AN EMPIRICAL STUDY FOR THE IMPACT OF MAINTENANCE ACTIVITIES IN CLONE EVOLUTION

by

Lionel Marks

A thesis submitted to the School of Computing

in conformity with the requirements for

the degree of Master of Science

Queen's University

Kingston, Ontario, Canada

(November, 2009)

Abstract

Code clones are duplicated code fragments that are copied to re-use functionality and speed up development. However, due to the duplicate nature of code clones, inconsistent updates can lead to bugs in the software system. Existing research investigates the inconsistent updates through analysis of the updates to code clones and the bug fixes used to fix the inconsistent updates. We extend the work by investigating other factors that affect clone evolution, such as the number of developers.

On two levels of analysis, the method and clone class level, we conduct an empirical study on clone evolution. We analyze the factors affecting bug fixes and co-change (i.e. update cloned methods at the same time) using our new metrics. Our metrics are related to the developers, code complexity, and stages of development. We use these metrics to find ways to improve the maintenance of cloned code. We discover that one way to improve maintenance of code clones is the decrease of code complexity. We find that increased code complexity leads to a decrease in co-change, which can lead to bugs in the software.

We perform our study on 6 applications. To maximize the number of clones detected, we use two existing code clone detection tools: SimScan and Simian. SimScan was used to find clones in 5 of the applications due to its versatility in finding code clones. Simian was used to detect clones due to its reliability to find code clones regardless of language or compilation problems. To analyze and determine the significance of the metrics, we use the R Statistical Toolkit.

Acknowledgements

I gratefully thank my supervisor Dr. Ying Zou for her supervision, patience, and encouragement that helped me to complete my study. I also gratefully thank Dr. Ahmed Hassan for the very helpful meetings that simulated many research ideas for my thesis.

I would like to thank all the members in the Software Reengineering Research Group: Mr. Xulin Zhao, Mr. Hua Xiao, Mr. Hua (Kobe) Yuan, Mr. Brian Chan, Mr. Lionel Marks, Mr. Kingchun (Derek) Foo, Mr. Ran Tang, Mr. Andrew Carter, Mr. Chris Fiorentino, Mr. Mohamed Al Guindy, Ms. Jin Guo and Ms. Liliane Barbour for making my master study a wonderful experience.

I would like to express my gratitude to my committee members: Dr. Diane Kelly, Dr. Juergen Dingel, and Dr. Ying Zou for their valuable comments and feedback on this thesis.

Finally, I am grateful to my family and friends for their support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software maintenance is the act of modifying applications to meet new user requirements and to resolve undesirable behavior (i.e. software bugs) in code. It has been estimated that 60% of development effort can be attributed to software maintenance [100] and that it accounts for 70% of the cost of a software system [100]. These statistics show that the majority of time and money is spent on software maintenance, making this activity important for investigation. Code clones are known to cause further problems to software maintenance [80]. A code clone is a duplicated code fragment. If duplicated code fragments are not updated together, it can lead to bugs in the software system [34], [72]. It has been found that the maintenance taken to correct bugs is known to take close to 20% of developers' time [77]. Also, cloned code increases the size of the code in the system, making it more complex and difficult to understand [34]. As much as 20% of code in industrial and open source software is cloned [6], [82]. This shows that a substantial portion of the overall code can cause problems and hinder software maintenance.

1.1 Clone Detection

Cloned code consists of at least two code fragments that are similar. Three levels of clone similarity are described in the literature [56]:

1. Type-1 clones are those exactly the same, line-by-line due to copying and pasting a code fragment.

2. Type-2 clones augment Type-1 clones by allowing variable, type or method names to be different, but the code statements must not have any additions, deletions, or reordering.

3. Type-3 clones are the same as Type-2 clones, but allow additions, deletions, or reordering of code statements.

All the code fragments that are similar to one another are called a *clone class*. A single fragment of code that is similar is called a *clone instance*. Two fragments of similar code are referred to as a *clone pair*.

An example of a *clone class* is shown in Figure 1-1. The code from two methods, M1 and M2 each contain a duplicated code fragment from lines 3-16. Each of these fragments would be considered a *clone instance*. These clone instances are of Type-2 because the method names the variable identifier "addMore" on line 5 in M1 has been changed to "someMore" on line 5 in M2. No statement additions, deletions, or reordering has taken place. Assuming that there are no other code fragments similar to these in the software system, these two methods would be the *clone class*.

```
1: int M1(int [] vals, int
2: numVals){
3: int increment = 2;
4: int sum = 0;
5: int addMore = 4;
6: for (int i=0;i<numVals;i++)
7: {
8:    vals[i]=vals[i]+increment;
9:    sum=sum+vals[i];
10: }
11: sum=sum+addMore;
12: for (int i=0;i<numVals;i++)
13: {
14:   sum=sum*sum;
15: }
16: return sum;
17: }
```

```
1: int M2(int [] vals, int
2: numVals){
3: int increment = 2;
4: int sum = 0;
5: int someMore = 4;
6: for (int i=0;i<numVals;i++)
7: {
8:    vals[i]=vals[i]+increment;
9:    sum=sum+vals[i];
10: }
11: sum=sum+someMore;
12: for (int i=0;i<numVals;i++)
13: {
14:   sum=sum*sum;
15: }
16: return sum;
17: }
```

Figure 1-1 – Sample code clone

Similar code fragments like the ones shown in Figure 1-1 can be detected using clone detection tools. The most widely used clone detection tools either use a token or abstract syntax tree based approach [94]. In the first approach, a token in software is any number of characters, surrounded by white space (i.e. a word). Token based clone detection tools read in code as tokens and can use a compiler-style analysis to search these tokens for code clones. The advantage of token based clone detection tools is that they are efficient and can work on any programming languages. Token based clone detection tools also work on code that has syntax errors, meaning that even not run-able code can be scanned successfully.

The second approach uses an abstract syntax tree. This level of analysis provides more depth and detail for analyzing code for clones. As a result, abstract syntax tree clone detection tools can detect less similar clones than the token based clone detection tools because the level of analysis has more depth and detail. The problem with abstract syntax based clone detection tools is that they are slower, cannot run on code with syntax errors, and require much more memory than the token based clone detection tools.

1.2 Clone Evolution

When code clones evolve, the similar code fragments can either be updated together or not. The act of updating similar code together is called a *co-change*. For example, in Figure 1-2, line 11 has been modified in both duplicated code fragments by changing the plus sign to an asterisk.

```
1: int M1(int [] vals, int
2: numVals){
3: int increment = 2;
4: int sum = 0;
5: int addMore = 4;
6: for (int i=0;i<numVals;i++)
7: {
8:    vals[i]=vals[i]+increment;
9:    sum=sum+vals[i];
10: }
11: sum=sum*addMore;
12: for (int i=0;i<numVals;i++)
13: {
14:   sum=sum*sum;
15: }
16: return sum;
17: }
```

```
1: int M2(int [] vals, int
2: numVals){
3: int increment = 2;
4: int sum = 0;
5: int someMore = 4;
6: for (int i=0;i<numVals;i++)
7: {
8:    vals[i]=vals[i]+increment;
9:    sum=sum+vals[i];
10: }
11: sum=sum*someMore;
12: for (int i=0;i<numVals;i++)
13: {
14:   sum=sum*sum;
15: }
16: return sum;
17: }
```

Figure 1-2 – Sample code clone for co-change

A lack of co-change can cause problems in software. For example, if the plus sign was not changed to an asterisk in line 11 of M2, even though this change was meant to be made in all instances of this code, an undesirable result would be returned when the code for M2 is run. The developer may not know M2 had the duplicated code. In this case, the developer modifying M1 on line 11 (i.e., changing the plus sign to an asterisk) would not know to change the cloned line in M2.

To perform analysis on co-change and bugs in software, data can be found in code versioning systems (CVS). A CVS is a convenient way of keeping code synchronized when two or more developers modify the same code and it is also an efficient storage system for large amounts of code. Whenever a code fragment is modified by a developer, she or he submits the code to the CVS to keep the code up-to-date for other developers to use. When code is submitted, a *changelist* is created. A *changelist* contains the following data:

- list of all the files that were modified
- when code has been submitted by the developer

- modification of those files (i.e. what lines changed)
- name of the developer who submitted the code
- description from the developer who submitted code

1.3 Research Questions

Research has been done on investigating corrective maintenance (e.g. fixing software bugs) in cloned code evolution [10], [27]. Cloned code can be more problematic when the duplicated fragments are not co-changed together [34]. However, in the current state of the art in the software community, there is limited research on investigating why inconsistent co-change occurs. It is not well studied what conditions in development teams cause the need for more bug fixes in cloned code. Without this knowledge, we cannot improve the software maintenance processes causing developers to be unaware of co-changing cloned code. We propose to investigate techniques to help improve co-change and decrease the need for bug fixes in cloned code.

In our thesis, we aim to address the following research questions:

1. *Does the number of developers working on cloned code affect co-change or bug fixes?* A greater number of developers tend to decrease co-change and increase the need for bug fixes. It would mean that development teams should limit the number of developers who work on cloned code. This would lead to a decreased need for bug fixes in software and less money spent on software maintenance.

2. *Does the complexity of the cloned code affect co-change or bug fixes?* Less complex cloned code leads to increased co-change and decreases the need for bug fixes. It would suggest to developers to keep cloned methods with a singular purpose. By allowing fewer conditional statements that allow for additional functionality unrelated to the cloned code in

a method, it can increase co-change and prevent bugs. By knowing that very complex cloned code may not be properly maintained, developers may utilize other techniques to duplicate functionality when coding to help improve the maintenance of the cloned code.

3. *Does locality of cloned code affect co-change?* The understanding that cloned code that is in the same location co-changes more frequently, can help improve co-change. We can suggest to development teams to keep clones in the same location to improve clone maintenance, saving developers time and money.

1.4 Research Issues

To address the aforementioned research questions listed in Section 1.3, we envision the following three challenges:

1. *Determine the code level to perform the code clone analysis.* From our study, we plan to propose techniques to improve software evolution. Any techniques must be feasible to be implemented. As a result, we use the method level because it is an appropriate granularity to assign to a developer. Only one developer can work on a method at a time. Additional changes from another developer would require an extra submission to the CVS.

2. *The amount of data is large.* Only one or two applications are analyzed in many of the clone evolution studies [10], [27], [63]. With six applications in our case study, we needed mining techniques and analyses that could handle the large amount of data. To reduce the code scanned by our clone detection tools, we filtered any methods that would be irrelevant for analysis. The AST-based clone detection tool was used as much as possible because it was able to find more clones than the token based clone detection tool. However, when it could not handle the large amount of data or errors existing in the code, the more efficient

token based clone detection tool was used instead. To perform statistical analyses on the large amount of data to confirm our research questions, the suite of statistical tools available in Excel was insufficient. To overcome this challenge, we used the R statistical analysis tool [107] that has its own language and environment suited to handling large amounts of data.

3. *Determine the relevant factors to be mined for this analysis.* There is a lack of studies that perform similar work on cloned code. There was little inspiration to draw from other sources. One study that analyzed clone coverage (i.e. the percent of a file that is cloned) that influences co-change in cloned code [43]. The factor is called *Clone Coverage*, which is the percent of the file that is cloned. However, this study that did not look into other factors such as developer, code complexity, or bug fix metrics.

1.5 Goals of Thesis

This thesis has three goals.

1. *Investigation of bug fixes in cloned and non-cloned methods*. Since the aspect of bug fixes is common to both cloned and non-cloned methods, we propose to compare the two and investigate if different factors affect the effort for fixing bugs.

2. *Investigation of co-change in cloned methods.* We investigate the effects of bug fixes, code complexity, and developer metrics on co-change. This investigation builds upon the analysis from the first goal on cloned methods and determines important factors for clone evolution of an individual method.

3. *Investigation of co-change in clone classes.* We determine factors involving maintenance of all the cloned methods in a clone class, instead of the individual cloned methods.

In summary, by finding the most effective factors influencing bug fixes and co-change, we can suggest to development teams techniques that can help improve co-change and minimize the need for bug fixes in cloned code.

1.6 Overview of Thesis

The remaining chapters of this thesis are organized as follows:

- **Chapter 2**: We present the related work in the field of clone detection, bug fixes and co-change for cloned code.
- **Chapter 3**: We introduce the techniques used to gather and analyze the data from the CVS to obtain the cloned methods for analysis. The five major steps of gathering method changes, filtering method changes, detecting clones, identifying cloned methods, and performing statistical analyses are described.
- **Chapter 4:** In this section, we discuss the experiment setup for our case study and the threats to validity.
- **Chapter 5**: We test and provide explanations of the hypotheses that compare the bug fix effort of cloned and non-cloned methods. The hypotheses in this section relate to the first goal of the thesis.
- **Chapter 6**: The hypotheses that deal with co-change on the method level are discussed for a more in-depth analysis of cloned methods. The second goal of the thesis is discussed in this section.

- **Chapter 7**: We take an overall perspective of cloned methods with an analysis of co-change on the clone class level. The hypotheses related to the third goal are presented in this section.
- **Chapter 8**: We describe the contribution of our work and the future work.

Chapter 2
Related Work

2.1 Clone Detection & Analysis

Even though 2 main types of clone detectors are used in practice, 4 types of clone detectors can be used to find duplicate code for analysis: token-based [12], [34], [60], [106], metrics-based [82], abstract syntax tree based [17], [105], or program dependence graphs based [69]. Studies have compared the strengths and weaknesses of the different approaches [25], [94] using various clone detector tools. Many of the approaches offer visualizations to help developers understand the distribution of code clones in an application. The visualizations typically use dot plots to represent the code clones. Geiger et al. found that the dot plot visualization technique was most useful for smaller fragments of source code but did not scale well for large systems [43].

With the availability of many clone detection tools [14], [17], [60], [67], [69], [78], [105], [106], clones have been investigated in great depth using different levels of analysis. Initial studies only analyzed clone pairs [17], [34], [72], [82]. As time has progressed, many researchers use both clone pairs and clone classes [12], [15], [60]. In cases where the number of method revisions to the cloned methods is important, analysis can be performed on the individual clone instances [79].

In recent years, there has been some controversy in the software community as to whether code clones are useful or harmful [61]. Studies have shown that extra effort is needed for maintenance of cloned code [79], [80]. However, code clones can be useful as a way of developing new features starting from similar ones that currently exist [10], [61]. While this creates duplications, it also permits the use of stable and already tested code. On the other hand, code clones can still be harmful in software evolution. Duplicated code fragments can significantly increase the work to be done when enhancing or modifying code due to the extra numbers of lines that need to be updated [82].

To further understand the extent of the problem cloning presents, the research community has investigated the percent of clones in a system over time. Research has found that the percent of clones in a system remains relatively stable throughout the lifetime of an application [7], [72]. Kim et al. also observed that over 50% of cloned code cannot be eliminated by refactoring [63]. These works show that cloning is an ongoing problem that cannot be easily removed.

Part of the problem with clone maintenance is the requirement of co-change. Lague et al. [72] reported that half of the changes to a clone were co-changed with other clone instances. A lack of co-change from one clone instance to another can lead to bugs in software [43]. Even though only Type-1 clones were considered for the study, from the statistic of 50% of changes as co-changes, we consider co-change a significant factor to be investigated in the study of clones.

2.2 Co-Change

The initial research on co-change was performed for all code together (i.e. not separating cloned and non-cloned code for analysis). The identification of co-change in code can be accomplished on the file level using CVS repository logs [5], [21], [43], [113] and on the module level using different releases [40]. Co-change represents a non-trivial dependency among code fragments and suggests that code fragments that co-changed in the past are likely to require developers to update all dependent code fragments when one of the code fragments is modified in the future [5]. The knowledge of dependent code fragments can help speed up maintenance, improve maintenance quality, and reduce reworking of parts of code that are left behind during updates. One technique for increasing the number of co-changes identified is to use time windows. Using time windows, all the files updated within a set interval of time are considered to have changed together instead of restricting co-change to the same changelist [5]. Using visualizations, co-change can be observed by developers to note dependencies between subsystems [22]

For cloned code specifically, co-change is especially important due to the fact that the lines of code are exactly the same and updates to the duplicated lines in one clone instance generally require updates in the duplicated lines in other clone instances. As a result, a lack of co-change in cloned code can lead to bugs in software [43]. Studies have analyzed the problem using a difference algorithm to analyze inconsistent changes in bug fixes [10], [27]. To help remind users to update clone instances together during development, a plug-in for Eclipse has been created [33].

2.3 Software and Clone Evolution

Software evolution (i.e. software maintenance) is the process of modifying a software system to correct faults, improve performance or other attributes, or adapt to a change in environment [20], [55]. A study of software evolution can be carried out by mining a CVS to obtain different versions of the code for analysis. The usage of product release versions as raw data can be used [40], [41], [42]. A more in depth approach can collect and analyze all the changes to code, not only product release versions of the code [9], [42], [88], [114]. One purpose of the studies is to find weaknesses in the software architecture and suggest subsystems of code that would benefit from restructuring.

The study of software evolution exclusively for cloned code is called clone evolution. Clone evolution studies can work on the individual clone level [72], [79] or the clone class level [10],

[63], [27]. The work by Kim et al [63] analyzes co-change specifically for clone classes and determined whether the clone classes evolved together or independently. The knowledge of the way clone classes evolve is useful for understanding the techniques used by developers to handle clones and helps software teams working on projects to understand the behavior of clone evolution. In our work we investigate co-change on both levels to provide a complete perspective on clone evolution and co-change.

2.4 Bug Fixes

Using simple data mining techniques, bug data for applications can be found in bug repositories such as Bugzilla [102]. The unique bug ids in a Bugzilla repository can be matched to changelist descriptions in a CVS repository to create mappings between the bug fixes and the code modified for the fix [37], [41], [89]. However, one drawback to this type of analysis is that bug IDs are not always present in CVS description logs. This lack of traceability between the bug repository and a CVS can undermine research into bug fixes for a software system [11].

Once the bug data is retrieved, it can be presented using visualizations to help managers make decisions for distributing development effort. Baker et al. present a tool for visualizing a software system with bug information for each subsystem and bug fixes applied on previous bug fixes that were performed incorrectly [13].

Given one of the goals of the thesis is to reduce the need for bug fixes in cloned code, another method to help reduce the need for bug fixes in software is to utilize tools to automatically detect bugs in the source code. Automatic detection of bugs using source code analysis has been examined at length [16], [26], [30], [38], [48], [54], [101], [104]. Three types of analysis are used in automatic bug detection from source code: formal proof, partial verification, and unsound techniques [54]. Each technique has its strengths and weaknesses. The formal proof technique is the most effective and is the best guarantee a program does not contain bugs, but is too costly for most programs due to the difficulty in constructing proper proofs. Partial verification techniques are less costly, but do not guarantee correctness in all executions. Unsound techniques are the least costly and identify probable bugs by detecting program patterns that are known to be buggy. The drawback to unsound techniques is that they may produce a lot of false positives and false negatives. Automatic bug detection tools are useful as aids to prevent bug fixes but cannot solve all the problems of bugs in software. As a result, we analyze trends for bug fixes in applications using static source code metrics.

2.5 Metrics to Analyze or Predict Bug Fixes

Some researchers have argued against the use of static code attributes claiming that their information content is very limited [36]. However, static code attributes are easy to collect [110], interpret and research has successfully used them to build defect predictors [86], [91], [109] . Furthermore, the information content of static code attributes can be increased using call graphs to reduce false positives [65].

There is research on predicting bugs based upon complexity metrics in software after release using the lines of code [18], [45], [64], [110], the Halstead volume metric [108], [110], and cyclomatic complexity [45], [91]. We focus our research on cyclomatic complexity because it calculates the conditional complexity of the code [83]. We are interested in the evolution of conditional statements of cloned and non-cloned methods over the lifetime of the methods. In addition, high cyclomatic complexity can lessen productivity of developers [44] and studies have correlated cyclomatic complexity with bug fixes [97]. Our unique contribution to the work on comparing complexity metrics to bugs in software is the separation of cloned and non-cloned code. All other studies have investigated the problem with cloned and non-cloned code together.

In addition to using complexity metrics to predict faults in software, the number of developers can be used to predict software defects using linear models [90]. This suggests a relationship exists between the files developers updated and the number of bug fixes in software [84]. We extend the work by analyzing developers and bug fixes in cloned and non-cloned code separately.

2.6 Metrics to Analyze or Predict Co-Change

Using metrics to predict co-change is a relatively new area of study in comparison to metrics to predict bug fixes. Specifically dealing with cloned code, Geiger et al. determined that clone coverage (percent of a file that was cloned) can be used to predict co-change of cloned code [43]. We continue this work by using different metrics to analyze co-changing cloned code.

We investigate metrics involving developers due to the research showing links between software artifacts and developers [47], [52], [95]. For example, a study has researched impact management, where developers try to minimize the impact of their work on others when making changes to code [98]. Increased awareness among developers about each other's relevant activities can speed up completion of tasks [47]. We focus our research on co-change and determine if developer metrics significantly impact co-change in cloned code.

In addition to the relationship between developers and co-change, we also analyze the relationship between the location of clones in the file system and co-change. Kasper et al. [62] analyzed two subsystems for cloned code and found 60% and 79% of the clones existed in the home directory for each subsystem. The result was explained that it might be harder to find clones when they are not in the home directory (i.e. directory with the majority of clones) of the clone class. We extend the work by testing if clones in the home directory co-change more frequently than clones not in the home directory.

To find stronger relationships between instances of cloned code, we use the names and signatures of methods inspired by earlier work [92]. Research has used abstract syntax tree matching on methods of the same name to help people understand similar code more easily and highlight what changes have been made [92]. This is an example of work that would help raise awareness for developers about cloned code in a system that has been modified. Since our work already finds similar portions of code through clone detection, we determine the level of similarity of method names between cloned methods in a clone class and investigate if cloned methods with similar names are co-changed more or less frequently.

Chapter 3
Detecting Cloned Methods for Analysis

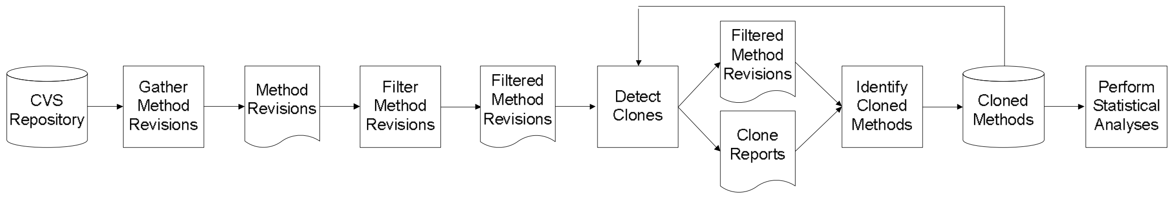

Figure 2-1 - Major steps of approach

The five major steps for detecting co-changing clone pairs are shown in Figure 2-1. A method revision occurs whenever the code of a method is updated. Our approach gathers the method revisions for every method from the CVS. By saving a copy of a method every time it was modified or created, we have a record of the code at every revision in the lifetime of every method. We filter out any methods that are either too trivial for analysis or those that would not be included in the final software product to be shipped to customers. We detect clones in the method revisions by using abstract syntax tree and token based clone detection tools on the method revisions to find Type-1, Type-2, and Type-3 clones. The clone reports generated from these clone detection tools are read by a tool we created to identify cloned methods. After the cloned methods are detected and ready for analysis, we perform statistical analyses by generating metrics from our tool. The metrics are then analyzed to find common trends in the applications under investigation.

Since the code under development is constantly changing, we incrementally update the identified cloned methods. An incremental update involves obtaining the new method revisions, filtering the unwanted methods, detecting clones in the new method revisions, merging the clone reports, and identifying the new cloned methods. All of these steps can be done in a day using the original (i.e. non-incremental update) techniques used to identify the cloned methods, except for the step of detecting clones. Clone detection of all the method revisions of an application can take hours, or even days. To speed up the process for clone detection of an incremental update, we scan the new method revisions with only the most recent existing method revisions because older method revisions cannot be cloned. For example, assume an application only had methods M1 and M2. M1_v4 and M2_v4 are the most recent method revisions before the incremental update and represent the fourth method revisions for both methods M1 and M2. M1_v5, M1_v6, M2_v5, and M2_v6 are the new method revisions. Only M1_v4, M1_v5, M1_v6, and M2_v4, M2_v5, M2_v6 need to be included in the clone scan. Earlier method revisions of M1 and M2 (i.e. M1_v1, M1_v2, M1_v3, M2_v1, M2_v2, M2_v3) can be excluded to save time and processing power. This ensures all possible clones are identified and also reduces the time for the clone detection for an incremental update to a couple hours.

