in the literature, it is required to have a preliminary knowledge of code clone and code clone related concepts. In this chapter, code clone, types of clone and clone changes are described to ease the understanding process. In the proceeding chapter, Clone related research and state-of-the-art techniques and tools in the literature are described elaborately.

Chapter 3

Literature Review

In software engineering and maintenance, Code clone is an ongoing research field because it may make the software development faster but it incurs much time and effort in software maintenance. It already has an enriched literature for more than two decades of active research. In software maintenance phase, cloned codes increase maintenance costs, efforts and times as inconsistent changes to duplicated code can lead to unexpected behavior of the system [16]. Studies have found that cloned codes require more effort than non cloned codes [3]. The reason behind this is the lack of clone management expertise and time limitation to fix a bug. Previous research mostly focuses on detecting code clones so a number of clone detection techniques and tools exists in the literature. It helps to get all clone information at a time but in incremental development, it may not be useful. To know the evolving pattern and evolution of cloned code, code clone evolution based research have also been conducted by the researcher. Considering intuition of existing research works, clone related research can be grouped broadly into two categories. Those are:

1. Clone detection
2. Clone evolution

The details of these two categories are given below.

3.1 Clone Detection

Clone detection research are all about finding clones more accurately. There are a number of techniques and tools exists in the literature to detect clones. Some of those papers concerned about precision, recall, execution time, or scalability issues. To achieve these requirements, researcher approach differently in clone detection. Clone detection techniques can be divided into five groups [9]. Those are given below-

1. Text-based techniques
2. Token-based techniques
3. Tree-based techniques
4. Metric-based techniques
5. Graph-based techniques

Comparison of these techniques and performance of proposed tools have been evaluated in the literature [9]. Since clone detection is the prerequisite to conduct research in clone evolution, a brief description of clone detection techniques and tools are discussed here.

3.1.1 Text-Based Techniques

In text-based techniques, the source code is used with/without transformation/normalization to compare in clone detection. Johnson et al. use fingerprints to identify exact repetitions of text in large programs [17]. Fingerprints have been defined as short strings which can be used for comparison purposes to represent larger data objects. A fingerprint is a mapping function from some data object domain into the set of fingerprints. How clone proneness the legacy systems are authors experiments the technique in a legacy software system. This experiment included

the construction of source trees and successful identification of repetitions in the system. This redundancy identification system has provided useful information that forms the basis of visualization and understanding of the programs. In the detection phase, authors employ a technique similar to a string searching method of Karp and Rabin [18].

Johnson et al. also use the source code as substrings to find duplicate code fragments [19]. A sliding window technique with an incremental hash function has used to identify a sequence of lines having same hash value as clones. Different size window has been used to get different length clones. This paper uses fingerprint approach which has been defined in the [17] paper and discussed earlier.

Nicad is also a text-based approach with tree-based structural analysis [20]. It uses a lightweight parsing by a grammar-based method for a specific language. It uses flexible pretty-printing after island-parsing which separates interesting parts of a program (features those are required) from uninteresting parts (other features, which need not be precisely parsed). In code normalization, identifiers name are changed into a normal form to ease the comparison. This helps to identify Type-2 clone with different identifiers and literals. After parsing and pretty-printing source code has been transformed and filtered before comparison. In text-comparison for duplicate code fragments, Longest Common Subsequence (LCS) is used. Other text-based approaches do the similar procedure to identify code clone.

3.1.2 Token-Based Techniques

Token-based techniques use the source code as a sequence of tokens using compiler-style lexical analysis. Tokens are programming language keywords, literals, and identifiers. The sequence of tokens is scanned and compared with a given threshold to find duplicated subsequences. Similar subsequences are reported as clones and corresponding original code is returned [9].

SourcererCC a token-based near-miss clone detection tool [15]. which uses an optimized inverted-index to quickly query the potential clones of a given block. To reduce the size of the index, heuristics based filtering are performed in token ordering. This helps to achieve large-scale clone detection. The authors use a global token ordering technique to compare a subset of tokens to find potential clones. This proposed technique has been evaluated in a large benchmark of real clone BigCloneBench and a mutation-based framework of thousand artificial clones [21]. Comparing with other state-of-the-art clone detectors SourcererCC outperforms others [15].

3.1.3 Tree-Based Techniques

These techniques belong to syntactic approaches of clone detection. The parser uses to parse source code into parse trees or abstract syntax tree which then be processed using tree-matching to find clones. In tree matching algorithm, clones are found by using similar subtrees. In the tree representation, variable names, literal values, and other tokens in the source are abstracted. Baxter have conducted research on clone detection using AST [7]. In this research, a compiler generator is used to generate a constructor for annotated parse tree. Subtrees are then hashed into buckets to reduce computational time as tree comparisons reduced drastically. Within the same bucket, subtrees are compared to each other by a tolerant tree matching.

Since token-based clone detection based on suffix tree is extremely fast and AST finds syntactic clone but less efficiently, combining two techniques can be useful to find clone efficiently. Abstract syntax suffix tree can find clone in linear time and space [22].

3.1.4 Metric-Based Techniques

Metric-based techniques use metrics to define and compare code fragments for clone detection. Metrics calculated for the syntactic unit such as a class, method, block, statement that yield values that can be compared to detect duplication of these units. Most cases source code parse into Abstract Syntax Tree (AST) or Control Flow Graph (CFG) on which metrics are calculated.

Different authors use different metrics to identify functions with similar metrics values as code clones from source code. Mayrand et al. [23] use names, layout, expressions and control flow of functions as metrics. A function clone is identified as a pair of whole function bodies with similar metrics values. Davey et al. [24] detect exact, parameterized, and near-miss clones by first computing certain features of code blocks and then training neural networks to find similar blocks based on the features [9].

3.1.5 Graph-Based Techniques

Graph-based techniques ar a part of semantics-aware approaches of clone detection. This type of proposed methods uses static analysis than syntactic similarity. Some techniques use Program Dependency Graph (PDG) to represent the program. Each node represents expressions and statements, while edges represent control and data dependencies. Then isomorphic subgraphs are used to search for clone in the generated graph [9].

Combining structure and identifiers information, learning-based clone detection is also proposed in the literature [10]. From the repository, terms and fragments of source code are mined. A framework, which relies on deep learning is used to automatically linking patterns mined at the lexical level with pattern mined at the syntactic level.

Code clone detection for better maintenance of source code has a vast amount

of techniques and tools. Still, ongoing research focuses on clone detection with different techniques to reduce execution time, to scale-up with large cloneBench or improve the accuracy [21].

3.2 Clone Evolution

In clone evolution, researchers have conducted research based on subsequent versions, revisions of software, and a specified period of time. There is a number of studies focused on evolution mapping between subsequent versions. Considering incremental software development, clone evolution-based research can be categorized into one of the following sections.

- clone genealogy related
- clone evolving patterns related
- consistent and inconsistent change related
- real-time detection and tracking related

Now, each of these categories will be discussed elaborately.

3.2.1 Clone Genealogy Related

Evolution of Linux kernel is the first study in cloning evolution [12]. Observing nineteen releases of Linux system, it has been found that the Linux system did not contain a relevant fraction of code clone. Moreover, Code duplication was stable across releases which indicate a fairly stable structure of Linux kernel.

