Studying Late Propagations Using Software Repository Mining

Master's Thesis

Hsiao Hui Mui

Studying Late Propagations Using Software Repository Mining

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Hsiao Hui Mui
born in Terneuzen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Studying Late Propagations Using Software Repository Mining

Author: Hsiao Hui Mui
Student id: 1047701
Email: hsiaomui@gmail.com

Abstract

While source code cloning is generally regarded as a bad software development strategy, there are, however, several studies which seem to indicate the contrary. One of the clone evolution patterns that can potentially lead to software defects is called the Late Propagation (LP). A LP occurs, when related clones are changed consistently, but not at the same time. The clone instance, which receives the update at a later time, might exhibit unintended behaviour if the modification was a bugfix. In this paper we present an approach to extract late propagations from software repositories. Using this approach, we will study LPs in four software system, which allows us to investigate the propagation time, commit activity and the effects of LPs. Finally, we present a number of suggestions to improve on the work done in this thesis.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. M.B. van Riemsdijk, Faculty EEMCS, TU Delft

Preface

This document describes the work I have done for my master thesis at the Delft University of Technology during 2009 - 2010. Firstly, I would like to thank my supervisor, Andy Zaidman, for helping me to find a very interesting research topic and for his advice, which has guided my work in the right direction and significantly improves the structure of this thesis. Finally, but not the least, I would like to thank my parents for their support and for giving me the opportunity to study.

Hsiao Hui Mui
Poortugaal, the Netherlands
September 16, 2010

Contents

List of Figures

Chapter 1

Introduction

Several studies have shown that about 7% to 23% of the code in a large software system contains duplicated source code fragments [44, 3]. These code fragments are called code clones and they are usually the result of copying and pasting code.
There are a number of reasons why duplicating source code is a common activity in software development [30]. If a programmer is evaluated by the number of lines of code he or she writes, code cloning would be an easy and fast way to increase the amount of code produced [3]. Inexperienced programmers may write more duplicated code fragments than veteran programmers and they have more problems managing them as well [29].
Code cloning may be a way to create variant modules. When for example a new driver is needed for a new hardware device, a similar hardware family may already have an existing driver. It would be easier to make minor modifications to the existing driver and use it than building everything from scratch [28]. Finally clones may be created due to the limitations of the programming language or libraries used by developers [30].
Code clones are generally assumed to be harmful [17, 24, 39, 26] and Fowler [18] ranked them as ‘no1 in the stink parade‘ of bad smells for the following problems:

- Code cloning can potentially lead to code bloat, which unnecessarily decreases code readability and maintainability.

- Code duplication is often an indication of bad software design like missing inheritance or missing abstractions [17].

- If the original code fragment is bugged, then copying this fragment will only spread bugs around the system. Chou et al. [14] have found that in a single source file under the Linux drivers/i2o directory, 34 out of 35 errors were the result of copy and paste activities. One of the errors was copied in 10 places and the other in 24 respectively.

- Modification of one code clone fragment means that it is necessary to find all related code clone fragments in order to change them consistently. If clones in a code clone group are changed inconsistently this may lead to unintended behaviour in the unchanged clones.

There are, however, a number of studies which seems to indicate the contrary [28, 31]. A study by Kapser and Godfrey [28] has identified eleven cloning patterns and they have found that cloning is an useful software development tool under some circumstances.
Currently, it is still uncertain which of these two opposing visions on the harmfulness of cloning is the right one or whether it depends on the software system analyzed [8].

1.1 Problem Statement

In order to study the effects of source code duplications, it is necessary to track these clones across different version of a software system.
By mining a repository of a software system, it is possible to observe how code clones are changing and correlate the clone evolution with bugs or maintenance problems [31].
Software repository mining employs data mining techniques to extract useful information from a large dataset [49]. Version control systems (VCS) contain a large amount of data on the history of a software system and mining VCS would allow us to find interesting information on the evolution of code clones in the source code repository. Well-known examples of version control systems are CVS and Subversion. Another useful software package is the issue tracking system (ITS) as it can be utilized for the purposes of finding and tracking of bugs and examples of ITS are Bugzilla, Jira and Trac [40]. So by mining those two systems we can see how code clones evolve and we can also see the impact it have on the underlying software system in terms of bugs and other issues.

Kim et al. [31] investigated the evolution of code clones and they found patterns of clone evolution, which can be used to gain insight on the effect of a specific clone evolution occurrence. Aversano et al. [2] expanded on the research of Kim by adding two new patterns and this thesis is focused on one of Aversano's clone evolution pattern which is shown in figure 1:

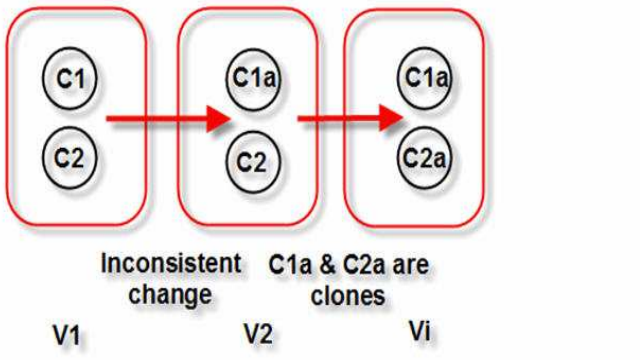

Figure 1.1: The Late Propagation clone evolution pattern.

In figure 1 we see two code fragments C1 and C2, both belonging to the same clone group. In the next version C1 is modified while C2 is not, which means that both clones are now inconsistent. In version Vi the missing update has finally reached code fragment C2 and both clones are now consistent again.
This clone evolution pattern is called *Late Propagation(LP)* and is defined as follows:

> *"a change is propagated consistently across clones, however this does not happen immediately. In other words, while a clone is maintained at revision k, others will be maintained at revision k + j."*

This kind of code clone occurrence can happen if a developer forgets one or more code fragments in a clone group when updating and adds the missing changes at a later date. If the update was intended to fix bugs then this type of inconsistent change can be dangerous as the unchanged clone instances will exhibit unintended behaviour until they are updated. Thus, software problems can be avoided if software developers have access to a clone management tool [15, 16], which can track code clones and prevents LPs from occuring. In this thesis, a clone management tool is made that will track LPs through repository mining and this leads us to the following central research question:

Central Research Question: *Can we study late propagations using software repository mining?*

In our previous literature study, where an evaluation of clone detection tools was conducted, we have taken the first step in answering the central research question by selecting a suitable clone detection tool. The next problem is to find a method to map code clones from one version i to version i + 1.
This lead us to following sub-question:

SQ1: *How do we track code clones in a software repository?*

The ability to track code clones, however, is not sufficient as we have to distinguish the late propagations from other code evolution patterns. This means that we have to consider the following issues [22]: Firstly, if we take figure 1 again into account, in version 2 code fragment C1 is changed and obviously both code fragments do not belong to the same clone group anymore. For a clone tracker, it would seem that this clone group disappears in version 2 and somehow a new clone group reappears in version i. When the clone group is larger, we run into a similar problem, a clone tracking tool would see one or more code fragments migrating to a new clone group.

This lead us to the next sub-question:

SQ2: *How do we determine if a late propagation has occured?*

To solve both subquestions, a tool is developed that can mine source code changes from a software repository. It will analyze the modifications made to a code clone and calculate the evolution pattern from those changes. With the ability to detect late propagations, we can then study them by answering the following subquestions:

- **SQ3:** *Is there a connection between the package distance and the propagation time of LPs?*
  If related clones are widely scattered across a code base, it would be difficult to track them down and this has as an effect that developers are more likely to forget them. As a result, it would take longer to update these clones.

- **SQ4:** *To what extent does the commit activity influence the appearances of LPs?*
  If a programmer is busy modifying several software components in one revision, he might overlook some code clones and this would lead to LPs. In order find out if this is the case, the change view visualisation technique by Zaidman et al. [48] is used to find out if there is a link between busy commits and late propagations.

- **SQ5:** *What are the effects of LPs?*
  As indicated earlier, LPs may cause bugs as related clones are not updated consistently. The commit logs of a software project is analyzed to determine if a LP has resulted into a software defect.

1.2 Outline

This thesis is structured as follows: In chapter 2 we will give some background information on code clones, clone detection and the relation between code clones and software evolution. In chapter 3 we describe the design and implementation of our clone tracker tool. The experiments are performed in chapter 4 and finally in chapter 5 conclusions are drawn.

Chapter 2

Background and Related Work

In this chapter we will give some background information on the subjects related to the work done in this thesis. In section 2.1 we will give some basic definitions related to code clones, like clone groups, clone typology etc. The various techniques for clone detection will be shown in section 2.2. Software evolution will be the final subject discussed in this chapter.

2.1 Code Clones

While code cloning has been the subject of research in many publications, there is, strangely enough, currently no precise definition of a code clone that is commonly agreed upon by the code clone research community. As a result, there exist various definitions of code duplications in literature, and, to make matters worse, some of these definitions are either inconsistent or too vague.
Before we begin to "define" a code clone, it is useful to know what a code fragment (CF) is. A fragment is a piece of source code that can be defined as a triple (file, beginline, endline). A CF is, thus, a sequence of source code lines with file being the file location where it appears and having begin- and endline numbers. In this thesis, a code fragment usually refers to a function definition, begin-end block or a list of statements.

The definition of code clone that is adopted in this thesis is proposed by Basit et al. [5]:

"Code clones, or simply clones, are code fragments of considerable length and significant similarity."

The above definition states that code duplications are code fragments of significant length and similarity. We are going to define "significant length" by stating the the minimum clone size used in this study. Here we run into a similar problem as defining code clones, some studies [26, 27] prefer to define minimum clone size in terms of tokens, while others use lines of code (LOC) [7]. We follow the definition used by Bellon et al. [7] by using 6 lines of code as minimum. As for similarity, there are two kinds of similarities: two code fragments can be similar in their program text or they can be similar only in their

semantics.
Clones of the first kind are usually the result of copying and pasting of code and these are exactly the kind of clones that we are looking for when we want to detect late propagations.