3.1 Gathering Method Revisions

The purpose of gathering the method revisions is to provide the necessary data to identify cloned methods. Every changelist in a CVS has a record of the files and lines of code modified from that revision. By synchronizing with the first changelist for a CVS, one can retrieve the code as it was after the very first code revision for a method. Every subsequent changelist can then be used to update the code to its next version. Every time the code of a method was created or updated, we save a copy so that we have all the versions of every method since the code is first created. During this process, a record of all the changelists for every method revision can be created so that mappings from each method revision can be made to its corresponding changelist. These mappings are later used as a chronological record of the exact times that methods changed together to determine when methods were cloned and for detecting co-change.

3.2 Filtering Methods

After gathering all the method revisions, we filter out methods that are not useful for our analysis. Any method that is abstract or a setter/getter was removed from analysis. Abstract methods do not have code within them and therefore cannot contain cloned code. A setter method only has assignment statements (i.e. no control flow, no method calls, etc.) A getter method only returns a value. Although a clone detection tool may find setters or getters to be clones of each other, they are trivial and not useful for analysis of cloned code.

Some methods are written for testing the functionality of code. These methods are not included in the final build of code to be shipped to customers. Such methods have the keyword "test" in the method name. As a result, we ignore these methods in our analysis.

3.3 Detecting Clones

To find code fragments that are similar, SimScan [105] and Simian [106] are used for clone detection. SimScan is used because it is an abstract syntax tree (AST) based clone detection tool that compares the AST trees of code fragments for similarity. It is able to find clones of varying sizes and similarities. An AST based clone detection tool can find clones of Type-1, Type-2, and Type-3. Simian is a token based clone detection tool that is able to insert wildcards for identifiers, characters, and literals in code during clone detection. Token based clone detection tools can find clones of Type-1 and Type-2. We use both clone detection tools to their maximum capabilities. SimScan was used to find Type-1, Type-2, and Type-3 clones while Simian was used to find Type-1 and Type-2 clones.

In our analysis, SimScan is the better clone detection tool when it worked because it is able to find more clones than Simian. However, SimScan cannot always find clones in an application due to its code size or methods containing coding errors. In these cases, Simian is used instead.

3.4 Identifying Cloned Methods

We identify cloned methods by finding the changelist that creates the duplicate code fragment in another method. In Figure 2-2, each box represents a method revision. Boxes labeled "clone" are method revisions with cloned code. To find the changelist that creates the duplicate code, we find the earliest method revision that contains the cloned code. In Figure 2-2, the changelist that creates the duplicate code is at t_4. For analysis of the cloned methods M1 and M2, we analyze all method revisions that occur after time t_4.

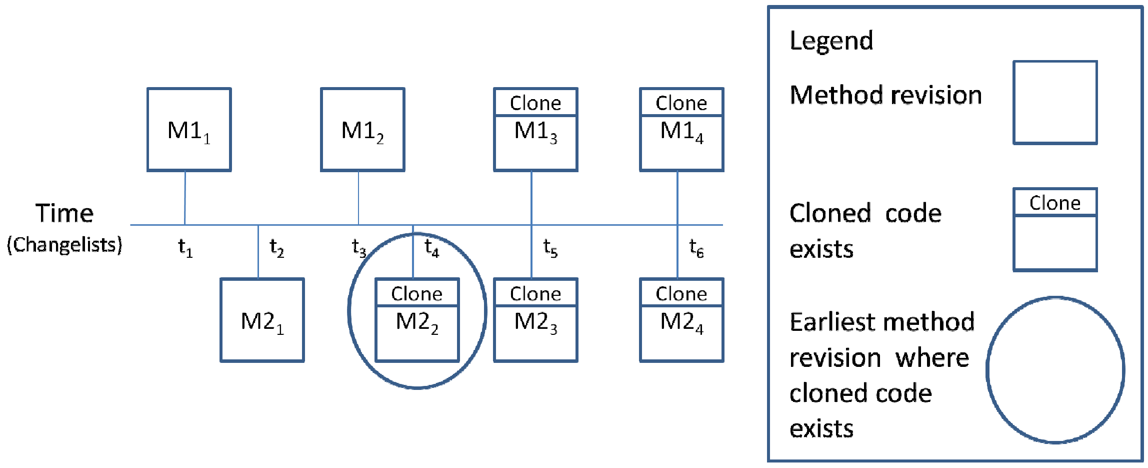

Figure 2-2 - Finding the earliest method revision with cloned code

In the case of more than two cloned methods in a clone class, we perform the same analysis for each method to determine the earliest method revision that contains the cloned code. All method revisions after the time the method is cloned are considered for analysis. All the method revisions before the method is cloned are not considered for analysis because no duplicate code exists in the method.

3.5 Performing Statistical Analysis

To find common trends in cloned methods for analysis of clone evolution, we analyze the characteristics of each method revision. From these characteristics, we formulate metrics to give us insight into the factors that affect bug fixes and co-change. Examples of metrics we investigate are code complexity, the number of developers who work on the method, and the similarity of method names.

To evaluate our hypotheses of the trends, we use the R statistical analysis tool [107] to analyze our metrics. The tool is used to calculate whether the trend responds positively or negatively to the factor under investigation. In addition, we determine if a statistical significance exists in the data. For statistical significance, we ensure the finding has a 95% confidence for each application under study.

Chapter 4
Case Study Introduction

We introduce the setup of the 6 applications for our case study system and the implementation for analysis using clone evolution metrics. We also present the technique used for data validation and the potential threats to validity.

4.1 Setup for case study

For our case study, we analyze 6 different applications. The decision to analyze the 6 applications is to ensure a variety of applications with different functionality are investigated.

Application 1 provides a user interface. Applications 2 and 3 implement protocols for communication. Application 4 is a security application that ensures secure transmission of e-mail. Applications 5 and 6 are used as hardware interfaces.

We scanned the code for clones using two clone detectors. Applications 1-5 have clones identified with SimScan [105], an AST based clone detection tool. The AST based clone detection tool failed to return a result for Application 6 due to an unexpected error in the SimScan clone detector. To compensate, we used Simian [106], a token based clone detection tool to identify clones in Application 6. All clones used for the study have a minimum of 7 lines of duplicated code.

The precision of a clone detector is a measure of the percent of relevant clones retrieved. The irrelevant clones (i.e. false positives) are removed from analysis.

App	Clone Detector	Precision
1	SimScan	85.44%
2	SimScan	88.89%
3	SimScan	91.07%
4	SimScan	85.96%
5	SimScan	95.92%
6	Simian	92.31%

Table 4-1 – Precision of clone detection in each application

From Table 4-1, we show the precision of clone detection for each application and its clone detector. Applications 1, 2, 4, 5, and 6 have false positive clones that contain many conditional statements. However, the content in the cloned fragments are not similar and are therefore rejected. Application 3 has false positive clones that contain hard coded statements for data output or initialization that are not similar.

The code is analyzed for each application individually because we discovered that each application had its own characteristics: different developers, different code lifetime, etc. This makes analyzing all the applications together less feasible and is consistent with past work from Nagappan et al. [91].

Cloned methods with independent evolutions never co-change. As a result, these methods do not require consistent updates of duplicated code and are removed from analysis of cloned code. We use a minimum number of two co-changes in the cloned methods to ensure a co-changing relationship exists in the methods studied [10].

4.2 Implementation

We have four general steps for analyzing the evolution of cloned and non-cloned methods in our study. First, we identify the method revisions appropriate for analysis in the method. For cloned methods, method revisions appropriate for analysis are after the method is cloned. For non-cloned methods, all method revisions are used for analysis. Second, we analyze each method revision to identify various characteristics for the production of metrics. Third, we perform calculations specific to the metric over all the method revisions under analysis. Fourth, to make the metrics relative to the number of method revisions or developers in the method, we use normalization techniques.

4.3 Data Validation

For verification of non-parametric data that has approximately the same shape, a Wilcoxon-Mann-Whitney T-Test is appropriate [24], [50]. We present the distributions of the data in H_1 of Chapter 5 that affirm the data has the same relative shape. The Wilcoxon-Mann-Whitney T-Test verifies the data by comparing the medians from the distributions [51]. We perform a two-tailed test to handle statistically significant cases of greater or lesser value than the expected result. Using a confidence level of 95%, if the p-value is less than 0.05, the data sets are not from the same population and we can reject the null hypothesis under examination. We use the Wilcoxon-Mann-Whitney T-Test in all our hypotheses for the case study.

4.4 Threats to Validity

We now discuss the different types of threats that may affect the validity of the results of our experiment.

External validity tackles the issues related to the generalization of the result. Our study performs analysis on 6 applications. The 6 applications represent a variety of types of applications. Each of the 6 applications has a large number of cloned and non-cloned methods.

Nevertheless, we should, in the future, test our hypotheses using other applications to determine if our results hold for other software systems.

In our hypotheses, we use the buggy cloned and non-cloned methods for analysis of bug fixes. A buggy method is defined as a method with at least one bug fix. We specifically analyze only methods with bug fixes because the results including the non-buggy methods are not as significant. In the future, we can extend the analysis of bug fixes to include methods without bug fixes to establish more general results on cloned non-cloned methods.

In the applications under study, a mapping between a bug repository and the CVS changelists was not available. As a result, to categorize a changelist as a bug fix, we search the changelists for words that involve bug fixes (i.e. bug, fix, etc.). We perform manual review of each changelist to ensure the only methods updated are associated with the bug fix. As a result of the manual review, we are certain no false positives occurred in our study. However, false negatives may occur due to developers that omitted mention of the bug fix in the changelist description or because the test suites missed bugs that exist in the software.

Internal validity is a concern with the issues related to the design of our experiment. The use of two different clone detectors can potentially introduce a bias in the study. Our AST-based clone detector can identify less similar duplicated code fragments (i.e. Type-3) than our token-based clone detector. We mitigate the discrepancy through manual review of the cloned methods to ensure each cloned method analyzed is appropriate for clone evolution study.

Bug fixes in cloned methods are counted on the method level because we consider a method to be a reasonable unit of work that can be assigned to a person. As a result, we do not differentiate if bug fixes in a cloned method are in the cloned or non-cloned code. The number of bug fixes reported in cloned methods may not reflect solely the number of bug fixes in cloned code.

Methods with more method revisions have more chances to change in complexity and have more bug fixes. For the metrics in our study, we use normalization techniques to make comparisons that take into account the numbers of method revisions or developers in each method studied. Our normalization techniques are bias-free of all metrics except for code complexity. We analyze the change in code complexity in software evolution by comparing the complexity of the initial, median, and final versions of each method. The technique is useful for obtaining data on the changes in complexity throughout the lifetime of the method. However, methods with a greater number of method revisions can be compared against methods with fewer method revisions in the median and final versions of the methods. For example, if we consider methods A and B. Method A has 5 method revisions. Method B has 13 method revisions. In a comparison of the complexity of the final versions of the methods, we compare the complexity of a method with only 5 method revisions, to a method with 13 method revisions. Although the number of method revisions does not necessarily increase the complexity, the analysis can be biased due to a larger number of method revisions in Method B. One possible solution is to use the 1st, 5th, and 10th method revisions of each method. However, methods with fewer than 10 method revisions are excluded from the analysis of the 10th method revision. Methods with greater than 10 method revisions have complexity data that is excluded from analysis. We currently do not have a solution to handle the problem and suggest further research to create a better normalized approach to the investigation of changes in complexity in software evolution.

To measure complexity, we chose to use cyclomatic complexity even though lines of code have been found to be as accurate in the prediction of bug fixes [45]. Cyclomatic complexity counts the number of decisions in the code as opposed to treating each line of code as equally complex. By tracking the number of decisions, we can note if new logical branches are added or removed during development. Other complexity metrics found to also have similar predictive power for forecasting of bug fixes [45] are the Halstead program volume metric [49] and the total number of operands in a program.

Construct validity is a concern as to the meaningfulness of the measurement. In our study, the results from our analyses are application specific. The characteristics of each application provide different reasons to explain the development effort of bug fixes in cloned and non-cloned methods. The coding styles used for the cloned methods influenced our results. Some coding styles inherently allow for a greater or fewer numbers of co-changes with respect to the total number of method revisions to the method. As a result, the applications and number of applications that support our hypotheses differs between experiments.

Chapter 5
Bug Fix Analysis of Cloned and Non-Cloned Methods

We analyze bug fixes in cloned methods and non-cloned methods to determine the factors that most influence bug fixes. Our work is similar to analysis of bug repositories[37], [40], [41] and fault prediction where correlation or linear models analyses are performed to determine if metrics are relevant to bug fixes [50], [91], [93]. In Table 5-1, we present the list of hypotheses for analyzing the impact of bug fixes in cloned and non-cloned methods. We first analyze the features of cloned and non-cloned methods to determine which has greater percent of development effort for bug fixes in H_1 and H_2. The intuition for a difference in development effort for bug fixes is that if cloned methods are not co-changed, bugs can be created. In H_1, we investigate the development effort and the distribution of bug fixes in all cloned and non-cloned methods. In H_2, we focus our analysis on the buggy cloned and non-cloned methods (i.e. the methods with at least one bug fix) and analyze the development effort for bug fixes in buggy cloned and non-cloned methods. Next, we continue our analysis of the buggy cloned and non-cloned methods to find the reasons that make the buggy cloned methods have greater development effort for bug fixes in H_3 and H_4. In H_3, the complexity of the buggy cloned and non-cloned methods is investigated to explain the greater bug fix effort for buggy cloned methods. The reason for analyzing complexity is that we expect the more complex methods to have needed more bug fixes. In H_4, we analyze the number of developer switches in buggy cloned and non-cloned methods. The explanation for the investigation of developer switches is that different developers changing the methods may require greater effort in re-understanding the code. Also, each time the code is re-understood, changes could be missed, and lead to new changes in logic that do not take into account how the current code functions. In our last step we determine the distribution of development effort in early and later development in order to improve maintenance of buggy cloned methods in H_5, H_6, and H_7. In H_5, an investigation of the development effort for bug fixes for buggy cloned and non-cloned methods in early development is conducted to note if buggy cloned methods have a greater proportion of bug fixes in early development. In H_6, we compare the development effort for bug fixes for buggy cloned and non-cloned methods in later development. In H_7, the development effort for bug fixes between early and later development is compared for buggy cloned and non-cloned methods. In H_8, we investigate the percent of developers who perform bug fixes in buggy cloned and non-cloned methods. The intuition for the hypothesis is that more developers performing bug fixes may lead to patchy code that continuously requires more fixes over the lifetime of the method.

	Hypothesis
H_1	Cloned methods have greater bug fix effort than non-cloned methods
H_2	Buggy cloned methods have greater bug fix effort than buggy non-cloned methods
H_3	Complexity of buggy cloned methods is higher than complexity of buggy non-cloned methods
H_4	The number of developer switches is lower in buggy cloned methods relative to buggy non-cloned methods
H_5	In the early development stage (first 6 months), the bug fix effort is greater in buggy cloned methods than buggy non-cloned methods
H_6	In the later development stage, the bug fix effort is less in buggy cloned methods and buggy non-cloned methods
H_7	Buggy cloned methods have greater bug fix effort in early development relative to later development than buggy non-cloned methods
H_8	The percent of developers performing bug fixes on buggy cloned methods is greater than for buggy non-cloned methods

Table 5-1 – List of hypotheses for analysis of bug fixes in cloned and non-cloned methods

5.1 H_1 –Cloned methods have greater bug fix effort than non-cloned methods

5.1.1 Definition of Metrics

To represent the bug fix effort (i.e. percent of method revisions that are bug fixes), the metric used is the bug fixes per revision shown in Equation (5-1).

$$BugFixPerRev = \frac{\#BugFixes}{\#Revisions} \tag{5-1}$$

The #BugFixes represents the number of bug fixes that are performed on the method under study. The #Revisions represents the number of method revisions made to the method.

5.1.2 Testing of H_1

The purpose of H_1 is to determine if the cloned methods have greater problems than non-cloned methods. A non-parametric test compares 2 independent samples of data that have an unpaired relationship. We use a statistical non-parametric test to study the significance of the bug fix effort in cloned and non-cloned methods. We formulate the following test hypothesis:

$$H_O\colon \mu(BugFixPerRev\_C) - \mu(BugFixPerRev\_NC) \leq 0$$

$$H_A\colon \mu(BugFixPerRev\_C) - \mu(BugFixPerRev\_NC) > 0$$

$\mu(BugFixPerRev\_C) - \mu(BugFixPerRev\_NC)$ is the difference in population means between the bug fixes per revision in cloned methods and the bug fixes per revision in non-cloned methods. The purpose of testing the hypothesis is to test the viability of the null hypothesis through the analysis of the experimental data. The null hypothesis H_O holds if the p-value is greater or equal to 0.05, then the difference in means is not statistically significant and is likely due to the variability of the data. If H_O is rejected with a high probability (i.e. $p\text{-value} < 0.05$), then we can accept the alternative hypothesis H_a, meaning our suggested hypothesis is proven true. We are then confident that the cloned methods have greater development effort for bug fixes than non-cloned methods for the application under study.

As support for using the Wilcoxon-Mann-Whitney T-test for verification test for our data, we investigate the distribution of the bug fix effort in cloned and non-cloned methods in the figures below.

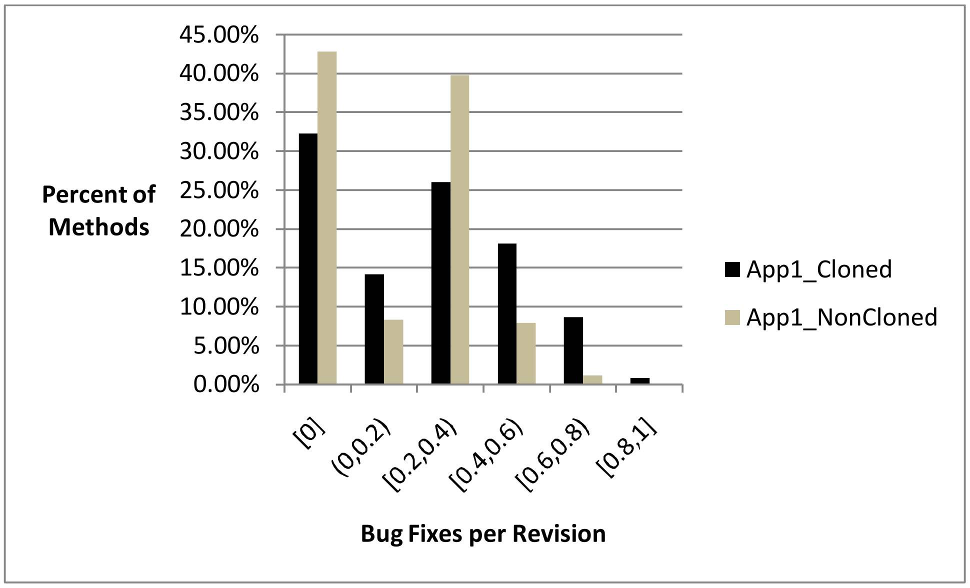

Figure 5-1 – Distribution of bug fixes per revision for cloned and non-cloned methods in Application 1

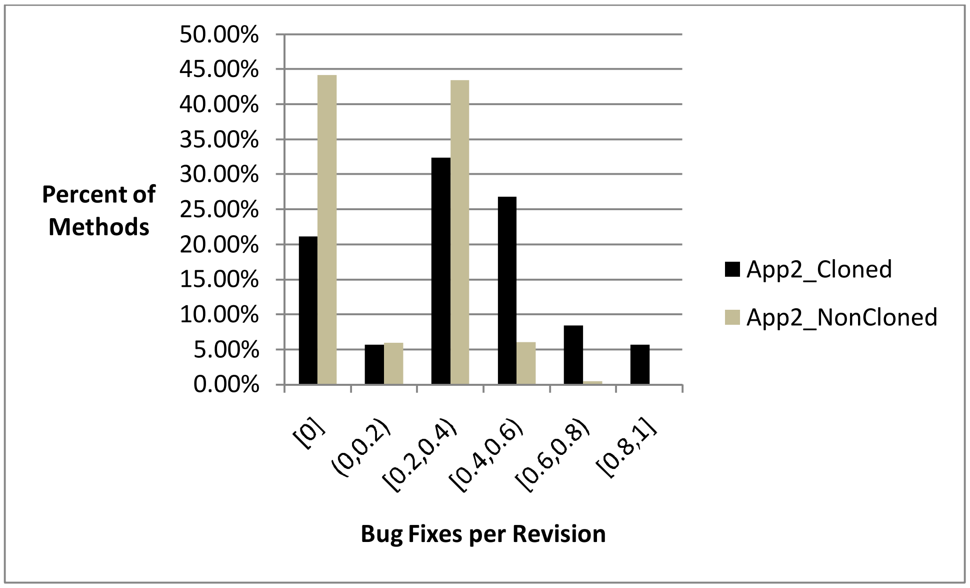

Figure 5-2 – Distribution of bug fixes per revision for cloned and non-cloned methods in Application 2

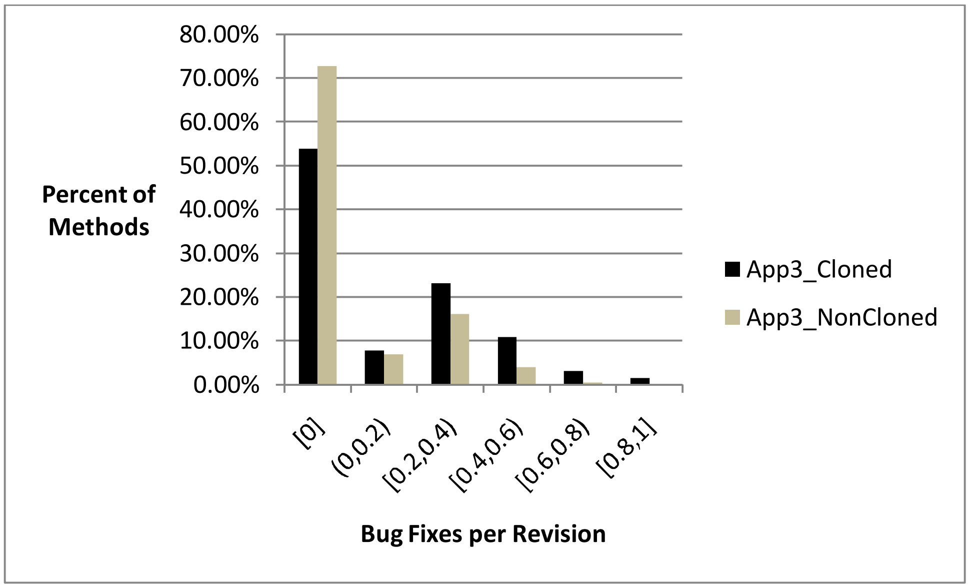

Figure 5-3 – Distribution of bug fixes per revision for cloned and non-cloned methods in Application 3

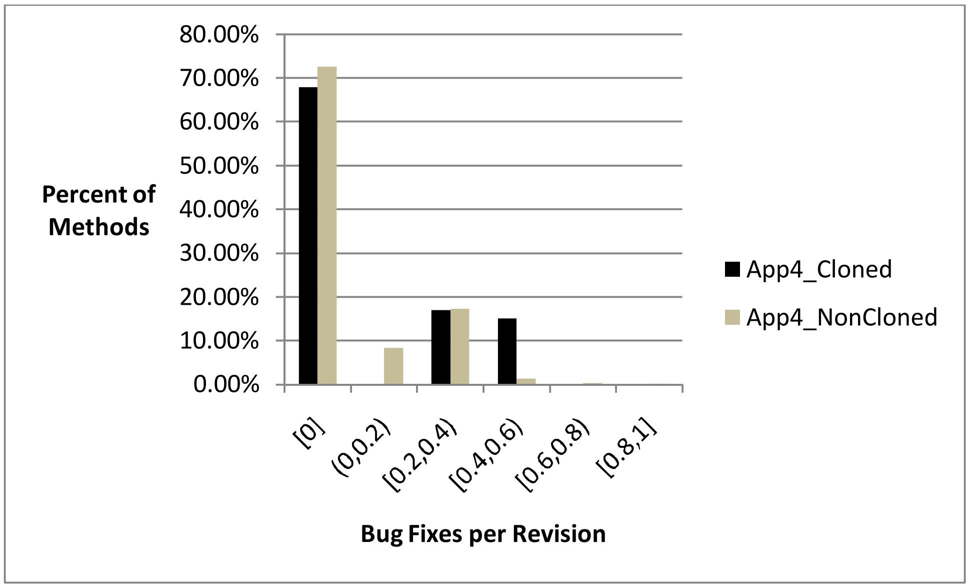

Figure 5-4 – Distribution of bug fixes per revision for cloned and non-cloned methods in Application 4

