EMPIRICAL INVESTIGATION OF CAUSES AND EFFECTS OF CODE CLONES

by

DEBARSHI CHATTERJI

JEFFREY C. CARVER, COMMITTEE CHAIR
JEFF GRAY
NICHOLAS A. KRAFT
RANDY SMITH
EUGENE SYRIANI

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2014
ABSTRACT

Code Clones, also known as Software Clones are similar code fragments mostly formed due to reuse of code. The literature is abundant with ambiguous and vague fundamental definitions of code clones. Over the years, researchers have shown increasing interest in code clones. However, most of the research lacks empirical validation. There is a dearth of empirical studies especially in the area of cause and effect. Often researchers have associated code clones with a negative connotation. However, there is little evidence to prove that code clones negatively affect the system. Although the research community unanimously agrees that it is critical to keep track of code clones, the available research is void of substantial efforts on maintenance related issues. Most efforts go into the software life-cycle process of maintenance. It is yet unknown how exactly code clones can affect the process of maintenance and this dissertation is a step in that direction. Good and bad coding practices, together give rise to code clones. Educating and providing assistance to developers in clone maintenance scenarios can save effort. A primary objective of this dissertation is to investigate developer behavior and ascertain ways to help developers during clone maintenance. Before reaching this goal, a major milestone to cross is, understanding the fundamentals of code clones. This dissertation proposes a ‘four pillar architecture’ with each pillar, namely - consistent definitions, causes and effects of clones, clone awareness, and clone management, focusing on questions closely related to the issues. For the purpose of answering the questions related to each pillar, this dissertation explains five research studies with respective empirical methods: systematic literature review, community survey, developer observation and
qualitative interview. Results highlight a degree of ambiguity in the literature and difference of opinion in the research community. The results also show that cloned code requires more effort to maintain, and given proper training and clone aware information, developers can be assisted. This dissertation also proposes a code clone categorization based on cloning intent with a classification of harmful and helpful clones.
DEDICATION

To Maa, Baba

To my lovely wife Debjani

To the rest of my family

To those special friends

&

To Sai Ram who said “Learn to give, Not to take. Learn to serve, Not to rule”
ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor, Dr. Jeffrey Carver, for the continuous support of my Ph.D study and research, for his patience, motivation, enthusiasm, and supervision. His guidance helped me in all the time of research and writing of this dissertation proposal. I could not have imagined having a better advisor and mentor for my Ph.D study.

Second, I would like to add a special thanks to Dr. Nicholas Kraft for advising me whenever I felt lost and reviewing my research from time to time.

Besides them, I would also like to thank the rest of my dissertation committee: Dr. Jeff Gray, Dr. Randy Smith and Dr. Eugene Syriani for their encouragement and assistance wherever I needed some.

My sincere thanks also goes to all the teachers and mentors from the Department of Computer Science at the University of Alabama, Tuscaloosa who have trained me, guided me and nurtured me for over four years.

I thank my fellow labmates in the Software Engineering group Ferosh Jacob, Aziz Naantaamornphong, Amiangshu Bosu, Dustin Heaton, Brian Eddy, Chris Corley and Amber Wagner for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we had in the last five years. Also I thank my friends in the Computer Science department, Gabriel Loween, Kazi Zunnurhain and Calvin Vreeland. It would not be out of place to mention all other fellow graduate students and friends who have played an important role in the last five years.
Last but not the least, I would like to thank my family: my parents Jogesh Chatterji and Gopa Chatterji, for giving birth to me at the first place and supporting me spiritually throughout my life.
# CONTENTS

ABSTRACT ............................................................... ii

DEDICATION ......................................................... iv

ACKNOWLEDGMENTS ................................................ v

LIST OF TABLES ...................................................... xii

LIST OF FIGURES ................................................... xiv

1 INTRODUCTION .................................................... 1

1.1 Classification of Code Clones .................................. 2

1.2 Problem Statement ............................................. 5

1.3 The Four Pillar Structure ....................................... 9

1.3.1 Consistent Definitions ..................................... 10

1.3.2 Causes and Effects of Code Clones ......................... 11

1.3.3 Clone Awareness ........................................... 12

1.3.4 Clone Management ......................................... 12

1.4 Plan of Action .................................................. 14

2 BACKGROUND AND LITERATURE REVIEW .................... 16

2.1 Detection Tools and Techniques ............................. 16

2.1.1 Text Based Detection Techniques ......................... 18

2.1.2 Token Based Detection Techniques ....................... 18

2.1.3 AST Based Detection Techniques ......................... 18
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1.4 PDG Based Detection Techniques</td>
<td>19</td>
</tr>
<tr>
<td>2.1.5 Metrics Based Detection Techniques</td>
<td>19</td>
</tr>
<tr>
<td>2.2 Performance Based Comparison of Detection Tools/Techniques</td>
<td>21</td>
</tr>
<tr>
<td>2.3 Visualization of Clone Aware Information</td>
<td>26</td>
</tr>
<tr>
<td>2.4 Developer Behavior and Related Studies</td>
<td>28</td>
</tr>
<tr>
<td>2.4.1 Retrospective Studies</td>
<td>28</td>
</tr>
<tr>
<td>2.4.2 Human-Based Studies</td>
<td>30</td>
</tr>
<tr>
<td>3 DEFINITIONS OF CODE CLONES AND INTENT BASED TAXONOMY</td>
<td>32</td>
</tr>
<tr>
<td>3.1 Ambiguous Structural Definitions</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1 Research Objectives</td>
<td>33</td>
</tr>
<tr>
<td>3.1.2 Selection of Source</td>
<td>33</td>
</tr>
<tr>
<td>3.1.3 Screening of Papers Returned from the Search</td>
<td>34</td>
</tr>
<tr>
<td>3.1.4 Results</td>
<td>37</td>
</tr>
<tr>
<td>3.2 Taxonomy Based on Cloning Intent</td>
<td>50</td>
</tr>
<tr>
<td>3.2.1 Study Overview</td>
<td>51</td>
</tr>
<tr>
<td>3.2.2 Data Analysis</td>
<td>54</td>
</tr>
<tr>
<td>3.2.3 Discussion</td>
<td>59</td>
</tr>
<tr>
<td>3.3 Summary</td>
<td>63</td>
</tr>
<tr>
<td>4 CLONE RESEARCH COMMUNITY BELIEFS - TWO SURVEYS</td>
<td>65</td>
</tr>
<tr>
<td>4.1 Research Topics</td>
<td>66</td>
</tr>
<tr>
<td>4.1.1 Research Topic 1: General Clone Usage Information</td>
<td>67</td>
</tr>
<tr>
<td>4.1.2 Research Topic 2: Clones and Developer Behavior</td>
<td>67</td>
</tr>
</tbody>
</table>
4.1.3 Research Topic 3: Clone Evolution .......................... 68
4.2 Survey Design and Analysis Process ............................ 68
4.3 Respondent Demographics ........................................ 71
4.4 Survey 1 .............................................................. 74
   4.4.1 General Clone Usage Information ............................ 76
   4.4.2 Developer Behavior ........................................... 78
   4.4.3 Evolution of Clones ........................................... 81
   4.4.4 Follow-up Survey .............................................. 82
4.5 Survey 2 .............................................................. 84
   4.5.1 General Developer Behavior ................................. 85
   4.5.2 How Can Developers Benefit from Clone Aware Tools? ... 88
   4.5.3 What Type of Clones should be Refactored? ............... 91
   4.5.4 Maintenance of Clones based on Location and Distance of the Fragments from Each Other ............................ 93
   4.5.5 Problems with the Current Widely Accepted Taxonomy of Code Clones .......................... 94
4.6 Discussion of Results Across Survey 1 and 2 ..................... 97
   4.6.1 Code Clone Taxonomy ........................................ 98
   4.6.2 Refactoring of Clones ....................................... 99
   4.6.3 Maintenance of Clones based on Location ................. 100
4.7 Conclusion .......................................................... 101
5 EMPirical human-based validation of clone maintenance ...... 104
   5.1 Defect-Localization (DL) Study ............................... 106
      5.1.1 Study Overview ........................................... 107
LIST OF TABLES