There exists three different types (Types 1-3) of textual similar clones [7]:

- **Type 1:** Code fragments that are completely identical, except for variations in whitespace, comments and formatting. Type 1 clones are also known as exact clones.
- **Type 2:** Syntactically identical fragments with possible changes in identifiers, literals, and types. Sometimes the identifiers are systematically changed as these replicated code fragments are adapted to their new environment. Let us consider the following code fragment:

```
1  public int gcd(int a, int b)
2  {
3      if (b==0)
4      {
5        return a;
6      }
7      return gcd(b, a % b);
8  }
```

Figure 2.1: Original code fragment.

A type 2 clone with changed identifiers would look like this:

```
1  public int getGCD(int x, int y)
2  {
3     if (y==0)
4     {
5        return x;
6     }
7     return getGCD(y,x % y);
8  }
```

Figure 2.2: A type 2 clone with changed identifiers.

- **Type 3:** Source code replications with further modifications; statements have been added, modified or removed. Figure 2.3 shows an example.

Clones that are semantically similar are called type 4 clones:

- **Type 4:** Code fragments that perform the same computation but with some syntactic differences. Figure 2.4 shows a type 4 clone pair which compute Fibonacci numbers.

```
1     for(double d : values) {
2         sum += d;
3     }
4     return sum;
```

```
1    for(double d : values) {
2         if(d > threshold){
3              sum += d;
4         }
5     }
6     return sum;
```

Figure 2.3: A type 3 clone pair.

```
1  int fib(int n)
2  {
3     if (n <= 2) return 1
4     else return fib(n-1) +
5        fib(n-2);
6  }
```

```
1  int fib(int n)
2  {
3     int f[n+1];
4     f[1] = f[2] = 1;
5     for (int i = 3; i <= n; i++)
6         f[i] = f[i-1] + f[i-2];
7     return f[n];
8  }
```

Figure 2.4: Type 4 clones.

This type classification not only ranks clones in increasing levels of differences, but it also offers an indication of how difficult it would be to detect these types of clones, with type 1 being relatively easy to find and type 4 the hardest.

Using the above definitions, we define a clone pair as a triple (CFA, CFB, type), where CFA and CFB are similar code fragments and type is one of the four different types of clones. Another relation between code fragments is called a clone group, which is a collection of more than two similar code fragments.

In the end, the question of whether a code fragment is a clone or not, is very dependent on the clone detection tool used. This is because of the fact that, clone detectors may detect clones that are actually not source code duplications (false positive) or they may fail to detect a clone completely (false negative).

2.2 Clone Detection

As mentioned in the introduction, avoiding code duplications is in many cases rather difficult to achieve as programmers are forced to copy source code for various reasons (e.g. limitation of the programming language used); in other cases, code cloning might actually be an useful way to develop software. Once a code clone is introduced in the source code, it can be forgotten or overlooked, which means that the clone is effectively lost in the software system.

Clone detection becomes, for this reason, a necessary tool to manage code clones. It can be used to find clones that can be refactored as well as making sure that all code fragments in

a clone group are changed consistently.

There are many different techniques for detecting code clones. These approaches can be classified into the following categories [7]:

- **Text-based:** A program text is divided into a sequence of strings, usually lines. Two code fragments are compared with each other to find sequences of similar strings/-lines. If there is little or no normalization used on the program text, then this technique might have problems detecting clones when there are variations in whitespace or comments in the source code [17].

- **Token-based:** Instead of comparing strings, a program is lexed into a token sequence in this approach. This sequence is then scanned for duplicated token subsequences and any matched subsequences are returned as potential clones. In comparison with text-based clone detection techniques, this method should be less likely to miss clones due to formatting or comments in the source code [26].

- **Abstract syntax tree(AST)-based:** In this technique an abstract syntax tree is generated from the source code. Subtrees in the AST are, then, compared with each other with some tree-matching algorithm and similar subtrees are returned as clones. Since the entire syntactic structure of the source code is captured within the AST, this approach is probably better in finding clones than the techniques describe above. On the other hand, it can be slower than either text- or token-based detection [34, 6].

- **Metric-based:** This approach gathers different metrics for code fragments and compare these metric vectors instead of using source code directly. Example of metrics are the number of function calls, parameters and local variables in a code fragment. If certain values of the metric vectors are similar in pair of code fragment, then they can be considered as clones [33].

- **Program dependency graph(PDG)-based:** A PDG is a representation of the control and data flow dependencies inside a program. Clones are found as isomorphic subgraphs inside the program dependency graph. PDG's also carries the semantic information of the source code and, hence, this approach should be able to handle the types of clones where other techniques would struggle, like reordered statements, deleted or inserted code. This approach, however, does not scale to large code bases [35, 32].

Each of these techniques has their advantages and weakenesses and, therefore, choosing the best clone detection method depends for a large part on the specific task encountered.

In addition to finding code duplications to improve the quality of a software system, there are several other applications for detecting clones:

- **Aspect Mining:** Code implementing a cross-cutting concern is usually replicated over the entire software system (e.g. code that is reponsible for logging) and clone detection tools can be used to track them [10].

- **Plagiarism:** Plagiarized source code are essentially some type of code clone of the original and, therefore, can be detected by code clone detectors.
- **Software evolution:** Clone detectors have been used to study the impact of code clones on software evolution by tracking their dynamic behaviour in multiple versions of a software system [4].

2.3 Clones and Software Evolution

Software development is usually a cycle of bug fixing, changes and improvements. The laws of software evolution that were proposed by Lehman [37] can give some insights on this occurence. After the initial release, a software system needs to be continually changed to meet the demands of the customers and, as a result, these adaptions will increase the complexity of the system. Studies have shown that software maintenance accounts for a significant amount of the total cost of a software system [23]. Several factors can influence the maintenance cost and code clones are one of them. As mentioned before, the following problems are associated with code cloning:

- Code cloning unnecessarily decreases code readability.
- Bugs can potentially be copied and pasted all over the software system.
- If change is not consistently propagated to all clones it will lead to defects.

This means that developers have to spend significant time tracking and inspecting code clones. For example, updating a clone would require a programmer to find all related clones and modify each of them consistently. Thus code cloning can lead to additional code reviews and inspection, which increases the workload of software developers and, as a result, they can have a negative influence on software maintainability. Several studies has investigated code clones with regard to their impact on software evolution.

Geiger et al. [20] have studied the relationship between code clones and change couplings. While they found some evidence of such relation, they could not statistically verify it. Kim et al. [31] investigated the evolution of clone classes (clone genealogy) by building a tool that automatically extracts the history of code clones from a software repository. They have found clone evolution patterns, which can be used to study how a clone class is affected by maintenance activities. They have proposed the following six clone evolution patterns:

- **Add:** A code fragment of a clone class that was not present in version i, has appeared in version i + 1.
- **Subtract:** A clone belonging to a given class that was present in version i, has disappeared in version + 1.
- **Same:** All fragments of a clone group in version i was not changed version i + 1.

- **Shift:** At least one code fragment in a new clone group overlaps with another code fragment in another clone class.
- **Consistent Change:** All code fragments have consistently changed from version i to version i + 1 and they have also changed their clone class accordingly.
- **Inconsistent Change:** At least one code fragment underwent a change differently from others from version i to version i + 1.

Kim concluded that, in most cases it is not practical, as a clone management policy, to aggressively refactor code clones as many of these source code duplications appear for a relative short time in the software system. The same problem also seems to apply to long-living clones, which might be desirable to refactor, but are hard to remove them due to programming limitations or other issues [31]. Aversano et al. [2] also studied clone patterns and they have found that majority of clone classes are always changed consistently. The results also show that developers tend to delay propagation of maintenance to clones if the updates are not intended to fix problems. Aversano expanded on the patterns proposed by Kim by specializing the Inconsistent Change pattern in two subcategories:

- **Independent Evolution:** Two or more clones in a group are evolved independently from version i to version i + 1. This means that they are changed inconsistently and in subsequent versions they are developed differently.
- **Late Propagation:** A change is propagated consistently across clones. A clone might be changed in version i, while others will be changed in version i + k.

Figure 6 shows the different clone patterns in a Venn diagram.

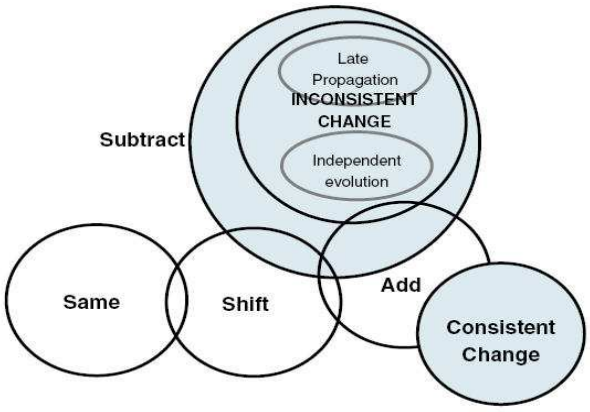

Figure 2.5: The clone evolution patterns proposed by Kim and Aversano [2].

Thummalapenta et al. [46] did a similar study and they have used an automatic approach to classify clone evolution patterns in four software systems. They have also reached similar

conclusions as Aversano as they have concluded that clones are propagated immediately if needed. They have discovered that around 70% of the clone classes are either changed consistently or underwent an independent evolution. A small percentage of cases, around 16%, was detected as late propagations. While they did not find an immediate cause-effect of late propagations and software defects, they have noted that "*it is possible to say that, when such a clone evolution pattern occurs, the code is more bug-prone than in other cases and thus worthy of more attention*". A further refinement of the *Late Propagation* pattern was also discovered by Thummalapenta and this pattern is called the *Delayed Propagation* pattern:

> "*A particular case of Late Propagation where changes between two clone fragments, although not propagated within the same snapshot, are propagated within 24 hours"*

This pattern occurs, when developers cannot commit all the files within the snapshot, but they are able to do it within the same day [46].
Another interesting study was conducted by Bakota et al. [4]. They try to find clones that have a negative impact on software quality which they classify into "smelly" clone classes. Their *Migrating clone instance* pattern is very similar to the *Late propagation* pattern of Aversano. Bakota studied twelve versions of *Mozilla firefox* in which they found eleven migration smells out of a total of sixty clone smells detected. Finally, Krinke [36] came to a different conclusion regarding late propagations than the studies described above. He carried out a study on five software systems to find consistent and inconsistent clones and one of his results was that late propagations are a very rare occurence.