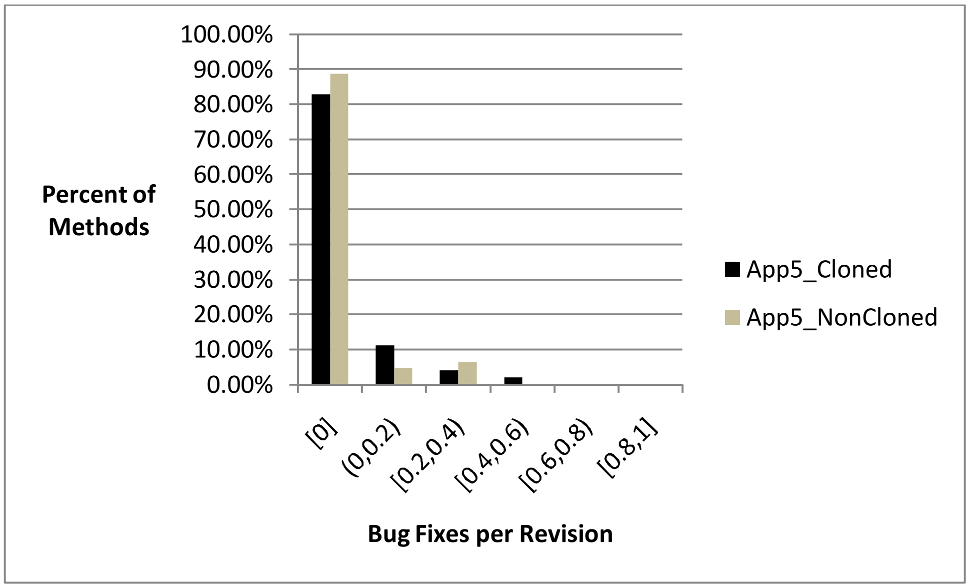

Figure 5-5 – Distribution of bug fixes per revision for cloned and non-cloned methods in Application 5

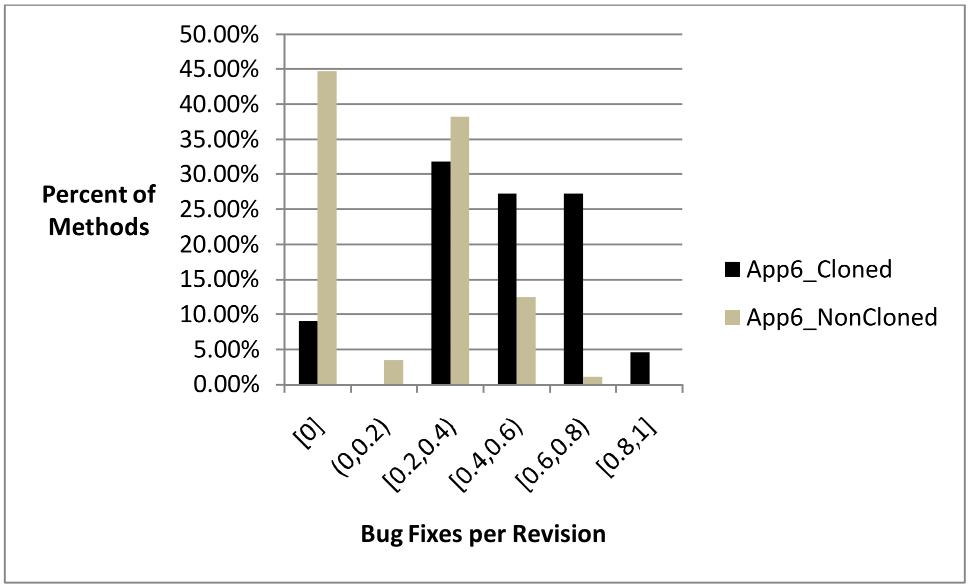

Figure 5-6 – Distribution of bug fixes per revision for cloned and non-cloned methods in Application 6

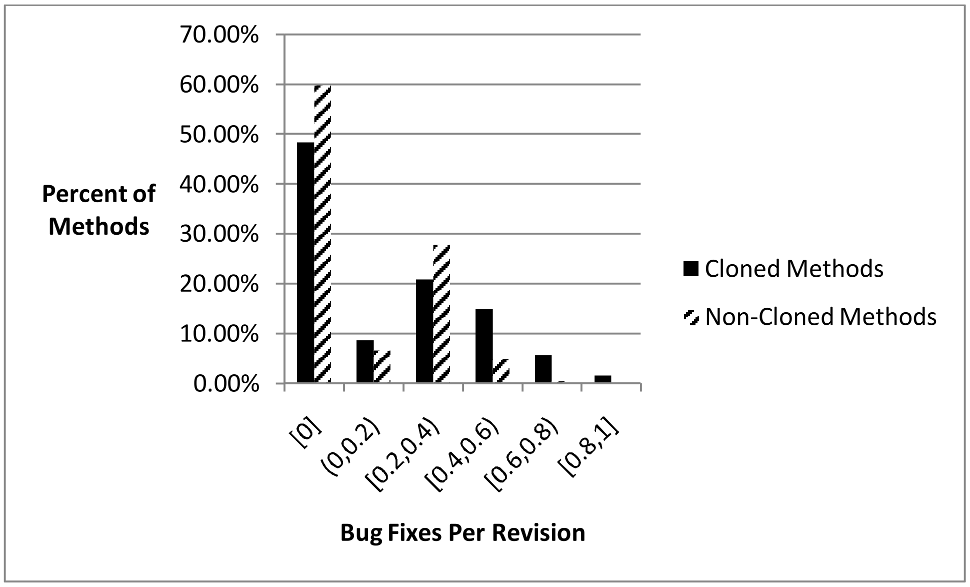

Figure 5-7 – Distribution of bug fixes per revision for cloned and non-cloned methods in all 6 applications

Figure 5-1, Figure 5-2, Figure 5-3, Figure 5-4, Figure 5-5, and Figure 5-6 relate to the distributions of bug fixes per revision in cloned and non-cloned methods for Applications 1, 2, 3, 4, 5, and 6 respectively. Figure 5-7 corresponds to the distribution of bug fixes per revision in cloned and non-cloned methods in all 6 applications. From visual analysis of the 7 figures, we note the distributions of cloned and non-cloned methods have the same approximate shape. Large percentages of the methods have bug fixes per revision in the ranges of [0], [0.2, 0.4), and [0.4, 0.6). Since the distributions of the cloned and non-cloned methods have comparatively the same shape, a *Wilcoxon-Mann-Whitney* T-Test is appropriate [24], [50].

5.1.3 Results

1	**2**	**3**	**4**
App	$\mu(BugFixPerRev\_C) - \mu(BugFixPerRev\_NC)$	$\frac{\mu(BugFixPerRev\_C) - \mu(BugFixPerRev\_NC)}{\mu(BugFixPerRev\_C)}$	P-Value
1	**0.08**	**0.33**	**0.00**
2	**0.18**	**0.52**	**0.00**
3	**0.08**	**0.53**	**0.02**
4	0.05	0.44	0.12
5	0.01	0.37	0.69
6	**0.28**	**0.61**	**0.00**

Table 5-2 – Comparison of bug fixes per revision for cloned methods and non-cloned methods

Of the 6 applications studied in Table 5-2, Applications 1, 2, 3, and 6 are statistically significant and have higher bug fixes per revision in cloned methods. From the fourth column in Table 5-2, the 2 applications that are not statistically significant at a confidence level of 95%, Applications 4 and 5 have greater bug fix effort in cloned methods. Application 4 has a confidence level of 88% and supports the hypothesis. Application 5 is not statistically significant because 83% and 89% of its cloned and non-cloned methods respectively have zero bug fixes.

Given 5 of the applications support the hypothesis with at least a confidence level of 88%, we conclude that H_1 holds.

From Figure 5-7, the data representing all 6 applications studied have cloned and non-cloned methods have 48% and 60% of its methods with zero bug fixes per revision. Because many of the methods in cloned and non-cloned code do not have any bug fixes, we focus our efforts on the buggy methods that have at least one bug fix. In these methods that have required bug fixes, we note the differences in these buggy cloned and non-cloned methods to investigate techniques to decrease the bug fix effort required in these methods.

5.2 H_2 – Buggy cloned methods have greater bug fix effort than buggy non-cloned methods

From H_1, over 48% of the cloned and non-cloned methods do not have bug fixes. For our analysis, we want to isolate the problems with the buggy methods (i.e. has at least one bug fix) to improve software evolution for cloned and non-cloned methods. Focusing our analysis on the buggy methods, we verify if buggy cloned methods require greater bug fix effort than buggy non-cloned methods to determine which is more problematic.

5.2.1 Definition of Metrics

The metric used for the analysis is the bug fixes per revision (5-1).

5.2.2 Testing of H_2

We use a statistical non-parametric test to compare the bug fix effort in buggy cloned and non-cloned methods. We formulate the following test hypothesis:

$$H_O: \mu(BugFixPerRev\_BC) - \mu(BugFixPerRev\_BNC) \leq 0$$

$$H_A: \mu(BugFixPerRev\_BC) - \mu(BugFixPerRev\_BNC) > 0$$

We test the null hypothesis with the difference between the bug fixes per revision in buggy cloned methods and the bug fixes per revision in buggy non-cloned methods. The null hypothesis can be rejected if the value of the difference is greater than zero with a p-value less than 0.05. We are then confident the buggy cloned methods have greater development effort for bug fixes than buggy non-cloned methods for the application under study.

5.2.3 Results

1	**2**	**3**	**4**
App	$\mu(BugFixPerRev\_BC) - \mu(BugFixPerRev\_BNC)$	$\frac{\mu(BugFixPerRev\_BC) - \mu(BugFixPerRev\_BNC)}{\mu(BugFixPerRev\_BC)}$	P-Value
1	**0.07**	**0.20**	**0.01**
2	**0.14**	**0.32**	**0.00**
3	**0.07**	**0.21**	**0.02**
4	**0.13**	**0.35**	**0.00**
5	0.01	0.03	0.41
6	**0.18**	**0.36**	**0.00**

Table 5-3 – Comparison of bug fixes per revision for buggy cloned and non-cloned methods

From Table 5-3, 5 out of the 6 applications studied are statistically significant with p-values under 0.05 and the buggy cloned methods have greater bug fix effort than the buggy non-cloned methods. We compare the statistically significant applications with the non-statistically significant application to determine the difference to explain the result. Applications 1, 2, 3, 4, and 6 are statistically significant and have buggy cloned methods that duplicate code from relatively new methods. Application 5 does not have a significant difference in the bug fix development effort because 18% of its buggy cloned methods duplicate code from older and more tested code (i.e. $\geq$ 6 changes). Less bug fix development effort is needed for methods cloned from the already tested and proven code. As a result, the buggy cloned and non-cloned methods for the application are similar in the level of bug fix effort, making the data verification result not statistically significant. Given that 5 of the 6 applications support the hypothesis and are statistically significant, we conclude that H_2 holds.

Due to the majority of the applications that support the hypothesis, we continue our investigation of the buggy cloned and non-cloned methods. For the remainder of the hypotheses in this chapter, we investigate the reasons to further explain and improve the bug fix effort of buggy cloned methods.

5.3 H_3 – Complexity of buggy cloned methods is higher than complexity of buggy non-cloned methods

To explain the greater bug fix effort for buggy cloned methods, we compare the code complexity of buggy cloned methods to buggy non-cloned methods. Research has shown a greater code complexity can lead to more bugs in software [68], [91]. We build upon existing research by comparing the code complexity of cloned and non-cloned methods separately. In addition, we investigate how cloned and non-cloned methods change in code complexity as the methods evolve.

5.3.1 Definition of Metrics

To investigate code complexity, we use a well-known complexity metric: cyclomatic complexity. Cyclomatic complexity is a metric that determines the complexity of the conditional statements in a method [83]. We show how to calculate cyclomatic complexity in Equation (5-2).

$$Cyclomatic = \#Decisions + 1 \tag{5-2}$$

The #Decisions is the number of conditional statements in a program that change the way the program executes.

Method Revision #1

```
int M1(int [] vals, int
numVals){
int increment = 2;
int sum = 0;
for (int i=0;i<numVals;i++)
{
     vals[i]=vals[i]+increment;
     sum=sum+vals[i];
}
return sum;
}

Number of decisions = 1
Cyclomatic complexity = 1+1 = 2
```

Method Revision #2

```
int M1(int [] vals, int
numVals){
int increment = 2;
int sum = 0;
int someMore = 4;
for (int i=0;i<numVals;i++)
{
     vals[i]=vals[i]+increment;
     sum=sum+vals[i];
}
sum=sum+someMore;
for (int i=0;i<numVals;i++)
{
     sum=sum*sum;
}
return sum;
}

Number of decisions = 2
Cyclomatic complexity = 2+1 = 3
```

Method Revision #3

```
int M1(int [] vals, int
numVals){
int increment = 2;
int sum = 0;
int addMore = 4;
for (int i=0;i<numVals;i++)
{
     vals[i]=vals[i]+increment;
     sum=sum+vals[i];
}
sum=sum+addMore;
for (int i=0;i<numVals;i++)
{
     sum=sum*sum;
}
if(sum<0)
{
sum=0;
}
else if (sum>10000000)
{
sum = 100;
}
return sum;
}

Number of decisions = 4
Cyclomatic complexity = 4+1 = 5
```

Method Revision #4

```
int M1(int [] vals, int
numVals){
int increment = 2;
int sum = 0;
int addMore = 4;
for (int i=0;i<numVals;i++)
{
     vals[i]=vals[i]+increment;
     sum=sum+vals[i];
}
sum=sum+addMore;
for (int i=0;i<numVals;i++)
{
     sum=sum*sum;
}
if(sum<0)
{
sum=0;
}
return sum;
}

Number of decisions = 3
Cyclomatic complexity = 3+1 = 4
```

Figure 5-8 – Sample code and calculations for cyclomatic complexity

In Figure 5-8, all the decisions in the code have been highlighted and the calculations for cyclomatic complexity are shown for each method revision. The values for cyclomatic complexity for the 4 method revisions shown are respectively 2, 3, 5, 4. To take into account the entire lifetime of a method and determine if the complexity changes as the methods evolve, we test the complexity of each method at three different stages of the lifetime of a method: the initial, median, and final revisions. For the example in Figure 5-8, the initial revision of the method refers to method revision 1. The cyclomatic complexity of method revision 1 is 2. The median revision of the method takes into account method revisions 2 and 3. We calculate the median cyclomatic complexity of method revisions 2 and 3 as 4. The final revision of the method takes into account method revision 4. The cyclomatic complexity of the final version of the method is 4. We can determine the cyclomatic complexity increased from the initial version to its median version by 2 and remained stable from its median version to its final version. By comparing the cyclomatic complexities of the methods from the initial to median, and median to final versions, we can determine the evolution of complexity in cloned and non-cloned methods.

5.3.2 Testing of H_3

For the initial, median, and final revisions, we compare the cyclomatic complexity in buggy cloned and non-cloned methods.We test if can we reject H_O with the difference in population means between the cyclomatic complexity in buggy cloned methods and the cyclomatic complexity in buggy non-cloned methods. If the difference is above zero and the data values are statistically significant, we can accept H_A. The test hypotheses in mathematical format are as follows:

$$H_O: \mu(Cyclomatic\_BC) - \mu(Cyclomatic\_BNC) \leq 0$$

$$H_A: \mu(Cyclomatic\_BC) - \mu(Cyclomatic\_BNC) > 0$$

5.3.3 Results

1	2	3	4
App	$\mu(Cyclomatic\_iniital\_BC) - \mu(Cyclomatic\_initial\_BNC)$	$\frac{\mu(Cyclomatic\_iniital\_BC) - \mu(Cyclomatic\_initial\_BNC)}{\mu(Cyclomatic\_iniital\_BC)}$	P-Value
1	**1.59**	**0.33**	**0.00**
2	**1.88**	**0.40**	**0.00**
3	**0.87**	**0.24**	**0.00**
4	**2.43**	**0.36**	**0.02**
5	**5.34**	**0.51**	**0.00**
6	-0.31	-0.12	0.75

Table 5-4 – Comparison of initial cyclomatic complexity of buggy cloned and non-cloned methods

1	2	3	4
App	$\mu(Cyclomatic\_median\_BC) - \mu(Cyclomatic\_median\_BNC)$	$\frac{\mu(Cyclomatic\_median\_BC) - \mu(Cyclomatic\_median\_BNC)}{\mu(Cyclomatic\_median\_BC)}$	P-Value
1	**1.48**	**0.27**	**0.00**
2	**2.26**	**0.42**	**0.00**
3	**0.70**	**0.19**	**0.00**
4	1.17	0.19	0.10
5	**7.09**	**0.57**	**0.00**
6	-0.11	-0.03	0.73

Table 5-5 – Comparison of median cyclomatic complexity of buggy cloned and non-cloned methods

1	2	3	4
App	$\mu(Cyclomatic\_final\_BC) - \mu(Cyclomatic\_final\_BNC)$	$\frac{\mu(Cyclomatic\_final\_BC) - \mu(Cyclomatic\_final\_BNC)}{\mu(Cyclomatic\_final\_BC)}$	P-Value
1	**2.14**	**0.34**	**0.00**
2	**2.65**	**0.44**	**0.00**
3	**0.58**	**0.14**	**0.01**
4	1.13	0.18	0.15
5	**10.21**	**0.64**	**0.00**
6	-0.11	-0.03	0.74

Table 5-6 – Comparison of final cyclomatic complexity of buggy cloned and non-cloned methods

From Table 5-4, Table 5-5, Table 5-6, 4 applications are consistently statistically significant and have greater complexity in cloned methods than non-cloned methods, supporting the hypothesis that greater complexity leads to greater bug fixes. We can explain the greater complexity in cloned methods. From manual review of the code, we note that the duplication of code for new features or purposes in most cases involves the addition of conditional complexity in order to add the functionality of the cloned code to the method. Of the 6 applications, only one application, Application 6 shows a greater complexity in buggy non-cloned methods and it lacks statistical significance with an average confidence level of 26%. From inspection of the methods in Application 6, 77% of the buggy cloned methods have a low initial, median, and final cyclomatic complexity of less than or equal to 3. This lack of a large number of buggy cloned methods with high complexity accounts for the comparison of complexity to be statistically insignificant.

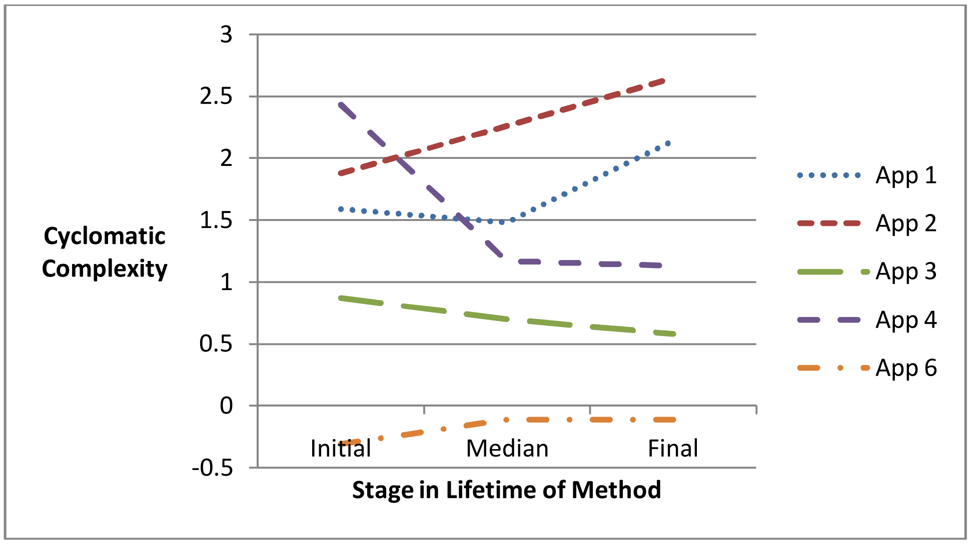

Figure 5-9 – Differences in cyclomatic complexity for Applications 1, 2, 3, 4, and 6 comparing buggy cloned and non-cloned methods

In Figure 5-9, the initial, median, and final values from the second columns from Table 5-4, Table 5-5, and Table 5-6 are shown. Due to the larger scale in values of cyclomatic complexity in Application 5, to keep the changes in other applications easily readable, we omit the application from the graph. Three of the statistically significant applications have an increasing trend from initial to final cyclomatic complexity. The increases in complexity for Applications 1, 2, and 5 from the initial revisions to final revisions, are 0.55, 0.77, 4.87 respectively. An increased value of 1 in cyclomatic complexity refers to one additional conditional statement added from initial to median revisions, and again from median to final revisions. We reviewed the code and the additional conditional statements make sense in complex methods for performing bug fixes. A developer can uncover new special cases when fixing code and not want to disrupt how the current code functions. As a result, conditional statements are added to handle special cases or exceptions while not disrupting the functionality of the code. Applications 3 and 4 have decreasing complexities from the initial to final versions of 0.3 and 1.3. The decreasing complexities represent removed conditional statements from special cases that are unnecessary or potentially incorrect program behavior.

Given that 5 out of the 6 applications support the hypothesis and have a confidence of at least 85%, we conclude that H_3 holds. In addition, we acknowledge that extra conditional statements can be added or removed as the cloned methods evolve during development. We provide a deeper analysis on cloned methods in the next chapter to determine the detrimental properties of complexity in cloned methods.

5.4 H_4 – The number of developer switches in buggy cloned methods is lower relative to buggy non-cloned methods

The definition of a developer switch is that the developer that updates a method is different from the developer of the previous method revision. The number of developer switches detects cases where the code is either new to the developer or the code within the method is different from how the developer left the code from his or her last method revision. This change in the code requires that the functionality of the code be (re-)understood before any modifications can be made. If there are errors or carelessness in (re-)understanding the code, new bugs can be introduced into the code when modifications are made. In addition, developers new to cloned methods may be unaware of the other cloned methods in the clone class. A lack of knowledge of other cloned methods can lead to a lack of co-change. We postulate a greater number of developer switches can cause a need for more bug fixes in cloned and non-cloned methods.

5.4.1 Definition of Metrics

We present Table 5-7 below to show the detection of the number of developer switches in a cloned or non-cloned method.

Method Revision	Developer
1	Dorothy
2	Brian
3	Jacob
4	Dorothy
5	John
6	John
7	Brian
8	John

Table 5-7 – Sample method for developer example

From Table 5-7, method revisions 2, 3, 4, 5, 7, and 8 are developer switches because the developer from the previous method revision is not the same as the developer of the current method revision under investigation. For example, in method revision 2, Brian performs the modification and in method revision 1, the previous method revision, Dorothy performed the method revision. Since the developer is different from method revision 1 to method revision 2, we say a developer switch occurred in method revision 2. Similarly for method revision 3, Jacob performs the update while Dorothy performed the previous method revision (i.e. method revision 2). Therefore, we say a developer switch occurred in method revision 3. For the example in Table 5-7, 5 developer switches occur in method revisions 2, 3, 5, 7, and 8.

To normalize the number of developer switches in relation to the number of method revisions in the method, we calculate the number of developer switches per revision as shown in Equation (5-3). The #DeveloperSwitches represents the number of developer switches for the method under study. The #Revisions represents the number of method revisions for the method.

$$DevSwitchPerRev = \frac{\#DeveloperSwitches}{\#Revisions - 1} \tag{5-3}$$

We subtract 1 from the denominator because a developer switch cannot occur in the first method revision to a method. As a result, the value of the number of developer switches per revision ranges from [0,1].

5.4.2 Testing of H_4

A statistical non-parametric test is used to compare the developer switches per revision in buggy cloned and non-cloned methods as shown in the following test hypothesis:

$$H_O: \mu(DevSwitchPerRev\_BC) - \mu(DevSwitchPerRev\_BNC) \geq 0$$

$$H_A: \mu(DevSwitchPerRev\_BC) - \mu(DevSwitchPerRev\_BNC) < 0$$

The difference in population means between the developer switches per revision in buggy cloned methods and the developer switches per revision in buggy non-cloned methods is $\mu(DevSwitchPerRev\_BC) - \mu(DevSwitchPerRev\_BNC)$. If the p-value is greater or equal to 0.05, the difference in means is not statistically significant. If H_O is rejected with a high probability (i.e. p-value < 0.05), we are confident the buggy cloned methods have fewer developer switches per revision than buggy non-cloned methods for the application under study.