2.1 List of Clone Detection Tools and Validation ............................................. 20
2.2 Precision and Recall From Burd and Bailey’s Experiment ............................. 22
2.3 Summary of Bellon’s Tool Comparison ....................................................... 22
2.4 Summary of Koschke’s Tool Comparison .................................................... 23
2.5 Evaluation of Code Clone Detection Techniques ......................................... 25
3.1 Paper Distribution ....................................................................................... 38
3.2 Data Extraction Sheet ................................................................................. 39
3.3 Terminologies for Code Clones .................................................................. 45
3.4 Type to Terminology Mapping ................................................................. 46
3.5 Minimum Clone Thresholds ........................................................................ 48
3.6 Interview Questions .................................................................................... 55
3.7 Resources to Consult .................................................................................. 57
3.8 Mapping of Developers’ Intentions to Kapser’s Taxonomy ......................... 60
3.9 Associated Categories to Intent .................................................................. 61
4.1 Demographic Questions .............................................................................. 74
4.2 General Questions ....................................................................................... 76
4.3 Systematic Qualitative Analysis of Maintenance Related Section ............. 80
4.4 Systematic Qualitative Analysis of Evolution Related Section .................. 82
4.5 Issues and Related Studies ......................................................................... 87
## LIST OF FIGURES

1.1 Type I Clones ......................................................... 3  
1.2 Type II Clones ......................................................... 4  
1.3 Type III Clones ......................................................... 4  
1.4 Type IV Clones ......................................................... 4  
1.5 Four Pillar Structure ............................................... 10  
1.6 Background Studies Lay the Foundation for Further Validation ................. 14  
3.1 Screening of Papers ............................................... 35  
3.2 Taxonomy Hierarchy ............................................... 47  
3.3 Experience of Participants ........................................... 56  
3.4 Preference of Resources ............................................. 57  
4.1 Methodology of Survey and Data Analysis ................................... 70  
4.2 Primary and Secondary Areas of Research ................................... 71  
4.3 Familiarity Distribution ............................................. 73  
4.4 D2: Credibility of Responses ......................................... 74  
4.5 D3: Years of Experience ............................................. 75  
4.6 D4: Countries ......................................................... 75  
4.7 Agreement of Respondents on the Current Taxonomy From Survey 1 ............. 77  
4.8 Handling Clones Across Boundaries .................................... 95  
4.9 Key Limitations to Current Taxonomy .................................... 98  

xiv
4.10 Does Distance Between Fragments Matter? . . . . . . . . . . . . . . . . . . . . . 100

5.1 Defect 1 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

5.2 Defect 2 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 108

5.3 Format of the Clone Report . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110

5.4 Study Design . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 112

5.5 Observer’s Data Recording Form . . . . . . . . . . . . . . . . . . . . . . . . . . . 114

5.6 Defect Identification Form . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 115

5.7 Overview of Data . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 117

5.8 Positive Relation Between Number of Clones Found and the Clone Efficacy . . . . . 119

5.9 Positive Relation Between Number of Clones Found and the Clone Efficacy Excluding the Data Points for which Number of Defects Found Was Zero . . . . . . . 120

5.10 Clone Report Usage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

5.11 Correct vs. Incorrect Use of Clone Report . . . . . . . . . . . . . . . . . . . . . . 122

5.12 Pseudo Code for FB Code Snippet . . . . . . . . . . . . . . . . . . . . . . . . . . 135

5.13 Pseudo Code for PM Code Snippet . . . . . . . . . . . . . . . . . . . . . . . . . . 136

5.14 Clone Report Template . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 136

5.15 Clone Group for FrozenBubble . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

5.16 Clone Group for Pacman . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 137

5.17 Distribution of Participants (Replication) . . . . . . . . . . . . . . . . . . . . . . 139

5.18 Task Load Index . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 141

5.19 Times for All Participants (In Seconds) . . . . . . . . . . . . . . . . . . . . . . . . 141

5.20 Distribution of Participants (Extension) . . . . . . . . . . . . . . . . . . . . . . . 144

5.21 Code Clone Awareness . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150
INTRODUCTION

While writing code, software developers sometimes copy and paste code fragments in various locations in order to implement similar or nearly similar functionality [Kim, Bergman, Lau, and Notkin, 2004]. These similar-looking code fragments are called code clones (or software clones). In other words, code clones are exact or slightly modified code fragments throughout a software system. After a clone has been created, it is not always easy to detect which code fragment is the original code (parent clone) and which is the cloned code (child clone). Most clone detection tools provide information about the cloned pairs or, in some cases, the clone classes as output.

From the original mention of code clones in the research literature, they have been looked down upon as a “bad smell” [Fowler, 1999]. For a long time code clones have been considered harmful for software maintenance and hence considered an inferior way of writing code. Several studies in the literature point towards the difficulty of maintaining cloned code [Baker, 1995; Johnson, 1993]. A code clone is considered a more serious threat in industrial software systems than it is in academic/research software [Baker, 2007; Baxter, Yahin, Moura, Sant’Anna, and Bier, 1998; Casazza, Antoniol, Villano, Merlo, and Di Penta, 2001; Ducasse, Rieger, and Demeyer, 1999; Johnson, 1994; Kontogiannis, 1997]. However, the view on code clones has changed. Clones are no longer considered to be harmful [Kapser and Godfrey, 2006a; Kim, Sazawal, Notkin, and Mur-
phy, 2005; Thummalapenta, Cerulo, Aversano, and Di Penta, 2010]. However, everybody agrees that awareness of code clones is critical to the maintenance tasks.

Code clone detection has been a frequently visited research topic for the community researchers. There are various tools available based on various techniques used for clone detection [Basit, Rajapakse, and Jarzabek, 2005b; Baxter et al., 1998; Ducasse et al., 1999; Kamiya, Kusumoto, and Inoue, 2002; Kontogiannis, 1997; Koschke, Falke, and Frenzel, 2006; Krinke, 2001; Mayrand, Leblanc, and Merlo, 1996]. There also have been a few important tool comparison studies which shed some light on the performance of the tools on the basis of recall and precision in terms of number of clones found [Bellon, Koschke, Antoniol, Krinke, and Merlo, 2007; Bruntink, van Deursen, van Engelen, and Tourwe, 2005; Burd and Bailey, 2002; Koschke et al., 2006; Van Rysselberghe and Demeyer, 2004]. These studies make many claims about the tools and techniques and provide some empirical evidence to support those claims. However, there is little human-based empirical evidence to support these claims.

1.1 Classification of Code Clones

Code clones have been classified into various types based upon their similarity with another code snippet. A widely accepted view of code clones in the literature is that they can be textually similar [Bellon et al., 2007; Koschke et al., 2006] or functionally similar [Komondoor and Horwitz, 2003; Krinke, 2001; Mishne and Rijke, 2004]. Textually similar code clones tend to result from copy and paste actions. Functionally similar code clones are not necessarily textual copies, but can have very subtle variations. The range of similarity can vary between different levels of syntactic and semantic similarities.

Understanding the types of clones is crucial as there are different difficulty levels when it comes to the detection of code clones. There have been claims for various types of techniques and
tools being better for detection of specific types. Evidently, the textual clones are easier to detect than functional clones. It is to be noted that the types are mere classification based on structural similarities between fragments. Although, these classifications are frequently used, there is a good bit of confusion on the exact definitions and the concrete boundaries between classes. A commonly used classification of clones was put forward by Bellon et al. [Bellon et al., 2007] which was reused by Roy et al. [Roy, Cordy, and Koschke, 2009] as stated below:

**Type I:** Code fragments are identical except for variations in whitespace, layout, and comments (Figure 1.1).

**Type II:** Code fragments are structurally and syntactically identical except for variations in identifiers, literals, types, layout and comments (Figure 1.2).

**Type III:** Code fragments are copies with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments (Figure 1.3).

**Type IV:** Two or more code fragments perform the same computation but are implemented through different syntactic variants (Figure 1.4).

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

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

Figure 1.1: Type I Clones

While detection of Type I and Type II clones is based on matching similar text, for Type III and Type IV clones a match occurs at a higher level of structural and semantic similarity. It
if (a >= b) {
    c = d + b;  //Comment 1
    d = d + 1;
} else
    c = d - a;  //Comment 2

if (m >= n)
    { //Comment 1'
    y = x + n;
    x = x + 5;  //Comment 3
    } else
    c = d - a;  //Comment 2'

---

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

if (a >= b) {
    c = d + b;  //Comment 1
    e = 1;  //This statement is added
    d = d + 1;
} else
    c = d - a;  //Comment 2'

---

int i, j = 1;
for (i=1, i<=VALUE; i++)
    j=j*i;

int factorial(int n) {
    if (n==0) return 1;
    else return n*factorial(n-1);
}
requires a greater degree of normalization in order to report matches as we move on to the higher clone groups. It can be said that the order of the analytical complexity increases from lower to the higher types of clones.

1.2 Problem Statement

Estimates indicate that software maintenance costs account for as much as 70% to 90% of total software cost [Boehm and Basili, 2001; Erlikh, 2000]. Techniques and tools can reduce the effort spent by developers on these activities and reduce maintenance costs. One characteristic of a software system that can adversely affect its comprehensibility is the presence of code clones.