In this research, G. Antoniol et al. [12] used metric-based approach to detect clone at function level granularity. Although different metrics could be considered, authors used only those metrics accounting for layout, size, control flow, function communication and coupling. To get the cloning ratio between two systems, the

authors performed different activities such as handling preprocessor directives, function identification, clone extraction etc. Since C, C++ languages contain different programming language peculiarities (e.g. union, struct etc.), it was necessary to preprocess those directives. In function extraction, island-driven parsing was used. Island-parsing separates interesting parts of a program (features we are interested in) from uninteresting parts (other features, which need not be precisely parsed). Duplicate functions were identified using those metrics. To get an unbiased result of cloning ratio, functions had less than five lines were discarded. Because small functions like this mostly used for setting or getting the value of a structure. The result of the analyzed Linux kernel was that the amount of code duplication was very low, nearly 10%. Having modular structure in Linux it can be assumed that different sub-systems had contained more clones. But the result of the study shown that different sub-systems had low clone rates.

However, this study mostly focused on clone ratio in subsequent releases. Moreover, the study did not consider cloning pattern in evolving software and how inconsistency among versions may be happened due to adapt same functionality for different architectural sub-systems.

Göde et al. analyzed clone evolution from the perspective of clone changes frequency and harmfulness of changes [14]. After analyzing three mature software, the authors had found that most of the clones were rarely changed and unintentional inconsistent changes to clones were small. Effective management of frequently changed clones was needed. Moreover, stable clones created extra overhead unnecessarily. Addressing two questions, they wanted to know how many clones required careful attention during software maintenance time and how many of those were unintentionally happened during software development. In this study, the authors used incremental clone detection approach which inherently used suffix tree to detect clones in multiple versions of a system. Although pre-

vious studies did not consider the frequency of changes to individual clones and preserve risks inherently from unintentional inconsistencies, this study divided changes to clones into three categories namely never changed, changed once and changed more than once. Studying on three subject system, authors found 47.5% of all genealogies were never changed, 40.3% changed once and less than 8% of all genealogies were changed more than once. From the assessment of unintentional inconsistency, authors also divided the severity levels of those clone changes. Reporting 59.9% changes were consistent changes, authors divided clones severity into two categories such as low and high severity. This was done based on clones should be or must be changed consistently. Some of those changes were semantic preserving which preserves actual implementation of logic. And 14.8% of changes were unwanted inconsistent and only 3% of those were severe.

The change frequency of clones provides knowledge of which clones should require more attention. This study also refers that a tool is needed in development time which tracks the changes to clones and prevents unintentional inconsistency. Source code cloning is a significant threat to the maintainability of a software system. When the system evolves, problems usually start to arise. Though removing all duplicated code smell can be a good solution, often it is not possible. Tracking the evolution of code clone to identify those fragments which really causes problems in the future versions can be useful.

Source code cloning is a significant threat to the maintainability of a software system. When the system evolves, problems usually start to arise. Though removing all duplicated code smell can be a good solution, often it is not possible. Tracking the evolution of code clone to identify those fragments which really causes problems in the future versions can be useful.

Bakota et al. presented an approach to map clone fragments based on similarity measures and defined which clones were smelly compared to those occurrences

[25]. Tracking how individual clone changed through the versions of a system, the authors proposed a mapping procedure of clone for different versions. In this paper, the authors divided the approach into three steps. First, clone identification from all versions of the target system. Then, computing clone instances evolution for all the extracted clone instances in different versions. Lastly, identifying clone smell based on the clone instance mappings. In clone identification step, authors used the AST-based approach of [22] which was abstract syntax tree with semantic edges in the tree representation. By this approach duplicated codes in different granularity namely classes, functions, blocks, iterations can be identified with linear time and space complexity. Evolution mapping was used as a partial injective mapping to clone instances. The authors mentioned six features to evaluate similarity in clone pairs. Moreover, in this paper, authors described some causes that may manipulate clone instances and may create problems in subsequent versions. Vanished Clone Instance (VCI) indicates clone instance was not mapped onto any subsequent version. This can be happened due to corresponding code fragment was removed to remove redundancy in the source code, code fragments had changed so much that evolution mapping could not find any relation between clone fragments of subsequent versions, and code fragment was no more clone to other fragments. In the last case, clone inconsistency arose because developers might have been forgotten to update other clone instances. Occurring Clone Instance (OCI) smell means a clone has found in a version which was not derived from the previous version. This may be happened by a new clone was created or a code fragment was updated so that it has fallen into its original clone class. Modified Clone Instance (MCI) means a clone instance might have been modified in a way which became a clone instance to another class. Migrating Clone Instance (MGCI) may be happening due to a clone instance was modified and left other clone instances of a clone class without modification. Later because of any bug issues other clone instances updated similarly and re-create original

clone class. To check the usefulness of these smell, above approach was evaluated on Mozilla Firefox internet browser. Overall result showed that the approach can be used to identify bugs because of these smells.

3.2.2 Clone Evolving Patterns Related

Kim et al. defined a model of code clone genealogies and provided patterns of clone evolution [11]. In this study, the authors derived six patterns of clone evolution namely same, add, subtract, consistent change, inconsistent change, a shift in clone group. They also divided clone evolution into consistently changing clones, volatile clones, locally unfactorable clones, and long-lived clones. Observing two system carol and dnsjava, authors found 38% and 36% clone genealogies followed consistently changing pattern. The stability behavior of clone in those subject system showed that clones were volatile. Within 5 to 10 version, almost 50% clone disappeared and changed independently. Using standard refactoring techniques like pull up a method, replace conditional with polymorphism etc., can be used to refactor locally. Whereas, 64% in carol and 49% in dnsjava comprised locally unfactorable clone group due to some constraints such as language limitation. Most of the clone genealogies of long-lived clones were locally unfactorable. This information suggests that conventional refactoring is not the solution to remove clone and which clones require more effort in maintenance should be identified.

3.2.3 Consistent and Inconsistent Change Related

Change in all clone instances in a clone group referred as consistent change whereas missing some of the clone instances to update accordingly referred as inconsistent change. Considering revision level as chaotic and experimental development process, Bettenburg et al. conducted release level empirical study on inconsistent changes to code clones [2]. As a release of a software is the end product that users use so research on release level might be helpful to know clone evolution.

In this research, two open source software system namely Apache Mina and jEdit were used to study clones. The study provided a coarse-grained level knowledge of long-lived clone characteristics, the effect of inconsistent changes to clones and types of cloning patterns at release level. In the study, authors used SimScan [26] to detect clones in different releases. Inherently SimScan uses the AST-based clone detection. Clone Region Descriptor (CRD) [26] was used to map clone group of subsequent releases. CRD uses an abstract representation of a clone region in an AST which combines syntactic, structural and lexical information. For a single clone region, CRD contains the type of node e.g., method declaration or block and contextual information about node e.g., method's signature. By this CRD information clone group mapped between subsequent versions. In case of a miss matched in a clone group between two consecutive versions, they did a textual comparison and report changes. Changes were classified into one of the patterns reported in [27]. Authors findings confirmed that many clones were short-lived and changed independently over time. Long-lived clone group survived 6.79 releases and contained 2.34 clones. From two subject systems, 1.26% and 3.23% defects had found from 462 and 62 inconsistent changes of cloned code. To get the cloning pattern information in different releases, authors found that most of those are replicated and specialized cloning pattern such as API cloning.