2.4 Software Repository Mining

Software repository mining (MSR) is an essential ingredient in the study of software evolution [21]. Software repositories contains a large amount of information about the history of software projects. By employing data mining techniques, it is possible to find useful information about a software evolutionary characteristic. Software repositories are usually mined to answer two classes of questions [25]:

- Causality relationship between two or more events. For example, if A occurs then how many times X do B and C happen.
- Finding out the metrics of a certain occurence, like determining how many functions and which function are called in a system [25].

Examples of sofware repositories are version control systems (VCS), bug repositories and deployement logs. A VCS tracks all changes to the source code together with metadata like commit comments, commit timestamp or the name of the developer responsible for the change. Mining a version control system can be used to support software development in various manners. For example, Gall et al. [19] developed a technique which uncovers hidden dependencies or logical couplings in the source code which can be used to identify modules that should be modified. Chen et al. [13] presented a tool called CVSSearch that

describes what the responsibility of a particular code fragment is by mining CVS comments. Malik and Hassan [41] examined the change propagation process in software development and proposed meta-heuristics to predict change propagations in a software system.

The focus of this research is primeraly on version control systems, which are used to track code clones in different versions of a software repository. There are several methods for tracking code clones. The first method is to use a clone detector on all versions of a software system and then use a matching algorithm to track clones in multiple versions of a software repository. This approach gives a (nearly) complete view of clone evolution, but suffers from several problems. First the matching algorithm is quite complex as it has to deal with extensive source code modifications in later versions and as a result it might fail to track some clones. Another problem is that this approach requires a lot of computational resources [31, 4].
Another approach for tracking code clones is to extract clones from a reference release and map clone changes using change and log information obtained from a software repository. This method is obviously faster and easier to to implement than the above method, but it comes at cost of not being able to track newer clones in subsequent versions [2]. Duala-Ekoko and Robillard [16] presented a clone tracking tool called *CloneTracker*, which can be seen as variant of the above method. *CloneTracker* is an eclipse plugin aimed to help software developers to track code clones in multiple versions of their software system. They use a *Clone Region Descriptor*, which identifies clone regions at the granularity of code blocks based on file name, class name, method name and block descriptors with some optional parameters. Since their tracking method is based on a (java) block, it has problems tracking clones if clone regions are not aligned to certain types of code blocks. *CloneTracker* is very accurate in tracking clones as only 139 clone regions out of 3275 were lost (around 4%) in a study of five software systems.

Chapter 3

Detecting Late Propagations

This chapter discusses our approach in detecting late propagations. To find this clone evolution pattern it is necessary to track changes made to code clones in a software system and to distinguish it from other patterns. A tool is presented that is be able to mine the change history of a VCS and combine this information with a clone detection tool output to discover late propagations. Furthermore, the components of our tool together with design decisions and challenges are also described.

3.1 Tracking Code Clone Evolution

Several approaches to track the evolution of code clones have been proposed in literature. For instance, Kim et al. [31, 4, 39] map clone fragments between version i and version i + 1 by using a location overlapping function in combination with unix diff, which is a file comparison tool that shows the changes between two versions of the same file. To determine how much the source code of a code clone has changed, a textual similarity function is defined that measures the text similarities between a given version and a previous version of a software system. The approach proposed by Kim, however, is not capable of finding late propagations as it does not distinguish late propagations from the independent evolution pattern as it only traces inconsistent changes. It also does not account for reappearing clones that might be a member of a certain clonegroup several revisions ago, which is essential for finding late propagations.

In order to find late propagations, several issues have to be considered [4, 36, 39, 22]:

- If a clone is modified in a clonegroup consisting of two code duplications, then that group will disappear. When finally the change has reached the second clone, a new clonegroup will appear. Our tool therefore has to compare new clone pairs with older or deleted clonegroups.
- Two consistent clones may evolve independently for a given number of revisions before they become consistent again. This means that we also have to track the independent evolution pattern to look for such an occurence.

- For clonegroups larger than 2 clones, a (partial) clone migration to another clonegroup should be observed. If a new clone is added to a clonegroup, the tool has to check if the existing code duplications in the group were related to the new clone in any previous revision.
- The granularity used to analyze the history of the software system is also important. Choosing a long interval means that a late propagation would probably be observed as a consistent change. Therefore we choose to analyze every revision in our selected projects.

3.2 Toolchain Structure

This section describes the different components of the late propagation finder tool, which is depicted in figure 3.1. The tool is written in Java and operates only on SVN repositories. This is not a big restriction as the cvs2svn[1] tool can migrate CVS repositories to Subversion. It can only analyze Java software systems as the modifications module needs a Java parser to work and the reason for using a Java-based approach will be explained in detail in the following subsections.
As stated before in chapter 2, clones can be traced by considering every relevant revision of a VCS or by analyzing clones from a single snapshot. We have chosen the former option as it allow us to find more late propagations in a sofware system than the second method, since it does not consider newly added clones in subsequent snapshots. Our method of tracking code clones is somewhat similar to Kim's approach [31], although we are not interested in constructing a clone genealogy of a software system. This means that we will only trace late propagations and filter out other irrelevant clone evolution patterns. Our approach consists of the following steps. Firstly, during initialization phase we extract the first revision or snapshot from a SVN repository and use a clone detection tool to find clones in the downloaded codebase and our tool will build clonegroups, which are saved into a xml database. Then for each subsequent snapshot:

- Determine if any changes are made to the code base, if so, download the sources of that revision.
- Extract all change sets of that snapshot.
- Analyze the change sets to see if there are any modifications made to clones in the xml database.
- Update the changes made to the code duplications into the database.
- Find out if any late propagations have occured by analyzing the changed clones.
- Run the clone detector to detect new clones and update the database accordingly.

The following subsections will explain some of the above steps in detail.

[1] http://cvs2svn.tigris.org/

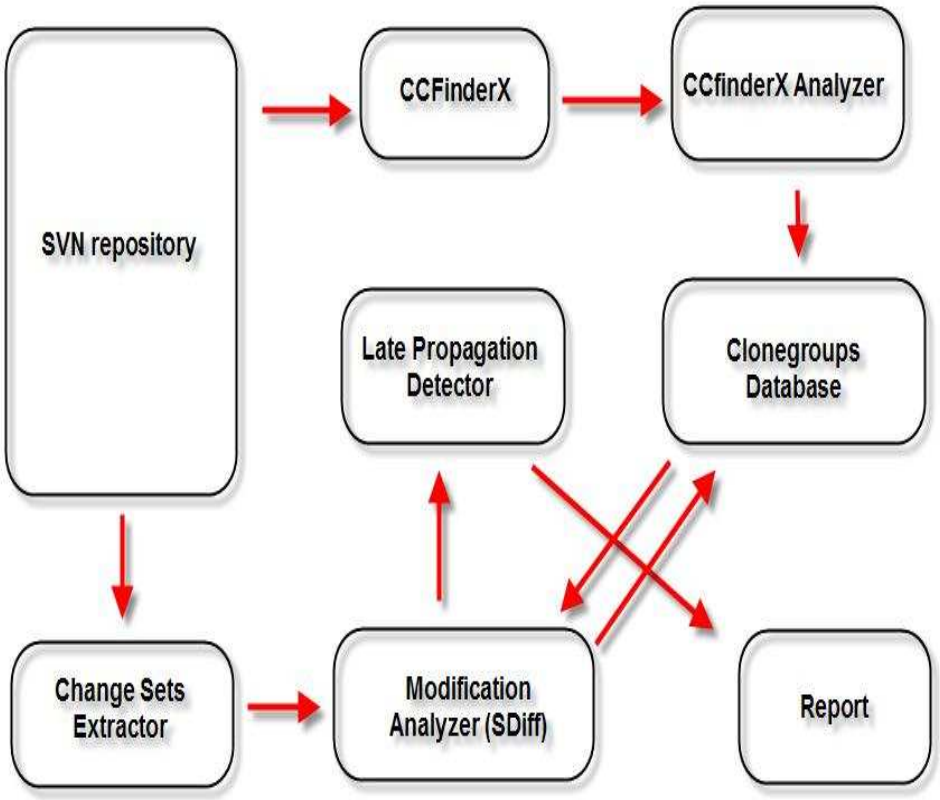

Figure 3.1: Overview of our late propagation detector

3.2.1 Initialization phase

In the first step, the first snapshot is downloaded from a Subversion repository and analyzed by a clone detection tool. Based on a previous literature study on clone detectors and also on existing comparison studies [7, 11], we have selected the CCFinderX [26] tool as the clone detection tool used in this paper. CCFinderX is an open source clone detection tool that can identify clones in Java, C, C++ and many other programming languages. The clone detection technique employed by this tool involves transforming source code into tokens and using a token-by-token comparison algorithm to find clones. The tool has been configured to find clones of at least 30 tokens, which is a reasonable minimum clone size to limit false positives according to other studies [31, 7, 39]. As mentioned in chapter 2, a clone has a minimum size of 6 source code lines and this means that any clone smaller than that limit

will be filtered out.
CCFinderX is used to find clones in a given revision of a software system and every clone detected by this tool is reported as a clone pair. The CCFinderX Analyzer module reads the entire output of CCFinderX to construct clonegroups by matching similar clone pairs with each other (e.g. clone pairs A-B and B-C are converted to a clonegroup containing A-B-C). For each clone found by the clone detector, our tool records the containing method, statement order and the code fragment. A Java abstract syntax tree parser is used to analyze each source file to extract the location of each clone found by CCFinderX. Finally, the clonegroups are saved in a xml database, which is later used for analysis.

3.2.2 Extraction and Analysis of Change Sets

This section describes how change sets are obtained from a VCS and then analyzed to determine if any modifications are made to the code clones in the database. Our tool uses the SVNKit library[2] to extract change sets from a Subversion repository. First the log entries are obtained from the VCS, which the Change Sets Extractor module will analyze. Each log entry contains information about a commit like the author name, revision number, timestamp, log message and also the added, deleted and modified files. The extractor module filters out any irrelevant change sets (e.g modification to property files) and passes the log entries it has fetched to the Modification Analyzer module. This modification module is responsible for determining if any changes are made to the code clones that are stored in the database.