A common approach employed by developers while coding is to identify sections of code that can be copied, pasted and possibly edited, rather than writing completely new code [Kim et al., 2004]. Other developer activities lead to the introduction of code clones, including: language construct recurrence, pattern or paradigm adherence, and framework reuse. On average, five to twenty percent of a software system is cloned code [Uchida, Monden, Ohsugi, Kamiya, Matsumoto, and Kudo, 2005]. Many studies have found the cloned code percentage to be even higher. For example, CCFinder, a popular clone detection tool [Kamiya et al., 2002] determined that almost 30% of JDK v.1.3.0 is cloned code. CP-Miner, a tool for finding copy-paste defects, found that over 22% of the code in version 2.6.6 of the Linux kernel was cloned [Li, Lu, Myagmar, and Zhou, 2006]. In addition, one study reported over 59% cloned code in a COBOL payroll application [Ducasse et al., 1999].

There are conflicting views on whether code clones are problematic. Earlier work by Fowler classified code clones as a “bad smell” that would increase the difficulty of maintenance [Fowler, 1999]. However, more recent work indicates that code clones may not be as harmful as originally believed [Kim et al., 2004; Thummalapenta et al., 2010]. In fact, code clones may actually im-
prove productivity by the use of “spring-boarding” and reusing robust and trusted code [Kapser and Godfrey, 2008]. Rahman et al. found little empirical evidence that clones negatively affect software maintainability, but did find that cloned code may be less fault prone than non-cloned code [Rahman, Bird, and Devanbu, 2010]. There is agreement that a developer’s awareness of code clones is critical for performing correct (and complete) software maintenance.

Code clone analysis techniques and tools are popular topics among the software engineering research community. Many studies draw conclusions solely based on an analytical analysis. These studies derive conclusions based on number comparison. Most of the time these studies compare parameters such as time taken to complete a process to parameters such as success, without considering other compounding factors that might affect the causality. These claims focus primarily on tool performance in terms of portability, scalability, robustness, precision, and recall. However, these types of analytical studies cannot adequately evaluate the behavior of the developers while using the tools. Human-based empirical studies are complementary to studies based on analytical data because they provide direct insight into developer behavior.

A primary goal of code clone research is to provide techniques and tools to help developers correctly and efficiently maintain source code that contains clones. Additional data about the clones is supposed to assist developers who must make decisions regarding these clones during software maintenance tasks. There are a few extensive tool evaluation studies in the literature, some of which are discussed later in Chapter 2. While these studies can rightfully make claims about properties of techniques and tools, they also tend to make claims about the behavior of the developer using the tool.

Due to the lack of human-based empirical studies in the literature, it appears that creators of clone detection approaches assume that if a developer is provided with the report from a code
clone detection tool, they will know how to use it to perform various software engineering tasks. In searching the literature, there were very few human-based empirical studies that verify the usefulness of the information provided in the code clone reports that are output from the tools.

Also, there is a substantial level of uncertainty regarding the fundamental definitions of code clones. This ambiguity is evident from the numerous definitions and terms that define and classify code clones [Baker, 1995; Basit and Jarzabek, 2009; Deissenboeck, Hummel, Jürgens, Schätz, Wagner, Girard, and Teuchert, 2008; Kim et al., 2005; Komondoor and Horwitz, 2001a; Lague, Proulx, Mayrand, Merlo, and Hudepohl, 1997; Martin and Cordy, 2011; Ueda, Kamiya, Kusumoto, and Inoue, 2002]. It appears that code clone research with its fast pace of moving forward has overlooked these fundamental issues. One of them is defining clones with more clarity. The lack of consistent definitions has impacted the opinions of the researchers for a long time. This problem has led to a considerable amount of concern in the code clone research area for the research community. Clarifying the ambiguity in this area would set standards for researchers to follow in future research. If as a community the clone researchers can arrive at standard definitions, arguably, it will be easier to focus on advantages or disadvantages of the standardized clones and types. Consequently, a high level research objective of this dissertation is to:

Concretely define code clones in order to better understand their benefits and drawbacks regarding software quality.

Moving on to the next problematic area in the field of code clone research, it is difficult to judge the causes and effects of clones in the practical field of development and maintenance. There
is little to no evidence or indication about the developer’s analysis and decision-making process regarding code clones.

Claims about human behavior cannot be fully validated or understood without the use of human-based empirical studies. Human-based empirical studies have commonly helped to understand developer behavior with regards to other aspects of software engineering. There is an entire discipline of empirical software engineering, which has a conference (Empirical Software Engineering and Measurement), a journal (Empirical Software Engineering) and a number of handbooks (e.g., [Juristo and Moreno, 2003]). Many types of human-based empirical studies have proved to be useful in different settings and for addressing different types of research questions. These types of studies range from very controlled laboratory-type studies to case studies which allow for naturalistic observation of behavior in practice, with many options and combinations in between.

Pate et al. conducted a systematic review of the literature on clone evolution [Pate, Tairas, and Kraft, 2013]. Of the 30 primary studies discussed in the review, 27 include empirical studies that involve software repository mining and source code analysis. In these 27 studies, the authors focus primarily on analytical findings regarding the efficacy of a tool, the characteristics of a subject software system, or the characteristics of a collection of subject software systems. However, some papers also include claims about developer behavior and intent. Until human-based empirical studies validate these claims, they still contain some uncertainty. For example, Hou et al. [Hou, Jablonski, and Jacob, 2009] describe CnP, a tool for proactive management of clones. CnP tracks clones from creation and provides features meant to facilitate clone evolution, including visualization, consistent renaming, and simultaneous editing. However, Hou et al. do not present a human-based empirical study of CnP. The (seemingly reasonable) assumption that the tool is us-
able and helpful for developers should be validated with such a study. This example stresses on the important factor of recording a developer’s intention during clone formation or clone maintenance.

Consequently, another high level objective of this dissertation is:

To investigate developer behavior during code clone formation and maintenance of reused code.

The next section discusses the roadmap that I propose to understand and investigate as a solution to the above problems. The roadmap focuses on clarifying the vagueness and ambiguity related to code clones.

1.3 The Four Pillar Structure

It is an established fact that cloning code is a common software development practice. Subsequently, it can be assumed that there would be clones in a system. Also, it is essential that developers keep track of code clones, especially in the parts of the code that they actively maintain. As a result, clone management practices become critical for developers. Hence, there is a need to simplify the overall process. For better understanding, we can compare with other software engineering issues such as robust and readable coding/system design [Boswell and Foucher, 2011]. The system design and architecture issues have been simplified by following guidelines which are commonly known as design patterns [Gamma, Helm, Johnson, and Vlissides, 1995]. Similarly, my research is directed towards the effects of code clones on software maintenance and towards devising an easier approach for the developers to manage them.

For my research, I followed a four pillar approach as shown in Figure 1.5, to attain the objectives of this dissertation work. The primary disposition for following the four pillar structure
is based upon my argument of providing balanced importance to all aspects of this field of research. The aspects or pillars are explained as follows.

![Figure 1.5: Four Pillar Structure](image)

1.3.1 Consistent Definitions

Essentially, code clones can be categorized under two broad categories, code fragments with similar text and code fragments with similar logic. There have been other definitions of code clones in various studies which have a degree of overlap with each other. Concrete and unambiguous definition of code clones and their types are required for further understanding the shortcomings and loopholes in the current research.

Various researchers have defined code clones differently and with a certain degree of ambiguity. For example there is not enough clear indication related to size and form of the code clones in the literature. Various studies have considered various threshold sizes for assigning snippets as code clones. Further, the threshold sizes have been measured using different metrics, for example number of tokens and lines of codes. Hence, there is a need to classify code clones into distinct
categories with concrete boundaries. It is also important to make sure that the vagueness of such
definitions are minimized.

Removing such vagueness would result in better understanding of code clones and further
the cause of understanding benefits and drawbacks of code clones. Additionally, it would also help
further the cause of code clone detection as strict boundaries of categories would in turn make the
process of detection, relevant and comparable.

My research questions related to this pillar are:

*RQ1.* What are the various definitions of code clones and how are code clones categorized?

*RQ2.* What type of clones are considered harmful to software quality?

*RQ3.* What are upper and lower bounds on the size of two fragments that can be considered as
clones?

1.3.2 Causes and Effects of Code Clones

Code clones occurrence is part of software development. Over the years, there have been
a few valuable insights into the intention of developers giving rise to clones in a software system.
Developers, in general, tend to use and reuse fragments. The decision to reuse code can be influ-
enced by a number of reasons. These reasons can be related to time constraints, re-usage because
of a piece of code works or because of the complexity of the part to be replicated elsewhere in
the system. Most of the reasons at this point can only be speculatively related to certain causality
between two different attributes out of a few, such as time, complexity or coding styles. De-
tailed verification is required to determine the presence of possible confounding variables. There
is a high possibility of these confounding variables influencing such practices. Since studies have
shown that cloning can be beneficial, it is critical to understand the effect of all such intentions.