Besides characteristics of clone genealogies, effects of inconsistent changes and cloning pattern at release level, Bettenburg et al. addressed six challenges to conducting research at release level [2]. Discrete changing information and changes between releases indicated information loss in intermediate revisions. This also incurred tractability problem of different clone group. Whereas irregular, parallel and selective releases are expected but difficult to address at release level.

Although this research had shown low inconsistency rate and developers were able to handle cloned code in their maintenance time, it is unlikely that all developers were able to handle clone changes and fixed a bug properly to all clone

instance of a group. Moreover, in development period, it increased the overhead of developers to track clone group changes. In this research authors mostly focused on gaining knowledge about inconsistent changes at release level. This information suggests that a real-time clone inconsistency tracker can be useful to maintain cloned code during development.

In software maintenance, it is assumed that change in one clone instance requires change is applied to others in a clone group. To study consistent and inconsistent changes to code clones, Krinke tried to validate two hypotheses [5]. In the study of five open source software systems, the author showed that usually half of the changes of clone groups were inconsistent change. Additional changes applied to near versions to get consistency among clone group was rare. Moreover, Krinke provided a framework for changes to retrieve consistency and lack of consistency information from consecutive versions [5].

In this study, the author used 200 weeks version history to know consistency and inconsistency in clones. First, CVS repository information was extracted and transformed which created each version repository to analyze changes. The simian tool was used to identify clone and clone groups. To get the change information of version repository Diff tool was used. A mapping of clone group and change to clone was classified and stored. From that information, roughly 45% - 50% consistent clone group and 50% inconsistent clone group have been reported. This study reflects that an approach is required to maintain consistency among clone group in development phase which allows the developer to track changes to clones. Inconsistency can be happened by overlooking code fragments for bug fixes, functional enchantment, and refactoring. How often unintended inconsistencies happen in due to overlooking code fragment is also exist in the literature. Higo et al. tried to derive modification patterns and detect overlooked code fragments in the source code to find relations with how often unintended inconsistencies happened in the repository. In a software repository, identical code fragments can be iden-

tified by GREP or code clone detection tool. After modifying source code these tools cannot help us to find inconsistent code fragments. To derive modification pattern, previous revisions of target system were analyzed. From the evolution of source code, modification patterns were extracted. This extracted information is used as modification quires on the target version of the system to get overlooked code fragments.

Applying the proposed method on FreeBSD and Apache HTTPD authors concluded that most of the past revisions contained unintended inconsistencies which finally modified in later revisions. These unintended inconsistencies arose from modification of bug fixes, functional enhancements, and refactoring. Based on historical information of previous revisions, authors proposed method could identify variable and statement level inconsistencies. Using SVN log and SVN diff commands of SVNKit tool, information was extracted. A numerical metric namely support was used to represent the number of equivalent modification patterns, confidence. And a probability method used to represent a code fragment was changed into another code fragment. On the subject systems, the precisions were 73.4% and 88.9% respectively.

3.2.4 Real Time Detection and Tracking Related

A clone increases maintenance effort and cost if a defect is found one of the clone instances in a clone group and it requires to inspect all clone instances of that group to fix the defect. So real-time detection of all similar clone instances is required. To support this type of maintenance tasks, a tool has been developed by Kawaguchi et al. [28]. In their paper, the authors propose how clone can be detected in real time. Before that, it is required to know programmers' behavior in clone detection and modification of existing clone. The authors have conducted two preliminary experiments to find out their behavior. First one is the number of clones exists in revised files when a functionality is added or fixes a bug. The second one is

observing three programmers coding, and interviewing eight programmers to know how they manage clones in the maintenance phase. The authors have implemented this proposed approach as Visual Studio Add-In and have been evaluated in a commercial CAD application [29].

In this paper, the authors use a sliding window technique [30] to get CVS information. Based on that information, authors have identified six problems. Those are (I) Programmers may not be aware of clone existences, (II) Novice programmers may not search and revise all clones instead of defective codes they found first (III) Lack of knowledge that where and how to search clones and necessity of revising the files, (IV) Using keyword search for duplicate instances in large software is difficult, (V) Difficulties to find a changed or modified clone instance, and (VI) Deciding revision strategy before revising the clones. To eliminate those problems, authors have proposed a real-time clone detector namely SHINOBI. SHINOBI consists of a SHINOBI server and SHINOBI client. In the server, CVS repository information is extracted and an index using suffix array technique is created. After that, CCFinderXs [29] preprocessor is used to parse the source code. It helps to unify identifiers to ignore identifiers difference. Mined information is stored in the SHINOBI mining data. By the SHINOBI client, the search key is sent to get ranked clone from suffix array index. Ranking value is the sum of the ratio of committed files at the same time and files edited or opened at the visual studio. In the IDE, there is clone list view and files information view from CVS information. Based on cursor location, it automatically finds clones of the corresponding code. This tool performs well in terms of execution time and memory because of its minimized quires in detection procedure. Authors also claim that comparing with GREP its performance has been good. So similar tools can be useful for maintenance tasks.

AlThough SHINOBI detects clone in real time, it does not provide clone inconsistency information. In addition, the tool only works on visual studio IDE. To

support other languages it should be included in the other platforms to manage code clone.

3.3 Summary

Although all of the above-described papers provide various knowledge in clone detection, clone genealogies in clone evolution, there is still a lack of appropriate approaches and tools to efficiently manage inconsistent changes in code clone. Having some proposed tools and techniques to track changes in the literature, a real-time inconsistency checker and notifier with integrated development environment (IDE) may help developers to effectively manage clone inconsistency.

Chapter 4

An Inconsistent Clone Change Tracking Technique in A Clone Group

4.1 Introduction

The existing works in code clones are mostly focused on clone detection [19, 7, 10] and clone changes in subsequent versions [12, 26, 14]. Authors have identified clones and derived patterns in clone changes [11]. Inconsistent change among different clone changes such as add, subtract, consistent, inconsistent changes are required more attention than others because studies have found that almost half of the changes in cloned codes are inconsistent [5]. It is necessary to manage cloned code changes especially inconsistent changes during software development. Although some of the studies in the literature focused on incremental clone detection [31] and real-time detection with plugin support in Visual Studio [28] but those are not solely concerning on inconsistent clone changes. In this report, a methodology for solving the problem is proposed. The aim of this method is to support developer to achieve consistency in clone changes within a clone group.

4.2 Inconsistent Clone Tracker

In this section proposed methodology is discussed to track the inconsistent changes. The proposed methodology can be broadly divided into two steps, first one is clone detection and the second one is change tracking in the cloned codes. In the proceeding sections, logical view of the model and each step is discussed elaborately.

4.2.1 Logical View of The Model

In this section, the logical structure of the model is discussed. According to the literature, different procedures such as syntactic, semantic etc. are used to detect code clone by considering different factors such as text, token etc. of the source code [9]. Afterward, subsequent version clones are mapped by those corresponding locations [25]. To track clone changes dynamic change tracking and resolution has also been proposed [32] in the literature. This proposed methodology is based on clone detecting and querying changed method fragment to find the duplicates of that fragment. This provides consistent/inconsistent changes information within the clone class/group of that fragment.

In the following subsections, the overview of the approach and the specific parts of it have been discussed.