There are three types of modifications that can occur in a clone fragment, which can make it inconsistent with other code duplications in a clone family:

- A statement is added to a clone fragment, that increases the size of the code duplication.
- A section of the clone is deleted or the entire clone is deleted. In the former case the size of the clone has decreased and in the latter, it means that the code duplication was present in the previous revision but has disappeared in the current one.
- One or more statements in the clone fragment have been modified and therefore the clone size remains the same.

Additionally, two types of changes can happen outside the clone, which does not affect its clone relationship:

- A code fragment has been added above the clone and as a result the clone has moved downwards.
- One or more statements above the clone have been removed and this has as an effect that the clone has moved upwards.

[2] http://svnkit.com/

In order to determine what kind of changes have occured in a method, it is necessary to compare the source file with the previous revision. Existing studies [36, 31, 2] have used the unix-diff tool to find out which part of a source file has changed. Unix-Diff or Diff is the standard tool for discovering the differences between two versions of a file, but it does have several limitations [45, 12].
Firstly, Diff has problems in distinguishing between code modifications and code additions/deletions. It records a line change as a deletion of the old line and an addition of a new one. It also can not detect method renamings/movements as shown in the figure below.

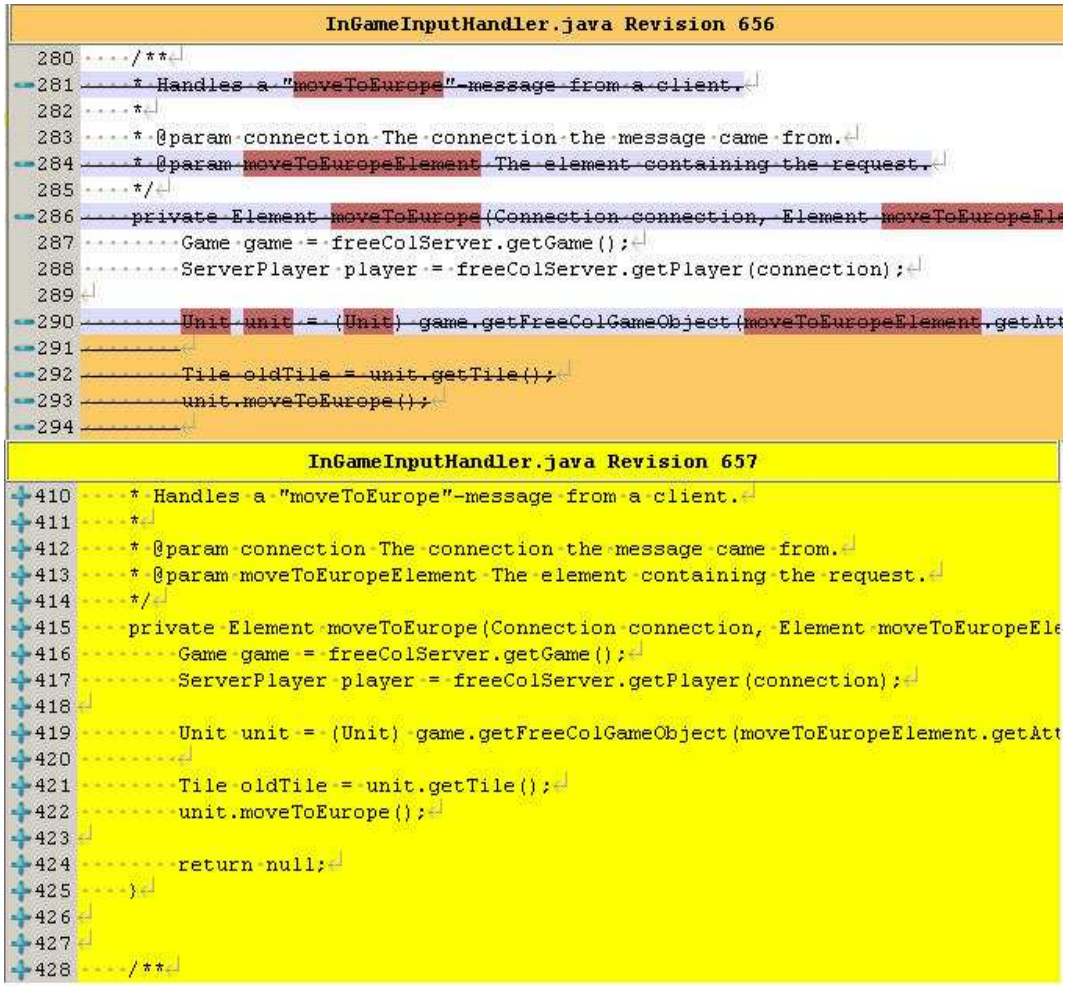

Figure 3.2: Due to new methods added above the function *moveToEurope*, Diff has wrongly detected a removal of the old method and addition of a new one. The only changes made to this method was the deletion of some statements.

While there is a degree of subjectiveness in determining whether a program text has been replaced or changed, recently there are several approaches proposed that can identify

edited lines with reasonable accuracy [12, 45, 43]. We have selected the Statement Diff or SDiff tool [45] to track the changes made to code clones in the database. The reason for choosing SDiff is the fact that the other options are either closed source [43] or are difficult to use in our toolchain as they are implemented in another programming language [12].
SDiff is an open source tool implemented in Java that uses a hybrid technique as it combines line-based (determining the edit distance between two lines) and structural (using the structure of the source code) approaches in tracking software artifacts. It uses the abstract syntax tree of a source file to break up the code into class declarations, import statements, field declarations and methods. Unlike Diff, which works on lines of code, it operates on Java statements and tries to match statements between two versions of a file. SDiff has the following advantages [45]:

- SDiff ignores whitespace, comments and brackets.
- Statements that are broken across multiple lines are also handled.
- It can also find changes by characters or by token.
- Movement or renaming of methods is also detected.

Since SDiff only reports changes per statement, we have to modify the source code of SDiff to make it suitable for our toolchain. Firstly, the modified SDiff reports changes per method instead of statement, so it can report changes inside a method as well as method addition, removal and renamings. The next figure shows the output of the modified SDiff for the *moveToEurope* function depicted in figure 3.2.

```
Code deleted in method private Element moveToEurope ( Connection connection , El
295,295d
< Iterator enemyPlayerIterator = game . getPlayerIterator ( ) ;
296,296d
< while ( enemyPlayerIterator . hasNext ( ) )
297,297d
< ServerPlayer enemyPlayer = ( ServerPlayer ) enemyPlayerIterator . next ( ) ;
299,299d
< if ( player . equals ( enemyPlayer ) )
300,300d
< continue ;
```

Figure 3.3: Modified SDiff only shows some statement removals and does not report the deletion of *moveToEurope*.

Secondly, if SDiff is not certain that a statement has changed it outputs *POSSIBLE* or *INVALID* leaving the user to decide whether a modification really has occured and since we need an automatic change analyzer, we have modified SDiff to regard these kind of changes as a removal of the old statement and addition of a new one.
Finally, we have made several minor changes to SDiff like how it handles inner classes and

outputs. Choosing SDiff, however, means that our tool can only analyze Java source files as it depends on a Java parser. This is not a big disadvantage as we have traded language independence for better accuracy in detecting changes made to code clones. Secondly, it is possible to use parser generators like JavaCC[3] to generate parsers for other programming languages. Finally, Java is often used in open source software development and therefore there are many software systems available to analyze. Any changes made to the clone fragments are passed to the Late Propagation Detector module and finally the clones in the database are updated as well.

3.2.3 Detecting Late Propagations

The Late Propagation Detector module is responsible for finding LPs by analyzing the change information from SDiff. If a clone fragment in a clone family has been changed to such an extent that its size is different (due to statement additions or deletions) from other code duplications in the same group. It is clear that a inconsistent change has happened. If, however, the size of the modified clone fragment is the same with some related code duplications, it is necessary to compare them to decide whether a consistent or inconsistent change has occured.
The Late Propagation Detector module uses the *Normalized Levenshtein edit distance*(NLD) [38] as a metric to measure the similarity between two clone fragments.
The Levenshtein distance(LD) between two strings *s1* and *s2* is defined as the minimum number of insertions, deletions and substitutions required to transform *s1* into *s2*. For example, for *s1* = 'qwerty' and *s2* = 'azerty', the edit distance between both strings is 2 and the higher the LD, the more different both strings are.
In order to conduct comparisons, the *Normalized Levenshtein edit distance* is used, which is ranged in the interval [0, 1], where 1 means that the strings match and 0 indicates that the strings are strictly different. The NLD for two non-empty string *s1* and *s2* is defined as [12, 46]:

$$NLD(s1,s2) = 1 - \frac{LD(s1,s2)}{max(s1,s2)}$$

where LD(s1, s2) is the *Levensthein Distance* and *max*(s1, s2) is the length of the longest string. We have used the threshold values of a previous study [46] to determine if a clone pair is consistent or inconsistent. A pair of clones is inconsistent if the NLD is smaller or equal to 0.87 and consistent if the NLD is greater or equal to 0.92 and between those two values the clone evolution is classified as unknown. While the inclusion of an unknown evolution pattern might be strange, Thummalapenta et al. [46] have shown through empirical analysis that these values lower the amount of false positives detected.
By using the NLD we can determine if the program texts of two clones are similar to each other and discover the clone evolution pattern. A consistent change is found if all clones in a clone family have received similar updates and this is determined by computing the NLD between each clone fragment. Similarly, an inconsistent change is detected if one of the

[3] https://javacc.dev.java.net/

clone fragments has been modified differently from other clone group members, which is also detected by the NLD.
A late propagation happens, as indicated earlier, when a clone in a clone group is changed inconsistently from the rest of the clone family. After a number of revisions the delayed update is propagated to the other related clones and as a result these code duplications are consistent again. This realignment of clones is detected in the same way as consistent changes are found. Finally, the detector generates a report if a late propagation is found.

3.2.4 Finding New Clones

CCfinderX is also used to detect clones that appear after the initial snapshot. Instead of processing the entire output of the clone detector, the CCFinderX Analyzer module uses the log entries to determine which file has actually changed and only process the clones that are in the modified files. The module checks if any new clone is related to any code duplication in existing clonegroups and if that is the case, then the clone is added to the clonegroup and the clone family is updated in the database.