My research questions related to this pillar are:
**RQ4. Why do developers reuse code?**

**RQ5. How are clones helpful/harmful?**

**RQ6. What preventive measures can minimize the bad effects arising from harmful practices?**

1.3.3 Clone Awareness

In order to build better tools for clone maintenance, there is a need for a clear understanding of code clones and their classifications. There are many claims that particular tools and techniques perform better in specific controlled environments [Baker, 2007; Bellon et al., 2007; Roy and Cordy, 2008a]. However, these studies lack evidence and are mere indications. Also, interestingly, there have not been many human-based empirical studies comparing the various tools and techniques to provide concrete evidence in support of the analytical claims. It is critical to understand the use of the clone aware information acquired from tools. Clone aware tools can generate a bulk of information. Some of this information can be misleading. If there is no clear way to use this information, building such tools can be proved less useful. ‘What to detect?’ and ‘How to present?’ keeping the developer as the subject, are two important questions to focus on.

My research questions related to this pillar are:

**RQ7. What assistance can be provided with clone aware information?**

**RQ8. How can developers use clone aware information to their maximum benefit?**

1.3.4 Clone Management

Much emphasis has been given to the research about the beneficial and harmful effects of code clones on a system. I argue that it is equally important to scrutinize the harmful effects related to the intent and rationale of clone formation by developers while coding or during maintenance activities.
The positive and negative effects of cloning can be connected to the intention of developers. Negative intentions such as duplicating code and in some cases hard coding rather than applying a proper abstraction are examples of intentional clones. Similarly, reusing code because of limited understanding of the code fragment gives rise to intentional clones. These types of intentional clones can be considered harmful for the software system. Such clones can give rise to unnecessary coupling and propagate defects throughout the system. However, in some cases intentional cloning can prove to be beneficial. Intentional cloning of robust code architectures with desired behavior, such as design patterns, can improve the readability and the overall quality of the system. Such practices are called ‘Templating’. Another example of good practices of intentional cloning is reusing a tested older version or a platform equivalent as a springboard for the software development. A lot of time and effort can be saved rebuilding the same architecture from scratch. However, in such cases there is always a risk of propagating defects to the parts of the reused code.

Unintentional clones pose a greater threat to the quality of the system. It is to be noted that it does not imply that unintentional clones are always harmful. However, it can be assumed that due to their obscure reasons of occurrence they are very easy to be overlooked. Unintentional clones can be overlooked easily as the creator is seldom aware of the fact that she or he is cloning. Programmers often use their experience to solve specific problems in specific ways. In other cases, developers have programming idioms and recurring constructs that they frequently use. Such practices can give rise to unintentional clones

Awareness of developers about the clones they create is essential. Keeping a track of them and making decisions related to clone maintenance is critical to the amount of effort. Better practices of keeping track of intentional and unintentional clones should be progressively adopted by developers.
My research questions related to this pillar are:

**RQ9.** How to manage clones after developers are aware of their existence?

**RQ10.** Which clones should be refactored, kept or encouraged?

### 1.4 Plan of Action

To attain the high level objectives of the dissertation, I have planned two types of studies. First, studies that lay the foundation of the research. Second, focused empirical validation studies that build upon the foundational studies in order to find more specific answers. Figure 1.6 shows the overview and represents the parts played by foundational studies laying foundation for the empirical validation studies.

As can be seen in the figure, most of the results are drawn from the human-based studies. The foundational studies lay a solid foundation for describing the goals and identifying the gaps in this area of research.
Until now I have tried to establish the need of empirical studies, especially human based, in the field of code clones, which is also the ultimate goal of this dissertation. Empirical evidence indicates that developers create, edit, and remove clones when maintaining source code. An understanding of developer behavior with regard to these operations is needed to produce techniques and tools that effectively address the clone-related issues which developers encounter. Further, the manner in which developers engage these techniques and tools should influence their design. To truly understand human behavior and to validate the claims about it, human-based empirical studies must be performed.

The next chapter helps the reader understand the background research in the related areas. Chapters 3 and 4, describe the foundational studies in detail. Chapter 5 presents the focused human-based empirical validation studies in detail. Finally, Chapter 6 holds the summary of the findings and conclusion based on them.
BACKGROUND AND LITERATURE REVIEW

This chapter discusses work related to this dissertation and to the code clone research area in general. The chapter starts with a brief summary of the available code clone detection literature, followed by some of the available ways of representing clones in a software system. This chapter also discusses some of the closely related empirical studies and the results indicated by them.

2.1 Detection Tools and Techniques

Any information from a software system related to code clones can be called clone aware information. For clone maintenance and management in a software system, some sort of clone aware information must be provided to a developer. Most of the clone aware information in a software system is provided by the detection tools and techniques. Without this information it is almost impossible to keep a track of clones and manage. Hence, before this dissertation goes into further discussion about other code clone related issues, it is important to understand the basic clone detection approaches. This section presents a review of the progress in the design and construction of various tools and techniques used in code clone detection. The motivation behind construction of a better tool is to make it helpful for a developer during maintenance tasks. Most of the detection tools provide the clone aware information in the form of a clone report. A clone report is usually in the form of a list of cloned pairs and classes, which in case of a large sized software system, can be large and unmanageable. Building better tools to aid programmers is an important part of the software engineering research. However, in the case of code clone detection tools which
provide clone aware information as an output, there is a chance of making the user overwhelmed with information [Tairas and Gray, 2006]. Therefore it is very important to make sure that the reports are easily readable, meaningful and not overloaded with clone aware information such as to make it tough for developers to find the relevant information.

Detection of code clones is essentially a two phase process. In the first phase the fragments are transformed into an internal structure based on the technique being used. Once the transformation is done, the actual comparison phase begins. All similar code fragments are classified into a clone classes/groups with each clone class being identified using a unique identifier, e.g. clone ID.

The clone detection techniques differ from each other on the basis of normalization of code, representation of the normalized code, granularity of comparison, algorithm used to compare the code snippets and compatibility with the language used. Code clone techniques are usually classified according to the internal source code representation.

The conventional approaches have their advantages and disadvantages, as described in Section 2.2. However, there are some hybrid techniques which use a combination of conventional techniques to use the advantages and remove some of the drawbacks. The primary motivation is to overcome the shortcomings of each conventional approach and build on its strengths. There are a few studies in the literature which describe hybrid techniques/tools and compare them to tools from the conventional approaches [Balazinska, Merlo, Dagenais, Lague, and Kontogiannis, 1999; Greenan, 2005; Jiang, Misheghi, Su, and Glondu, 2007; Jiang and Hassan, 2007; Koschke et al., 2006; Roy and Cordy, 2008a; Synytskyy, Cordy, and Dean, 2003; Tairas and Gray, 2006]. The primary focus of this dissertation not being code clone detection, it is reasonable not to go into further details of hybrid techniques. Hybrid techniques are are based on complex hybrid algorithms combining basic detection techniques. However, It is important to have a firm grasp on the basic
clone detection techniques and tools as it would help the readers to gain an overall understanding of code clones.

Given the interest in building new and better detection techniques, it is out of scope of this dissertation to list all of them. However, the following subsections discuss some of the basic detection techniques and also some popular tools, based on their use in comparison studies, that follow the respective approaches.

2.1.1 Text Based Detection Techniques

In this approach, the target source program is considered as sequence of lines/strings. Fragments of text/strings are compared with each other for similarity. The detection technique reports code clones based on the level of similarity and are documented as clone pairs or clone classes. Baker’s DUP [Baker, 1999] is an example of a lexical (text based) analyzer.

2.1.2 Token Based Detection Techniques

A token-based detection approach parses the entire source system into a sequence of tokens, followed by a scan to find duplicated sub-sequences of tokens. Finally it returns the sub-sequences that match the original code portion as code clones. Some studies claim that token-based tools are more robust than text-based tools [Bellon et al., 2007; Koschke et al., 2006; Rysselberghe and Demeyer, 2003]. Token-based tools are highly precise and have a higher recall rate. CCFinder [Kamiya et al., 2002] is the most well-known token-based clone detector.

2.1.3 AST Based Detection Techniques

In the tree-based approach, a parser for the language of interest pairs a program to a parse tree or an abstract syntax tree (AST). Some tree matching techniques search for similar sub-trees in the tree and the corresponding source code of the similar sub-trees is returned as clone pairs or classes. The parse tree or AST contains the complete information about the source code. Although
the variable names and literal values of the source are lost in the tree representation, more sophis-
ticated methods for the detection of clones still can be applied. Clone DR [Baxter et al., 1998] is
one of the well-known and more frequently used clone detection tool.