5.4.3 Results

1	**2**	**3**	**4**
App	$\mu(DevSwitchPerRev\_BC) - \mu(DevSwitchPerRev\_BNC)$	$\frac{\mu(DevSwitchPerRev\_BC) - \mu(DevSwitchPerRev\_BNC)}{\mu(DevSwitchPerRev\_BC)}$	P-Value
1	**-0.06**	**-0.10**	**0.01**
2	**-0.19**	**-0.33**	**0.00**
3	-0.05	-0.12	0.47
4	-0.13	-0.22	0.10
5	**-0.13**	**-0.22**	**0.04**
6	-0.17	-0.35	0.06

Table 5-8 – Comparison of developer switches per revision for buggy cloned and non-cloned methods

In Table 5-8, of the 6 applications studied, Applications 1, 2, and 5 are statistically significant and have more author switches per revision in buggy non-cloned methods. Applications 4 and 6 have confidence levels close to the level required for statistical significance of at least 90% and have more author switches per revision in buggy non-cloned methods as well. Application 3 is not statistically significant and has a confidence level of 53% (i.e. 1-p). Upon review of the number of developers per revision in the 6 applications studied, from Table A-1 in Appendix A, Application 3 has the fewest developers per revision in cloned and non-cloned methods. For Application 3, in non-cloned methods the impact is especially pronounced with 0.58 developers per revision, in comparison with 0.71 developers per revision in the application with the next highest value. With respect to the number of revisions in each method from Application 3, fewer developers worked on each method in the application. Therefore, fewer developer switches could occur, resulting in the number of developer switches per revision for buggy cloned and non-cloned methods to not be significantly different. Given that 5 out of the 6 applications studied support the hypothesis and have a confidence level of at least 90%, we conclude that H_4 holds.

To explain the effect of greater developer switches per revision on the development effort required for bug fixes in buggy cloned and non-cloned methods, we inspected the methods to note the effect of developer switches. From our manual review of the code, developer switches can introduce different coding styles among revisions or some developers remove and redevelop code to produce new features or solve problems during development. In the case of non-cloned methods, since no cloned method counterpart exists, the new coding styles or redevelopment is relatively safe. The developer switches for cloned methods are more problematic. New coding styles can be incompatible with other cloned methods in the clone class if parts of the cloned method have been specialized. As a result, new coding styles implemented in the cloned methods with a lack of co-change result in inconsistent cloned fragments of code that can later require a bug fix. As a result, developer switches can be considered a safe practice for non-cloned methods. However, developer switches in cloned methods should raise caution because the updates without co-change can result in greater bugs in the application.

5.5 H_5 – In the early development stage, the bug fix effort is greater in buggy cloned methods than buggy non-cloned methods

To determine if buggy cloned methods require greater bug fix effort in early development, we compare the bug fix effort of buggy cloned and non-cloned methods. If we can determine that buggy cloned methods require greater bug fix effort in early development, we can suggest techniques to improve clone maintenance.

For our analysis, we have the problem of choosing an appropriate amount of time that represents early development. After review of the bug fix data in our applications, we decide to use the first 6 months to represent early development. First, a sufficient number of bug fixes occur in the applications in the first 6 months. Second, 3 months does not have a sufficient number of bug fixes. Therefore, we use 2 fiscal quarters to represent early development of software for the 6 applications under study.

5.5.1 Definition of Metrics

We use the metric of the bug fixes per revision (5-1) from H_1 to represent the bug fix effort for our analysis.

5.5.2 Testing of H_5

We test the null hypothesis with the difference in population means between the bug fixes per revision in the first 6 months in buggy cloned methods and the bug fixes per revision in the first 6 months in buggy non-cloned methods. If the null hypothesis H_O can be rejected (i.e. p-value<0.05), we are then confident the buggy cloned methods have greater bug fix effort in the first 6 months than buggy non-cloned methods for the application under study. The null and alternative hypotheses are as follows:

$$H_O: \mu(BugFixPerRev\_First6Mths\_BC) - \mu(BugFixPerRev\_First6Mths\_BNC) \leq 0$$

$$H_A: \mu(BugFixPerRev\_First6Mths\_BC) - \mu(BugFixPerRev\_First6Mths\_BNC) > 0$$

5.5.3 Results

1	**2**	**3**	**4**
App	$\mu(BugFixPerRev\_First6Mths\_BC) - \mu(BugFixPerRev\_First6Mths\_BNC)$	$\frac{\mu(BugFixPerRev\_First6Mths\_BC) - \mu(BugFixPerRev\_First6Mths\_BNC)}{\mu(BugFixPerRev\_First6Mths\_BC)}$	P-Value
1	**0.16**	**0.55**	**0.00**
2	**0.16**	**0.81**	**0.00**
3	**0.21**	**0.56**	**0.00**
4	0.01	0.14	0.49
5	0.09	0.68	0.07
6	0.13	0.32	0.10

Table 5-9 –Comparison of bug fixes per revision for first six months of buggy cloned and non-cloned methods

Of the 6 applications analyzed in Table 5-9, for buggy cloned and non-cloned methods 3 are statistically significant at a confidence of 99% and have a higher bug fixes per revision in cloned methods. Applications 5 and 6 have confidence levels of at least 90% and have more bug fixes per revision in buggy cloned methods.

From manual review of Applications 1, 2, 3, 5, and 6, the cloned methods have more bug fixes in the first 6 months due to a lack of co-change and cases of unchecked pre-conditions that can result in software crashes. Application 4 has a confidence level of 51% and suggests that cloned methods are approximately the same level of bug fix effort as non-cloned methods in the first 6 months. From inspection of the methods in Application 4, only 18% and 20% of its buggy cloned and non-cloned methods respectfully have bug fixes in the first 6 months. We can explain the lack of bug fixes in the first 6 months in cloned methods because a great deal of the content in the conditional statements abstracted into method calls. As a result, a lack of co-change in the cloned methods was less problematic because most updates are done in the methods called as opposed to the cloned methods. The lack of bug fixes in the first 6 months in the non-cloned methods of Application 4 can be attributed to the fact the application deals with data security and has greater code review during development.

Given that 5 out of 6 applications support the hypothesis with a confidence level of at least 90%, we conclude that H_5 holds.

5.6 H_6 – In the later development stage, the bug fix effort is less in buggy cloned methods than buggy non-cloned methods

To determine if buggy cloned methods require less bug fix effort in later development (i.e. after 6 months), we compare the bug fix effort of buggy cloned and non-cloned methods. If the hypothesis holds, we can suggest that development teams focus their attention on buggy non-cloned methods during later development.

5.6.1 Definition of Metrics

The metric used for the analysis of H_6 is the bug fixes per revision (5-1).

5.6.2 Testing of H_6

The bug fixes per revision after the first 6 months in buggy cloned and non-cloned methods are compared to test the sixth hypothesis in the thesis. We present the null and alternative hypotheses as follows:

$$H_O: \mu(BugFixPerRev\_After6Mths\_BC) - \mu(BugFixPerRev\_After6Mths\_BNC) \geq 0$$

$$H_A: \mu(BugFixPerRev\_After6Mths\_BC) - \mu(BugFixPerRev\_After6Mths\_BNC) < 0$$

$\mu(BugFixPerRev\_After6Mths\_BC) - \mu(BugFixPerRev\_After6Mths\_BNC)$ is the difference in population means between the bug fixes per revision after the first 6 months in buggy cloned methods and the bug fixes per revision after the first 6 months in buggy non-cloned methods. If we can reject the H_O, we are confident the buggy cloned methods have less bug fix effort after the first 6 months than buggy non-cloned methods for the application under study.

5.6.3 Results

1	**2**	**3**	**4**
App	$\mu(BugFixPerRev\_After6Mths\_BC) - \mu(BugFixPerRev\_After6Mths\_BNC)$	$\frac{\mu(BugFixPerRev\_After6Mths\_BC) - \mu(BugFixPerRev\_After6Mths\_BNC)}{\mu(BugFixPerRev\_After6Mths\_BC)}$	P-Value
1	**-0.27**	**-0.93**	**0.00**
2	**-0.17**	**-0.30**	**0.00**
3	**-0.20**	**-0.77**	**0.02**
4	-0.01	-0.01	0.67
5	**-0.28**	**-1.20**	**0.01**
6	0.04	0.12	0.47

Table 5-10 –Bug fixes per revision for the later development of buggy cloned and non-cloned methods

Of the 6 applications analyzed in Table 5-10, Applications 1, 2, 3, and 5 have a statistically significant difference between buggy cloned and non-cloned methods and are buggier in non-cloned methods after 6 months of development. In Chapter 6, we find that cloned methods with frequent co-change in the first 6 months co-change frequently in later development. For cloned methods that infrequently co-change, the clone methods tend to have a more independent evolution in later development due to greater specialization in the methods. This makes a lack of co-change not result in as many bugs because the clone coverage (i.e. percent of the method that is cloned code) is reduced by the specializations in the method. Applications 4 and 6 are not statistically significant and have confidence levels of 33% and 53%. Because the development of Application 4 had greater code review and planning due to the security nature of the application, only 10% of the bug fixes in cloned methods for the application were performed in early development of the application (i.e. first 6 months). The remainder of the bug fixes occurred in later development after 1 year. Application 6 is one year old and relatively new compared to the other applications in the study. The other five applications are all over 3 years old. This accounts for the lack of statistical significance of its bug fixes in later development. Given that 4 out of the 6 applications studied are statistically significant and support the hypothesis, we conclude that H_6 holds.

In conclusion, with the knowledge of the results from H_5 that cloned methods can be buggier in the first 6 months and 4 out of 6 statistically significant applications supporting our hypothesis, we can suggest that greater time be given to developers while working with buggy cloned methods in the first 6 months of development. Also, less development effort need be expended on buggy cloned methods in later development.

5.7 H_7 – Buggy cloned methods have greater bug fix effort in early development relative to later development than buggy non-cloned methods

The previous hypotheses separately compare the bug fix effort in early development (H_5) or later development (H_6) of buggy cloned and non-cloned methods. From analysis of the hypotheses, H_5 shows that buggy cloned methods have greater bug fix effort in early development than buggy non-cloned methods. H_6 shows that buggy cloned methods have less bug fix effort in later development than buggy non-cloned methods. However, H_5 and H_6 have not proven that buggy cloned methods reduce in bug fix effort in later development or that buggy non-cloned methods have less bug fix effort in early development. We want to determine if bug fix effort remains stable or changes over time for buggy cloned methods and similarly for buggy non-cloned methods.

5.7.1 Definition of Metrics

To investigate the bug fix effort between early and later development of buggy cloned and non-cloned methods, we calculate the ratio of the bug fixes per revision in the first 6 months to after 6 months as shown in Equation (5-4).

$$RatioBugFixPerRev = \frac{BugFixPerRev\_First6Mths}{BugFixPerRev\_After6Mths} \tag{5-4}$$

The $BugFixPerRev\_First6Mths$ represents the bug fixes per revision in the first 6 months. The $BugFixPerRev\_After6Mths$ represents the bug fixes per revision after 6 months. We avoid division by zero with the removal of methods with no bug fixes after 6 months from analysis.

5.7.2 Testing of H_7

We formulate the following test hypothesis:

$$H_O: \mu(RatioBugFixPerRev\_BC) - \mu(RatioBugFixPerRev\_BNC) \leq 0$$

$$H_A: \mu(RatioBugFixPerRev\_BC) - \mu(RatioBugFixPerRev\_BNC) > 0$$

The difference of the ratios of the bug fixes per revision for buggy cloned and non-cloned methods is taken to test the null hypothesis. If H_O is rejected, we are confident the buggy cloned methods have greater bug fix effort in early development relative to later development than buggy non-cloned methods for the application under study.

5.7.3 Results

1	2	3
App	$\mu(RatioBugFixPerRev\_BC)$	$\mu(RatioBugFixPerRev\_BNC)$
1	0.78	0.16
2	0.20	0.03
3	0.94	0.20
4	0.00	0.01
5	0.00	0.01
6	0.22	0.14

Table 5-11 – Average ratios of bug fixes per revision for corrective maintenance effort of buggy cloned and non-cloned methods

1	2	3	4
App	$\mu(RatioBugFixPerRev\_BC) - \mu(RatioBugFixPerRev\_BNC)$	$\frac{\mu(RatioBugFixPerRev\_BC) - \mu(RatioBugFixPerRev\_BNC)}{\mu(RatioBugFixPerRev\_BC)}$	P-Value
1	**0.62**	**0.80**	**0.00**
2	**0.16**	**0.83**	**0.00**
3	**0.75**	**0.79**	**0.00**
4	-0.01	-0.12	0.36
5	-0.01	-0.07	0.51
6	0.08	0.37	0.78

Table 5-12 – Comparison of ratios of bug fixes per revision of buggy cloned and non-cloned methods

From Table 5-11, the average ratio of bug fixes per revision for buggy cloned methods represents the ratio of the bug fixes per revision of buggy cloned methods before and after 6 months of development in buggy cloned methods. A value less than 1 denotes the methods in the application have more corrective maintenance effort in later development relative to the effort of other activities such as developing new features, optimizing the code, etc. Conversely, a value over 1 denotes the methods in the application have more corrective maintenance effort in the first 6 months relative to other activities. The same analysis of the values can be applied to the average ratio of bug fixes per revision for buggy non-cloned methods representing the ratio of the bug fixes per revision of buggy non-cloned methods before and after 6 months of development. For Table 5-12, the second and third columns present values to compare the ratio of bug fixes per revision in buggy cloned methods to buggy non-cloned methods. A positive value supports our hypothesis that buggy cloned methods have greater bug fixes in early development relative to later development than buggy non-cloned methods. A negative value contradicts our hypothesis and we accept the null hypothesis for the application. The sixth column presents the p-values for data verification. If the value is under 0.05, the ratio of bug fixes per revision in early and later development is statistically significant for buggy cloned and non-cloned methods. We highlight the values of the statistically significant applications in bold.

We now discuss the differences in bug fix effort for cloned and non-cloned methods in applications that are statistically significant. From Table 5-12, of the 6 applications studied, Applications 1, 2, and 3 are statistically significant in a comparison of buggy cloned and non-cloned methods with a confidence over 99%. We recognize the reason why this occurs in Table 5-11. For the 3 statistically significant applications, referring to the data in the second column, the cloned methods have all six values less than 1. As explained in the earlier paragraph, this means cloned methods use a greater percent of bug fix effort in later development. This makes sense because the features of the method have already been programmed and corrective maintenance is needed more frequently in later development to keep the cloned methods aligned with other code changes to the software system. For buggy non-cloned methods, the 3 applications that are statistically significant follow behavior similar to the buggy cloned methods with corrective maintenance as the primary activity of the methods after the first 6 months of development.

We now present the reasons why the other applications are not statistically significant in the analysis of this hypothesis. In our analysis, Applications 4, 5, and 6 are not statistically significant. Applications 4 and 5 have an average ratio of bug fixes per revision in the first 6 months to bug fixes per revision after 6 months of approximately 0 in both cloned and non-cloned methods. We attribute the values of the ratios of both cloned and non-cloned methods due to the methods as not having reoccurring coding problems that cause bugs. The methods either had bug fixes in early development (i.e. first 6 months) or later development (i.e. after 6 months). As we explained before, Application 4 is well-coded using method calls in the conditional statements of the cloned methods to reduce problems with co-change because updates to the methods called affect all cloned methods in the clone class. Application 5 uses cases of cloning tested and proven code as stated in H_2. Application 6 is not statistically significant and has cloned methods that are mainly reciprocal in nature (i.e. setActive() and setInactive()). These cloned methods are more intuitive to co-change and missed fewer co-changes which resulted in fewer bugs.

Given that 3 of the applications support the hypothesis and are statistically significant, we conclude that H_7 holds. We notice that cloned methods follow the trend of non-cloned methods. Both types of methods require greater bug fix effort in later development than early development to properly work with new features or development in the system. From H_5, we established that cloned methods have more bug fixes in the first 6 months in comparison with non-cloned methods. Since the cloned methods have an even greater amount of bug fix effort in later development in comparison with the first 6 months, we suggest refactoring the cloned methods that require more bug fixes in the first 6 months in comparison with non-cloned methods.

5.8 H_8 – The percent of developers performing bug fixes on buggy cloned methods is greater than for buggy non-cloned methods

When a method is new to a developer, before modifying the code in the method, she or he must first understand how the current code functions. If the developer misunderstands the current code, this can lead to programming errors when modifying the code. Cloned code can be harder to understand [80] and this raises the likelihood of a programming error when modifying code. In addition, a new developer (i.e. never modified the method before) working on cloned code may not know where the other cloned methods exist. She or he can miss co-changing the other cloned methods and create bugs in software. We want to determine if a greater number of developers performing bug fixes is problematic and causes further bugs in software.

5.8.1 Definition of Metrics

We calculate the percent of developers performing bug fixes as shown in Equation (5-5).

$$\%DevBugFix = \frac{\#DevelopersBugFix}{\#Developers} \tag{5-5}$$

The number of developers who have performed a bug fix on the method under study is represented by #DevelopersBugFix. The number of developers who have made modifications to the method is represented by #Developers.

5.8.2 Testing of H_8

We use a statistical non-parametric test to compare the percent of developers who perform bug fixes in buggy cloned and non-cloned methods. We formulate the following test hypothesis:

$$H_O: \mu(\%DevBugFix\_BC) - \mu(\%DevBugFix\_BNC) \leq 0$$

$$H_A: \mu(\%DevBugFix\_BC) - \mu(\%DevBugFix\_BNC) > 0$$

To test H_O, we calculate the difference in population means between the percent of developers who perform bug fixes for buggy cloned and non-cloned methods. If H_O is rejected, we are confident the buggy cloned methods have a greater percent of developers who perform bug fixes than buggy non-cloned methods for the application under study.

5.8.3 Results

1	2	3	4
App	$\mu(\%DevBugFix\_BC) - \mu(\%DevBugFix\_BNC)$	$\frac{\mu(\%DevBugFix\_BC) - \mu(\%DevBugFix\_BNC)}{\mu(\%DevBugFix\_BC)}$	P-Value
1	**-0.10**	**-0.19**	**0.00**
2	0.04	0.06	0.16
3	0.01	0.01	0.50
4	**0.11**	**0.17**	**0.04**
5	**-0.10**	**-0.33**	**0.02**
6	**0.21**	**0.26**	**0.00**

Table 5-13 – Comparison of the percent of developers who have performed bug fixes for buggy cloned and non-cloned methods

From Table 5-13, 4 out of the 6 applications studied are statistically significant with p-values under 0.05. Applications 4 and 6 are statistically significant and support our hypothesis that the percent of developers performing bug fixes on buggy cloned methods is greater than the percent of developers performing bug fixes on buggy non-cloned methods. Developers with different coding styles or experience can solve coding problems differently [8], thereby potentially making the cloned code less familiar and making coding new features or non-corrective maintenance activities more difficult for other developers working on the method. Applications 1 and 5 have results that are contrary to our hypothesis, suggesting that there is a greater number of developers performing bug fixes in buggy non-cloned methods.

To explain the results from Applications 1 and 5, we calculate the difference of the averages of two metrics to understand the frequency new developers perform revisions and the frequency new developers perform bug fixes in buggy cloned and non-cloned methods. For the first metric, we use the number of developers per revision (5-6) to represent the rate that new developers are introduced to the method to perform revisions. For the second metric, we use the number of developers who have performed bug fixes per revision (5-7) to represent the rate new developers are introduced to the method to perform bug fixes.

$$DevPerRev = \frac{\#Developers}{\#Revisions} \tag{5-6}$$

$$DevBugFixPerRev = \frac{\#DevelopersBugFix}{\#Revisions} \tag{5-7}$$

The #Developers represents the number of developers that have updated the method under investigation. The #Revisions represents the number of method revisions for the method under investigation. The #DevelopersBugFix represents the number of developers who performed a bug fix on the method under investigation.

1	2	3	4
App	$\mu(DevPerRev\_BC) - \mu(DevPerRev\_BNC)$	$\frac{\mu(DevPerRev\_BC) - \mu(DevPerRev\_BNC)}{\mu(DevPerRev\_BC)}$	P-Value
1	**-0.12**	**-0.19**	**0.00**
2	**-0.17**	**-0.27**	**0.00**
3	-0.08	-0.15	0.11
4	-0.11	-0.17	0.10
5	**-0.17**	**-0.32**	**0.01**
6	**-0.11**	**-0.18**	**0.03**

Table 5-14 – Comparison of the developers per revision for buggy cloned and non-cloned methods

1	2	3	4
App	$\mu(DevBugFixesPerRev\_BC) - \mu(DevBugFixesPerRev\_BNC)$	$\frac{\mu(DevBugFixesPerRev\_BC) - \mu(DevBugFixesPerRev\_BNC)}{\mu(DevBugFixesPerRev\_BC)}$	P-Value
1	**-0.02**	**-0.14**	**0.00**
2	-0.01	-0.03	0.10
3	-0.01	-0.09	0.24
4	0.00	0.02	0.31
5	**-0.04**	**-0.37**	**0.04**
6	0.02	0.10	0.24

Table 5-15 – Comparison of the number of developers who performed bug fixes per revision for buggy cloned and non-cloned methods

We use Equation (5-6) and Equation (5-7) in comparisons of buggy cloned and non-cloned methods to produce Table 5-14 and Table 5-15 respectively. The second column in Table 5-14 presents the difference between the average developers per revision in buggy cloned methods and the average developers per revision in buggy non-cloned methods. In Applications 1, 2, 5, and 6, the result is statistically significant and buggy non-cloned methods have greater developers per revision. The contrary result from Applications 1 and 5 from our original hypothesis could be due to a larger number of developers per revision performing code modifications in buggy non-cloned methods.

We also investigate the second column in Table 5-15 that represents the difference in average developers who have performed bug fixes per revision. The results are only statistically significant for Applications 1 and 5 with more developers performing bug fixes per revision in buggy non-cloned methods. The result is similar to our analysis in hypothesis H_4, where a greater number of developer switches made buggy non-cloned methods less buggy. However, rather than changing between developers, Applications 1 and 5 also have many new developers constantly working on the non-cloned methods. In Applications 1 and 5, more developers in non-cloned methods helped prevent bugs.

Given that 4 of the applications studied are statistically significant, but split with 2 of applications supporting the hypothesis and the other 2 applications contrary to the hypothesis, H_8 is rejected. We notice that the percent of developers performing bug fixes can have a positive or negative effect. The result of whether the percent of developers who perform bug fixes increases or decreases the bug fix effort is dependent upon the application.

5.9 Summary

	Hypothesis	% Applications Supporting Hypothesis	Status
H_1	Cloned methods have greater bug fix effort than non-cloned methods	67%	Holds
H_2	Buggy cloned methods have greater bug fix effort than buggy non-cloned methods	83%	Holds
H_3	Complexity of buggy cloned methods is higher than complexity of buggy non-cloned methods	67%	Holds
H_4	The number of developer switches is lower in buggy cloned methods relative to buggy non-cloned methods	50%	Holds
H_5	In the early development stage (first 6 months), the bug fix effort is greater in buggy cloned methods than buggy non-cloned methods	50%	Holds
H_6	In the later development stage, the bug fix effort is less in buggy cloned methods and buggy non-cloned methods	67%	Holds
H_7	Buggy cloned methods have greater bug fix effort in early development relative to later development than buggy non-cloned methods	50%	Holds
H_8	The percent of developers performing bug fixes on buggy cloned methods is greater than for buggy non-cloned methods	33%	Rejected

Table 5-16 – Status of hypotheses for analysis of bug fixes in cloned and non-cloned methods

5.9.1 Reasons Each Application Rejected a Hypothesis

Application 1 was rejected in only one hypothesis, H_8 due to a larger number of developers per revision performing code modifications in buggy non-cloned methods. Application 2 was also only rejected in H_8 from a lack of statistical significance at a confidence level of 84% in favor of the hypothesis. In hypotheses H_3 and H_8, Application 3 was not accepted due a deficiency in developers per revision in both cloned and non-cloned methods. Application 4 was rejected in hypotheses H_1, H_3, H_4, H_5, H_6, and H_7 because the application dealt with security and underwent greater code review. Application 5 was not accepted in hypotheses H_1, H_2, H_5, and H_7 because 18% of the cloned methods obtained duplicated code from older and more tested cloned code (i.e. ≥ 6 changes). Application 5 was also rejected in H_8 for the reason that a larger number of developers per revision performing code modifications in buggy non-cloned methods. In H_3 and H_4, Application 6 was rejected due to greater complexity in the non-cloned methods and the cloned methods to have a complexity less than or equal to 3 in 77% of cases. Application 6 lacked statistical significance in H_5, H_6, and H_7 due to the application as an implementation of a protocol that lends itself to methods that are reciprocal in nature (i.e. setActive() and setInactive()). In addition, Application 6 was only a year old and relatively less mature than the other applications studied.