Chapter 4

Expirement Results

This chapter describes the experiments conducted to find late propagations(LP). We have selected four software systems to analyze the evolution of code clones.

4.1 Subject Systems

The subject systems chosen for our case study have to meet several requirements that are based on our tool and for finding late propagations. Firstly, the software systems must be written in Java and accessible from Subversion. The projects should also differ in size and come for different domains to avoid bias. Finally, they must be in a mature development phase to have enough revisions from which we can observe the late propagation pattern.

We have chosen the following four software systems as subjects for our experiments:

- **Subclipse.** Subclipse is an Eclipse Team Provider plug-in providing support for Subversion within the Eclipse IDE. It is the smallest VCS considered for our case study with only 1946 revisions. The analysis of Subclipse starts in June 2003 and a maximum of 223 KLOC of code has been processed by our tool.
- **JEdit.** JEdit is a popular open source programmer's text editor and we have examined 2418 revisions starting in September 2001.
- **FreeCol.** FreeCol is a turn-based strategy game, which can be considered an open source version of the old Colonization game. The project is analyzed starting in April 2005 and a total of 4935 have been extracted. It is the largest project considered for our case study.
- **Seam.** Seam is an application framework for building Web 2.0 applications. While having over 3000 revisions, its code base is the smallest of the four ranging from 2 to 148 KLOC of code.

Table 4.1 provides an overview of the open source software projects selected for our case study. It shows the domain, the total number of snapshots considered in this case

study, the number of authors and the minimum and maximum number of KLOC (thousands lines of code) of each software project.

System	Purpose	Snapshots Analyzed	Authors	KLOC	Start Date
Subclipse	Subversion plug-in	1946	15	42 - 223	Jun, 2003
JEdit	Text editor	2481	20	102 - 335	Sep, 2001
FreeCol	Game	4935	35	40 - 445	Apr, 2004
Seam	Application framework	3005	29	2 - 148	Aug, 2005

Table 4.1: Overview of selected open source projects.

4.2 Systems Analysis

This section displays the results of our clone tracking tool. For each project, we try to find the cause and effect of late propagations, where we also make the distinction between short term (described in chapter 2 as delayed propagations) (SLP) and long term LPs (LLP). Seam is, however, not analyzed as we have found no LPs in that project. The following questions are discussed:

- **Is there a relation between the propagation interval and the clone radius/package distance?.**
  Clone radius is defined by Ueda et al. [47] as: *'For a given clone class C, let F be a set of files which include each code fragment of C. Define RAD(C) as the maximum length of path from each file F to the lowest common ancestor directory of all files in F'.*
  Since we are dealing with software systems written in Java, this means that we are actually looking at the package distance between clones. A high package distance implies that the code duplications are scattered over packages, which has as a result, that related clones are more difficult to find and update. A clone pair in the same source file has a distance of zero and if the pair of clones only share the same package, the distance becomes one. If they are in distinct packages, the distance is defined as by Ueda et al. and can range from two to the maximum package distance.

- **Is there a connection between commits and the appearances of LPs?.**
  Ideally, each commit should be self-contained[1] [42], which means that related source artifacts should be changed together. If this is not the case, then it will result in inconsistent changes. If a commit contains a large number of source code modifications, then a software developer might forget to change clones consistently and this can potentially lead to late propagations. We consider a commit to be busy if 6 changes [1] are made in the revision or around the revision in a timespan of 30 minutes [9].

[1]The KDE project prescribes the following in its SVN commit policy: 'Please commit all related changes in multiple files [. . .] in the same commit' and 'Every bug-fix, feature, refactoring or reformatting should go into an own commit'; see http://techbase.kde.org/Policies/ SVN_Commit_Policy#Commit_complete_changesets

The change history view, which is a sofware visualization technique by Zaidman et al. [48], is used to observe the commit data from a VCS. We had to make minor modifications to the change history view as its intended use is to visualize the co-evolution of production code and unit tests. The visualization of the test files is removed in favor of displaying late propagations.

- **What are the effects of LPs?.** Late propagations have as a consequence, that a bug fix is not properly propagated to all clones or that a few clones might miss an update, which results in unintended behaviour in these code duplications. In order to detect these occurences, we look at the commit comments in a VCS, where keywords like 'fix' or 'issue' indicate that a software defect was fixed. We also look into other activities that may cause LPs by analyzing the source code.

4.2.1 Subclipse

```
119 if (!SVNWorkspaceRoot.getSVNFolderFor(destination.getParent())
120                         .isManaged()) {
121     SVNTeamProvider provider = (SVNTeamProvider) RepositoryProvider
122                 .getProvider(destination.getProject());
123     if (provider == null) //target is not SVN project
124         throw new SVNException(Policy.bind("SVNMoveHook.moveFileException"));
125     provider.add(new IResource[] { destination.getParent() },
126                 IResource.DEPTH_ZERO, new NullProgressMonitor());
127     ISVNLocalResource parent = SVNWorkspaceRoot.getSVNResourceFor(destination.
        getParent());
128     if (parent != null) parent.refreshStatus();
129 }
```

```
194 if (!SVNWorkspaceRoot.getSVNFolderFor(destination.getParent())
195                         .isManaged()) {
196     SVNTeamProvider provider = (SVNTeamProvider) RepositoryProvider
197                 .getProvider(destination.getProject());
198     if (provider == null) {
199         throw new SVNException(Policy.bind("SVNMoveHook.moveFolderException"));
200     }
201     provider.add(new IResource[] { destination.getParent() },
202                 IResource.DEPTH_ZERO, new NullProgressMonitor());
203 }
```

Figure 4.1: An example of a late propagation in Subclipse: A code fragment (in blue) added to one clone instance, the other clone was updated after two months (SVNMoveDeleteHook.java).

Our tool has discovered 7 late propagations in the Subclipse project, where only one can be classified as the long term type and the remaining LPs have a propagation time of shorter than a day. Figure 4.1 shows an example of a late propagation in Subclipse. An

update was introduced in revision 2352 to a clone fragment in the method *moveFile* in the class *SVNMoveDeleteHook.java*, but this change was not propagated to a related clone in the method *moveFolder* (in the same class) until revision 2606 (about 2 months). In the commit log of revision 2606 the developer states: '*Fix a refactoring problem where it can error out because it does not know that a folder has already been added to Subversion.*', which means that this particular case of LP has resulted into bug. Another software defect was found that was due to a delayed bugfix, which we also confirmed through the commit log, between two dialogs in Subclipse. The remaining LPs caused minor inconsistencies to the UI subsystem as a result of delayed updates to the layout.

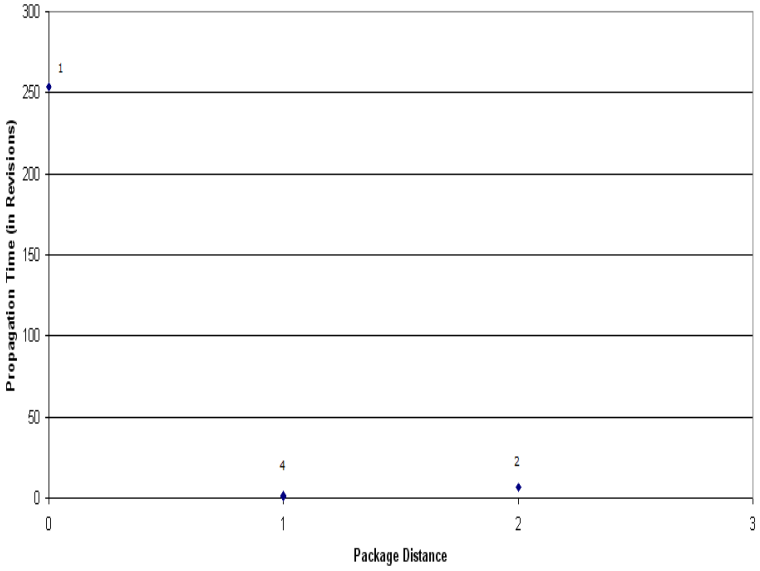

Figure 4.2: Propagation Time and Package Distance

Figure 4.2 shows the propagation time and package distance for all 7 late propagations (the numbers above the dots display the LPs which have the same package distance and propagation time). It does not seem to show any relation between both metrics, the longest propagation time was between a clone pair in the same source file and they belong to methods that were in close proximity of each other (about 3 lines!). The other LPs have a much shorter propagation time, but larger package distances.
There is a group of 4 LPs in the same clone group in Subclipse and figure 4.3 depicts their package distances (two LPs have the same package distance and are shown as 1 dot). It shows that 3 LPs in the same package are updated in 1 revision, the remaining one, how-

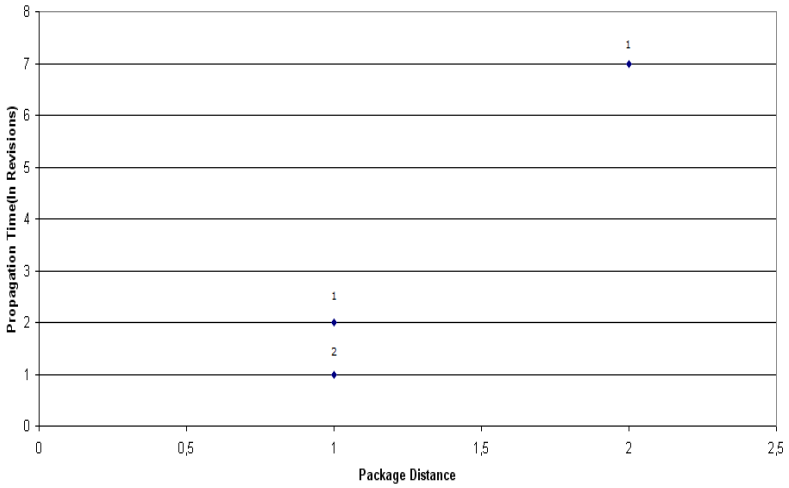

Figure 4.3: Propagation Time and Package Distance between a clone group of Late Propagations.

ever, resides in another package and is updated with a difference of 7 revisions. This might be an indication that the developer forgot about this clone instance. While this is hardly a conclusive evidence of a connection between propagation time and package distance, it is an interesting observation.