2.1.4 PDG Based Detection Techniques

Program dependence graphs (PDG) for the subject programs produce the required semantic
information. Once the PDGs are obtained, isomorphic sub-graph matching algorithms are applied
to look for similar patterns. PDG-DUP [Komondoor and Horwitz, 2001b] is one of the tools based
on this approach. PDG based approaches [Komondoor and Horwitz, 2001b; Liu, Chen, Han, and
Yu, 2006; Prechelt, Malpohl, and Philippsen, 2002] are considered to be better for detection of
Type 3 clones.

2.1.5 Metrics Based Detection Techniques

In metrics-based approaches, the tools define metrics for code fragments, based on the AST
or PDG representation of the code. The tools then use those metrics for comparison rather than
using the code itself. Kontogiannis et al. [Demori, Kontogiannis, Galler, and Demori, 1995], Buss
et al. [Buss, De Mori, Gentleman, Henshaw, Johnson, Kontogiannis, Merlo, Müller, Mylopoulos,
Paul, Prakash, Stanley, Tilley, Troster, and Wong, 1994] and Dagenais et al. [Dagenais, Merlo,

There are some empirical validation studies reported in the related literature comparing the
performance of tools. However, these studies fail to provide any conclusive evidence of any kind
related to usability of tools, particularly, when the usability is in the context of practical use rather
than detecting clones for academic purposes. Table 2.1 provides a general overview of the tools
that have been evaluated in the literature.

The following sections describe the tool evaluation and empirical validation studies. The
<table>
<thead>
<tr>
<th>Tool</th>
<th>Technique</th>
<th>Supported languages</th>
<th>Lines of input</th>
<th>Validation</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dup</td>
<td>Text</td>
<td>C, C++, Java</td>
<td>1.1 MLOC</td>
<td>With two systems</td>
<td>Baker [Baker, 1995]</td>
</tr>
<tr>
<td>DupLoc</td>
<td>Text</td>
<td>Language Independent</td>
<td>46K LOC</td>
<td>With 4 systems of different languages</td>
<td>Ducasse et al. [Ducasse, Nierstrasz, and Rieger, 2006; Ducasse et al., 1999]</td>
</tr>
<tr>
<td>JPlag</td>
<td>Token</td>
<td>Java, C, C++, Scheme, NL text</td>
<td>236 KLOC</td>
<td>Student assignments/artificial data</td>
<td>Prechelt et al. [Prechelt et al., 2002]</td>
</tr>
<tr>
<td>CCFinder</td>
<td>Token</td>
<td>C, C++, Java, COBOL</td>
<td>10M LOC</td>
<td>With 4 systems</td>
<td>Kamiya et al. [Kamiya et al., 2002]</td>
</tr>
<tr>
<td>i-clones</td>
<td>Token</td>
<td>C, C++, Java, COBOL, VB</td>
<td>235K LOC</td>
<td>With 4 systems and 8 tools</td>
<td>Koschke [Koschke et al., 2006]</td>
</tr>
<tr>
<td>CloneDr</td>
<td>AST</td>
<td>C, C++, Java, COBOL</td>
<td>400K LOC</td>
<td>Process Control System</td>
<td>Baxter et al. [Baxter, Pidgeon, and Mehlich, 2004; Baxter et al., 1998]</td>
</tr>
<tr>
<td>Deckard</td>
<td>AST</td>
<td>C, Java</td>
<td>5,287K LOC</td>
<td>With Linux kernel and JDK</td>
<td>Jiang et al. [Jiang and Hassan, 2007]</td>
</tr>
<tr>
<td>Tairas’ Tool</td>
<td>AST</td>
<td>C, C++, C-Sharp</td>
<td>1500K LOC</td>
<td>With several systems</td>
<td>Tairas et al. [Tairas and Gray, 2006]</td>
</tr>
<tr>
<td>Duplix</td>
<td>PDG</td>
<td>C</td>
<td>25K LOC</td>
<td>Several test cases</td>
<td>Krinke et al. [Krinke, 2001; Krinke and Breu, 2004]</td>
</tr>
<tr>
<td>Covet/CLAN</td>
<td>Metric</td>
<td>C</td>
<td>507K LOC</td>
<td>With two telecommunication systems</td>
<td>N. Mayrand [Mayrand et al., 1996]</td>
</tr>
</tbody>
</table>
discussion will focus on the lack of empirical validation which is more closely related to the problem statement of this dissertation.

2.2 Performance Based Comparison of Detection Tools/Techniques

In general, the two dimensions: sensible results and effective output, can broadly measure the effectiveness of a tool. The motivation behind this section is to understand the effectiveness of various clone detection techniques and tools and to suggest ways of making the tools more useful for developers. Most of the studies discussed in this section are based on analytical evidence. Although, analytical data can indicate towards performance based results, it is hard to gauge usability without human-based empirical studies. Human-based studies help instill more faith in beliefs and provide additional support for the analytical data.

There are a few tool evaluation experiments in the literature, which compare techniques based on: portability, scalability, precision and recall. The reminder of this discussion focuses on precision and recall. Precision accounts for a low percentage of false positives and recall accounts for a low percentage of false negatives. The following studies are, in chronological order, Bailey and Burd [Burd and Bailey, 2002], Bellon et al. [Bellon et al., 2007], Koschke et al. [Koschke et al., 2006], Rysselberghe and Demeyer [Rysselberghe and Demeyer, 2003]. All of the papers discussed below provide an analytical comparison of tools. These studies have a set of tools based on different techniques which are tested in a controlled environment. Once the experiments have been run, the analytical data was compared to draw some conclusions.

Burd and Bailey’s experiment [Burd and Bailey, 2002] compares three clone detection tools and two plagiarism tools based on recall and precision. I have left the two plagiarism tools out of the discussion as in this review the focus is on clone detection tools only. The system used for this experiment was a medium-sized system, which turned out to be a threat to validity for the
Table 2.2: Precision and Recall From Burd and Bailey’s Experiment

<table>
<thead>
<tr>
<th>Technique</th>
<th>CC Finder</th>
<th>CloneDR</th>
<th>Covet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>72%</td>
<td>100%</td>
<td>63%</td>
</tr>
<tr>
<td>Recall</td>
<td>72%</td>
<td>9%</td>
<td>19%</td>
</tr>
</tbody>
</table>

Table 2.3: Summary of Bellon’s Tool Comparison

<table>
<thead>
<tr>
<th>Technique</th>
<th>Tool</th>
<th>Precision</th>
<th>Recall</th>
<th>RAM</th>
<th>Speed</th>
<th>Type1</th>
<th>Type2</th>
<th>Type3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>CloneDr</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>X</td>
</tr>
<tr>
<td>Token Dup</td>
<td>DUP</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Token CCFinder</td>
<td>-</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>PDG</td>
<td>Duplix</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td>-</td>
<td>-</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Metrics</td>
<td>CLAN</td>
<td>+</td>
<td>-</td>
<td>++</td>
<td>++</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Line</td>
<td>Duploc</td>
<td>-</td>
<td>+</td>
<td>?</td>
<td>?</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

study and its results being questionable. Results from Burd and Bailey’s experiment are shown in Table 2.2.

From Table 2.2 it can be seen that the precision of CloneDR, which is an AST based tool, is 100% which means that there were no false positives. However, the recall is too low for it to be used in situations where finding a high percentage of the clones is critical. In comparison to the precision levels, Covet, the metric-based tool, seems to be good in terms of precision but its low recall rate makes it vulnerable. CC Finder, the token-based tool, is the most appropriate tool for clone detection when both factors are taken into account. Its high recall value and reasonably good precision rate make CCFinder the clear winner in this study. However, this study had its own shortcomings, as discussed above.