Overview

Figure (4.1) depicts the overview of the proposed method. It consists of two steps which are *Clone Detection* and *Clone Change Tracking*. *Clone Detection* is used to collect information about cloned code from the source code. This information is stored in a repository. Then, based on clones similarity, clone classes/groups are created and stored in a file. *Clone Change Tracking* is used to detect changes in cloned fragment and check inconsistencies. Clone classes will be used as a repository to query clone instances of a code fragment where code fragment is a

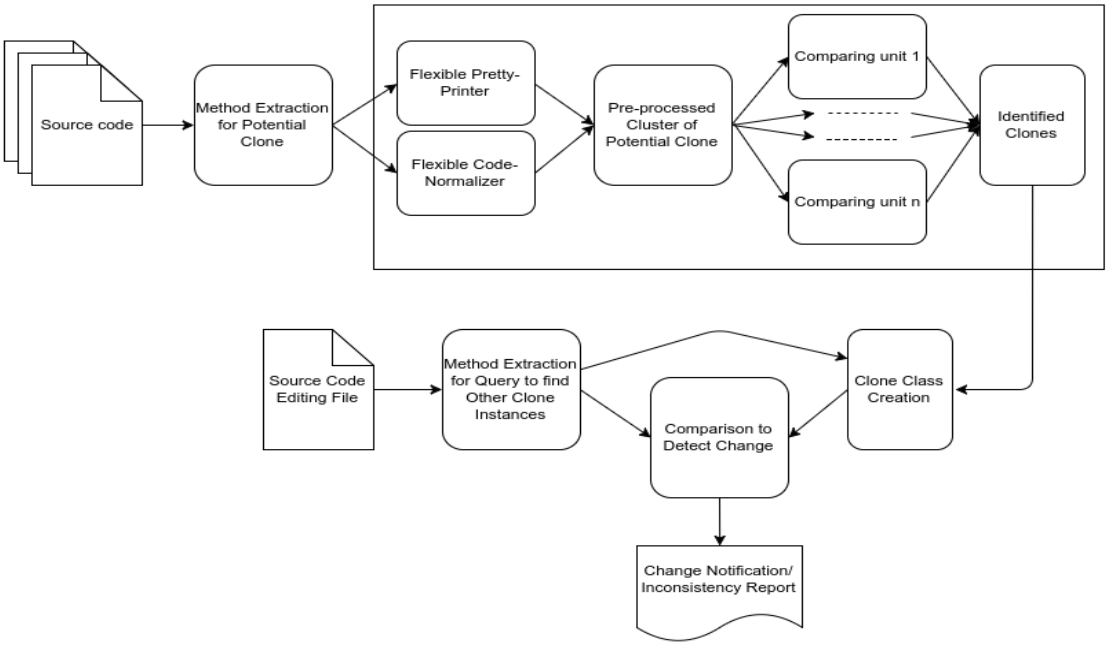

Figure 4.1: The Logical View of the Proposed Methodology

method. If the currently inspecting method is a clone instance of a clone group, then updated fragment and the previous fragment is compared to get the change information. In case of a change, other clone fragments of that clone group are going to be inconsistent. So the developer will be notified of the change. It also suggests changing other clone instances of that group by an editor notifier event. Afterward, clone log and group will be updated with the new code substituting the previous corresponding fragments. On the other hand, if there is no change occurred in the inspected or required code fragment then no inconsistency happens. So all clone instances remain same.

4.2.2 Clone Detection

Clone detection is the pre-requisite to track clone changes. To detect code clone in a source code repository, many approaches can be considered which are discussed in the literature review section. In this project text-based clone detection approach has been used. Figure 4.2 depicts the whole procedure simplified way with flow chart diagram.

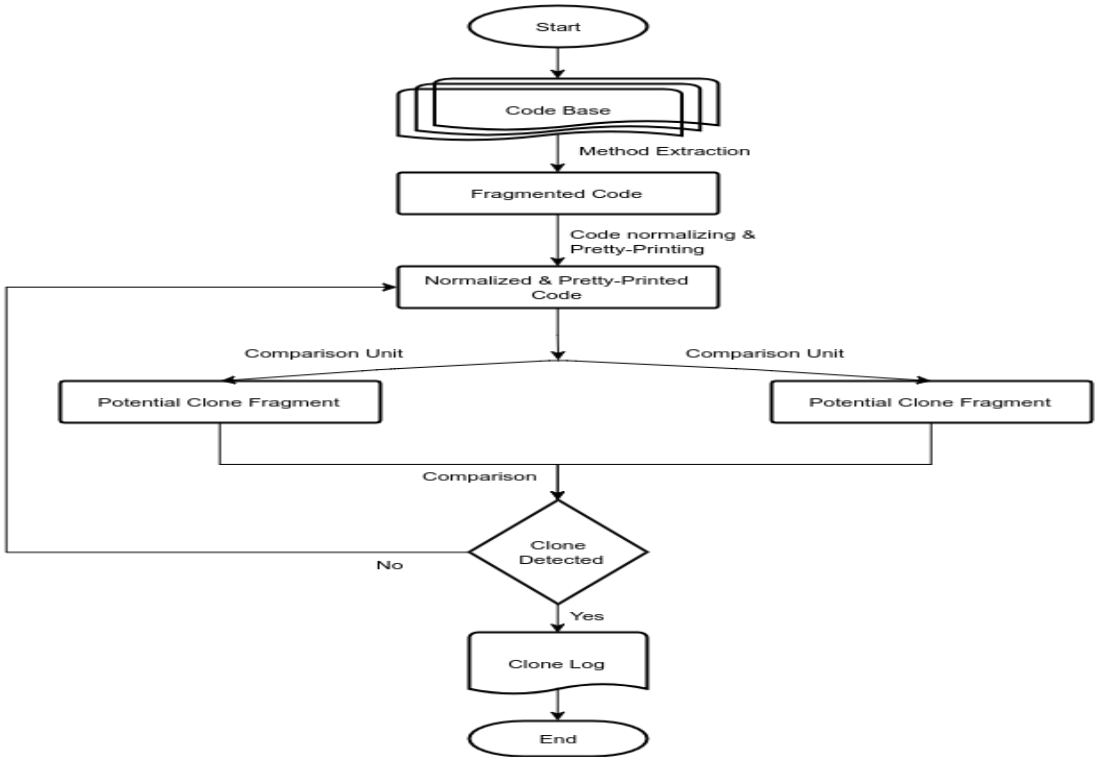

Figure 4.2: Clone Detection Process

A source code repository is parsed into a comparable unit to detect clones. This unit is called granularity level which will be used to find duplicate codes or clones. The granularity level can be a single token, a statement, a code block, a method or an entire class. Since a method is an implementation of an object's behavior or functionality, it is more relevant to be considered as the comparable unit. In this project, method level granularity is used. Moreover, method level granularity is used in the literature as well [8, 15].

Depending on developer tastes one could have written code with different layouts. Whitespace, comments and block markers are moved around according to the developer preference. So formatting all code fragments into the similar format is necessary. All potential clone fragments are stripped of formatting and comments, and prettyprinted by TXL [33] according to grammar's rules of TXL for a language. This standard pretty-printing guarantees that all code has the uniform layout and line breaks. This special pretty-printing helps to break different parts

of a statement into several lines so that local changes to the parts of a statement can be isolated using a simple line-comparison. These pre-processed potential clone fragments are used as comparing unit to find clones.