Next, the change history view of Subclipse is depicted in figure 4.4. The green triangle in the view reveals the file from which a late propagation originates and the revision in which it appears and since this visualization only shows changes at file level, only 6 are displayed as one of the source files contains 2 late propagations. In the figure it seems that 71% of the LPs (all short term) appeared around busy revisions. For the group of 4 related LPs, however, we can see a lot of commit activities around those revisions. If we combine this with our observation of the package distance, as described earlier, both factors might have contributed to the delayed propagation of 7 revisions.

4.2.2 FreeCol

A total of 18 late propagations have been detected by our tool in the FreeCol project, where ten LPs are of the long term category and 8 delayed propagations. Figure 4.7 shows a harmless example of a late propagation in the class *InGameInputHandler.java*, where a line of

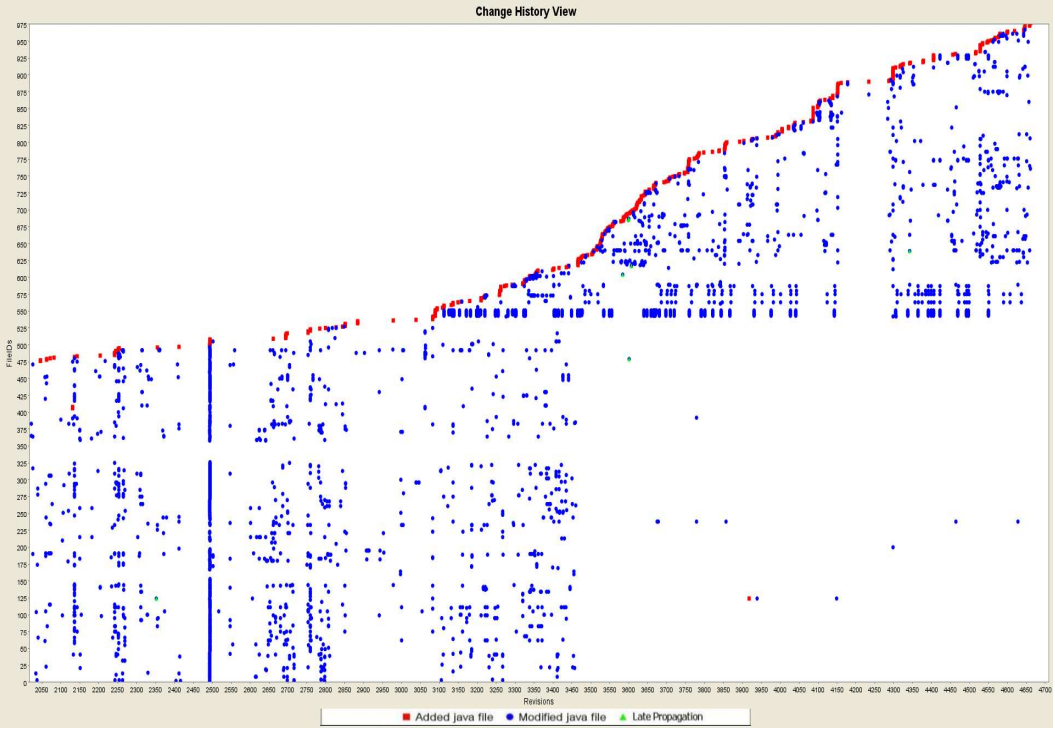

Figure 4.4: Subclipse ChangeHistory View.

dead code has been removed from the method *armedUnitDemandTribute* in revision 1859 and in revision 1864 the method *attack* (in the same class) received an identical modification. While there are more LPs found than in Subclipe, we have not found any strong evidence of late propagation related bugs. Most LPs involves improvements which did not cause any problems as they were due to activities such as code beautification, dead code removal and movement of common code to a method. For example in revision 6640 in the class *BuyPropositionMessage.java*, some of the logic has been moved to the class *InGameController.java* and after 12 hours a similar change was introduced to a duplicate in the class *BuyMessage.java*. We did, however, see some delayed updates between revisions 4900 and 5200, which relate to layout modifications of several UI components and thus may lead to minor inconsistencies.

In the figure 4.6 we see a dot plot of the propagation time vs. the package distance. It is apparent that most LPs happened between clones with a package distance of 1 and they also have the longest propagation time. This means for FreeCol that the package distance

```
1763 if (unit.getOwner() != player) {
1764     throw new IllegalStateException("Not your unit!");
1765 }
1766 Tile oldTile = unit.getTile();
1767 Tile newTile = game.getMap().getNeighbourOrNull(direction, unit.getTile());
1768 if (newTile == null) {
1769     throw new IllegalArgumentException("Could not find tile in direction " +
          direction + " from unit with ID "
1770                     + attackElement.getAttribute("unit"));
1771 }
```

Figure 4.5: An example of a LP in FreeCol: removal of dead code *(InGameInputHandler.java)*.

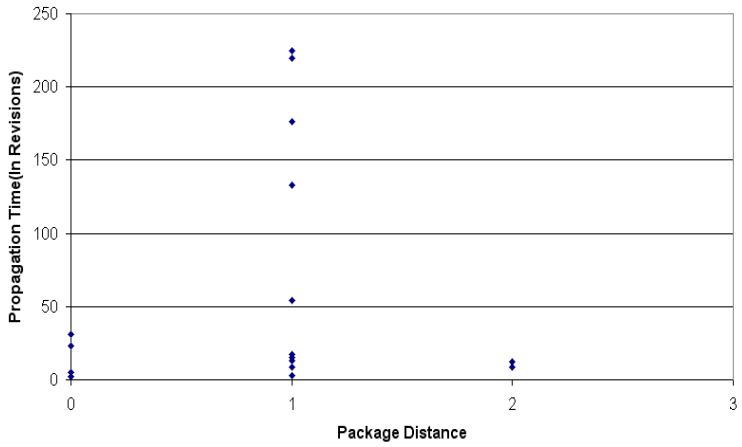

Figure 4.6: Propagation Time and Package Distance.

probably has no influence on the propagation time. The figure does show that some of the refactoring activities concerning LPs have a very long propagation time, which might be an indication of forgetfullness on the part of the developers.

The change view history of FreeCol is depicted in figure 4.7. It shows that there is a busy commit activity around 8 LPs in FreeCol, which amounts to around 50% of all LP.

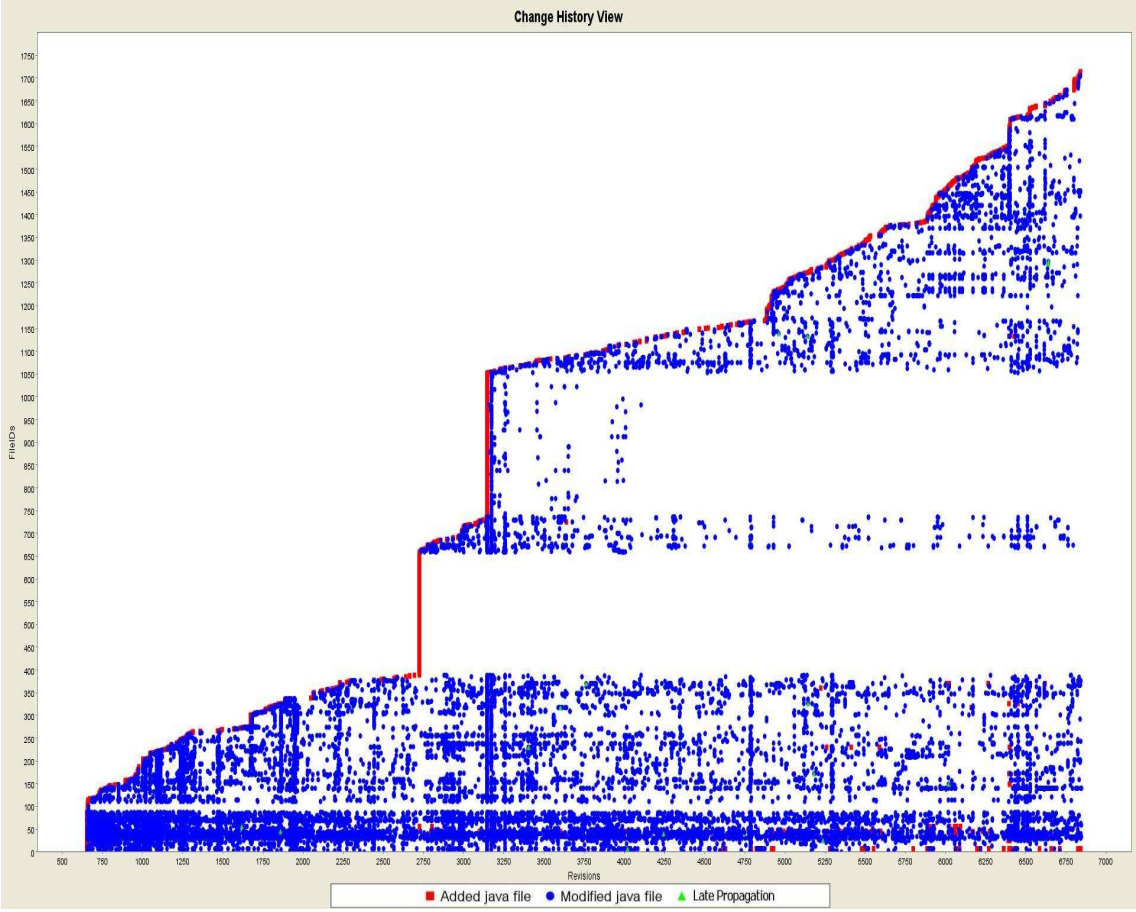

Figure 4.7: FreeCol ChangeHistory View.

This means that for FreeCol, there is no clear indication for a link between commits and LPs. It is, however, interesting to note that 6 short term LPs are associated with active commit actions.