Bellon et al. [Bellon et al., 2007], performed a large-scale experiment comparing six different multilingual tools. The comparison was rigorous as they tested the tools using four Java and four C software systems. The results from that study are shown in Table 2.3. The data is reported at an ordinal scale –, -, +, ++ where – is worst and ++ is best.
Table 2.4: Summary of Koschke’s Tool Comparison

<table>
<thead>
<tr>
<th>Technique</th>
<th>Tool</th>
<th>Precision</th>
<th>Recall</th>
<th>Number of Candidates</th>
<th>Running Time</th>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>CloneDr</td>
<td>++</td>
<td></td>
<td>1434</td>
<td>++</td>
<td>-</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AST</td>
<td>ccdiml</td>
<td>+</td>
<td>+</td>
<td>18245</td>
<td>--</td>
<td>++</td>
<td>+--</td>
<td>-</td>
</tr>
<tr>
<td>Token</td>
<td>Dup</td>
<td>+</td>
<td>+--</td>
<td>8978</td>
<td>+--</td>
<td>++</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>Token</td>
<td>CCFinder</td>
<td>+</td>
<td>+--</td>
<td>18961</td>
<td>+</td>
<td>++</td>
<td>+--</td>
<td>-</td>
</tr>
<tr>
<td>Token</td>
<td>clones</td>
<td>+</td>
<td>--</td>
<td>32975</td>
<td>--</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Token</td>
<td>cscope</td>
<td>-</td>
<td>+--</td>
<td>17758</td>
<td>+--</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>ASST</td>
<td>cpdetector</td>
<td>+</td>
<td>+--</td>
<td>4852</td>
<td>+</td>
<td>++</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>PDG</td>
<td>Duplix</td>
<td>-</td>
<td></td>
<td>12181</td>
<td>++</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metrics</td>
<td>CALN</td>
<td>+++</td>
<td></td>
<td>318</td>
<td>+--</td>
<td>++</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Line</td>
<td>Duploc</td>
<td>+</td>
<td>-</td>
<td>5212</td>
<td>?</td>
<td>++</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

The most interesting fact that stands out from the experiment is that precision and recall ratings highly complement each other for each tool. It can be summarized that while making a choice of a clone detection tool, a developer has to decide on which property is more important in the context of the scenario at hand. From the above discussion, it is apparent that CCFinder is the best tool.

Koschke et al. [Koschke et al., 2006] used the tools from Bellon et al. [Bellon et al., 2007], with slight modifications to fit the model of his experiment, and compared the results with his own developed AST based tool. The results are shown in Table 2.4. The data is reported at an ordinal scale of --,+-,+, ++, +++ where – is worst and +++ is best.

Apparently, this experiment had similar results to those in Bellon’s experiment. The token-based tools tend to find the most clones. However, they are not as good as the AST or Metrics-based tools when it comes to precision. The PDG-based tools are the best for finding Type 3 clones.
Table 2.5 summarizes the findings of the reported studies about the evaluation of clone detection techniques on the basis of the two most important properties precision and recall (High: 90% to 100%, Medium: 70% to 90% and Low: below 70%).

To conclude, it can be stated that there are no clear winner in terms of technique and tool used. However, a clever selection of a technique and a tool based on it can be useful depending on the job at hand. There is an open opportunity for development of tools based on hybrid techniques where the shortcomings can be improved keeping the strengths of the techniques. The results from the analytical data can be summarized as:

- Text-based tools are not very good at finding clones, except for Type 1, though they are very precise for detecting exact clones.

- Token-based tools are best at recall and reasonably good at precision, but they are not very efficient in detecting functional clones.

- PDG-based tools are efficient in finding Type 3 clones but lack precision and recall.

- AST-based tools have a high precision with Type 1 clones and are helpful in refactoring purposes. Cannot be used where recall rate is crucial.

- Metrics-based tools can execute fast with consumption of minimal memory.

All of these results are based on analytical data, which is crucial in suggesting trends and making claims. However, there is a need for conclusive evidence on the grounds of efficacy of these tools for software maintenance. Human-based empirical studies can provide that much needed evidence that can support or refute the claims made on basis of the analytical data. In this dissertation,
<table>
<thead>
<tr>
<th>Approach</th>
<th>Precision</th>
<th>Recall</th>
<th>Discussion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Line-Based</td>
<td>100%</td>
<td>Low</td>
<td>High precision but low recall. Not good for detection of Type 3 clones. 100% confidence in result as text matching is targeted over only the exact matches. Because of high precision can be helpful in refactoring tasks.</td>
</tr>
<tr>
<td>Token-Based</td>
<td>Medium</td>
<td>High</td>
<td>Best approach according to the empirical data available. Only apparent shortcoming is in detection of Type 3 clones. It has a high recall with an equally good precision rate only missing 28% of the clones in the system. However token-based tools have high recall but low precision. Good in terms of memory usage and running time</td>
</tr>
<tr>
<td>AST-based</td>
<td>High</td>
<td>Low</td>
<td>Generally considered to not produce false positives, very high on precision. Recall is very low, so not considered suitable for clone detection in a system where the percentage of hits is crucial. Considered to be not effective for Type 3 clones. Suffix tree approach is claimed to be the best for refactoring tasks.</td>
</tr>
<tr>
<td>PDG-Based</td>
<td>Low</td>
<td>High (for Type 3), Low otherwise</td>
<td>According to Bellon. PDG-based tools are incompetent, both in case of precision and recall. Also bad in terms of speed. Not good for detecting Type 1 and Type 2 clones. However, in the evaluation literature, it has been claimed to be the best for detecting Type 3 clones.</td>
</tr>
<tr>
<td>Metrics-Based</td>
<td>Low</td>
<td>Medium</td>
<td>Good in finding clones (recall) but very low in precision, which would mean lots of false positives. Usually considered to have high execution speed and low memory consumption. Worst in finding candidate clones. They provide the duplicate code and scope of the fragment in contention.</td>
</tr>
</tbody>
</table>
I will investigate the efficacy of such clone aware information and measure the effects when this information is provided to developers during maintenance.

2.3 Visualization of Clone Aware Information

Any clone related information in the form of an artifact, such as a clone report in CC-Finder [Kamiya et al., 2002] or in the form of a live clone tracking information environment such as CnP [Hou et al., 2009], can be called clone aware information. Different tools use various methods to provide such information to the user. A good tool should complement a good detection result with an effective output visualization that makes it easier for the user while working with the output. As a starting point, the following section summarizes some of the visualization methods of clone aware information provided by tools. Please note that these are some of the most widely used methods and not a complete list.

The most common clone visualization method uses a textual format to output the clone aware information. The reports usually contain a list of clone classes or pairs with information about the specific location of the code fragments in terms of file, function/method and line numbers. In general the reports do not carry information about the types of clones, the degree of similarity, granularity or size. A potential issue with text based visualization lies in the verbose nature of the reports. As seen in Kasper and Godfrey [Kapser and Godfrey, 2005] in large software systems there can be as many as 13,000 clone pairs reported. Such a large amount of information could be overwhelming and hard to deal with.

Church and Helfman [Helfman, 1996] proposed another famous approach of clone visualization, the scatter plot. It is a common practice to use a 2-D charts to represent the software units on the axes and a dot representing two similar code fragments. There is an issue of scalability for large software systems in expressing the clone pairs/classes using this approach. Higo et al. [Higo,
Ueda, Kamiya, Kusumoto, and Inoue, 2002] proposed an enhanced scatter plot to overcome this shortcoming. An enhanced scatter plot can automatically filter out uninteresting clones and also provide the directory span of a cloned class.

Johnson [Johnson, 1994] proposed another approach for expressing the relationship between clones by using Hasse Diagrams. A Hasse diagram consists of nodes and edges. For each of the clones and its related cluster of files, the copied source text and the source files are shown as nodes and the relation between clones are shown as edges. The height of a node in the graph is determined by its size, the large files or code segments are towards the bottom while similar segments of code towards the top.

Over the years, evident from the frequent use in tool comparison studies, CC-Finder [Kamiya et al., 2002] has been a favored choice of the academic research community. Some studies have reportedly used the output provided by CC-Finder to visualize it in a more efficient way. Gemini and Aries [Higo, Kamiya, Kusumoto, and Inoue, 2005] visualize the output of CCfinder as a metric graph and queries which makes browsing through the list of clones easier. Recall that CC-Finder does not differentiate between the types of clones. Kasper and Godfrey. [Kapser and Godfrey, 2005] developed a comprehension tool, CLICS, which used the output from CCfinder to provide a detailed report of the types of clones in the system. CLICS also provides a query-based approach for developers to utilize the information from the detection tool. Jiang et al. [Jiang, Hassan, and Holt, 2006] mine the clone information from the clone classes generated by CC Finder. They provide an interactive representation of the cloned data to query the cloned candidates.

Some approaches, for example in CloneDR [Tairas and Gray, 2006], integrate the features of the Eclipse IDE for providing the output. In another approach, Adar and Kim [Adar and Kim, 2007] formulated SoftGUESS (extension of GUESS [Adar, 2006]) to support the visualization of
code clones. The visualization consisted of a code library with a few mini applications supporting the analysis of code clones with respect to system dependencies.

One aim of this dissertation is also to gauge the efficacy of the clone aware information provided to developers in maintenance scenarios. The visualization techniques listed above are merely a way to project the information. The degree of usability of these techniques is yet to be validated.