To detect clones, potential clone fragments are then compared with each other. In a comparison of two pretty-printed fragments, Longest common subsequence (LCS) algorithm is used. This incurs so much comparison in the large code base. To reduce the comparison Unique Percentage of Items (UPI) similarity technique is used. Based on similarity percentage a large amount of comparison is reduced. Details of the technique described in the paper nicad [8]. Since nicad is used for clone detection, default setup of the tool is used for clone detection.

4.2.3 Clone Change Tracking

After detecting the clone in the repository, change of the clone is being detected and tracked to get the change information. Through this information, decision can be made whether the clones are consistent or inconsistent in a clone group. To detect the changes in a cloned fragment in real time, a file wise mapping is created to get search result faster. Figure 4.3 illustrated the mapping procedure.

From the clone log, clones are mapped into the corresponding file. So a class may have a different number of clones. The mapping helps to reduce search space for which clone is changing now in the editor by the developer. It also helps to provide real time notification of inconsistency. In a scenario, where currently editing class does not have any clone, there is no need to check clones of that class's method fragments. The mapping is done based one file name, method signature, method body, start and end line number.

Using diff [34] utility, edited method fragment, and corresponding previous method are compared to get change information. If any change is detected by the diff then other clone instances of the clone group which are in the inconsistent state regarding with the edited clone instance is identified. After that, inconsistency

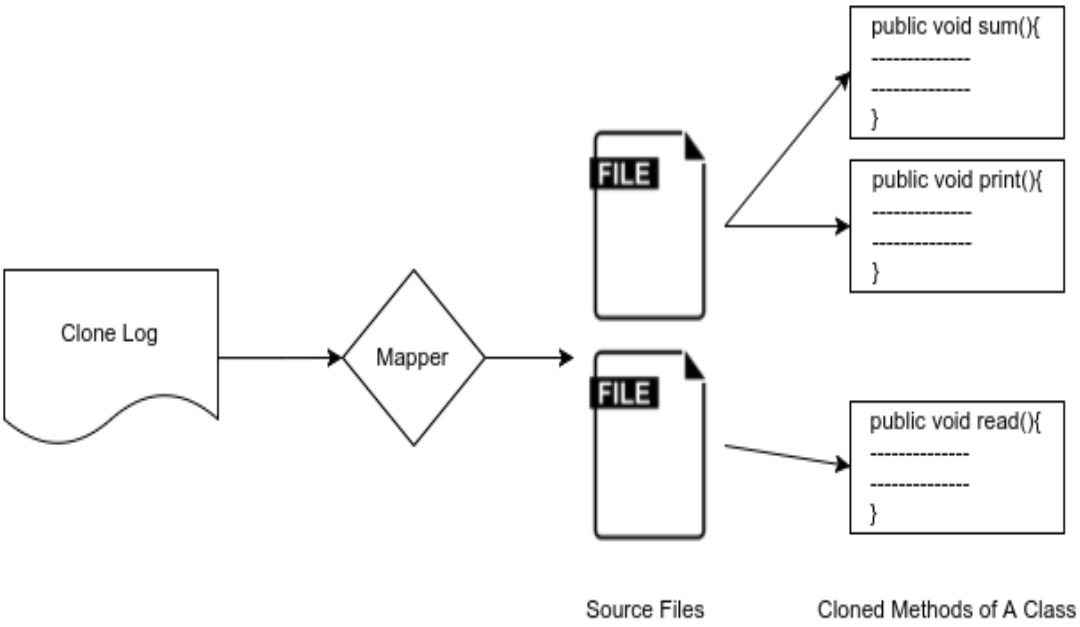

Figure 4.3: File Wise Clone Mapping

among clones are notified and changed fragment is shown with different label like insertion, deletion or modification in the editor.

4.3 Summary

From the above discussion, it is evident that the proposed approach can be applied to track inconsistency among clone instances within a clone group in real time. Clone detection ensures existing clones of the source codes. Section 4.2.2 discussed proposed methodology. In tracking phase updated code comparison with previous code is required to get change information. Section 4.2.3 discussed mechanism to do that. Thus the proposed technique can track clone changes in a clone group.

Chapter 5

Implementation and Result Analysis

Code clone changes in the evolving software and their effective management are research topics in the literature [2, 5, 32]. It is found that almost half of the cloned codes are inconsistent [5] and this lack of consistency increases the probability of bug creation. So it is necessary to fix in the upcoming versions. To detect code clone some plugins such as [28] and to effectively manage code clone plugins such as Clone-Board [32] are exist in the literature. Fully focusing on code clone inconsistency and how those can be managed during development, there exists a lack of proper tool in the literature. In this section, the implementation of the proposed methodology as a plugin and consequences of this supportive tool is discussed. The model is tested by manually changing fragments of some software such as JHotDraw.

5.1 Implementation Details

The proposed methodology was implemented in Java using IntelliJ IDEA 2017.1.5 Integrated Development Environment (IDE) [35]. An Object-Oriented Program-

ming language such as Java mostly uses in the development. It provides extensive support for polymorphism, inheritance, and encapsulation which are necessary for reusability. Moreover, this research project is detecting method level clone change inconsistency tracking in software which is written in Java language. This is why it was chosen to implements the approach. The following tools and libraries were used to develop the system.

- Txl [33]: TXL is a unique programming language designed to support computer software analysis and source transformation tasks. It is the evolving result rule-based structural transformation. This is used as pre-requisite of Nicad4 to detect clone.

- Nicad4 [8]: NiCad is a flexible TXL-based hybrid language-sensitive / text comparison software clone detection system developed by James R. Cordy and Chanchal K. Roy. NiCad4 is a significantly improved and optimized version. It is a scalable, flexible hybrid clone detection method. This is used to detect clone in the repository.

- JUnit 4.12.0 [36]: JUnit is a library that helps to write unit tests in Java. So, it was used to write unit tests for the modules in the system.

- Intellij PSI [37]: PSI is the Program Structure Index in IntelliJ. It is the layer in the IntelliJ Platform that is responsible for parsing files and creating the syntactic and semantic code model. This is used to parse the source code and to retrieve method from the cursor.

- IntelliJ Plugin Development [38, 39] Plugins are extensions to IntelliJ IDEA core functionality. They provide support for various development technologies, frameworks and programming languages, and so on to the IDE. It is used as the platform to fully implement the methodology.