5.9.2 Discussion of Hypotheses and Usage

Table 5-16 shows the percent of applications that support each hypothesis. A threshold of 50% is used to determine if the hypothesis holds. We recognize from H_1 and H_2 that cloned methods that do not copy from older and more tested code require greater bug fix effort than non-cloned methods. In our study, Application 5 had cases of older and more tested cloned code. We analyzed the past development of the methods that are copied from. The cloned methods that copied from more tested code had at least 6 changes before the code was duplicated. We suggest that cloned methods duplicated from relatively untested and younger code can be used as a warning sign of greater bug fix effort in the future, especially in the first 6 months (i.e. H_5 and H_6). Managers who determine the release schedules or assignment of customer support for applications can benefit because greater problems in the first 6 months of the cloned methods can be anticipated before the release of the product.

We notice a higher code complexity in general in cloned methods than non-cloned methods. Studies have shown the connection between complexity and bug fixes [91], [97]. However, to the best of our knowledge, we are the first to perform this analysis with cloned and non-cloned methods separately. Cloned methods have greater complexity and require greater bug fix effort (i.e. H_3). We can suggest that cloned methods that are duplicated from new code and that have high code complexity be considered for refactoring.

From the results of H_4, we found developer switches to be greater in non-cloned methods. Since the cloned methods have greater bug fix effort in general, we find developer switches to not be a significant factor in bug fix effort. In addition, the hypothesis concerning the percent of developers who perform bug fixes (i.e. H_8) was not found to be a significant factor in bug fix effort due to conflicting results amongst 4 statistically significant applications. The conflict in results may have occurred due to a larger number of developers per revision performing code modifications in buggy non-cloned methods for the 2 statistically significant applications with results opposing the hypothesis.

Chapter 6
Co-Change Analysis of Cloned Methods

We analyze co-change in cloned methods to determine the factors that help or hinder co-change in cloned methods. A similar work investigated the percent of a file that was cloned to predict co-change of cloned code [43]. Other works have determined that a relationship exists between software artifacts and developers [47], [52], [95] and that with more developers, extra communication is required to modify code quickly and without introducing bugs. In Table 5-1, we present the list of hypotheses for investigating the influences affecting co-change in buggy cloned methods. For each hypothesis, we compare buggy cloned methods with frequent co-change to buggy cloned methods with infrequent co-change. We quantify "co-change frequently" as 75% or more of the revisions in the method are co-changes. We first determine in H_9 if greater co-change in development (i.e. coding new features) leads to greater co-change of bug fixes. H_9 is the only hypothesis in this chapter that strictly uses the revisions during development in the calculation for grouping cloned methods as "co-changing frequently". The purpose is to separate the co-changes during development from the co-changes of bug fixes. For the next three hypotheses, we investigate reasons to explain the frequent co-change of the buggy cloned methods co-change frequently. In H_{10}, we determine if code complexity is a significant reason to explain the co-change behavior of buggy cloned methods with frequent co-change and buggy cloned methods with infrequent co-change. The intuition of the hypothesis is greater code complexity makes understanding the code more difficult and decrease co-change. In H_{11}, we analyze the effect of developer switches that have on frequent co-change of buggy cloned methods. We hypothesize greater developer switches can also hinder understanding of the code and decrease co-change. In H_{12}, we compare the names of methods in clone classes to determine if similar naming increases or decreases co-change. We hypothesize similar naming can act as a cue to developers to co-change the methods. We then focus our analysis to determine the distribution of development effort to improve co-change of buggy cloned methods in H_{13}, H_{14}, and H_{15}. In H_{13}, we compare the co-change development effort in the first 6 months to determine if greater co-change effort is spent in early development for buggy cloned methods with frequent co-change compared with buggy cloned methods with infrequent co-change. In H_{14}, we focus on the buggy cloned methods with infrequent co-change to determine if co-change increases in later development due to greater developer familiarity with the cloned methods. In H_{15}, we investigate the percent of developers that co-change the buggy cloned methods to determine if greater awareness of other cloned methods leads to greater co-change.

	Hypothesis
H_9	Buggy cloned methods with frequent co-change in development have greater co-change in bug fixes than buggy cloned methods with infrequent co-change in development
H_{10}	The complexity of buggy cloned methods with frequent co-change is lower in comparison with the complexity of buggy cloned methods with infrequent co-change
H_{11}	Buggy cloned methods with frequent co-change have fewer developer switches than buggy cloned methods with infrequent co-change
H_{12}	The method names in buggy cloned methods with frequent co-change are more similar than the method names of buggy cloned methods with infrequent co-change
H_{13}	In early development, buggy cloned methods with frequent co-change have greater co-change than buggy cloned methods with infrequent co-change
H_{14}	Buggy cloned methods with infrequent co-change in early development improve with co-change in later development than buggy cloned methods with infrequent co-change
H_{15}	Buggy cloned methods that co-change frequently have a greater percent of its developers co-changing methods than buggy cloned methods with infrequent co-change

Table 6-1 – List of hypotheses for analysis of co-change of cloned methods

6.1 H_9 – Buggy cloned methods with frequent co-change in development have greater co-change in bug fixes than buggy cloned methods with infrequent co-change in development

A lack of co-change in bug fixes constitutes a potential problem in cloned methods. The bug is fixed in one cloned method, but because the developer does not update the other cloned methods in the clone class, the same bug can appear later in the cloned methods not co-changed in the previous bug fix. To better understand cases of co-change in bug fixes, we investigate if co-change in development (i.e. development of new features, optimizations, etc.) leads to greater co-change in bug fixes.

6.1.1 Definition of Metrics

We introduce two metrics to test the hypothesis: co-change per revision and percent bug fixes as co-change. The co-change per revision is shown in Equation (6-1).

$$CoChangePerRevision = \frac{\#CoChanges}{\#Revisions} \tag{6-1}$$

The #CoChanges represents the number of co-changes for the method under investigation. The #Revisions represents the number of method revisions for the method.

To study the co-change of bug fixes in buggy cloned methods with frequent co-change and those with infrequent co-change, we present the percent bug fixes as co-change in Equation (6-2).

$$\%BugFixCoChg = \frac{\#BugFixesCoChange}{\#BugFixes} \tag{6-2}$$

The #BugFixesCoChange refers to the number of bug fixes co-changed for the method under investigation. The #BugFixes represents the number of bug fixes for the method.

6.1.2 Testing of H_9

We use the co-change per revision to divide the buggy cloned methods into two data sets for comparison. Buggy cloned methods with co-change per revision greater or equal to 75% during development are considered to co-change frequently. Buggy cloned methods with co-change per revision less than 75% during development are considered to co-change infrequently.

We use the metric of the percent bug fixes as co-change for our analysis and formulate the following test hypothesis:

$$H_O: \mu(\%BugFixCoChg\_BCF) - \mu(\%BugFixCoChg\_BCI) \leq 0$$

$$H_A: \mu(\%BugFixCoChg\_BCF) - \mu(\%BugFixCoChg\_BCI) > 0$$

We test the null hypothesis (i.e. H_O) with the difference in population means between the percent bug fixes as co-changes of the buggy cloned methods that co-change frequently and the percent bug fixes as co-change as co-changes of the buggy cloned methods that co-change infrequently. If the null hypothesis H_O rejected, we are confident the buggy cloned methods that co-change frequently have a greater percent of bug fixes as co-change than the buggy cloned methods that co-change infrequently.

6.1.3 Results

1	**2**	**3**	**4**
App	$\mu(\%BugFixCoChg\_BCF) - \mu(\%BugFixCoChg\_BCI)$	$\frac{\mu(\%BugFixCoChg\_BCF) - \mu(\%BugFixCoChg\_BCI)}{\mu(\%BugFixCoChg\_BCF)}$	P-Value
1	**0.20**	**0.27**	**0.01**
2	0.12	0.13	0.21
3	**0.44**	**0.47**	**0.04**
4	-0.07	-0.10	0.66
5	**0.48**	**0.72**	**0.05**
6	-0.18	-0.23	0.23

Table 6-2 – Comparison of the percent of bug fixes as co-change for buggy cloned methods with frequent and infrequent co-change

From Table 6-2, Applications 1, 3, and 5 are statistically significant with p-values under 0.05 and have greater percent bug fixes as co-change for buggy cloned methods with frequent co-change. From the developers past behavior of co-changing the cloned methods, bug fixes are also co-changed in the 3 statistically significant applications for the hypothesis. These applications follow the expected trend of co-change during development leads to co-change of bug fixes. Application 2 supports the hypothesis with a confidence level of 79%. Applications 4 and 6 have cases of specialization in the cloned methods that caused our expected trend not to hold. The results in Applications 4 and 6 contradict our hypothesis, with cases respectively of 9% and 19% greater co-change in bug fixes in the specialized methods to fix the coding errors or special cases common to both methods.

Given that 3 of the applications support the hypothesis with statistical significance, we conclude that H_9 holds. We note that in general, cloned methods that are not specialized (i.e. both methods have clone fragments, but other fragments of the cloned methods are different) benefit from co-change in development and leads to greater co-change in bug fixes.

6.2 H_{10} – The complexity of buggy cloned methods with frequent co-change is lower in comparison with the complexity of buggy cloned methods with infrequent co-change

To explain the reason that some buggy cloned methods co-change more frequently than others, we further investigate code complexity in buggy cloned methods.

6.2.1 Definition of Metrics

We continue to use the cyclomatic complexity (5-2) to represent code complexity.

6.2.2 Testing of H_{10}

To perform a statistical non-parametric test and test our hypothesis, we divide the buggy cloned methods into two groups: buggy cloned methods that co-change frequently and buggy cloned methods that co-change infrequently. Since the complexity of a method can affect co-change during development of new features and bug fixes, we use co-change per revision (5-1) for all code changes to divide the buggy cloned methods into methods with frequent co-change and methods with infrequent co-change. We remain consistent with H_9 and use 75% as the threshold between methods with frequent co-change and those that do not.

To determine whether the additional conditional statements are added to the buggy cloned methods that co-change frequently or infrequently, we test the complexity at three different stages of the lifetime of a cloned method: the initial, median, and final revisions.

For the initial, median, and final revisions, we use a statistical non-parametric test to compare the cyclomatic complexity of the buggy cloned methods that co-change frequently and buggy cloned methods that co-change infrequently. The null and alternative test hypothesis are as follows:

$$H_O: \mu(Cyclomatic\_BCF) - \mu(Cyclomatic\_BCI) \geq 0$$

$$H_A: \mu(Cyclomatic\_BCF) - \mu(Cyclomatic\_BCI) < 0$$

We test H_O with the difference in population means between the cyclomatic complexity in buggy cloned methods that co-change frequently and the cyclomatic complexity in buggy cloned methods that co-change infrequently. If the null hypothesis H_O is rejected, we are confident the buggy cloned methods that co-change frequently have lower cyclomatic complexity than buggy cloned methods that co-change infrequently for the application under study.

6.2.3 Results

1	**2**	**3**	**4**
App	$\mu(Cyclomatic\_initial\_BCF) - \mu(Cyclomatic\_initial\_BCI)$	$\frac{\mu(Cyclomatic\_initial\_BCF) - \mu(Cyclomatic\_initial\_BCI)}{\mu(Cyclomatic\_initial\_BCF)}$	P-Value
1	**-2.47**	**-0.68**	**0.00**
2	-2.69	-0.69	0.09
3	0.58	0.15	0.44
4	2.63	0.36	0.66
5	-6.04	-0.88	0.14
6	**-2.38**	**-1.36**	**0.01**

Table 6-3 – Comparison of initial cyclomatic complexity of buggy cloned methods with frequent and infrequent co-change

1	2	3	4
App	$\mu(Cyclomatic\_median\_BCF) - \mu(Cyclomatic\_median\_BCI)$	$\frac{\mu(Cyclomatic\_median\_BCF) - \mu(Cyclomatic\_median\_BCI)}{\mu(Cyclomatic\_median\_BCF)}$	P-Value
1	**-3.29**	**-0.86**	**0.00**
2	**-4.56**	**-1.12**	**0.00**
3	0.27	0.07	0.56
4	3.10	0.45	0.27
5	**-9.54**	**-1.39**	**0.05**
6	**-3.00**	**-1.50**	**0.03**

Table 6-4 – Comparison of median cyclomatic complexity of buggy cloned methods with frequent and infrequent co-change

1	2	3	4
App	$\mu(Cyclomatic\_final\_BCF) - \mu(Cyclomatic\_final\_BCI)$	$\frac{\mu(Cyclomatic\_final\_BCF) - \mu(Cyclomatic\_final\_BCI)}{\mu(Cyclomatic\_final\_BCF)}$	P-Value
1	**-5.30**	**-1.43**	**0.00**
2	**-6.01**	**-1.40**	**0.00**
3	0.22	0.05	0.58
4	3.83	0.54	0.13
5	**-16.03**	**-2.44**	**0.03**
6	**-3.04**	**-1.46**	**0.02**

Table 6-5 – Comparison of final cyclomatic complexity of buggy cloned methods with frequent and infrequent co-change

From Table 6-3, Applications 1 and 6 are statistically significant and have a lower initial complexity in buggy cloned methods with frequent co-change. From Table 6-4, Table 6-5, Applications 1, 2, 5, and 6 are statistically significant and have a lower median and final complexity in buggy cloned methods with frequent co-change. Applications 2 and 5 have a confidence level of at least 86% in initial cyclomatic complexity, suggesting a reasonable difference exists in the applications between the buggy cloned methods that co-change frequently and those that co-change infrequently. Applications 3 and 4 are not statistically significant in initial, median, or final cyclomatic complexity. From review of the Applications 3 and 4, 32% of cloned methods with cyclomatic complexity greater than or equal to 7 co-changed frequently. These cloned methods that exhibit frequent co-change are contrary to our hypothesis and explain the lack of statistical significance in Applications 3 and 4. In the statistically significant applications, the methods that co-change less frequently have on average 3.41 more conditional statements in the initial versions. These statements account for special cases specific to the individual cloned method and do not require co-change with other clone methods in the clone class.

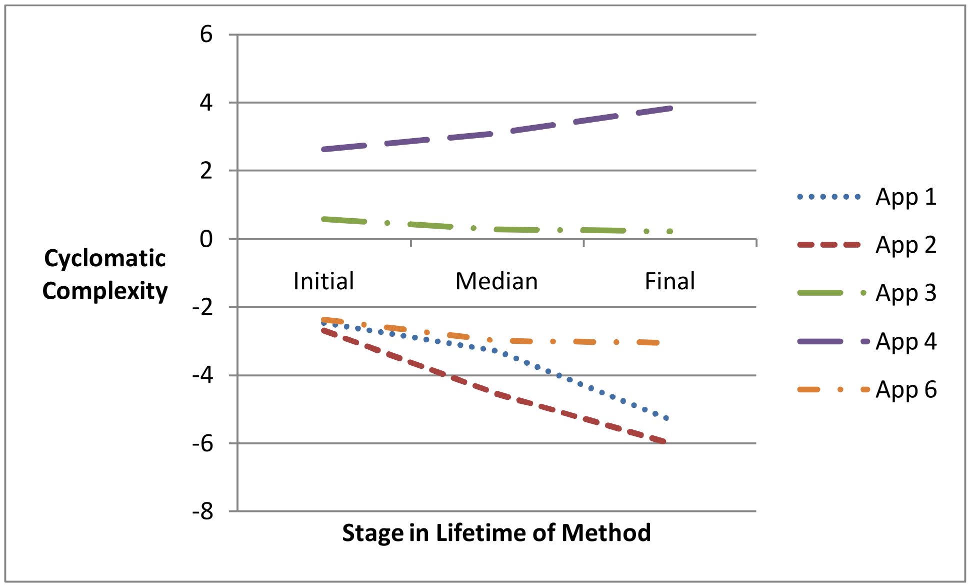

Figure 6-1 – Differences in cyclomatic complexity for Applications 1, 2, 3, 4, and 6 comparing buggy cloned methods with frequent and infrequent co-change

In Figure 6-1, the initial, median, and final values from the second columns from Table 6-3, Table 6-4, and Table 6-5 are shown. Because Application 5 has much higher cyclomatic complexity values than all other applications, its inclusion in the graph would cause the changes in cyclomatic complexity of other applications to be difficult to read. As a result, we omit Application 5 from the graph. Four of the statistically significant applications have an increasing trend from initial to final cyclomatic complexity in buggy cloned methods that co-change infrequently. The increases in complexity for Applications 1, 2, 5 and 6 from the initial revisions to final revisions, are 2.83, 3.32, 9.99 and 0.66 respectively. An increased value of 1 in cyclomatic complexity refers to one additional conditional statement added from the initial to its final version. As the buggy cloned methods that co-change infrequently evolve, more conditional statements are added to handle special cases or different features supported by the application.

Given that 4 of the applications support the hypothesis and have a confidence of at least 86%, we conclude that H_{10} holds. From the results of this hypothesis, we note that cloned methods with more complex code are less likely to co-change frequently. If a developer wishes the cloned method to be co-changed frequently, the method should be given a singular purpose with minimal special cases. With less complex code, co-change occurs more frequently due to less specialization of the method.

6.3 H_{11} – Buggy cloned methods with frequent co-change have fewer developer switches than buggy cloned methods with infrequent co-change

To further clarify the reason that some buggy cloned methods co-change more frequently than others, we investigate developer switches in buggy cloned methods.

6.3.1 Definition of Metrics

The metric of developer switches per revision (5-3) is used for the analysis of H_{11}.

6.3.2 Testing of H_{11}

The developer switches per revision of the buggy cloned methods that co-change frequently are compared with the buggy cloned methods that co-change infrequently. We calculate the difference in population means between the developer switches per revision in buggy cloned methods that co-change frequently and infrequently using the following test hypotheses:

$$H_O: \mu(DevSwitchPerRev\_BCF) - \mu(DevSwitchPerRev\_BCI) \geq 0$$

$$H_A: \mu(DevSwitchPerRev\_BCF) - \mu(DevSwitchPerRev\_BCI) < 0$$

If the null hypothesis H_O is rejected, we are confident the buggy cloned methods that co-change frequently have fewer developer switches per revision than buggy cloned methods that co-change infrequently for the application under study.

6.3.3 Results

1	**2**	**3**	**4**
App	$\mu(DevSwitchPerRev\_BCF) - \mu(DevSwitchPerRev\_BCI)$	$\frac{\mu(DevSwitchPerRev\_BCF) - \mu(DevSwitchPerRev\_BCI)}{\mu(DevSwitchPerRev\_BCF)}$	P-Value
1	-0.03	-0.05	0.57
2	0.18	0.28	0.09
3	**-0.28**	**-0.78**	**0.04**
4	-0.06	-0.11	0.75
5	0.10	0.15	0.57
6	**-0.37**	**-1.15**	**0.03**

Table 6-6 – Comparison of developer switches per revision for buggy cloned methods with frequent and infrequent co-change

From Table 6-6, Applications 3 and 6 are statistically significant and have fewer developer switches per revision in buggy cloned methods with frequent co-change. From manual review of Applications 3 and 6, 79% of the cloned methods that co-change frequently do not have code changes specific to the method in the conditional statements. The lack of developer switches in these cloned methods limited the number of developers and maintained knowledge of the cloned relationship. Application 2 has a confidence level of 91% and 28% greater developer switches per revision in buggy cloned methods with frequent co-change. We investigated the result and the main difference between this application and the others is that these methods implemented protocol requirements. Each new conditional statement provided well-defined functionality. As a result, developer switches did not obstruct co-change of these methods. Applications 1, 4, and 5 are not statistically significant and have a confidence level of less than 43%, meaning that developer switches per revision have little effect on the cloned methods in these applications.

Given that only 2 of the applications are statistically significant and support the hypothesis, we conclude that H_{11} is rejected. From the results of this hypothesis, we note that the effects of developer switches are application specific. The results of developer switches are dependent upon the cloned methods under investigation.

6.4 H_{12} – The method names in buggy cloned methods with frequent co-change are more similar than the method names of buggy cloned methods with infrequent co-change

Developers give names to methods so that they can refer to them later and understand what the method does. The name of a method implies its functionality. Since cloned methods already represent similar functionality, a similar name for a method implies that there is a greater relationship between them than if the methods do not have similar naming. We hypothesize the relationship is more recognizable and that cloned methods with similar naming co-change more frequently.

6.4.1 Definition of Metrics

We test our hypothesis by analyzing the Levenshtein distance [74] between the cloned method names in their respective clone classes. The Levenshtein distance computes the minimum number of method revisions required to change one sequence of characters to another. To make comparisons of method names irrespective of length, we compute the percent Levenshtein distance for each clone pair in a clone class as shown in (6-3). In the equation, mn_1 and mn_2 refer to two method names.

$$\%LevDist = \frac{LevenshteinDistance(mn_1, mn_2)}{\text{maxLength}\ (mn_1, mn_2)} \tag{6-3}$$

The $LevenshteinDistance(mn_1, mn_2)$ represents the Levenshtein distance between mn_1 and mn_2. The maxLength(mn_1, mn_2) represents the number of characters from the larger of the two method names mn_1 and mn_2. The percent Levenshtein distance ranges from values [0,1].

Given clone classes can have more than 2 cloned methods; we investigate cases of the most similar naming that exists in a clone class. The most similar naming corresponds to the minimum percent Levenshtein distance between the cloned method under study and the other cloned methods in the clone class.

6.4.2 Testing of H_{12}

A statistical non-parametric test is used to compare the minimum percent Levenshtein distance in buggy cloned methods with frequent co-change and buggy cloned methods with infrequent co-change. The null hypothesis H_O is tested with the difference in population means between the minimum percent Levenshtein distance in buggy cloned methods with frequent co-change and the minimum percent Levenshtein in buggy cloned methods with infrequent co-change. If the null hypothesis is rejected (i.e. $p\text{-value} < 0.05$), then we are confident the buggy cloned methods with frequent co-change have a lower minimum percent Levenshtein than buggy cloned methods with infrequent co-change for the application under study. The null and alternative hypotheses are as follows:

$$H_O: \mu(Min\%LevDist\_BCF) - \mu(Min\%LevDist\_BCI) \geq 0$$

$$H_A: \mu(Min\%LevDist\_BCF) - \mu(Min\%LevDist\_BCI) < 0$$

6.4.3 Results

1	**2**	**3**	**4**
App	$\mu(Min\%LevDist\_BCF) - \mu(Min\%LevDist\_BCI)$	$\frac{\mu(Min\%LevDist\_BCF) - \mu(Min\%LevDist\_BCI)}{\mu(Min\%LevDist\_BCF)}$	P-Value
1	**-0.11**	**-1.37**	**0.04**
2	-0.04	-0.21	0.55
3	**-0.19**	**-0.78**	**0.01**
4	-0.07	-1.39	0.18
5	**-0.26**	**-0.82**	**0.04**
6	0.11	0.31	0.11

Table 6-7 – Comparison of Levenshtein distance for buggy cloned methods with frequent and infrequent co-change

From Table 6-7, Applications 1, 3, and 5 are statistically significant with p-values under 0.05 and have lower Levenshtein distances for buggy cloned methods with frequent co-change. Application 2, 4, and 6 were not statistically significant and the results are contrary to our hypothesis that similar naming increases co-change. Manual inspection of the method names in Application 2 revealed that 83% of cloned methods in different classes with the exact same name co-changed infrequently. Application 4 showed 31% less similar naming in the buggy cloned methods that co-changed frequently. In Application 6, 50% of the cloned methods with frequent co-change have substring matches in the method names (i.e. setActive() and setInactive()). Given that 3 of the applications are statistically significant and support the hypothesis, we can conclude that H_{12} holds. Similar naming of method names can improve co-change.

6.5 H_{13} – In early development, buggy cloned methods with frequent co-change have greater co-change than buggy cloned methods with infrequent co-change

To continue our study of co-change, we investigate if greater co-change in early development leads to greater co-change for the lifetime of the method. We determine if co-change in early development is significantly different between buggy cloned methods with frequent and infrequent co-change. For the analysis of co-change in early development of software, we remain consistent with Section 5.5 and continue to use the first 6 months to represent early development. From the results of this hypothesis, we can further determine the reasons co-change is higher in some cloned methods in comparison with others.

6.5.1 Definition of Metrics

We use the metric of the co-change per revision (6-1) for our analysis.