2.4 Developer Behavior and Related Studies

A very important objective of this dissertation is to investigate the effects of clones on development scenarios. Hence, developer behavior becomes a very important factor towards the outcome of this dissertation. It will not be out of place to mention the developer behavior related studies in this section to focus the discussion towards what empirical results are available and what is yet to be validated.

Two primary methods adopted in the software engineering field to gauge developer behavior are through retrospective studies and user based human interaction studies. Although the later chapters of this dissertation discuss the differences in detail, it will be befitting to at least report the findings briefly to some of the related developer-oriented studies.

2.4.1 Retrospective Studies

Bettenburg et al. [Bettenburg, Shang, Ibrahim, Adams, Zou, and Hassan, 2009] indicated in their findings, how developers manage long-lived clones. They reported that the majority of long-lived clones in Apache Mina and jEdit are instances of the replicate and specialize pattern, in which code is cloned and then customized to implement a new feature. Moreover, the authors found that errors to replicate-and-specialize clones were not introduced at a high rate by inconsistent changes.
Based on this finding, the authors concluded that developers of both Mina and jEdit are aware of those long-lived clones and are able to effectively manage their independent evolution.

Balint et al. [Balint, Marinescu, and Girba, 2006] performed a retrospective study in which they created an author-centric view of clone evolution in Apache Ant, ArgoUML, and Ptolemy II. Although the authors did not directly observe developers, they mined software repository data to track the activities of project developers over time. By analyzing visualizations of the clone evolution data, Balint et al. identified five cloning activity patterns: (1) consistent line/block cloning with unique author, (2) creation of clones by multiple authors using consistent block cloning, (3) consistent line/block cloning with multiple authors, (4) inconsistent line cloning fixed by same author, and (5) inconsistent line cloning fixed by different authors. The results indicated that the rate of detection of inconsistent changes correlated with the number of developers. Thus, the specific developers involved should be considered when analyzing code clones.

Other researchers have studied the potential impacts that code clones can have on the software development and maintenance process. Thummalapenta et al. [Thummalapenta et al., 2010] claim that developers tend to consistently propagate clone changes immediately where needed. Krinke et al.’s [Krinke, 2007] study of open source software indicated that half of the changes to code clone groups are inconsistent with each other. A study by Cordy et al. [Cordy, Dean, and Synytskyy, 2004] reports that removing clones actually increases risk in large software systems. Similarly, Kapser et al. [Kapser and Godfrey, 2006a] claim that clones have a positive impact on maintainability. They describe several patterns of cloning and discuss their benefits for the long term evolution of software. Instead of eradicating repeated code there should be an effort towards developing tools to support long term maintenance of clones. My dissertation proposal takes a step in this direction by providing insight into how developers use code clone information.
2.4.2 Human-Based Studies

Similar to the human-based developer studies performed towards the fulfillment of the objectives of this dissertation, there are some other human-based studies that are related to this context. The following paragraphs briefly describe them.

de Wit et al. [De Wit, Zaidman, and van Deursen, 2009] developed CLONEBOARD, an Eclipse plug-in that tracks changes to clones to help prevent inconsistently modified clones. To evaluate CLONEBOARD, de Wit et al. conducted a user study of seven software engineers performing a programming assignment. The study results indicated that the developers saw some value in CLONEBOARD. However, in practice, CLONEBOARD failed to meet the expectations of the users when fixing clone related defects. The users believed that it would need a better user interface to be more useful.

Kim et al. [Kim et al., 2004] performed an ethnographic study of copy-and-paste programming practices. They observed nine developers for about 10 hours of Java/C++/Jython programming. The researchers manually logged edit operations (copy, cut, paste, delete, undo, and redo). As these operations were performed, the developers indicated their intention for copy-and-pasting. Kim et al. also used an Eclipse plug-in to automatically track 50 hours of Java source code edits by five other developers. The researchers used this information to infer the programmers’ intentions for copying-and-pasting. They then interviewed each programmer twice to confirm the accuracy of the inferences. The study conclusion was that developers typically wait until after several copy-and-paste operations before restructuring the code.

Rysselberghe and Demeyer [Rysselberghe and Demeyer, 2003] report the suitability of various clone detection techniques with respect to a specific maintenance task, refactoring. The
results of this study showed that the token suffix tree approach stands out to be the best approach that can be used with respect to refactoring activities.

This chapter discussed some studies that are closely related to the objective of this dissertation. The future chapters mention some of these studies and a few other studies that fall in context of the discussion.
DEFINITIONS OF CODE CLONES AND INTENT BASED TAXONOMY

This chapter presents an overview of the first foundational study. In the literature, researchers have ambiguously defined and classified code clones. Over the years, research studies have defined code clones based on two properties, *structural* and *intentional*. Structural properties are based on semantic and syntactic similarities. Intentional properties are based on the intent and rationale of a developer for clone formation. Research studies have widely used the structural definitions which are vague and unclear. The first part of the chapter presents a systematic literature review (SLR) aimed at summarizing the structural definition of code clones in the literature to gauge the level of ambiguity. On the other hand, researchers have not been using the intent based definitions and classifications as frequently. From a clone management perspective, it is very important to understand the intention of developers while forming code clones. The second part of the chapter examines the available categorizations and proposes a taxonomy based on cloning intent of developers. In addition to that, this chapter also presents a classification of harmful and non-harmful clones based on intent and rationale.

3.1 Ambiguous Structural Definitions

The current literature related to code clones is scattered with ambiguous definitions. This section presents the SLR which closely follows guidelines provided by Kitchenham et al. [Kitchenham, Pearl Brereton, Budgen, Turner, Bailey, and Linkman, 2009]. This section presents the syn-
thesis from 39 published and peer reviewed papers. The goal of this SLR is to integrate information regarding the definition of code clones to help reduce the vagueness.

On a high level, to better understand code clone definitions, the expected outcomes of the SLR are:

– Summarize existing evidence;
– Identify gaps in research for further investigation;
– Help position new research activities.

3.1.1 Research Objectives

The primary research objective for the SLR is to assimilate information on code clones and their classification from the existing literature. This objective motivated three research questions to guide the SLR.

*RQ1. How are clones defined?*

*RQ2. How are code clones categorized?*

*RQ3. What are the terms associated to the various types of clones?*

*RQ4. What is a lower/higher threshold for the size of two fragments to be considered as clones?*

3.1.2 Selection of Source

The first step in the systematic review process was searching databases for relevant publications. Several online databases archive valuable software engineering literature. However, I left out some of the databases due to the evident redundancy in the search results. Peterson et al. found that ACM Digital Library, Compendex, Inspec, IEEE Explore, and Web of Science return unique articles [Petersen and Ali, 2011]. Additionally, the Engineering Village portal\(^1\) includes

\(^1\) http://www.engineeringvillage2.org/
both Compendex and Inspec databases and claims to de-duplicate. I excluded Google Scholar as it returns publications that are not peer reviewed. The following list shows the final database list:

- IEEExplore
- ACM Digital Library
- Web of science
- Engineering Village

I used the following search string to search databases for papers that talk about definitions or categorization of code clones.

("code clone" OR "software clone" OR "code clones" OR "software clones")
AND ("clone" AND ("definition" OR "define"))

3.1.3 Screening of Papers Returned from the Search

This section presents the review process based on relevance of papers to the research questions. Specific inclusion and exclusion criteria guided the screening process of papers. The criteria are as follows:

**Inclusion Criteria**

- Papers that focus on code clones and their types;
- Papers that provide a new/different explanation of the same or other type of clone;
- Papers that focus on other terms used to define types of clones;
- Papers that contain some empirical evidence to support all such claims.
Exclusion Criteria

- Papers that are only based on expert opinion;
- Studies not related to any of the research questions;
- Preliminary conference versions of included journal papers;
- Papers where results are not supported by evidence;
- Papers that are not peer reviewed.

The search string as stated above extracted a list of 338 papers from the four databases. After removing the repetitions the number came down to 305 papers. I performed the screening of papers in four stages as shown in Figure 3.1. In each of these three phases, a second reviewer reviewed a sample of no less than 20% of the excluded articles to determine a fair exclusion of the articles. This practice helped to nullify my bias, if any, in excluding papers and made sure that the exclusions were fair.

Figure 3.1: Screening of Papers
Stage 1 - Exclusion Based on Titles:

This stage of screening excluded papers based on the title of the papers. Although I took special care to make sure that only the most unlikely papers were rejected, a second reviewer reviewed a randomly-selected 20% of the rejected papers to ensure there was no bias in the rejection of papers. We rejected 200 papers based on the Title. This stage produced 105 papers for me to review in stage 2.