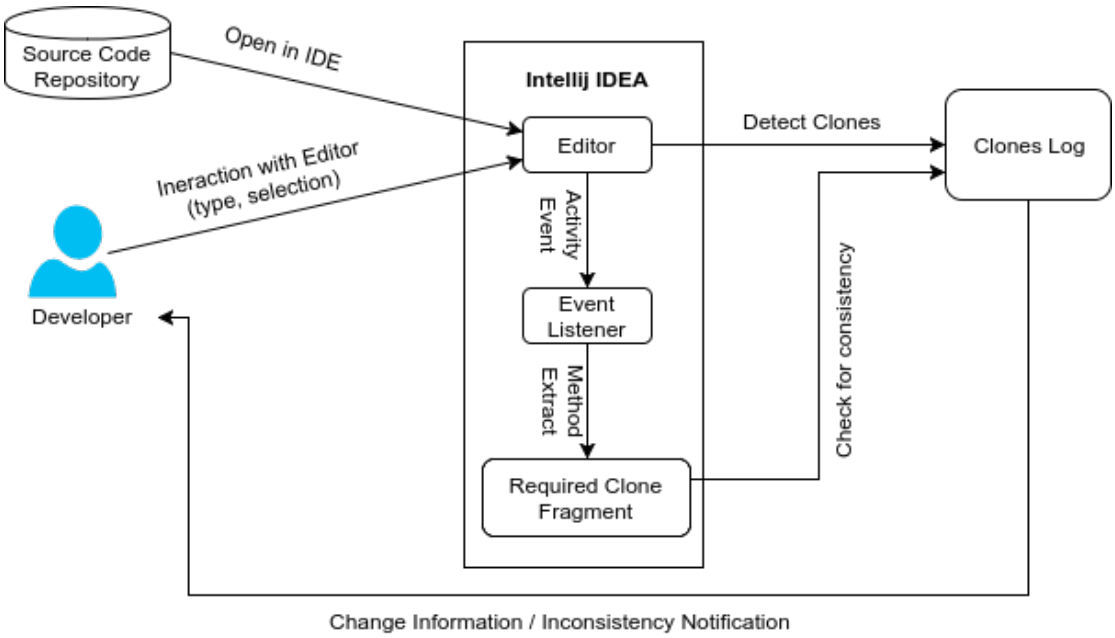

Figure 5.1: Plugin Interaction with Developer

- Java-diff-utils 1.3 [34] Java Diff Utils library is an OpenSource library for performing the comparison operations between texts: computing diffs, applying patches, generating unified diffs or parsing those. It generates diff output for easy future displaying like the side-by-side view. It used for getting change information of cloned fragments.

The implementation was executed on a computer with the following configurations.

- Operating System: Ubuntu 16.04 LTS
- RAM: 8.00 GB
- CPU: 3.30GHz Intel Core i7 Processor
- Platform: 64 bit

The proposed methodology is implemented as an IntelliJ IDEA plugin. Because In IntelliJ IDEA is a great and mostly used editor for the Java developers. Figure 5.1 depicts the overall scenario of the interaction between the developer and the editor.

While a source code repository is opened in the IDE, the plugin automatically detects code clone in that repository. Through a background *service* code clone has been detected and stored in a file in XML format. In any time clone log can be updated by *clone detection* action of the editor. From detected clones, clone group or clone class is formed based on similarity of clones. At any time, all clones of a selected fragment can be shown through *show clones* action. To check inconsistency of a clone group a fragment must be selected. The selected fragment should be a method since we are interested to track method level inconsistency. After selecting a fragment changes of that clone is shown with other clones of that group. Through the plugin, it is possible to jump one of the inconsistent fragment and fix accordingly.

5.2 Tool Description

The proposed approach is implemented as an IntelliJ Idea plugin. Clone detection and real-time change information notification is required to resolve clone inconsistency within a clone group. To support the developer in their development the tool should be incorporated into their development IDE. Since IntelliJ Idea is so much popular to developer their Java-based project so the IDE is selected as integration environment. Below plugin mechanism is described by *plugin.xml* file.

Figure 5.2 shows the plugin window extensions. In the plugin there is two window. *Clones View* tool window is for clones of a clone group which is found by

```
27    <extensions defaultExtensionNs="com.intellij">
28        <!-- Add your extensions here -->
29        <!-- Declare the project level service -->
30        <applicationService serviceInterface="services.ProjectInstance"
31                            serviceImplementation="services.ProjectInstance">
32        </applicationService>
33
34        <toolWindow id="Clones View" anchor="right" factoryClass="views.ClonesView"/>
35        <toolWindow id="Clone Changed View" anchor="right" factoryClass="views.ClonedCodeView"/>
36    </extensions>
```

Figure 5.2: Plugin's Extensions

```
<actions>
    <action id="CaretPosition.FragmentExtractor" class="actions.CaretPositionFragment"
            text="Show Clones" description="Get all clones of selected Fragment">
        <add-to-group group-id="EditorPopupMenu" anchor="first"/>
    </action>
    <!-- This action extract code fragment from selected model -->
    <action id="SelectedFragment" class="actions.SelectedFragment" text="Select Fragment"
            description="a method fragment is selected for searching its clone instances">
        <add-to-group group-id="EditorPopupMenu" anchor="first"/>
        <keyboard-shortcut keymap="$default" first-keystroke="ctrl S F"/>
    </action>
    <!-- This action compare changes in a cloned fragment -->
    <action id="ShowChanges" class="actions.ChangeDetection" text="Show changes"
            description="a method fragment is selected for searching its clone instances">
        <add-to-group group-id="EditorPopupMenu" anchor="first"/>
        <keyboard-shortcut keymap="$default" first-keystroke="ctrl S F"/>
    </action>
    <group id="Plugin.CloneTrack" text="CloneTrack" description="Custom Menu for SPL3">
        <add-to-group group-id="MainMenu" anchor="last"  />
        <!-- This action detect clones in the whole repository -->
        <action id="Plugin.actions.CloneDetectionAction" class="actions.CloneDetectionAction"
                text="Detect Clone"description="clone detection in the whole repository" />
        <action id="actions.MethodFragment" class="actions.MethodFragment"
                text="Method Fragment"description="Method Fragment Extraction" />
    </group>
</actions>
```

Figure 5.3: Plugin's Actions

detecting from a code fragment. Another one is *Clone Changed View* for showing the changes of a cloned fragment regrading with the clone instances of a clone group.

Figure 5.3 shows the actions need to be performed to get required result. Through these actions (in the picture) clone fragments are shown in the plugin window. Performing different tasks such as show clones, show changes one can easily find out clone change information and take decision to resolve inconsistency.

In the following figure, plugin *Project Component* is shown which is used for detecting clones in the repository whenever a project is opened in the IDE 5.4. This is a back-end service which is used only when the project is opened.

In plugin development, Nicad tool is used to detect clones in a repository [8].

```
65      <project-components>
66          <component>
67              <implementation-class>services.CurrentProject</implementation-class>
68          </component>
69      </project-components>
```

Figure 5.4: Plugin's Component

Which inherently uses TXL for parsing the source code [33]. The plugin is used with IDE in Linux operating system. The plugin performs well in better computer configuration.

Tool View

The plugin associated with IntelliJ Idea provides tool window view for changing any cloned code. Figure 5.5 shows clone change view. In the left side of the window shows the changes occurred in a clone fragment. This information is shown with different labels like modification, insertion, and deletion. This indicates the action performed in the currently editing code. The right side of tool window shows other clone instances of the clone group. It has a drop-down button to view complete fragment of other clones. It helps the developer to decide how current changes need to be applied to other clones accordingly.