6.5.2 Testing of H_{13}

To compare the co-change per revision in the first 6 months in buggy cloned methods with frequent and infrequent co-change, we formulate the following test hypothesis:

$$H_O: \mu(CoChgPerRev\_First6Mths\_BCF) - \mu(CoChgPerRev\_First6Mths\_BCI) \leq 0$$

$$H_A: \mu(CoChgPerRev\_First6Mths\_BCF) - \mu(CoChgPerRev\_First6Mths\_BCI) > 0$$

$\mu(CoChgPerRev\_First6Mths\_BCF) - \mu(CoChgPerRev\_First6Mths\_BCI)$ is the difference in population means between the co-change per revision in the first 6 months in buggy cloned methods with frequent and infrequent co-change. If the null hypothesis H_O is rejected, we are confident the buggy cloned methods with frequent co-change have greater co-change per revision in the first 6 months than buggy cloned methods with infrequent co-change for the application under study.

6.5.3 Results

1	**2**	**3**	**4**
App	$\mu(CoChgPerRev\_First6Months\_BCF) - \mu(CoChgPerRev\_First6Months\_BCI)$	$\frac{\mu(CoChgPerRev\_First6Months\_BCF) - \mu(CoChgPerRev\_First6Months\_BCI)}{\mu(CoChangePerRev\_First6Months\_BCF)}$	P-Value
1	**0.18**	**0.29**	**0.00**
2	**0.25**	**0.35**	**0.01**
3	**0.41**	**0.62**	**0.00**
4	-0.08	-0.17	0.77
5	**0.35**	**0.47**	**0.04**
6	**0.39**	**0.56**	**0.01**

Table 6-8 – Co-change per revision for first six months of buggy cloned methods with frequent and infrequent co-change

From Table 6-8, 5 of the 6 applications analyzed have a statistically significant increase in the co-changes per revision in the first 6 months comparing the methods with frequent co-change and the methods with infrequent co-change. The result makes sense that initial co-change of a cloned method maintains the cloned relationship for greater co-change in the future. Application 4 has a confidence level of 23% and showed that the first 6 months did not necessarily lead to greater co-change in the future. From review of the buggy cloned methods in Application 4, after the initial cloning, 31% of the methods did not co-change with other buggy cloned methods in the first 6 months. However, co-change occurred in later development to handle new updates for the application.

Given that 5 of the 6 applications studied are statistically significant and support the hypothesis, we conclude that H_{13} holds. Therefore, we conclude that a high co-change per revision in the first 6 months can lead to high co-change per revision for future and the overall lifetime of the method.

6.6 H_{14} – Buggy cloned methods with infrequent co-change in early development improve with co-change in later development than buggy cloned methods with infrequent co-change

This hypothesis focuses only on the buggy cloned methods with infrequent co-change (i.e. co-change per revision < 75% for the lifetime of the method). We hypothesize that co-change increases with time because the developers become familiar with the cloned methods in later development. For the analysis of co-change in early and later development of software, we use the first 6 months and after 6 months respectively to remain consistent with our analysis from Section 5.5 and Section 5.6

6.6.1 Definition of Metrics

For our analysis, we use the metric of the co-change per revision (6-1).

6.6.2 Testing of H_{14}

We compare the co-change per revision in the first 6 months in buggy cloned methods with infrequent co-change to after 6 months buggy cloned methods with infrequent co-change. We test the null hypothesis with the difference in population means between the co-change per revision in the first 6 months in buggy cloned methods with infrequent co-change and the co-change per revision after 6 months in buggy cloned methods with infrequent co-change. If the null hypothesis is rejected, we are confident the buggy cloned methods with infrequent co-change have less co-change per revision in the first 6 months than buggy cloned methods with infrequent co-change after 6 months for the application under study. We formulate the following null and alternative test hypothesis:

$$H_O\colon \mu(CoChgPerRev\_First6Mths\_BCI) - \mu(CoChgPerRev\_After6Mths\_BCI) \geq 0$$

$$H_A\colon \mu(CoChgPerRev\_First6Mths\_BCI) - \mu(CoChgPerRev\_After6Mths\_BCI) < 0$$

6.6.3 Results

1	2	3	4
App	$\mu(CoChgPerRev\_First6Mths\_BCI) - \mu(CoChgPerRev\_After6Mths\_BCI)$	$\frac{\mu(CoChgPerRev\_First6Mths\_BCI) - \mu(CoChgPerRev\_After6Mths\_BCI)}{\mu(CoChgPerRev\_First6Mths\_BCI)}$	P-Value
1	**0.17**	**0.39**	**0.00**
2	0.09	0.20	0.54
3	**-0.40**	**-1.65**	**0.04**
4	0.00	0.00	0.88
5	-0.10	-0.27	0.65
6	-0.11	-0.38	0.70

Table 6-9 –Co-change per revision for first six months to after six months of buggy cloned methods with infrequent co-change

From Table 6-9, Applications 1 and 3 are statistically significant at a confidence level of 95%. Application 3 supports the hypothesis that buggy cloned methods with infrequent co-change in the first 6 months can improve in co-change in later development. Application 1 is contrary to the hypothesis and shows more co-change in the first 6 months and in later development co-change decreases. From manual review of Application 3, 75% of the cloned methods that co-changed infrequently are simplified after early development with conditional statements removed. In Application 1, 37% of the cloned methods that co-changed infrequently are updated with 2 to 3 co-changes after cloning in the first 6 months. After the initial development, the functionality for these methods has already been established and only 1 or 2 code modifications are required to maintain the software. As a result, in these methods early development has an average of 50% co-change and later development has a co-change of 0%.

The other 4 applications have a maximum confidence level of 55%. We can explain the lack of statistical significance in the 4 applications. Application 4 has a confidence level of 12%, meaning that the co-change in early development does not change in later development. We attribute this lack of change to the more structured development process of Application 4. From

H_{10}, our earlier results on conditional complexity in H_{10}, we note that Applications 2, 5 and 6 decline in co-change in later development due to increased conditional statements for coding specialization of the methods.

Given that 2 of the applications are statistically significant, with 1 application supporting the hypothesis and the other contrary to the hypothesis, we reject H_{14}. We conclude that applications generally do not improve in co-change in later development due to additional control statements for specialization.

6.7 H_{15} – Buggy cloned methods that co-change frequently have a greater percent of its developers co-changing methods than buggy cloned methods with infrequent co-change

The purpose of this hypothesis is to determine whether buggy cloned methods with frequent co-change have a statistically higher percent of developers co-changing the methods in comparison to buggy cloned methods with infrequent co-change. We hypothesize that reducing the number of developers is not relevant. Rather we propose the percent of developers that recognize the need for co-change is statistically significant in comparison of buggy cloned methods with frequent and infrequent co-change.

6.7.1 Definition of Metrics

To test the hypothesis, we use the metric of the percent of developers who perform co-changes (6-4).

$$\%DevCoChg = \frac{\#DevelopersCoChange}{\#Developers} \tag{6-4}$$

The #DevelopersCoChange represents the number of developers who have performed a co-change for the method under investigation. The #Developers is the total number of developers who worked on the method under investigation. The values for this metric range from [0,1].

6.7.2 Testing of H_{15}

To compare the percent of developers who perform co-changes in buggy cloned methods with frequent and infrequent, we formulate the following null and alternative test hypothesis:

$$H_O: \mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI) \leq 0$$

$$H_A: \mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI) > 0$$

$\mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI)$ is the difference in population means between the percent of developers who perform co-changes in buggy cloned methods with frequent and infrequent co-change. If H_O is rejected (i.e. p-value < 0.05), then we are confident the buggy cloned methods with frequent co-change have a greater percent of developers who perform co-changes than buggy cloned methods with infrequent co-change for the application under study.

6.7.3 Results

1	**2**	**3**	**4**
App	$\mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI)$	$\frac{\mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI)}{\mu(\%DevCoChg\_BCF)}$	P-Value
1	**0.31**	**0.34**	**0.00**
2	**0.34**	**0.34**	**0.00**
3	**0.33**	**0.33**	**0.00**
4	**0.26**	**0.28**	**0.01**
5	**0.38**	**0.40**	**0.00**
6	**0.25**	**0.25**	**0.00**

Table 6-10 – Comparison of percent developers co-change for buggy cloned methods with frequent and infrequent co-change

From Table 6-10, all 6 applications studied have p-values under 0.05 and the percent developers co-change is statistically significant between the methods with frequent co-change and the methods with infrequent co-change. To further verify our result that the methods with frequent co-change do not simply have fewer developers, we use the metric of the developers per revision (5-6). We calculate the difference of the average developers per revision as described in Section 5.8 for the buggy cloned methods with frequent co-change and the buggy cloned methods with infrequent co-change. For data verification, we perform a two-tailed Wilcoxon-Mann-Whitney T-Test at a confidence level of 95%.

1	**2**	**3**	**4**
App	$\mu(DevPerRev\_BCF) - \mu(DevPerRev\_BCI)$	$\frac{\mu(DevPerRev\_BCF) - \mu(DevPerRev\_BCI)}{\mu(DevPerRev\_BCF)}$	P-Value
1	0.08	0.12	0.12
2	**0.22**	**0.31**	**0.01**
3	-0.20	-0.45	0.11
4	-0.05	-0.08	0.57
5	0.18	0.28	0.12
6	-0.15	-0.27	0.28

Table 6-11 – Comparison of developers per revision for buggy cloned methods with frequent and infrequent co-change

In Table 6-11, the results of the second column are only statistically significant in one application, meaning the number of developers does not play a significant role in whether a cloned method has high co-change or not. The significant factor is the number of developers recognizing the clone relationship between the methods to perform co-change. Given that all 6 of the applications studied support the hypothesis and are statistically significant, we conclude that H_{15} holds.

6.8 Summary

	Hypothesis	% Applications Supporting Hypothesis	Status
H_9	Buggy cloned methods with frequent co-change in development have greater co-change in bug fixes than buggy cloned methods with infrequent co-change in development	50%	Holds
H_{10}	The complexity of buggy cloned methods with frequent co-change is lower in comparison with the complexity of buggy cloned methods with infrequent co-change	67%	Holds
H_{11}	Buggy cloned methods with frequent co-change have fewer developer switches than buggy cloned methods with infrequent co-change	33%	Rejected
H_{12}	The method names in buggy cloned methods with frequent co-change are more similar than the method names of buggy cloned methods with infrequent co-change	50%	Holds
H_{13}	In early development, buggy cloned methods with frequent co-change have greater co-change than buggy cloned methods with infrequent co-change	83%	Holds
H_{14}	Buggy cloned methods with infrequent co-change in early development improve with co-change in later development than buggy cloned methods with infrequent co-change	33%	Rejected
H_{15}	Buggy cloned methods that co-change frequently have a greater percent of its developers co-changing methods than buggy cloned methods with infrequent co-change	100%	Holds

Table 6-12 – Status of hypotheses for analysis of co-change of cloned methods

6.8.1 Reasons Each Application Rejected a Hypothesis

Application 1 was rejected in H_{11} because the methods that co-changed frequently had code changes not exclusively dealing with the cloned code. Application 2 was rejected in H_9 and H_{14} due to specialization in the cloned methods (i.e. sections of the cloned methods are never resynchronized). Application 2 was also rejected in H_{12} because 83% of cloned methods with infrequent co-change with the same name, but in different classes. In H_{11}, Application 2 had a confidence level of 91%, which is consistent with the high confidence level in Application 3 that also implemented protocol requirements. Applications 3 and 4 were rejected under H_{10} because 32% of the cloned methods in the applications had a cyclomatic complexity greater than or equal to 7 and co-changed frequently. Application 4 was not statistically significant in hypothesis H_9 and H_{10} due to specialization in the cloned methods. For hypotheses H_{12}, H_{13}, and H_{14}, Application 4 was rejected due to the secure nature and increased code review of the application. Application 5 was not accepted due to specialization in the cloned methods in hypotheses H_{11} and H_{14}. Application 6 was rejected in hypotheses H_9 and H_{14} from specialization in the cloned methods. In H_{12}, Application 6 was not accepted because 50% of the cloned methods with frequent co-change were substring matches (i.e. setActive() and setInactive()), which outperformed exact matches in cloned methods of frequent co-change.

6.8.2 Discussion of Hypotheses and Usage

The summary of the results of the hypotheses for Chapter 6 are shown in Table 6-12. A hypothesis is considered to hold if the percent of applications supporting the hypothesis is at least 50%. From H_9, cloned methods that co-change frequently also co-change their bug fixes. These cloned methods are not causing the software to require greater bug fix effort because inconsistent code changes are rarely occurring. From the frequency of co-change in the first 6 months of a cloned method, we can predict frequent or infrequent co-change for the rest of its lifetime (i.e. H_{13}). To improve the overall co-change of the cloned methods in the application, we can suggest removing the cloned methods with low co-change. From the results from H_{13} and H_{14}, if the cloned method does not co-change well in the first 6 months, the cloned method is unlikely to co-change well in the future. Refactoring has been suggested in the past to remove code clones with *bad smells* [39]. Using techniques to refactor clones [75], [76], [111], the clones with low co-change can be refactored and the overall co-change of the cloned methods in the application can be increased.

To improve co-change and clone evolution in general, the techniques for cloning methods can be improved. From the results of H_{10}, we suggest for developers to keep cloned methods with a singular purpose (i.e. not handling many different situations) to minimize chances for increased complexity and specialization in the future. Also, with the outcome of H_{12}, we advise for developers to use similar naming of cloned methods that the developer feels should be co-changed frequently in the future.

From H_{15}, a high percent of developers who co-change cloned methods leads to high co-change. Increasing only the frequency of co-change in a cloned method may result in one developer co-changing the cloned method more frequently while other developers do not. As a result, the developers who do not co-change the cloned method may make inconsistent updates. We suggest a general understanding amongst the developers working on the cloned methods must be made to present that co-change is important for these cloned methods in order to achieve frequent co-change in a cloned method.

The hypothesis concerning developer switches and co-change (i.e. H_{11}) was rejected due to a lack of applications with statistical significance. In 3 of the 4 applications that were not statistically significant, the developer switches were not a significant factor because the methods that co-changed frequently had code changes not exclusively dealing with the cloned code. Hypothesis H_{15} tested whether cloned methods that co-change infrequently improve in co-change over time. The hypothesis was rejected due to divergence in the code content over time in the cloned areas of cloned methods that co-change infrequently.

Chapter 7
Co-Change Analysis of Clone Classes

Three established ways of analyzing cloned code are to analyze the individual clone instances [79], clone pairs [17], [34], [72], [82], and clone classes [12], [15], [60], [62]. The three established ways correspond to analysis of one, two, or all clone instances in a clone class. In the previous two chapters, we analyze individual clone instances to investigate how cloned methods are affected by bug fixes and co-change. In this chapter, we analyze the clone classes to understand how all the cloned methods in a clone class are affected by bug fixes and co-change using the hypotheses in Table 7-1. By analyzing clone classes, we can provide a different perspective on development effort because our analysis is using the changelist, as opposed to using method revision in the previous two chapters. We performed the analysis for individual methods because it is more accurate for the time analysis (e.g. the first 6 months of each cloned method as opposed to the first 6 months of a clone class where other methods may be cloned after 6 months). The advantage of analysis from the changelist is that co-change better approximates development effort. With changelist analysis, we can detect the number of clone instances that are co-changed each time a developer updates a cloned method in the clone class. To obtain a full overview of the clone classes associated with the buggy cloned methods, we include the non-buggy cloned methods that belong to the same clone classes. Also in this chapter, similar to Chapter 6, we compare buggy clone classes with frequent co-change to buggy clone classes with infrequent co-change in each hypothesis. We quantify "frequent co-change" as 75% or more of the changelists in the clone class are co-changes. We first analyze the features of buggy clone classes with frequent co-change and buggy clone classes with infrequent co-change in H_{16} and H_{17}. In H_{16}, we analyze if a greater number of co-change changelists in development (i.e. coding new features) leads to a greater number of co-change changelists of bug fixes. Similar to H_9 from Chapter 6, H_{16} is the only hypothesis in this chapter that strictly uses the revision during development for grouping clone classes as "co-changing frequently". In H_{17}, we study whether buggy clone classes with frequent co-change have a higher average percent of methods co-changed. We hypothesize that the frequency of co-change has the added benefit of a greater number of the cloned methods in the clone class co-changing in each changelist. Next, we investigate reasons to explain the frequent co-change of the buggy clone classes with frequent co-change in $H_{18,}$ and H_{19}. In H_{18}, we determine if code complexity affects the co-change behavior of buggy clone classes with frequent co-change. In H_{19}, we study the buggy clone classes that co-change more frequently to note if the percent of methods in the home directory is higher and suggest that more cloned methods in the same directory increases co-change. We hypothesize more cloned methods in the same directory make co-change more convenient for developers. We then focus our analysis to determine the distribution of development effort to improve co-change of buggy clone classes in H_{20} and H_{21}. In H_{20}, we determine if buggy clone classes with frequent co-change have a greater development effort on co-change in the first 6 months of the clone class. In H_{21}, we investigate if the percent of developers that perform co-change on the methods in the buggy clone classes is statistically higher in buggy clone classes with frequent co-change. We hypothesize limiting the number of developers is not relevant to co-change; rather awareness to co-change amongst developers is of importance for frequent co-change.

	Hypothesis
H_{16}	Buggy clone classes that co-change frequently during development have greater co-change in bug fixes than buggy clone classes that co-change infrequently during development
H_{17}	The average percent of methods co-changed in buggy clone classes that co-change frequently is greater than the average percent of methods co-changed in buggy clone classes that co-change infrequently
H_{18}	Buggy clone classes that co-change frequently have lower code complexity than buggy clone classes that co-change infrequently
H_{19}	Buggy clone classes that co-change frequently have a greater percent of methods in the home directory than buggy clone classes that co-change infrequently
H_{20}	In early development of the clone class, buggy clone classes that co-change frequently have greater co-change than buggy clone classes that co-change infrequently
H_{21}	Buggy clone classes that co-change frequently have a greater percent of its developers co-changing methods than buggy clone classes that co-change infrequently

Table 7-1 – List of hypotheses for analysis of co-change of clone classes

7.1 H_{16} – Buggy clone classes that co-change frequently during development have greater co-change in bug fixes than buggy clone classes that co-change infrequently during development

Following from the first hypothesis from Chapter 6, greater co-change in development leads to greater co-change of bug fixes, we extend the work from using method revisions to changelists level. The benefit of performing the tests using the changelist is to note whether the clone class also benefits from greater co-change in development as opposed to the individual cloned methods.

7.1.1 Definition of Metrics

We introduce two metrics to test the hypothesis: co-change per changelist and percent bug fix changelists as co-change. The co-change per changelist is expressed in Equation (7-1).

$$CoChgPerChglist = \frac{\#CoChangeChangelists}{\#Changelists} \tag{7-1}$$

The number of co-change changelists for the method under investigation is represented by #CoChangeChangelists. The number of changelists for the method is represented by #Changelists.

To study the co-change of bug fixes in buggy clone classes with frequent co-change and those with infrequent co-change, we present the percent bug fixes changelists as co-change in Equation (7-2).

$$\%BugFixChglistCoChg = \frac{\#BugFixChglist\_CoChg}{\#BugFixChangelists} \tag{7-2}$$

The $\#BugFixChangelists_{CoChange}$ refers to the number of bug fix changelists co-changed for the method under investigation. The #BugFixChangelists represents the number of bug fix changelists for the method.

7.1.2 Testing of H_{16}

We use the co-change per changelist to divide the buggy clone classes into two data sets for comparison. Buggy clone classes with co-change per changelist greater or equal to 75% during development are considered to co-change frequently. Buggy clone classes with co-change per changelist less than 75% during development are considered to not co-change frequently.

We use the metric of the percent bug fix changelists as co-change for our analysis. A statistical non-parametric test is used to compare the percent bug fix changelists as co-change of the buggy clone classes that co-change frequently and the buggy clone classes that co-change infrequently. We formulate the following null and alternative hypotheses:

$$H_O: \mu(\%BugFixChglistCoChg\_BCF) - \mu(\%BugFixChglistCoChg\_BCI) \leq 0$$

$$H_A: \mu(\%BugFixChglistCoChg\_BCF) - \mu(\%BugFixChglistCoChg\_BCI) > 0$$

We test the null hypothesis (i.e. H_O) with the difference in population means between the percent bug fix changelists per revision as co-changes of the buggy clone classes that co-change frequently and the percent bug fixes as co-change as co-changes of the buggy clone classes that co-change infrequently. If the null hypothesis is rejected, we are confident the buggy clone classes that co-change frequently have a greater percent of bug fixes as co-change than the buggy clone classes that co-change infrequently.

7.1.3 Results

1	**2**	**3**	**4**
App	$\mu(\%BugFixChglistCoChg\_BCF) - \mu(\%BugFixChglistCoChg\_BCI)$	$\frac{\mu(\%BugFixChglistCoChg\_BCF) - \mu(\%BugFixChglistCoChg\_BCI)}{\mu(\%BugFixChglistCoChg\_BCF)}$	P-Value
1	**0.37**	**0.41**	**0.00**
2	**0.38**	**0.38**	**0.00**
3	**0.25**	**0.25**	**0.01**
4	0.26	0.28	0.14
5	**0.45**	**0.45**	**0.04**
6	**0.29**	**0.29**	**0.04**

Table 7-2 – Comparison of the percent of bug fix changelists as co-change between buggy clone classes with frequent and infrequent co-change

From Table 7-2, 5 out of the 6 applications studied are statistically significant with p-values under 0.05 and have greater percent bug fix changelists as co-change for buggy clone classes with frequent co-change.   The one application that is not statistically significant, Application 4, has a confidence level of 86% and supports the hypothesis. We compare our results in this hypothesis to the results in the first hypothesis in Chapter 6 and notice the increased agreement with the hypothesis in Applications 2, 4, and 6. The increased agreement makes sense because analysis using the clone class provides a better perspective of developers that notice the cloned relationship in the clone class. The analysis using the clone class includes co-changes during development of other methods aside from the one requiring the bug fix. By including co-changes of the other cloned methods, we take into account all code changes in the clone class that show developers have experience or knowledge of cloned methods in the clone class to co-change bug fixes in the future.

Given that all 6 of the applications studied support the hypothesis and are statistically significant, we conclude that H_{16} holds. We note that even though the buggy cloned methods did not co-change in many cases, other cloned methods from the same clone class performed co-changes. As a result, we deduce the developers missed or forgot to co-change the buggy cloned methods during development.

7.2 H_{17} – The average percent of methods co-changed in buggy clone classes that co-change frequently is greater than the average percent of methods co-changed in buggy clone classes that co-change infrequently

The purpose of this hypothesis is to determine whether clone classes with frequent co-change lead to co-change of a small or large number of the clone instances co-changed in each changelist relative to the size of the clone class (i.e. the percent of methods co-changed). The motivation for the hypothesis is to determine if frequent co-change equates to a greater number of clone instances co-changed.

7.2.1 Definition of Metrics

We use the metric of the co-change per changelist (7-1) to divide our data into two sets for comparison.

To determine whether the number of methods co-changed is proportionally higher for clone classes that co-change more frequently, we calculate the average percent of methods co-changed for all changelists for each clone class as shown in Equation (7-4).

$$Avg\%MethodsCoChg = \frac{\sum \frac{\#MethodsChanged\_Changelist}{\#MethodsInCloneClass\_Changelist}}{\#Changelists} \qquad (7\text{-}4)$$

The #MethodsChanged_Changelist represents the number of methods changed in the changelist under investigation. The #MethodsInCloneClass_Changelist represents the number of methods that exist in the clone class at the time of the changelist under investigation. The #Changelists represents the number of changelists in the clone class under investigation.

7.2.2 Testing of H_{17}

Since the number of cloned methods co-changed is relevant during development and bug fixes, we use the co-change per changelist for all changes (i.e. development of new features, optimizations, bug fixes, etc.) to divide the buggy clone classes into buggy clone classes with frequent co-change and those that do not. We continue to use 75% as the threshold between methods with frequent co-change and those that do not.