4.2.3 JEdit

JEdit has the largest amount of LPs of all 4 projects with 35 instances of this clone evolution pattern. However, JEdit also has a remarkable number of unparsable revisions, for example between revisions 3907 and 5020 there are more than 30 revisions with the comment such as 'didn't compile' in the log. These revisions can not be processed by our tool and this means that some LPs will not be detected. However, we have found more than enough LPs to conduct an analysis and our tool has detected 30 long term LPs and 5 delayed propogations.

```
3671 public void goToNextFold(boolean select)
3672 {
3673         int line = caretLine;
3674
3675         while(!buffer.isFoldStart(line))
3676         {
3677             if(line == 0)
3678             {
3679                 getToolkit().beep();
3680                 return;
3681             }
3682             else
3683                 line--;
3684         }
```

```
3577 public void goToPrevFold(boolean select)
3578 {
3579         int line = caretLine;
3580
3581         while(!buffer.isFoldStart(line))
3582         {
3583             if(line == 0)
3584             {
3585                 getToolkit().beep();
3586                 return;
3587             }
3588             else
3589                 line--;
3590         }
```

Figure 4.8: An example of a late propagation in JEdit: A code fragment (in red) deleted in one clone instance, the other clone was updated after 79 revisions (JEditTextArea.java).

Figure 4.8 shows an example of a long term LP. In revision 3939 the beep function call was removed in the *goToNextFold* method due to a bugfix, but the developer somehow forgot to modify an exact duplicate in the method *goToPrevFold* in the same class. This instance of a late propagation was fixed after 79 revisions (over a month) and it is interesting to note that both functions are in the same class and they are seperated by 4 lines. The commit log of revision 3939 states 'Various JDK 1.4 bug fixes' and since it was the only

modification made to the class, it seems very likely that this is a bug. In similar fashion, we have found 4 other bugs by inspecting the commit logs. For 3 LPs we could not confirm if they resulted in software defects as their commit logs do not cleary link a bugfix with those late propagations, but due to the nature of the modication we suspect that they have caused a bug. In total we have found that around 8 LPs (5 confirmed and 3 suspected) have resulted in bugs and 27 are due refactoring activities like deletion of useless code, code movements or code restyling. However, some LPs involved with refactoring seem to have a very long propagation time, which is shown in the next figure.

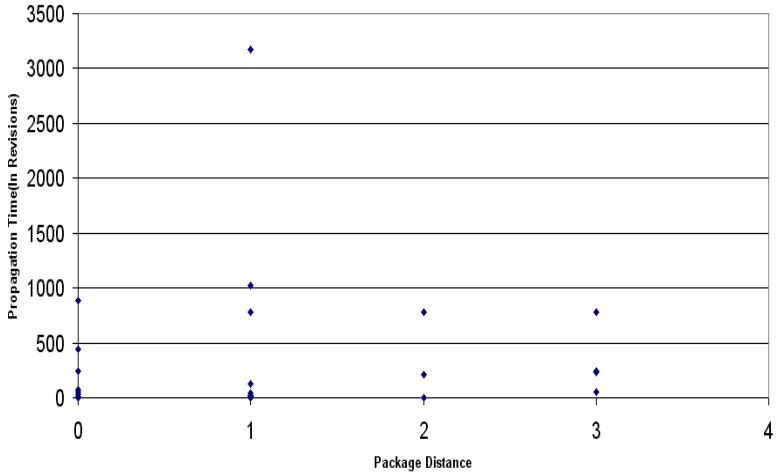

Figure 4.9: Propagation Time (in Revisions) and Package Distance.

In figure 4.9, we can see a refactoring LP with propagation time of 3000 revisions. In the class *BufferInsertedRequest*, code to close an inputStream has been replaced by a method in the *IOUtilities* class in revision 5486 and after more than 6 months in revision 8660 the class *PluginJar* received a similar modification to its clone instance. The figure seems to show that there is no link between propogation time and package distance as most LPs have a distance of just 1.

The change history view of JEdit is shown in figure 4.10. It is interesting to see a dense population of blue dots (changed Java files) between revisions 3600 and 5500 and the bulk of the late propagations (26) detected by our tool comes from that interval. For 24 late propagations, there seems to be a lot of commit activity in and around the revisions from

which they have appeared. This amounts to 68% of the LPs found in JEdit and this number is not unsimilar to FreeCol. The number of short term LPs , however, is different from FreeCol as only 2 out of 5 LPs come from busy revisions.

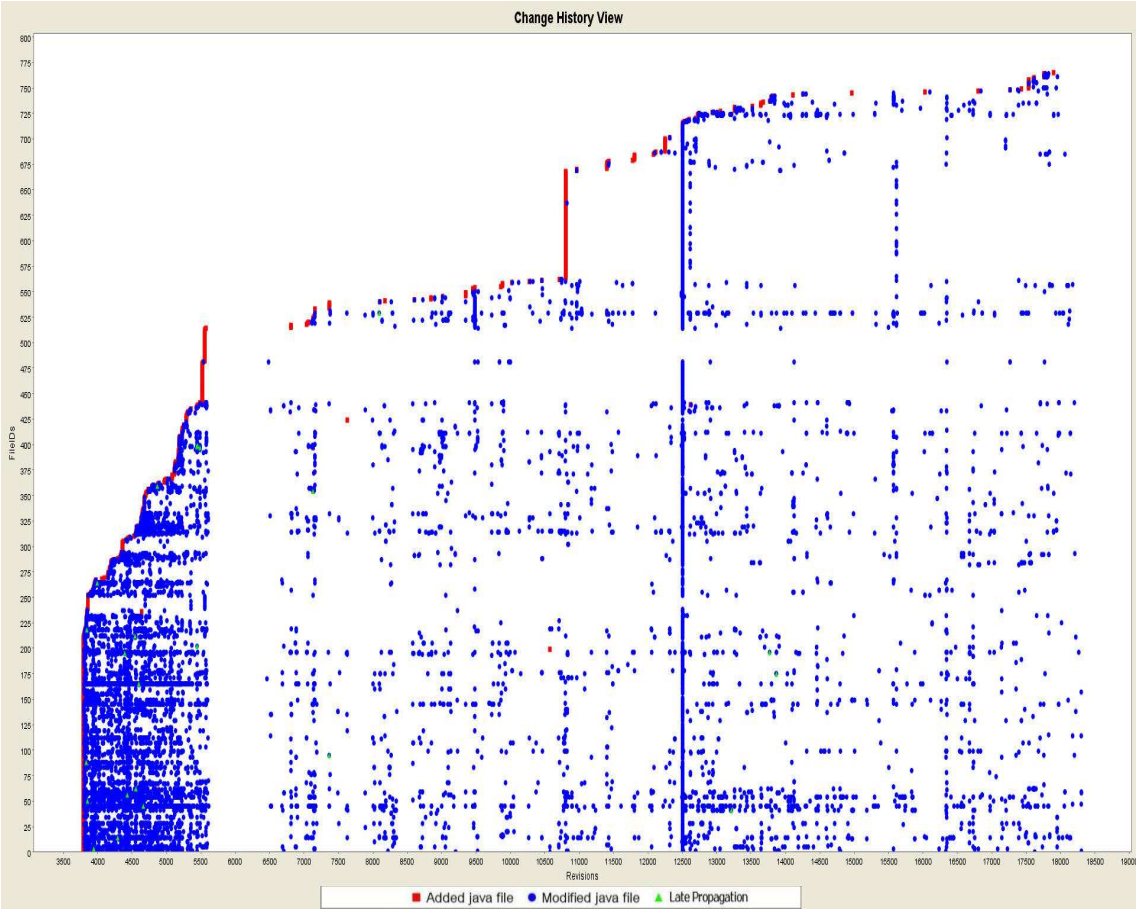

Figure 4.10: JEdit ChangeHistory View.

4.3 Discussion

Finally the results of each software system are summarized in the next table. The last column 'Busy LPs' shows the percentages of LPs where we have observed busy commit activities in connection.

Project	Late propagations	LLP	SLP	Bugs	Bugs(%)	Busy LP(%)
Subclipse	7	1	6	2	28%	71.4%
FreeCol	18	10	8	0	0%	50%
Jedit	35	30	5	8	22%	68.5%
Seam	0	0	0	0	0%	0%

Table 4.2: Projects Summary.

As the table shows, we have detected a total 60 LPs of which 41 long term and 19 delayed propagations and they have caused a total of 10 bugs (7 LLP and 3 SLP).

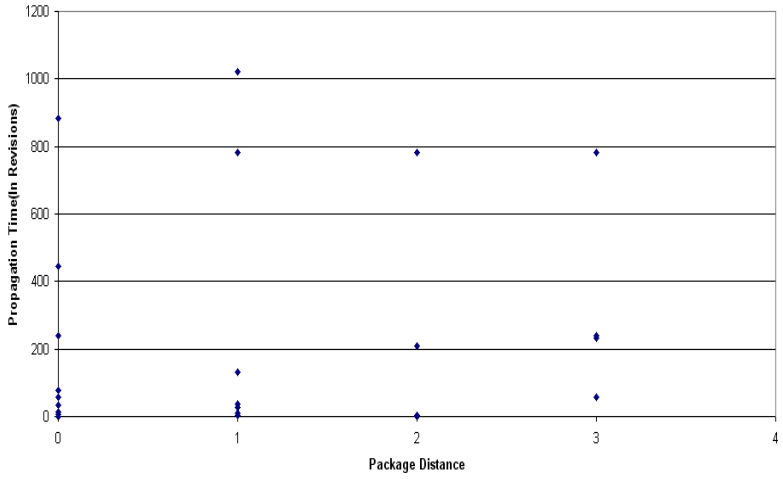

Figure 4.11: Propagation Time (in Revisions) and Package Distance (extreme values removed for better overview).

In figure 4.11 we can see the dotplot of all 4 projects and it is clear that LPs with a package distance of 1 appear more frequently with the longest propagation time in total.

Therefore, it seems that the package distance has no real influence on the propagation time and this is quite unexpected as the package distance is an indication of how widely clones are scattered in the Java packages. It is interesting to observe LPs with long propagation times and a package distance of zero (same source file) and we have seen late propagations and bugs occuring between methods in close proximity of each other. We have also seen refactoring activities negatively affected by LPs as they have caused a long delay of updates to related clones and this seems to point out that even 'harmless' LPs have a bad influence on software maintenance.
Table 4.2 also seems to indicate a link between late propagations and commit activities of developers for two software systems. The percentages of LPs affected by commits ranges from 50% to 71% and if we combine this with our observation of package distance, we can state that clone management tools might be helpful for software developers.