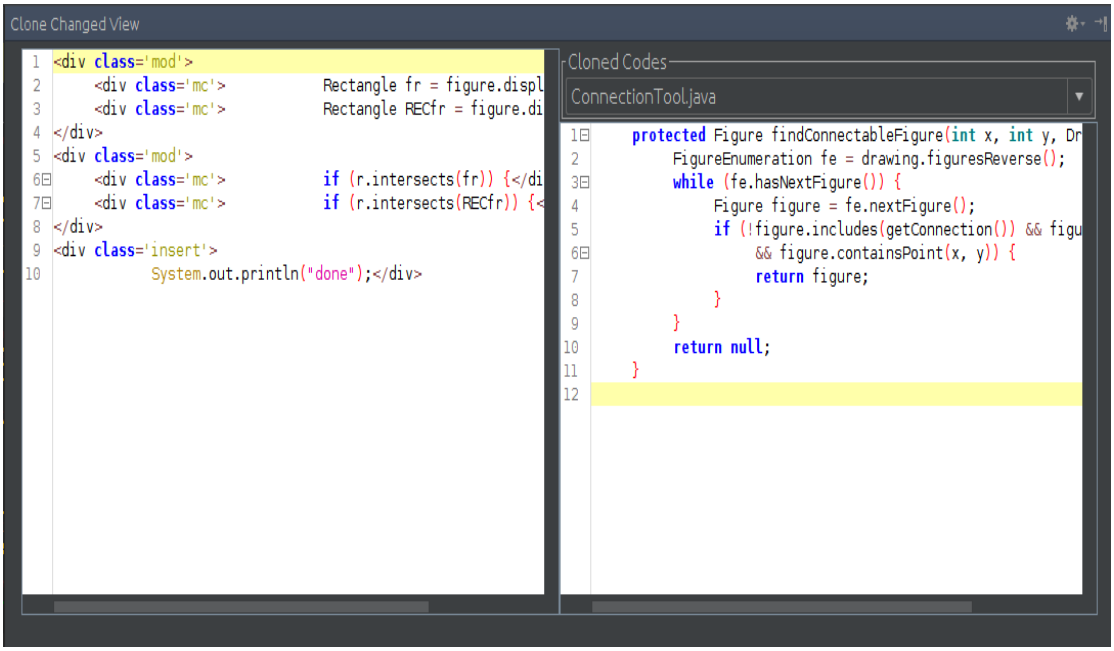

Figure 5.5: Plugin Tool View

5.3 Case Study

In academic or research works controlled experiments are conducted to evaluate tools and techniques [40]. A controlled experiment is also performed for this project. This experiment is performed to evaluate the proposed approach and developed plugin as clone inconsistency tracker.

From the perspective of Goal, Question, Metrics (GQM) [41], the study has been conducted. The *Goal* of the experiment was how effective the plugin was in development time to manage code clone inconsistency. In the experiment, the plugin was set up with the necessary environment with the idea in three different computer. Three developers developed their java based project with the IntelliJ Idea. In the development phase, they used the plugin to detect and track clones in the project. The experiments duration was 2 days. After using the plugin, developers were asked to answer some questions about the usefulness of the plugin. *Questions* were selected focused on the *Goal* of the experiment. *Questions* are given below:

- How many times you use the plugin support during development?
- Is clone view helpful to know the clones of a selected code fragment?
- Is clone change information supportive?
- Is clone change information descriptive enough to make a decision whether changes should be updated other clones or not?
- How many times you resolve inconsistency of clones by this support?

From the answer of those questions, some *Metrics* were derived to measure the effectiveness of the plugin. Those are

- Number of uses
- Supportive
- Descriptive information
- Number of inconsistencies resolved

After analyzing the accumulated result of developer's answers, it was found that on an average 5 times developers used the plugin support. They rated 8 on a scale of 1-10 that the plugin support was helpful. Average value of descriptive information was 7. Total 10 times they resolved inconsistencies of clones from the change information view of the plugin. Although the result was less significate by means of usefulness, it was promising to be worthwhile in clone consistency management during software development.

From the study, it can be concluded that the plugin is useful in code clone changes in a clone group. Along with developers, the author also thinks that considering a long history of clone changes information may be more useful to manage evolving clone inconsistency. This will be considered in the future work.

5.4 Result Analysis

The proposed approach is implemented as a plugin and evaluated to know the effectiveness. Three open source projects namely *JHotDraw, dnsjava* and *Eclipse JDT Core* are used to evaluate the plugin [42, 43, 44]. *JHotDraw* is a Java GUI framework for technical and structured Graphics. *dnsjava* is an implementation of DNS (Domain Name System) in Java. It supports record types, queries, zone transfers, and dynamic updates etc. *Eclipse JDT Core* is the Java infrastructure of the Java IDE. It provides different types of API such as Java compiler, Java element tree etc.

From three software, code clones are detected using the Nicad tool. After detecting clones, based on the similarities clone classes are formed. It is found that each of those software contains 25, 36 and 470 clone classes. From the clone classes the maximum number of clone instances are 9, 9 and 34 respectively. Table 5.1 shows the experiment software with those these information.

Table 5.1: List of experiment software with clone Information

No	Application	Software Category	Number of Clone Classes	Max. Number of Clone in a Class
1	JHotDraw	GUI Framework	25	9
2	dnsjava	Java DNS	36	9
3	Eclipse JDT Core	Eclipse API	470	34

After detecting clones and classifying into different classes based on similarity a repository is created. The repository is used to track the changes into one of the clones from one of the classes. With plugin support in the IDE, source codes of the mentioning software are opened one by one. Manually 2 files

clones are manipulated to get the change information. Changing one of the clone instances of a clone class, the plugin can notify the changes with the previous fragment. It can also show the other clone instances of that clone class.

The accuracy of clone tracking is dependent on clone detection. By using Nicad [8] as clone detector and changing only one clone fragment in a file, the accuracy is 100%. In case of multiple change accuracy is reduced and varied depending on clone location is changed completely or not. Because changing in a fragment may create a different type of clone with a different class. So again clone detection is needed to get the updating effects.

By diff utility, change is detected and shown. So this may provide a false positive result. However, after clone detection, any change in clone can be considered as a change. Because it may change the clone location.

Performance

The performance of the plugin is dependent on whether the opening file is too large or small. It requires indexing of potential clones and clone detection on the opening of IDE. In change detection, it is fast enough to provide a response in tool window namely clone changed view. The performance may degrade by many time clone detection since the tool is not incorporated with automatic update.

Usability

The overall usability of the plugin is good because it has some strict requirement to use in development. In clone information view, it dependents on user action to show clones. To detect any change in clone fragment, it is needed to mark the whole method fragment to get the change information. It also dependents on user show change action. So the plugin has some limitations in use.

5.5 Summary

This chapter provides the implementation details, case study and result analysis of the proposed method. The proposed approach is implemented in Java and demonstrated in IntelliJ IDEA plugin. This plugin supports clone management through detect clones in the repository and tracks those changes. It shows similar clones of a code fragment. In case of inconsistency among clones, it provides a useful view of clone changes. This helps to take decision whether the change should be updated into other clones or not. Through a study of five developers in their project development, it is found that the tool is useful in clone inconsistency management.

Chapter 6

Conclusion

In this report, a code clone inconsistency tracking methodology is proposed and based on that methodology a plugin is developed to support clone management. The core contribution of this work is the development of a real time change tracking of cloned code with an IDE integrated tool such as Plugin. This chapter summarizes the report and concludes by providing direction for future work.

6.1 Discussion

This project is done by focusing on clone inconsistency during software development. Literature has shown that most of the clone related bugs are created from inconsistent clone change. Because similar fragments may require similar update based on the requirements. However, the missing adaptation of some of those clones leads to cloning related bugs and need to be fixed in later versions. So a plugin is developed to resolve the problem.

This report proposed an approach to track clone changes in a clone group. Change in one fragment is detected through the plugin. Relevant change information of a clone is shown with its other clones in a clone group. This is evaluated with three sample software by manually change fragments. The plugin is performed well in this regard.