To test the null hypothesis we calculate the difference in population means between the average percent methods co-changed in buggy clone classes that co-change frequently and the average percent methods co-changed in buggy clone classes that co-change infrequently. If H_O is rejected, we are confident the buggy clone classes that co-change frequently have a greater average percent methods co-changed than buggy clone classes that co-change infrequently for the application under study. The null and alternative hypotheses are as follows:

$$H_O: \mu(Avg\%MethodsCoChg\_BCF) - \mu(Avg\%MethodsCoChg\_BCI) \leq 0$$

$$H_A: \mu(Avg\%MethodsCoChg\_BCF) - \mu(Avg\%MethodsCoChg\_BCI) > 0$$

7.2.3 Results

1	**2**	**3**	**4**
App	$\mu(Avg\%MethodsCoChg\_BCF) - \mu(Avg\%MethodsCoChg\_BCI)$	$\frac{\mu(Avg\%MethodsCoChg\_BCF) - \mu(Avg\%MethodsCoChg\_BCI))}{\mu(Avg\%MethodsCoChg\_BCF)}$	P-Value
1	**0.29**	**0.31**	**0.00**
2	**0.24**	**0.27**	**0.00**
3	**0.24**	**0.25**	**0.00**
4	0.11	0.13	0.13
5	**0.43**	**0.43**	**0.04**
6	**0.24**	**0.24**	**0.01**

Table 7-3 – Comparison average percent of cloned methods co-changed for buggy clone classes with frequent and infrequent co-change

From Table 7-3, 5 out of the 6 applications studied are statistically significant with p-values under 0.05. Application 4, the application that is not statistically significant, has a confidence level of 87% and supports the hypothesis. All 6 applications have a greater percent of cloned methods co-changed for buggy clone classes with frequent co-change. The result makes sense because if more methods co-change in each changelist, more methods maintain the cloned relationship for future changes. Even small numbers of methods that co-change together in a changelist help to maintain the cloned relationship in the clone class. From Table 7-3, on average, clone classes that co-change frequently have 26% greater percent of its cloned methods co-changed. Given that 5 of the 6 applications studied are statistically significant and support the hypothesis, we conclude that H_{17} holds. Clone classes with frequent co-change have a higher average percent of methods co-changed in its changelists.

7.3 H_{18} – Buggy clone classes that co-change frequently have lower code complexity than buggy clone classes that co-change infrequently

We continue our analysis of complexity from Section 6.2 using cyclomatic complexity. In Section 6.2, we recognize that increases in conditional complexity reduce co-change in general. In the analysis of the buggy clone classes, we can analyze the minimum, maximum, and median complexity of the methods in a clone class. The purpose of this hypothesis is to determine if the buggy clone classes that co-change frequently have a significantly lower complexity than the buggy clone classes that co-change infrequently.

7.3.1 Definition of Metrics

We use the metric cyclomatic complexity (5-2) to represent code complexity for our analysis.

7.3.2 Testing of H_{18}

As in Section 6.2, we calculate the complexity of the initial, median, and final revisions of each method in a clone class. To determine if the smallest, median, or largest complexities in a clone class influence co-change, we calculate the minimum, median, and maximum of the initial, median, and final complexity values.

For the initial, median, and final revisions, we use a statistical non-parametric test to compare the minimum, median, or maximum cyclomatic complexity of the buggy clone classes that co-change frequently and buggy clone classes that co-change infrequently. The null hypothesis is tested with the difference in population means between the cyclomatic complexity in buggy clone classes that co-change frequently and infrequently. We formulate the following null and alternative test hypotheses:

$$H_O: \mu(Cyclomatic\_BCF) - \mu(Cyclomatic\_BCI) \geq 0$$

$$H_A: \mu(Cyclomatic\_BCF) - \mu(Cyclomatic\_BCI) < 0$$

If H_O is rejected, we are confident the buggy clone classes that co-change frequently have lower cyclomatic complexity than buggy clone classes that co-change infrequently for the application under study.

We create a total of 9 null hypotheses equations: 3 using the minimum of the initial, median, and final revisions, 3 using the median of the initial, median, and final revisions, and 3 using the maximum using the initial, median, and final revisions. We present below the tables representing the 3 null hypotheses equations using the maximum, which yielded the best results. The results from the other 6 null hypotheses equations can be found in Appendix B.

7.3.3 Results

1	**2**	**3**	**4**
App	$\mu(Cyclomatic\_max\ \_initial\_BCF) - \mu(Cyclomatic\_max\ \_initial\_BCI)$	$\frac{\mu(Cyclomatic\_max\ \_initial\_BCF) - \mu(Cyclomatic\_max\ \_initial\_BCI)}{\mu(Cyclomatic\_max\ \_initial\_BCF)}$	P-Value
1	-3.82	-1.12	0.16
2	-3.15	-0.79	0.11
3	-0.75	-0.24	0.71
4	-2.92	-0.61	0.71
5	-8.60	-1.56	0.20
6	**-3.65**	**-2.28**	**0.02**

Table 7-4 – Comparison of maximum initial cyclomatic complexity of buggy clone classes with frequent and infrequent co-change

1	**2**	**3**	**4**
App	$\mu(Cyclomatic\_max\ \_median\_BCF) - \mu(Cyclomatic\_max\ \_median\_BCI)$	$\frac{\mu(Cyclomatic\_max\ \_median\_BCF) - \mu(Cyclomatic\_max\ \_median\_BCI)}{\mu(Cyclomatic\_max\ \_median\_BCF)}$	P-Value
1	-4.38	-1.09	0.14
2	**-4.05**	**-0.93**	**0.04**
3	-1.16	-0.35	0.65
4	-2.33	-0.49	0.73
5	-13.85	-2.77	0.15
6	**-4.35**	**-2.29**	**0.05**

Table 7-5 – Comparison of maximum median cyclomatic complexity of buggy clone classes with frequent and infrequent co-change

1	2	3	4
App	$\mu(Cyclomatic\_max\ \_final\_BCF) - \mu(Cyclomatic\_max\ \_final\_BCI)$	$\frac{\mu(Cyclomatic\_max\ \_final\_BCF) - \mu(Cyclomatic\_max\ \_final\_BCI}{\mu(Cyclomatic\_max\ \_final\_BCF)}$	P-Value
1	-6.59	-1.65	0.15
2	**-5.75**	**-1.26**	**0.05**
3	-1.60	-0.49	0.65
4	-3.00	-0.67	0.41
5	-20.10	-4.02	0.11
6	**-4.25**	**-2.13**	**0.04**

Table 7-6 – Comparison of maximum final cyclomatic complexity of buggy clone classes with frequent and infrequent co-change

From Table 7-5 and Table 7-6, referring to the maximum of the median and final complexity values, Applications 2 and 6 are statistically significant and have a lower complexity in buggy clone classes with frequent co-change. Applications 1, 2, 5, and 6 have a confidence level of at least 84% in all three tables and support the hypothesis. The same applications have similar results to the code complexity study in Chapter 6. We note from manual review of Applications 1 and 5 that 10% and 40% cloned methods that co-change frequently respectively have a complexity greater than or equal to 7 and account for the lack of statistical significance in the application. From review of Applications 3 and 4, respectively, the applications have 69% and 83% of its clone classes with 2 cloned methods. As a result, the findings are similar to our previous results in the code complexity analysis in Chapter 6.

The results from the minimum and medians of the complexities in Appendix A have either 0 or 1 applications as statistically significant. Given that only 2 of the applications are statistically significant and support the hypothesis, we conclude that H_{18} is rejected.

7.4 H_{19} – Buggy clone classes that co-change frequently have a greater percent of methods in the home directory than buggy clone classes that co-change infrequently

From Kasper et al. [62], the two subsystems analyzed for their case studies in cloning had 60% and 79% of the clones in the home directory. Two potential reasons are cited for the large number of clones in the home directory: developers tend to duplicate code from files within subsystems they are working in and clones just might be harder to find when they are in other directories. To extend the work, we hypothesize that buggy clone classes that co-change more frequently have a higher percent of methods in the home directory.

7.4.1 Definition of Metrics

To determine the home directory for a clone class, we analyze the number of methods in each directory. Using Figure 7-1, in the topmost table, there are five different methods in the clone class and the directory is stated for each method. These directories represent the file paths to the directories where the methods are stored. In the middle table in Figure 7-1, the frequency for each directory is stated. In the example, directory "B" has the highest frequency and is concluded to be the home directory for the clone class. In the case where there is an equal distribution of frequency with a frequency greater than 1, then the first directory alphabetically that has the highest common value is considered the home directory. If all the directories for the clone class have a frequency of 1, we consider there to be no home directory. In the lower table in Figure 7-1, each of the methods is assigned a Boolean value of true or false, whether the method is in the home directory or not.

Data for Clone Class #1	
Methods in Clone Class	Directory
M1	A
M2	B
M3	C
M4	B
M5	B

$$\downarrow$$

Directory	Frequency
A	1
B	3
C	1
SUM	5

$$\downarrow$$

Data for Clone Class #1		
Methods in Clone Class	Directory	Is In Home Directory of Clone Class
M1	A	False
M2	B	True
M3	C	False
M4	B	True
M5	B	True

Figure 7-1 – Steps to determine if a method is in the home directory

We calculate the percent of methods in the home directory as shown in Equation (7-5).

$$\%MethodsInHomeDir = \frac{\#MethodsInHomeDir}{\#MethodsInCloneClass} \tag{7-5}$$

The #MethodsInHomeDir represents the number of methods in the home directory for the clone class under investigation. The #MethodsInCloneClass represents the number of methods in the clone class.

7.4.2 Testing of H_{19}

We compare the average percent methods co-changed of the buggy clone classes that co-change frequently and infrequently using the following test hypothesis:

$$H_O: \mu(\%MethodsInHomeDir\_BCF) - \mu(\%MethodsInHomeDir\_BCI) \leq 0$$

$$H_A: \mu(\%MethodsInHomeDir\_BCF) - \mu(\%MethodsInHomeDir\_BCI) > 0$$

$\mu(\%MethodsInHomeDir\_BCF) - \mu(\%MethodsInHomeDir\_BCI)$ is the difference in population means between the percent of methods in the home directory in buggy clone classes that co-change frequently and infrequently. If the null hypothesis H_O is rejected, we are confident the buggy clone classes that co-change frequently have a greater percent of methods in the home directory than buggy clone classes that co-change infrequently for the application under study.

7.4.3 Results

1	2	3	4
App	$\mu(\%MethodsInHomeDir\_BCF) - \mu(\%MethodsInHomeDir\_BCI)$	$\frac{\mu(\%MethodsInHomeDir\_BCF) - \mu(\%MethodsInHomeDir\_BCI)}{\mu(\%MethodsInHomeDir\_BCF)}$	P-Value
1	0.00	0.00	NA
2	0.00	0.00	NA
3	0.06	0.06	0.47
4	0.17	0.33	0.53
5	0.07	0.07	0.44
6	0.40	0.40	0.20

Table 7-7 – Comparison of percent of cloned methods in home directory between buggy clone classes with frequent and infrequent co-change

From Table 7-7, 4 out of the 6 applications studied have a greater percent of cloned methods in the home directory for buggy clone classes with frequent co-change. The other 2 applications, applications 1 and 2 have the same average percent of cloned methods in the home directory in buggy clone classes with frequent co-change and in buggy clone classes with infrequent co-change. Application 6 shows the most promise with a confidence level of 80%. To explain the result of Application 6, we investigate the number of directories and files in the applications under study.

1	**2**	**3**	**4**
App	*Number of Directories*	*Number of Files*	$\frac{Number\ of\ Files}{Number\ of\ Directories}$
1	27	6810	252.22
2	7	5100	728.57
3	17	5346	314.47
4	12	3118	259.83
5	4	3434	858.5
6	24	1527	63.63

Table 7-8 – Ratio of number of files to number of directories in the 6 applications under study

From the fourth column in Table 7-8, we note that Application 6 has the smallest ratio of files to directories with an average of 64 files per directory. We expect that for applications with even smaller ratios, the co-change in the home directory can become more significant. For the two reasons stated in Godfrey et al. [62] and our results, we can presume that cloning in the home directory is a strong concept that occurs in the majority of clone instances in a clone class. Given that none of the applications are statistically significant, we reject H_{19}. However, we note in 4 of the 6 applications studied the same or higher percent of cloned methods in the home directory for clone classes with frequent co-change. We can suggest further research into the locality of clones in applications with small file to directory ratios to note if co-change is greater in the home directory of a clone class.

7.5 H_{20} - In early development of the clone class, buggy clone classes that co-change frequently have greater co-change than buggy clone classes that co-change infrequently

For our study, a clone class contains 2 or more cloned methods. If all the methods in a clone class are cloned at the same time or only 2 cloned methods exist in the majority of the clone classes, the results of this hypothesis should be similar to the study of co-change in early development of individual methods in Chapter 6. The purpose of this hypothesis is to investigate co-change in early development (first 6 months) of a clone class. We determine if co-change in early development corresponds with greater co-change for the lifetime of the clone class.

7.5.1 Definition of Metrics

We use the metric of the co-change per changelist (7-1) for our analysis.

7.5.2 Testing of H_{20}

The average percent methods co-changed in the first 6 months of the buggy clone classes that co-change frequently are compared with the buggy clone classes that co-change infrequently.We test the null hypothesis with the difference in population means between the co-change per changelist in the first 6 months of buggy clone classes that co-change frequently and the co-change per changelist in the first 6 months in buggy clone classes that co-change infrequently. If H_O is rejected, we are confident the buggy clone classes that co-change frequently have a greater co-change per changelist in the first 6 months than buggy clone classes that co-change infrequently for the application under study. The null and alternative test hypotheses in mathematical form are as follows:

$$H_O\!: \mu(CoChgPerChglist\_First6Mths\_BCF) - \mu(CoChgPerChglist\_First6Mths\_BCI) \leq 0$$

$$H_A\!: \mu(CoChgPerChglist\_First6Mths\_BCF) - \mu(CoChgPerChglist\_First6Mths\_BCI) > 0$$

7.5.3 Results

1	**2**	**3**	**4**
App	$\mu(CoChgPerChglist\_First6Mths\_BCF) - \mu(CoChgPerChglist\_First6Mths\_BCI)$	$\frac{\mu(CoChgPerChglist\_First6Mths\_BCF) - \mu(CoChgPerChglist\_First6Mths\_BCI)}{\mu(CoChgPerChglist\_First6Mths\_BCF)}$	P-Value
1	**0.43**	**0.46**	**0.00**
2	**0.48**	**0.50**	**0.00**
3	**0.55**	**0.58**	**0.00**
4	0.13	0.16	0.32
5	**0.72**	**0.72**	**0.04**
6	0.44	0.46	0.10

Table 7-9 – Comparison of co-change per changelist for first 6 months of development between buggy clone classes with frequent and infrequent co-change

From Table 7-9, Applications 1, 2, 3, and 5 have a statistically significant increase in the co-changes per changelist in the first 6 months comparing the buggy clone classes with frequent co-change and the buggy clone classes with infrequent co-change. Application 6 is not statistically significant, but has a confidence level of 90% and supports the hypothesis. The result makes sense that initial co-change of a cloned method maintains the cloned relationship for greater co-change in the future. Application 4 is not statistically significant and has a confidence level of 68%. From review of the clone classes in Application 4, 72% of the clone classes with infrequent co-change have a co-change per change of at least 50% in the first 6 months. The results from the application are not statistically significant because the co-change per change in the first 6 months in the clone classes that co-change frequently and those that co-change frequently are similar.

Given that 5 of the 6 applications support the hypothesis with a confidence of at least 90%, we conclude that H_{20} holds. A high co-change per changelist in the first 6 months corresponds with high co-change per changelist for the lifetime of the method.

7.6 H_{21} – Buggy clone classes that co-change frequently have a greater percent of its developers co-changing methods than buggy clone classes that co-change infrequently

The purpose of this hypothesis is to determine whether buggy clone classes with frequent co-change have a statistically higher percent of developers who co-change the cloned methods in the clone class than the buggy cloned methods with infrequent co-change. We hypothesize that reducing the number of developers is not relevant and that a statistically significant percent of developers recognize the need for co-change in comparison of buggy clone classes with frequent and infrequent co-change.

7.6.1 Definition of Metrics

To test the hypothesis, we use the metric of the percent of developers who perform co-changes (6-4).

7.6.2 Testing of H_{21}

A statistical non-parametric test is used to compare the percent of developers who perform co-changes in buggy clone classes with frequent co-change and buggy clone classes with infrequent co-change. We formulate the following test hypothesis:

$$H_O: \mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI) \leq 0$$

$$H_A: \mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI) > 0$$

We test the null hypothesis H_O with the difference in population means between the percent of developers who perform co-changes in buggy clone classes with frequent co-change and the percent of developers who perform co-changes in buggy cloned classes with infrequent co-change. If the null hypothesis H_O is rejected, we are confident the buggy clone classes with frequent co-change have a greater percent of developers who perform co-changes than buggy clone classes with infrequent co-change for the application under study.

7.6.3 Results

1	**2**	**3**	**4**
App	$\mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI)$	$\frac{\mu(\%DevCoChg\_BCF) - \mu(\%DevCoChg\_BCI)}{\mu(\%DevCoChg\_BCF)}$	P-Value
1	**0.37**	**0.41**	**0.00**
2	**0.38**	**0.38**	**0.00**
3	**0.25**	**0.25**	**0.01**
4	0.26	0.28	0.14
5	**0.45**	**0.45**	**0.04**
6	**0.29**	**0.29**	**0.04**

Table 7-10 – Comparison of percent developers co-change between buggy clone classes with frequent and infrequent co-change

From Table 7-10, 5 of the 6 applications studied have p-values under 0.05 and the percent developers who perform co-changes is statistically significant between the buggy clone classes with frequent co-change and the buggy clone classes with infrequent co-change. Application 4 is not statistically significant, but has a confidence level of 86% and supports the hypothesis.

To further verify our result that the methods with frequent co-change do not simply have fewer developers, we normalize the developer metric with respect to the number of changelists in a clone class. We calculate the developers per changelist as shown in Equation (7-6).

$$DevPerChglist = \frac{\#Developers}{\#Changelists} \tag{7-6}$$

The #Developers represents the number of developers that have changed a method in the clone class under investigation. The #Changelists represents the number of changelists for the clone class under investigation.

We calculate the difference of the average developers per changelist for the buggy clone classes with frequent co-change and the buggy cloned classes with infrequent co-change. For data verification, we perform a two-tailed Wilcoxon-Mann-Whitney T-Test at a confidence level of 95%.

1	**2**	**3**	**4**
App	$\mu(\%DevPerChglist\_BCF) - \mu(\%DevPerChglist\_BCI)$	$\frac{\mu(\%DevPerChglist\_BCF) - \mu(\%DevPerChglist\_BCI)}{\mu(\%DevPerChglist\_BCF)}$	P-Value
1	**0.40**	**0.46**	**0.00**
2	0.11	0.19	0.38
3	-0.06	-0.14	0.53
4	0.17	0.25	0.30
5	0.43	0.58	0.11
6	0.00	-0.01	0.31

Table 7-11 – Comparison of developers per revision between buggy clone classes with frequent and infrequent co-change

The second column in Table 7-11 presents the difference in average developers per changelist in buggy cloned methods with frequent co-change and buggy cloned methods with infrequent co-change. The results are only statistically significant in one application, meaning the number of developers does not play a significant role in whether a cloned method has high co-change or not. The significant factor is that the developers recognize the cloned relationship between the cloned methods to perform co-change. Given that all 6 of the applications support the hypothesis and have a confidence of at least 86%, we conclude that H_{21} holds.

7.7 Summary

	Hypothesis	% Applications Supporting Hypothesis	Status
H_{16}	Buggy clone classes that co-change frequently during development have greater co-change in bug fixes than buggy clone classes that co-change infrequently during development	83%	Holds
H_{17}	The average percent of methods co-changed in buggy clone classes that co-change frequently is greater than the average percent of methods co-changed in buggy clone classes that co-change infrequently	83%	Holds
H_{18}	Buggy clone classes that co-change frequently have lower code complexity than buggy clone classes that co-change infrequently	33%	Rejected
H_{19}	Buggy clone classes that co-change frequently have a greater percent of methods in the home directory than buggy clone classes that co-change infrequently	0%	Rejected
H_{20}	In early development of the clone class, buggy clone classes that co-change frequently have greater co-change than buggy clone classes that co-change infrequently	66%	Holds
H_{21}	Buggy clone classes that co-change frequently have a greater percent of its developers co-changing methods than buggy clone classes that co-change infrequently	83%	Holds

Table 7-12 – Status of hypotheses for analysis of co-change of clone classes

7.7.1 Reasons Each Application Rejected a Hypothesis

For all 6 applications, we speculate hypothesis H_{19} was rejected because the large number of files per directory could have made it difficult for developers to find cloned methods by locality of existence in the same directory. Application 1 was not statistically significant on the class level in H_{18} due to differences in maximum complexity of the cloned methods in the clone classes. Applications 3 and 4 were rejected in H_{18} for the same reasons stated in H_{10} from Chapter 6. Application 4 was rejected in H_{16}, H_{17}, H_{19}, H_{20}, and H_{21} due to the greater code review of the application. Application 5 was rejected on the class level in H_{18} though the application was statistically significant in the comparable hypothesis H_{10} from Chapter 6 due to differences in maximum complexity of the cloned methods in the clone classes. Application 6 had a confidence level of 90% in H_{20}. The lack of the other necessary 5% agreement for statistical significance could be explained by a lack of data because the application is only one year old and smaller than the other applications tested in the study.

7.7.2 Discussion of Hypotheses and Usage

In Table 7-12, the result of each hypothesis in Chapter 7 is presented. Hypotheses with less than 50% support from the applications tested were rejected. Using the clone class for analysis, from hypothesis H_{16}, we have further proof that greater co-change in development leads to greater co-change for bug fixes. From hypothesis H_{17}, the clone classes that co-change frequently also co-change with more of its cloned methods in the clone class. Both H_{16} and H_{17} are supported with statistical significance in 5 of the 6 applications studied. This also means that the clone classes that co-change infrequently have statistically lower co-change of bug fixes and perform co-changes in fewer of its cloned methods. As a result, we suggest assigning higher priority for refactoring for these clone classes with less frequent co-change

The hypothesis H_{20} held and support that greater co-change for a clone class in the first 6 months leads to greater co-change for the clone class. This result is consistent with H_{13} that tested if co-change in the first 6 months leads to greater co-change for the lifetime of the method.

From H_{21}, we notice the percent of developers who co-change the methods in the clone class was significant using the clone class. The result is consistent with H_{15} from the analysis using the cloned method. A general understanding amongst the developers working on cloned methods in a clone class must be made that co-change is important in order to achieve frequent co-change in the clone class.

The highest, median, and lowest cyclomatic complexity values of cloned methods in clone classes (i.e. H_{18}) were not found to be a significant factor in co-change of the clone class due to a lack of statistically significant applications. For 2 of the 4 applications that were statistically significant in the similar hypothesis H_{10} from Chapter 6, were not statistically significant on the class level. The difference in results can be attributed to differences in maximum complexity of the cloned methods in the clone classes. We conclude that the complexity of one specific cloned method in a clone class does not significantly affect co-change of the clone class. The other hypothesis that was rejected in the chapter is H_{19}. Although the results support the hypothesis with greater co-change in the home directory, the results were not statistically significant. We speculate that the large number of files per directory could have made it difficult for developers to find cloned methods by locality of existence in the same directory.

Chapter 8
Conclusion and Future Work

We present an approach to study the influences of different factors on bug fixes and co-change in cloned code. The effect of each influence tested is presented through the case study using 6 applications. In this chapter, we summarize the contributions of our thesis and discuss future work related to our research.

8.1 Thesis Contributions

Bug Fix Analysis for Cloned and Non-Cloned Methods: We show that cloned methods that do not copy from older and more tested code require greater bug fix effort than non-cloned methods. It suggests that applications with cloned methods that copy from new code can require greater support and testing time before the application is released to the market.

We determined that code complexity of cloned methods have a higher complexity in general than non-cloned methods. Previous work has shown that higher complexity leads to more bug fixes [91], [97]. However, these works did not analyze cloned and non-cloned methods separately. As a result, we submit that the greater complexity in cloned methods contributes more to bug fixes than non-cloned methods. We recommend that applications with cloned methods of high complexity and that are duplicated from new code (i.e. not old and tested code) be allotted more time for testing before release to market.

Co-Change Analysis for Cloned Methods: In our analysis, we find that cloned methods that co-change frequently also co-change their bug fixes. Conversely, cloned methods that co-change infrequently do not co-change their bug fixes. The lack of co-change leads to inconsistent changes in the cloned code in the methods. As a result, bugs are created in the code.

We showed that methods that do co-change infrequently in the first 6 months do not co-change often during the lifetime of the method, especially in cases with a growing code complexity during software evolution. We suggest refactoring the cloned code in these methods to eliminate the problem of inconsistent change.

Clone evolution in general can be improved by cloning methods with a particular purpose that minimizes chances for increased complexity and specialization in the future. In addition, similar naming of cloned methods can help make co-change more intuitive and increase co-change in the future.

Co-Change Analysis for Clone Classes: We provide further evidence with analysis using the clone class that greater co-change in development leads to greater co-change for bug fixes. In addition, the clone classes that co-change frequently also co-change with more of its cloned methods in the clone class. These results in the clone class analysis are supported by 5 out of the 6 applications tested. This means that the clone classes that co-change infrequently have statistically lower co-change of bug fixes and perform co-changes in fewer of its cloned methods. As a result, we suggest assigning a higher priority for refactoring of all the cloned methods in cloned classes with infrequent co-change before refactoring singly low co-changing cloned methods due to greater agreement in the results from the clone class analysis.