4.4 Threats to Validity

There are some factors that may limit the validity of the work conducted:

- **Clone Detection Technique:** We have used CCFinderX to detect clones in a code base. As stated in chapter 2, it may find false negatives (miss clones) or false positives (detect clones that are not code duplications).
- **Clone Tracking Method:** The SDiff tool is used to determine the changes made to code clones found by CCFinderX. While it is an improvement to standard unix diff, it also has similar limitations as CCFinderX. It is possible that it may find false positives(e.g. false statement changes) and as a result, this can be a threat to our clone tracking approach.
- **Language Support:** The clone tracker tool can only track clones in Java, which limits our findings to this programming language.
- **AST-based Approach:** Our tool uses an AST-based approach, which needs a parsable Java source file. We have encountered revisions where the developers have commited uncompilable code (especially JEdit) that is rejected by the AST parser in the clone tracker tool. As a result, some revisions are ignored and this means that there will be late propagations in the sofware system that can not be detected by our tool.

Chapter 5

Conclusions

5.1 Conclusions

The central research question of this thesis was 'Can we study late propagations using software repository mining?'. To help us answer this question we have formulated five subquestions:

- **SQ1:** *How do we track code clones in a software repository?*
  We have used CCFinderX to find code clones in every revision of a software project and they are tracked with SDiff. SDiff is an improvement to standard unix diff as it operates on Java statements instead source code lines, which is very useful as we have only analyzed sofware systems written in Java.

- **SQ2:** *How do we determine if a late propagation has occured?*
  A late propagation occurs, if clones in a clone group received delayed modifications. This means that several clones in the clone family are inconsistent for a certain amount of time until they are updated. We have used the *Normalized Levenshtein Distance* as a metric to determine if two clones are consistent or inconsistent with each other.

- **SQ3:** *Is there a connection between the package distance and the propagation time of LPs?*
  We have extracted late propagations from three software systems and we have found no connection between package distance and propagation time. But we have discovered that some cases of LPs have a very long propagation time for code clones that are either in the same file or the same package, which seems to indicate that the developer has forgotten about the clones.

- **SQ4:** *To what extent does the commit activity influence the appearances of LPs?*
  Using the change view history, we have analyzed the commit activities for the 3 software projects and we have observed that the number of LPs affected by busy commit activities ranges from 50% to 70%. It seems for these systems there may be a weak link between commit acitivity and LPs appearances.

- **SQ5:** *What are the effects of LPs?*
  We have observed LPs causing bugs or a long delay to refactoring activities and these late propagations have a negative influence of software maintenance. Other LPs involve changes that can be delayed as they introduce no problems to the software systems.

With subquestion 1 and 2, we discussed our approach in finding late propagations in a software project. As stated earlier, ideally each commit should be self-contained and if we combine this statement with our observations of LPs in SQ3 to SQ5, it would seem that clone management tools may help software developers to change related clones consistently.

5.2 Contributions

With our research, we have made the following contributions:

- We have developed a tool that can extract late propagations for sofware repositories. We have used SDiff, which employs a different diff mechanism in comparison to other related studies [31, 2], to track changes made to code clones.
- We have used our tool on 4 software systems and we have extracted a large number of late propagations from 3 projects, which we have studied.

5.3 Future work

- **Incremental Clone Detection based on CCFinderX:** We have used CCFinderX to detect clones on every revision that we want to study. Since CCFinderX does not remember its previous output in revision i, it will analyze every source code in a project in revision i + 1. This is very wasteful as source files that have not been modified are analyze as well. Recently, the source code of CCFinderX has been released[1] and modifying CCFinderX to remember its previous results can reduce redundant processing of source files. Furthermore, the clone tracking ability should be added to CCFinderX, which removes the need for a seperate code change detector like SDiff. By doing these kind of modifications, it is possible to transform a clone detection tool to a clone evolution detector.
- **Analyze More Systems:** We have analyzed 4 software systems and only found LPs in 3 of them. It would be better to evaluate more systems to find out to get more data for SQ3 to SQ5.
- **Unparsable Snapshots:** We have encountered unparsable revisions with our tool, which means that some LPs might be missed. One solution to this problem, is to build a late propagation detector that does not employ a parser for tracking changes to code clones.

[1] http://www.ccfinder.net/

Bibliography

[1] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? a characterization of open source software repositories. In *ICPC '08: Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, pages 182–191, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society.

[3] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86, Washington, DC, USA, 1995. IEEE Computer Society.

[4] T. Bakota, R. Ferenc, and T. Gyimothy. Clone smells in software evolution. In *ICSM '07: IEEE International Conference on Software Maintenance*, pages 24 –33, oct. 2007.

[5] Hamid Abdul Basit, Simon J. Puglisi, William F. Smyth, Andrew Turpin, and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 513–516, New York, NY, USA, 2007. ACM.

[6] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[7] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Trans. Softw. Eng.*, 33(9):577–591, 2007.

[8] Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 85–94, Washington, DC, USA, 2009. IEEE Computer Society.

[9] Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Magiel Bruntink, Arie van Deursen, Tom Tourwe, and Remco van Engelen. An evaluation of clone detection techniques for identifying crosscutting concerns. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 200–209, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Elizabeth Burd and John Bailey. Evaluating clone detection tools for use during preventative maintenance. In *SCAM '02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, page 36, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Ldiff: An enhanced line differencing tool. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 595–598, Washington, DC, USA, 2009. IEEE Computer Society.

[13] A. Chen, E. Chou, J. Wong, A.Y. Yao, Qing Zhang, Shao Zhang, and A. Michail. Cvssearch: searching through source code using cvs comments. In *ICSM '01: Proceedings of IEEE International Conference on Software Maintenance*, pages 364 – 373, 2001.

[14] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 73–88, New York, NY, USA, 2001. ACM.

[15] Michiel de Wit, Andy Zaidman, and Arie van Deursen. Managing code clones using dynamic change tracking and resolution. In Tao Xie and Kostas Kontogiannis, editors, *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, pages 169–178, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Ekwa Duala-Ekoko and Martin P. Robillard. Clonetracker: tool support for code clone management. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 843–846, New York, NY, USA, 2008. ACM.

[17] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *ICSM '99: Proceedings of the IEEE In-*

ternational Conference on Software Maintenance, page 109, Washington, DC, USA, 1999. IEEE Computer Society.

[18] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999.

[19] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[20] Reto Geiger, Beat Fluri, Harald C. Gall, and Martin Pinzger. Relation of Code Clones and Change Couplings. In *Proceedings of the 9th International Conference of Funtamental Approaches to Software Engineering*, number 3922 in Lecture Notes in Computer Science, pages 411–425. Springer, March 2006.

[21] Michael W. Godfrey, Ahmed E. Hassan, James D. Herbsleb, Gail C. Murphy, Martin P. Robillard, Premkumar T. Devanbu, Audris Mockus, Dewayne E. Perry, and David Notkin. Future of mining software archives: A roundtable. *IEEE Software*, 26(1):67–70, 2009.

[22] Jan Harder and Nils Göde. Modeling clone evolution. In *Proceedings of the 3rd International Workshop on Software Clones*, pages 17–21. IEEE Computer Society, 2009.

[23] Takeo Imai, Yoshio Kataoka, and Tetsuji Fukaya. Evaluating software maintenance cost using functional redundancy metrics. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 299–306, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 485–495, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

[26] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[27] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems: Research articles. *J. Softw. Maint. Evol.*, 18(2):61–82, 2006.

[28] Cory J. Kapser and Michael W. Godfrey. ”cloning considered harmful” considered harmful: patterns of cloning in software. *Empirical Softw. Engg.*, 13(6):645–692, 2008.

[29] Shinji Kawaguchi, Takanobu Yamashina, Hidetake Uwano, Kyohei Fushida, Yasutaka Kamei, Masataka Nagura, and Hajimu Iida. Shinobi: A tool for automatic code clone detection in the ide. In *WCRE ’09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 313–314, Washington, DC, USA, 2009. IEEE Computer Society.

[30] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *ISESE ’04: Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, 2005.

[32] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *SAS ’01: Proceedings of the 8th International Symposium on Static Analysis*, pages 40–56, London, UK, 2001. Springer-Verlag.

[33] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Journal of Automated Software Engineering*, 3:108, 1996.

[34] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *WCRE ’06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 253–262, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Jens Krinke. Identifying similar code with program dependence graphs. In *WCRE ’01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE’01)*, page 301, Washington, DC, USA, 2001. IEEE Computer Society.

[36] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *WCRE ’07: Proceedings of the 14th Working Conference on Reverse Engineering*, pages 170–178, Washington, DC, USA, 2007. IEEE Computer Society.

[37] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[38] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[39] Angela Lozano and Michel Wermelinger. Tracking clones’ imprint. In *Proc. 4th Int’l Workshop on Software Clones*. ACM, 2010. To appear.

[40] Bart Luijten, Joost Visser, and Andy Zaidman. Assessment of issue handling efficiency. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, pages 94–97. IEEE, 2010.

[41] H. Malik and A.E. Hassan. Supporting software evolution using adaptive change propagation heuristics. In *ICSM '08: IEEE International Conference on Software Maintenance*, pages 177 –186, 28 2008-oct. 4 2008.

[42] Frank Mulder and Andy Zaidman. Identifying cross-cutting concerns using software repository mining. In *Proceedingsof the 4th International Joint ERCIM/IWPSE Symposium on Software Evolution (IWPSE-EVOL)*. ACM, 2010. To appear.

[43] Steven P. Reiss. Tracking source locations. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 11–20, New York, NY, USA, 2008. ACM.

[44] Chanchal K. Roy and James R. Cordy. An empirical study of function clones in open source software. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 81–90, Washington, DC, USA, 2008. IEEE Computer Society.

[45] Jaime Spacco and Chadd Williams. Lightweight techniques for tracking unique program statements. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:99–108, 2009.

[46] Suresh Thummalapenta, Luigi Cerulo, Lerina Aversano, and Massimiliano Di Penta. An empirical study on the maintenance of source code clones. *Empirical Software Engineering*, 15(1):1–34, 2010.

[47] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *In Proceedings of the Eighth IEEE Symposium on Software Metrics*, pages 67–76. IEEE Computer Society Press, 2002.

[48] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 220–229, Washington, DC, USA, 2008. IEEE Computer Society.

[49] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Trans. Softw. Eng.*, 31(6):429–445, 2005.