6.2 Threats to Validity

In this section, the validity of the proposed method and observations resulting from the experiment is discussed. Here two types of validity can be considered namely internal validity, external validity. Internal validity focuses on proposed method's cause and effects in each step and external validity focuses on the outcome of the experiment of the methodology as a plugin.

6.2.1 Internal Validity

Internal validity can be divided parts three parts, one is clone detection and another one is matching to get the change information. In clone detection, text based approach is used with pretty-printing and code normalization [20]. So the performance of the tool in clone detection affects the outcome of the proposed methodology. Here other approaches like token-based, AST-based or metrics-based approaches can be used so that results are also depended on those approaches validation.

In comparison to previous clone fragment and changed or updated code fragment, string matching algorithm is used. Here, before comparing two fragments, converting both of those into token, AST or metrics may have an affect in the results. This also limits the validation of proposed approach.

The plugin can not support with multiple changes in clone fragment because of clone information lost after changes. This invalidates the approach with the text based comparison. Fragments modification should be stored as tree-based architecture so that changes in fragments can incorporate simultaneously in the node. In the implemented plugin, clones have detected while the IDE is open. Afterward, clone change inconsistency has been tracked. Clones log can be updated at any time by an action. Here, previous versions of the software or changelog in the commit history have not been considered. So the relation with commit history is

missing. Although any time clone detection provides a convenient way of tracking clone changes this also incurs lack of validation of the proposed methodology due to not considering past history.

6.2.2 External Validity

To evaluate the proposed method as clone inconsistency tracking relies on the advantages it provides during software development. This depends on the usefulness of the plugin which is integrated into the IDE. Since the plugin is not used in a real-life project, general claim or result cannot be validated. To resolve this problem, this tool needs to be used in a software project. However, this project duration is only six months. Within this short period of time, an experiment in a real-world software development is not possible.

Moreover, the usefulness of the tool also depends on the developer. Developers are the only actor who can use and get the advantages of the tool. In a development team, there exists novice to expert developers. Their behavior to use an additional tool to manage code clone need to be considered. Since before developing the tool, a survey was not conducted, how they use the tool and get benefited from the tool cannot be validated. However, experimentation of Kawaguchi et al. have considered resolving the issue [28].

At last, a case study is also required to validate the effectiveness of the tool. The study reports performance of two teams where one team use the tool and others are not in software development. Since this also cannot be conducted due to lack of fund, time and proper arrangement, this also comes as an external threat to validate the proposed method and the implemented tool.

6.3 Future Work

The proposed methodology provided a code clone inconsistency tracking mechanism with real time notification integrated in an IDE. In the method, all clone instances of a clone group are notified by the editor about any change in one of those clone instances. This is required to apply the proper change on those and create consistency with all clone instances of that group. However, some of those clone instances may not have relation with bug propagation. Because the group of clone instances is created by the similarity level of clone instances. In this case, exact similar clones are fully related to each other but Type-2 or Type-3 clones are may fully not related with each other. So considering data flow and control dependency should be considered in clone group creation. This might be useful to get proper inconsistency information of clone instances.

In future work, data flow and control dependency will be considered in clone group creation. Thus in case of inconsistency among clone instances of one group, Related and should be modified clones will be notified only. This can help a developer to take decision easily weather regarding change should be propagated to all others or some of those. Moreover, past history of commit log can be considered to effectively manage clone inconsistency.

Bibliography

[1] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 55–64, ACM, 2007.

[2] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. E. Hassan, "An empirical study on inconsistent changes to code clones at release level," in *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 85–94, IEEE, 2009.

[3] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort?," in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pp. 1–7, IEEE, 2017.

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

[5] K. Jens, "A study of consistent and inconsistent changes to code clones," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pp. 170–178, IEEE, 2007.

[6] Y. Higo and S. Kusumoto, "How often do unintended inconsistencies happen? deriving modification patterns and detecting overlooked code fragments," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 222–231, IEEE, 2012.

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

[8] "Nicad4 clone detector." `http://www.txl.ca/nicaddownload.html`.

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

[10] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 87–98, ACM, 2016.

[11] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 187–196, ACM, 2005.

[12] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Analyzing cloning evolution in the linux kernel," *Information and Software Technology*, vol. 44, no. 13, pp. 755–765, 2002.

[13] A. Goon, Y. Wu, M. Matsushita, and K. Inoue, "Evolution of code clone ratios throughout development history of open-source c and c++ programs," in *Software Clones (IWSC), 2017 IEEE 11th International Workshop on*, pp. 1–7, IEEE, 2017.

[14] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 311–320, ACM, 2011.

[15] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "Sourcerercc: Scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pp. 1157–1168, IEEE, 2016.

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

[17] 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: software engineering-Volume 1*, pp. 171–183, IBM Press, 1993.

[18] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[19] J. H. Johnson, "Substring matching for clone detection and change tracking.," in *ICSM*, vol. 94, pp. 120–126, 1994.

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

[21] "Ambient software evoluton group. ijadataset 2.0.," 10 October 2017. `https://sites.google.com/site/asegsecold/projects/seclone`.

[22] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pp. 253–262, IEEE, 2006.

[23] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics.," in *icsm*, vol. 96, p. 244, 1996.

[24] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley, "The development of a software clone detector," *International Journal of Applied Software Technology*, 1995.

[25] T. Bakota, R. Ferenc, and T. Gyimothy, "Clone smells in software evolution," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 24–33, IEEE, 2007.

[26] E. Duala-Ekoko and M. P. Robillard, "Tracking code clones in evolving software," in *Proceedings of the 29th international conference on Software Engineering*, pp. 158–167, IEEE Computer Society, 2007.

[27] C. Kapser and M. W. Godfrey, ""cloning considered harmful" considered harmful," in *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pp. 19–28, IEEE, 2006.

[28] 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 *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pp. 313–314, IEEE, 2009.

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

[30] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[31] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based code clone detection: incremental, distributed, scalable," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pp. 1–9, IEEE, 2010.

[32] M. De Wit, A. Zaidman, and A. Van Deursen, "Managing code clones using dynamic change tracking and resolution," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pp. 169–178, IEEE, 2009.

[33] "The txl programming language "source transformation by example"," 20 August 2017. `http://www.txl.ca/`.

[34] "java-diff-utils." `https://code.google.com/archive/p/java-diff-utils/`. accessed 20 Nov. 2017.

[35] "Intellij idea." `https://www.jetbrains.com/idea/`.

[36] "Junit - about." `http://junit.org/`. Online; accessed 29 Nov. 2014.

[37] "Intellij idea psi." `https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi.html`.

[38] "Plugin development guidelines," 1 October 2017. https://www.jetbrains.com/help/idea/plugin-development-guidelines.html.

[39] "Intellij platform sdk," 10 November 2017. https://www.jetbrains.org/intellij/sdk/docs/welcome.html.

[40] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[41] V. Caldiera and H. D. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, vol. 2, no. 1994, pp. 528–532, 1994.

[42] "Jhotdraw (two-dimensional graphics framework)." https://sourceforge.net/projects/jhotdraw/.

[43] "Dnsjava." http://www.dnsjava.org/.

[44] "Eclipse jdt core." https://www.eclipse.org/jdt/core/.