We notice the percent of developers who co-change the methods in the clone class was statistically significant in 5 out of the 6 applications studied. This suggests that a higher percent of the developers recognize the need to co-change in the clone classes that co-change frequently. We suggest that in order to improve co-change of cloned methods that the developers who work on applications with many cloned methods, be told which methods are cloned. As a result, we hope that the percent of developers who co-change the methods can increase, and the co-change of the clone class increase as well.

8.2 Future Work

In the future, the work can be extended in following 5 directions:

Analyze more applications from different sources: we plan to analyze more applications from different sources, such as open-source projects to verify our findings from our current applications. Then, based on the agreement between their analyses we can get more fair evaluation of factors that influence bug fixes and co-change.

Analyze different metrics: through analysis of different metrics, better metrics than code complexity or developer switches per change can be found that influence bug fixes and co-change. Other code complexity metrics may be useful, as well as different metrics that relate to developers.

Investigate the effect of locality on co-change on smaller applications or subsystems: our analysis of cloned methods in the home directory was most significant in the smallest application studied in this thesis. Most of the cloned methods analyzed in this thesis are in the home directory. Smaller applications with fewer files per directory may have more cases of cloned methods not in the home directory and tests can be conducted to determine if a statistically significant difference exists in the co-change of cloned methods in the home directory.

Analyze the bug fixes and co-change of categorized cloned methods: by performing a more in-depth study on the bug fixes and co-change on each category of cloned method (i.e. overloading, reciprocal operations, methods for high-level control flow, etc.), we can determine greater insight into clone evolution. The different design patterns for cloned methods can be scrutinized to investigate the patterns that lead to more stable code.

Raise awareness of cloned methods in developers: We suggest further study as to the technique to convey the information to developers about the existence of a cloned method in order to increase co-change. The effectiveness of different verbal and tool approaches can be explored.

References

[1] F. B. Abreu and W. Melo. “Evaluating the impact of object-oriented design on software quality,” *Proceedings of the 3rd International Symposium on Software Metrics: From Measurement to Empirical Results*, pp. 90-99, Mar. 1996.

[2] R. Al-Ekram, C. Kasper, R. Holt and M. Godfrey. “Cloning by accident: an empirical study of source code cloning across software systems,” *International Symposium on Empirical Software Engineering*, pp. 376-385, Nov. 2005.

[3] M. D’Ambros and M. Lanza. “BugCrawler: visualizing evolving software systems,” *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pp. 333-334, Mar. 2007.

[4] M. D’Ambros, M. Lanza and M. Lungu. “Visualizing co-change information with evolution radar,” *IEEE Transactions on Software Engineering,* Mar. 2009

[5] G. Antoniol, V. Rollo and G. Venturi. “Detecting groups of co-changing files in CVS repositories,” *International Workshop on Principles of Software Evolution*, pp. 23-32, Sept. 2005.

[6] G. Antoniol, E.Merlo, U. Villano and M. Di Penta, “Analyzing cloning evolution in the Linux kernel,” *Information and Software Technology*, pp. 755-765, Sept. 2002.

[7] G. Antoniol, M. Di Penta and E. Merlo, “An automatic approach to identify class evolution discontinuities,” *International Workshop on Principles of Software Evolution*, pp. 31-40, 2004,

[8] E. Arisholm and Dag I.K. Sjøberg, “Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software,” *IEEE Transactions on Software Engineering*, pp. 521-534, Aug. 2004.

[9] D. Atkins, T. Ball, T. Graves and A. Mockus, "Using version control data to evaluate the impact of software tools: a case study of the version editor," *IEEE Transactions on Software Engineering*, pp. 625-637, July 2002.

[10] L. Aversano, L. Cerulo and M. D. Penta, "How clones are maintained: an empirical study," *European Conference on Software Maintenance and Engineering*, pp. 81-90, Mar. 2007.

[11] K. Ayari, P. Meshkinfam, G. Antoniol and M. D. Penta, "Threats on building models from CVS and Bugzilla repositories: the Mozilla case study," *CASCON*, pp. 215-228, 2007.

[12] B.S. Baker, "A program for identifying duplicated code," *Proceedings of Computing Science and Statistics: 24th Symposium Interface*, vol. 24, pp. 49- 57, Mar. 1992.

[13] M. J. Baker and S. G. Eick, "Visualizing software systems," *International Conference on Software Engineering* (ICSE), pp. 59-67, May 1994.

[14] B. Baker, "On finding duplication and near-duplication in large software systems," *Proceedings of the 2nd Working Conference on Reverse Engineering (WCRE)*, pp. 86-95, July 1995.

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

[16] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 1-3, January 2002.

[17] I. Baxter, A. Yahin, L. Moura and M. Anna, “Clone detection using abstract syntax trees,” *Proceedings of the 14th International Conference on Software Maintenance (ICSM)*, pp. 368-377, Nov. 1998.

[18] R. M. Bell, T. J. Ostrand and E. J. Weyuker, “Looking for bugs in all the right places,” *Proceedings of the ACM/international symposium on software testing and analysis* (ISSTA). pp. 61–71, 2006.

[19] S. Bellon. Vergleich von Techniken zur Erkennung duplizierten Quellcodes. *Diplomarbeit, Universit ̈at Stuttgart*, 2002. (In German).

[20] K. Bennett, “Software evolution: past, present and future,” *Elsevier Science B.V*, Feb. 1999.

[21] D. Beyer and A. Noack, “Clustering software artifacts based on frequent common changes,” *Proceedings of the 13th International Workshop on Program Comprehension*, pp. 259-268, May 2005.

[22] D. Beyer and A. E. Hassan, “Animated visualization of software history using evolution storyboards,” *Working Conference on Reverse Engineering (WCRE)*, pp. 199-210, Oct. 2006.

[23] D. Beyer and A. E. Hassan, “Evolution storyboards: visualization of software structure dynamics,” *International Conference on Program Comprehension*, pp. 248-251, 2006.

[24] L. C. Briand and J. Wust, “Modeling development effort in object-oriented systems using design properties,” *IEEE Transactions on Software Engineering*, pp. 963-986, Nov. 2001.

[25] E. Burd and J. Bailey, “Evaluating clone detection tools for use during preventative maintenance,” *Proceedings of the 2nd International Workshop on Source Code Analysis and Manipulation* (SCAM), pp. 36, Oct. 2002.

[26] W. R. Bush, J. D. Pincus and D. J. Sielaff, “A static analyzer for finding dynamic programming errors,” *Software—Practice & Experience*, , pp. 775–802, June 2000.

[27] G. Canfora, L. Cerulo and M. Di Penta, “Tracking your changes: a language-independent approach,” *IEEE Transactions on Software Engineering*, pp. 50-57, Jan. 2009.

[28] G. Canfora, L. Cerulo and M. Di Penta, “On the use of line co-change for identifying crosscutting concern code,” *International Conference on Software Maintenance*. Pp. 213-222, Sept. 2006.

[29] S. Chidamber and C. Kemerer, “A metrics suite for object-oriented design,” *IEEE Transactions of Software Engineering*, pp. 476-493, June 1994.

[30] J. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar and M. Sridharan, “Efficient and precise data race detection for object-oriented programs,” *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 258-269, May 2002.

[31] J. Cordy, “Comprehending reality – practical barriers to industrial adoption of software maintenance automation,” *11th IEEE International Workshop on Program Comprehension*, pp. 196, 2003.

[32] S. Demeyer, S. Ducasse and M. Lanza, “A hybrid reverse engineering approach combining metrics and program visualization,” *Working Conference Reverse Engineering*, pp. 175, 1999.

[33] W. Duala-Ekoko and M. Robillard, “Tracking code clones in evolving software,” *Proceedings of the 29th international conference on Software Engineering*, pp. 158-167, May 2007

[34] S. Ducasse, M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code," *Proceedings of International Conference on Software Maintenance*, pp. 109-118, August 1999.

[35] H. J. Eghbali, "K-S test for detecting changes from landsat imagery data," *IEEE Transactions on Software Engineering*, pp. 17-23, Jan. 1979.

[36] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions in Software Engineering*, pp. 675-689, Sept. 1999.

[37] M. Fischer, M. Pinzger and H. Gall, "Populating a release history database from version control and bug tracking systems," *Proceedings of the International Conference on Software Maintenance*, pp. 23-32, Sept. 2003.

[38] C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata, "Extended static checking for Java," *Programming Language Design and Implementation* (PLDI), pp. 234-245, May 2002.

[39] M. Fowler, *Refactoring: improving the design of existing code,* Addison Wesley, 1999.

[40] H. Gall, K. Hajek and M. Jazayeri, "Detection of logical coupling based on product release history," *Proceedings of the International Conference on Software Maintenance*, pp. 190-198, Nov. 1998.

[41] H. Gall, M. Jazayeri, R. Klosch, and G. Trausmuth, "Software evolution observations based on product release history," *International Conference on Software Maintenance*, pp. 160-166, Oct. 1997.

[42] H. Gall, M. Jazayeri and J. Krajewski, "CVS release history data for detecting logical couplings," *International Workshop on Principles on Software Evolution*, pp.13-23, Sept. 2003.

[43] R. Geiger, B. Fluri, H. C. Gall and M. Pinzger, "Relation of code clones and change couplings," *Proceedings of the 9th International Conference of Fundamental Approaches to Software Engineering (FASE)*, pp. 411-425, Mar. 2006.

[44] G. Gill and C. Kemerer, "Cyclomatic complexity density and software maintenance productivity," *IEEE Transactions on Software Engineering, pp. 1284-1288,* Dec. 1991

[45] T. L. Graves, A. F. Karr, J.S. Marron and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on Software Engineering*, pp. 653-661, July 2000.

[46] P. Grubb and A. T. Takang, *Software Maintenance: Concepts and Practice*, 2nd ed., Singapore: World Scientific, 2003.

[47] C. Gutwin and S. Greenberg, "The effects of workspace awareness support on the usability of real-time distributed groupware," *Transactions on Computer-Human Interaction*, pp. 243-281, Sept. 1999

[48] S. Hallem, B. Chelf, Y. Xie and D. Engler, "A system and language for building system-specific, static analyses," *Programming Language Design and Implementation*, pp. 69-82, July 2002.

[49] M.H. Halstead, *Elements of Software Science*. Elsevier North-Holland, 1979.

[50] A. E. Hassan, "Predicting faults using the complexity of method revisions," *International Conference on Software Engineering*. May 2008.

[51] W. Hays, *Statistics*, 5 ed., New York: Harcourt Brace College Publishers, 1994.

[52] J. Herbsleb and R. Grinter, "Splitting the organization and integrating the code: Conway's Law revisited," *International Conference on Software Engineering*, pp. 85-95, May 1999

[53] Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue, "ARIES: refactoring support environment based on code clone analysis," *Software Engineering and Applications,* pp.222- 229, Nov. 2004.

[54] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *19th Object Oriented Programming Systems Languages and Applications*, pp. 132-136, Dec. 2004.

[55] IEEE Std. 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, IEEE, New York, 1991.

[56] Y. Jia, D. Binkley, M. Harman, J. Krinke and M. Matsushita, "KClone: a proposed approach to fast precise code clone detection," *International Workshop on Software Clones*. Mar. 2009.

[57] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner, "Do code clones matter?" *International Conference on Software Engineering*, pp. 485-495, May 2009.

[58] E. Juergens, B. Hummel, F. Deissenboeck and M. Feilkas, "Static bug detection through analysis of inconsistent clones," *Testmethoden fur Software (TESO)*, 2008

[59] J. Kajihara, G. Amamiya and T. Saya, "Learning from bugs," *IEEE Software*, pp. 46-54, Sept. 1993.

[60] T. Kamiya, S. Kusumoto and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transcations on Software Engineering*, pp. 654-670, July 2002.

[61] C. Kasper and M. W. Godfrey, "Cloning considered harmful," *Working Conference on Reverse Engineering*, pp. 19-28, Oct. 2006.

[62] C. Kasper and M. Godfrey, "Aiding comprehension of cloning through categorization," *Proc. Int'l Workshop on Principles of Software Evolution*, pp. 85-94, Mar. 2004.

[63] M. Kim, V. Sazawal, D. Notkin and G. Murphy, "An empirical study of code clone genealogies," *Proceedings of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering*, pp. 187–196, September 2005.

[64] S. Kim, "Adaptive bug prediction by analyzing project history," *Dissertation from University of California Santa Cruz*. 2006

[65] G. Kocak, B. Turhan, and A. Bener, "Software defect prediction using call graph based ranking (CGBR) framework," *Euromicro Conference*, pp. 191-198, Sept. 2008.

[66] K. Kontogiannis, "Evaluation experiments on the detection of programming patterns using software metrics," *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE)*, pp. 44-54, Oct. 1997.

[67] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler and M. Bernstein, "Pattern matching for clone and concept detection," *Journal of Automated Software Engineering*, pp.77-108, July 1996.

[68] T. M. Koshgoftaar, E. B. Allen, K. S. Kalaichelvan and N. Goel, "Early quality prediciton: a case study in telecommunications," *IEEE Software, pp. 65-71,* Jan. 1996

[69] J. Krinke, "Identifying similar code with program dependence graphs," *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE)*, pp. 301-30, October 2001.

[70] J. Krinke, "Is cloned code more stable than non-cloned code?" *8th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 57-66, Sept. 2008.

[71] J. Krinke, "A study of consistent and inconsistent changes to code clones," *14th Working Conference on Reverse Engineering*, pp. 170-178, Feb. 2008.

[72] B. Lague, D. Proulx, J. Mayrand, E. M. Merlo and J. Hudeophl, “Assessing the benefits of incorporating function clone detection in a development process,” *Proceedings of IEEE International Conference on Software Maintenance*, pp. 314-321, Oct. 1997.

[73] M. Lehman and L. Belady, “Program evolution processes of software change,” *Academic Press*. 1985

[74] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals,” *Soviet Physics,* pp. 707-710, Feb. 1966.

[75] H. Li and S. Thompson, “Clone detection and removal for Erlang/OTP within a refactoring environment,” *Proceedings of the ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pp. 169-178, Jan. 2009

[76] H. Li, S. Thompson, G. Orosz and M. T’oth, “Refactoring with wrangler,” *ACM SIGPLAN Erlang Workshop,* pp. 61-72, Sept. 2008.

[77] B. Lientz and E. Swanson, Software Maintenance Management. Addison-Wesley. 1980.

[78] Z. Li, S. Lu, S. Myagmar and Y. Zhou, “CP-Miner: finding copy-paste and related bugs in large-sale software code,” *IEEE Transactions on Software Engineering*, pp. 176-192, Mar. 2006.

[79] A. Lozano, M. Wermelinger and B. Nuseibeh, “Evaluating the harmfulness of cloning: a change based experiment,” *Mining Software Repositories*, pp. 18, May 2007.

[80] A. Lozano and M. Wermelinger, “Assessing the effect of clones on changeability,” *International Conference on Software Maintenance,* pp. 227-236, Oct. 2008.

[81] F. J. Massey, Jr., “The distribution of the maximum deviation between two sample cumulative step-function,” *Ann. Math. Statistic*. pp. 125-128, 1951.

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

[83] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, pp. 308-320, Dec. 1976.

[84] A. Meneely, L. Williams, W. Snipes and J. Osborne, "Predicting failures with developer networks and social network analysis," *ACM SIGSOFT International Symposium on the Foundations of Software Engineering,* pp. 13-23, 2008.

[85] T. Mens and S. Demeyer, "Future trends in software evolution metrics," *Proceedings of 4th International Workshop on Principles of Software Evolution,* pp. 83-86, 2001.

[86] T. Menzies, J. Greenwald and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Transactions on Software Engineering*, pp. 2-13, Jan 2007.

[87] R. Von Mises, *Mathematical Theory of Probability and Statistics*. New York: Academic, 1964, pp 490-493.

[88] A. Mockus and L. Votta, "Identifying reasons for software change using historic databases," *Proceedings of IEEE International Conference on Software Maintenance*, pp. 120-130, Oct. 2000.

[89] A. Mockus, R. Fielding and J. Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Trans. Software Eng. and Methodology* (TOSEM), pp. 309–346, July 2002.

[90] A. Mockus and D. Weiss, "Predicting risk of software changes," *Bell Labs Technical Journal,* pp. 169-180, Aug. 2002.

[91] N. Nagappan, T. Ball and A. Zeller, "Mining metrics to predict component failures," *International Conference on Software* Engineering, pp. 452-451, May 2006.

[92] I. Neamtiu, J. Foser and M. Hicks, "Understanding source code evolution using abstract syntax tree matching," *Mining Software Repositories*, pp. 1-5, July 2005.

[93] N. Ohlsson and H. Alberg, "Predicting fault-prone software modules in telephone switches," *IEEE Transactions on Software Engineering*, pp. 886-894, Dec. 1996.

[94] 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*, pp. 470-495, May 2009.

[95] A. Sarma, L. Maccherone, P. Wagstrom and J. Herbsleb, "Tesseract: interactive visual exploration of socio-technical relationships in software development," *International Conference on Software Engineering*, pp. 23-33, May 2009.

[96] S. Siegel, *Non-Parametric Statistic for Behavioral Science*. New York: McGraw-Hill, 1956, pp. 127-135.

[97] M. Sheppard, "A critique of cyclomatic complexity as a software metric," *Software Engineering*, pp. 30-36, Mar. 1988.

[98] C. Souza and D. Redmiles, "An empirical study of software developers' management of dependencies and changes," *International Conference on Software Engineering*, pp. 241-250, May 2008.

[99] F. Simon, F. Steinbruckner and C. Lewerentz, "Metrics based refactoring," *European Conference on Software Maintenance and Re-engineering*, pp. 30-38, Mar. 2001.

[100] Software Maintenance, http://hebb.cis.uoguelph.ca/~dave/27320/quality/maintain.html, 1995.

[101] B. A. Tate, *Bitter Java*, London: Manning Publications, 2002.

[102] Tool BugZilla Mozilla, https://bugzilla.mozilla.org/, 1998

[103] Tool Eclipse: Java Development Tools, http://www.eclipse.org/jdt/, 2009

[104] Tool JLint – Find Bugs in Java Programs, http://jlint.sourceforget.net, 2006

[105] Tool SimScan, http://www.blue-edge.bg/download.html, 2005.

[106] Tool Simian, http://www.redhillconsulting.com.au/products/simian/, 2003

[107] Tool R, www.**r**-project.org/, 2008

[108] K.S. Tso, M. Hecht and K. Littlejohn, "Complexity metrics for avionics software," *Aerospace and Electronics Conference*, pp. 603-609, May 1992.

[109] B. Turhan and A. Bener, "A multivariate analysis of static code attributes for defect prediction," *International Conference on Quality Software*, pp. 231-237, Oct. 2007.

[110] B. Turhan, G. Kocak and A. Bener, "Data mining source code for locating software bugs: A case study in telecommunication industry," *Expert Systems with Applications*, pp. 9986-9990, Aug. 2009.

[111] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto and K. Inoue, "On refactoring support based on code clone dependency relation," *Software Metrics, 11th IEEE International Symposium*, pp. 16, Sept. 2005.

[112] T. J. Yu, V. Y. Shen and H. E. Dunsmore, "An analysis of several software defect models," *IEEE Transactions on Software Engineering*, pp. 1261-1270, Sept. 1998.

[113] A. Zaidman, B. Rompaey, S. Demeyer and A. Deursen, "Mining software repositories to study co-evolution of production & test code," *International Conference on Software Testing, Verification, and Validation*, pp. 220-239, May 2008.

[114] T. Zimmermann, P. Weißgerber, S. Diehl and A. Zeller, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering,* pp. 429-445, June 2005.

Appendix A

Bug Fix Analysis of Cloned and Non-Cloned Methods Chapter

App	$\mu(DevelopersPerRevision\_BC)$	$\mu(DevelopersPerRevision\_BNC)$
1	0.62	0.74
2	0.64	0.81
3	0.50	0.58
4	0.65	0.76
5	0.54	0.71
6	0.60	0.71

Table A-1 – Average developers per revision for buggy cloned and non-cloned methods

Appendix B

Clone Class Co-Change Chapter

1	2	3	4
App	$\mu(Cyclomatic\_min\_iniital\_BCF) - \mu(Cyclomatic\_min\_initial\_BCI)$	$\frac{\mu(Cyclomatic\_min\_iniital\_BCF) - \mu(Cyclomatic\_min\_initial\_BCI)}{\mu(Cyclomatic\_min\_iniital\_BCF)}$	P-Value
1	-0.23	-0.07	0.78
2	-0.45	-0.12	0.66
3	-0.30	-0.10	0.65
4	-2.33	-0.52	0.82
5	-3.30	-0.60	0.57
6	**-2.15**	**-1.34**	**0.05**

Table B-1 – Comparison of minimum initial cyclomatic complexity of buggy clone classes with frequent co-change and buggy clone classes do not co-change frequently

1	2	3	4
App	$\mu(Cyclomatic\_min\_median\_BCF) - \mu(Cyclomatic\_min\_median\_BCI)$	$\frac{\mu(Cyclomatic\_min\_median\_BCF) - \mu(Cyclomatic\_min\_median\_BCI)}{\mu(Cyclomatic\_min\_median\_BCF)}$	P-Value
1	0.27	0.07	0.56
2	-1.67	-0.42	0.27
3	-0.47	-0.15	0.82
4	-2.25	-0.56	0.82
5	-5.00	-1.00	0.41
6	-2.60	-1.37	0.15

Table B-2 – Comparison of minimum median cyclomatic complexity of buggy clone classes with frequent co-change and buggy clone classes do not co-change frequently

1	**2**	**3**	**4**
App	$\mu(Cyclomatic\_min\_final\_BCF) - \mu(Cyclomatic\_min\_final\_BCI)$	$\frac{\mu(Cyclomatic\_min\_final\_BCF) - \mu(Cyclomatic\_min\_final\_BCI)}{\mu(Cyclomatic\_min\_final\_BCF)}$	P-Value
1	0.25	0.06	0.52
2	-1.62	-0.38	0.41
3	-0.63	-0.20	0.78
4	-3.08	-0.95	0.24
5	-6.50	-1.44	0.26
6	-2.75	-1.38	0.12

Table B-3 – Comparison of minimum final cyclomatic complexity of buggy clone classes with frequent co-change and buggy clone classes do not co-change frequently

1	**2**	**3**	**4**
App	$\mu(Cyclomatic\_med\_iniital\_BCF) - \mu(Cyclomatic\_med\_initial\_BCI)$	$\frac{\mu(Cyclomatic\_med\_iniital\_BCF) - \mu(Cyclomatic\_med\_initial\_BCI)}{\mu(Cyclomatic\_med\_iniital\_BCF)}$	P-Value
1	-1.08	-0.32	0.32
2	-1.91	-0.51	0.23
3	-0.52	-0.17	0.69
4	-2.63	-0.57	0.82
5	-5.55	-1.01	0.41
6	**-2.90**	**-1.81**	**0.03**

Table B-4 – Comparison of median initial cyclomatic complexity of buggy clone classes with frequent co-change and buggy clone classes do not co-change frequently

1	**2**	**3**	**4**
App	$\mu(Cyclomatic\_med\_median\_BCF) - \mu(Cyclomatic\_med\_median\_BCI)$	$\frac{\mu(Cyclomatic\_med\_median\_BCF) - \mu(Cyclomatic\_med\_median\_BCI)}{\mu(Cyclomatic\_med\_median\_BCF)}$	P-Value
1	-0.95	-0.24	0.40
2	-2.84	-0.68	0.12
3	-0.91	-0.29	0.65
4	-2.04	-0.44	0.73
5	-9.33	-1.87	0.18
6	**-3.48**	**-1.83**	**0.04**

Table B-5 – Comparison of median median cyclomatic complexity of buggy clone classes with frequent co-change and buggy clone classes do not co-change frequently

1	**2**	**3**	**4**
App	$\mu(Cyclomatic\_med\_final\_BCF) - \mu(Cyclomatic\_med\_final\_BCI)$	$\frac{\mu(Cyclomatic\_med\_final\_BCF) - \mu(Cyclomatic\_med\_final\_BCI)}{\mu(Cyclomatic\_med\_final\_BCF)}$	P-Value
1	-1.36	-0.34	0.37
2	-3.51	-0.81	0.11
3	-1.25	-0.40	0.69
4	-2.54	-0.58	0.34
5	-13.05	-2.75	0.13
6	**-3.50**	**-1.75**	**0.04**

Table B-6 – Comparison of median final cyclomatic complexity of buggy clone classes with frequent co-change and buggy clone classes do not co-change frequently