Stage 2 - Exclusion based on Quick Scan:

Based on Kitchenham et al.’s [Kitchenham et al., 2009] guidelines, the second stage incorporates screening by reading abstracts. However, considering that I was looking for information that may or may not be the primary objective of the paper, I had to substitute this phase. Based on observation, most papers on code clones provide some sort of definition for code clones in the introductory sections of the paper. In some cases, papers reuse common definitions and in some cases authors provide slightly different definitions. Hence, for this stage I decided to scan quickly through the 105 papers without getting into the details. The primary aim was to gauge if a paper has relevant information on clone definitions or their types. Exclusion was based on the criterion, ‘Studies not related to any of the research questions.’ After the screening process in this stage, there were 86 papers left for review in the next stage. Similar to the earlier stage, a second reviewer reviewed a random sample of 20% of the excluded papers.

Stage 3 - Exclusion based on reading full text:
For this stage of screening, I read full papers. A quick scan (Stage 2) through the papers made sure that all papers in this stage had relevant information on clone definitions, threshold sizes or clone classification. In this stage, I rejected some papers on the grounds of relevant information referencing a different paper to make sure that selected papers had original information. At the end of the screening process, there were 39 papers for the final stage of data extraction. Because of time constraints the second reviewer was not asked to read 20% of 47 rejected papers. Instead, he read a random 20% of selected papers to make sure that the selections are all valid. Table 3.1 represents the paper distribution categorized according to the sources.

Stage 4 - Data Extraction:

The final list of 39 papers went through a data extraction process. Data extraction forms ensure consistent and accurate extraction of important information from a paper. The form follows the research questions to make sure the extraction of relevant information only. The data extraction form is shown in Table 3.2. To ensure the validity of the process, the second reviewer was asked to extract data from randomly selected five papers. We compared our extraction forms for the set of randomly selected papers to check for consistency.

3.1.4 Results

This section presents the results of the systematic literature review. Each subsection is focused on the three research questions explained in Section 3.1.1.

3.1.4.1 Ambiguity in Defining Code Clones

Various researchers have defined code clones differently. These definitions fall under the broad umbrella of fragments that are similar to each other on different levels of syntactic and semantic similarities. One observation that is very eminent from the study on the related literature
### Table 3.1: Paper Distribution

<table>
<thead>
<tr>
<th>Source</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE Transactions Software Engineering</td>
<td>5</td>
<td>12.8</td>
</tr>
<tr>
<td>International Conference on Software Maintenance</td>
<td>2</td>
<td>5.1</td>
</tr>
<tr>
<td>International Symposium on Software Metrics</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Working Conference on Reverse Engineering</td>
<td>6</td>
<td>15.4</td>
</tr>
<tr>
<td>International Workshop on Principles of Software Evolution</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Source Code Analysis and Manipulation</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>International Conference on Software Maintenance</td>
<td>2</td>
<td>5.1</td>
</tr>
<tr>
<td>International Symposium on Empirical Software Engineering</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>SIGSOFT Software Engineering</td>
<td>2</td>
<td>5.1</td>
</tr>
<tr>
<td>International Conference on Automated Software Engineering</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Conference on Computer Assisted Information Retrieval</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Malaysian Conference on Software Engineering</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Software Quality Control</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>International Workshop on Software Clones</td>
<td>2</td>
<td>5.1</td>
</tr>
<tr>
<td>SIGPLAN Workshop on Partial Evaluation and Program Manipulation</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Scientific Computer Programming</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>International Conference of Software Engineering</td>
<td>3</td>
<td>7.6</td>
</tr>
<tr>
<td>Asia-Pacific Software Engineering Conference</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>European Symposium on Programming Languages and Systems</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Journal of Software Maintenance and Evolution</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>International Conference on World Wide Web</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>International Conference on Program Comprehension</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>ACM Transactions on Software Engineering and Methodology</td>
<td>1</td>
<td>2.6</td>
</tr>
<tr>
<td>Data Items</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>Unique identifier or the reference number for a paper.</td>
<td></td>
</tr>
<tr>
<td>Bibliographic</td>
<td>Author, year, title, source.</td>
<td></td>
</tr>
<tr>
<td>Type of article</td>
<td>Journal/conference/workshop/book chapter</td>
<td></td>
</tr>
<tr>
<td>Study Aims</td>
<td>The aims or goals of the primary study.</td>
<td></td>
</tr>
<tr>
<td>Context</td>
<td>Research area(s) the paper focus on.</td>
<td></td>
</tr>
<tr>
<td>Related papers</td>
<td>Short references of closely related papers.</td>
<td></td>
</tr>
<tr>
<td>Study design</td>
<td>Type of study - controlled experiment, survey, empirical, lessons learned, etc.</td>
<td></td>
</tr>
<tr>
<td>Level of analysis</td>
<td>Single/more researchers, project team, organization, department.</td>
<td></td>
</tr>
<tr>
<td>Empirical Study</td>
<td>Yes, no; If &quot;Yes&quot;: Is the study about human subjects?</td>
<td></td>
</tr>
<tr>
<td>Control group</td>
<td>Yes, no; if &quot;Yes&quot;: number of groups and size per group.</td>
<td></td>
</tr>
<tr>
<td>Data collection</td>
<td>How the data was collected, e.g., interviews, questionnaires, measurement forms, observations, discussion, and documents</td>
<td></td>
</tr>
<tr>
<td>Data analysis</td>
<td>How the data was analyzed; qualitative, quantitative or mixed.</td>
<td></td>
</tr>
<tr>
<td>Clone Definition</td>
<td>What is the term used, what is the definition and what are specific parameters used</td>
<td></td>
</tr>
<tr>
<td>Concepts</td>
<td>Threshold and clone sizes etc.</td>
<td></td>
</tr>
<tr>
<td>Study findings</td>
<td>Major findings and conclusions from the primary studies.</td>
<td></td>
</tr>
</tbody>
</table>
is that code clones lack a concrete unambiguous definition. Most of the definitions provided in the literature are either ambiguous or lack distinct boundaries. For example, Mayrand et al. [Mayrand et al., 1996] defined "Distinct Name" clones, which based on their definitions, can only differ by identifier names. Similarly, Balazinska et al. [Balazinska et al., 1999] used the attribute “long difference,” which means the amount of change between fragments. In retrospect, these definitions are specific to the researcher’s study and fail to adhere to widely accepted definitions. Although fairly concretely defined in the context of the respective studies, they fail to offer a standardized solution to this issue of unclear definitions.

The literature is abundant with unclear and vague definitions. Kamiya et al. [Kamiya et al., 2002] defined code clones as fragments of the source code which are similar or identical to each other. They did not provide any explanation of the terms related to similarity. Some studies have defined clones based on tree similarity thresholds which ideally apply to the near-miss clones [Baxter et al., 1998]. However, as the name suggests, near miss clones are vague and hard to define concretely. In my understanding from reading the literature, near miss clones lie in a range between semantically similar fragments to textually similar fragments which are different enough not to be detected as textual clones. Evidently, the vagueness is large. Burd defined clones as additional occurrences of a code fragment with or without modification [Burd and Bailey, 2002]. This is an example of broadly defined code clones without any concrete explanation of the term ‘modification.’ In a similar attempt to define clones, Higo et al. [Higo et al., 2005] defined them as "sequences" having an equivalence relation between themselves, which again is inconclusive because of the vagueness introduced by the term ‘equivalence.’ Baxter et al. [Baxter et al., 1998] used a contrasting way to define code clones. They used the similarity threshold in the matching sub-trees to define fragments as clones. Although, concrete in nature, these definitions deviate from
the general structural similarity metrics of comparing fragments. Nguyen et al. [Nguyen, Nguyen, Pham, Al-Kofahi, and Nguyen, 2008] defined code clones as fragments that can be symmetric in relation but cannot be transitive to each other in similarity. This definition is another example of a definition which is confusing as well as vague.

After reading through the available literature, the most prominent observation was that code clones require better and standardized definitions. In all the above examples and elsewhere throughout the literature, it is highly likely that a code snippet that falls under the category of code clones using one of the above definitions can be categorized otherwise using another. From the point of view of a developer, this problem gives rise to two other issues. First, detection of clones in a software system becomes futile because for maintenance purposes the clones being reported can vary. A shortcut to this issue is to stick to a definition to make sure the results are not affected. However, if and when there is a requirement to change because of an issue with the current definition or availability of a better one, the same issue pops up. Depending on the maturity of the software system this might lead to greater problems. Second, educating or training developers on advantages and drawbacks of clones and their types becomes highly complex to comprehend. From the perspective of a developer, broadly or vaguely defined clones could lead to confusion in terms of keeping track of clones, or whether or not to apply maintenance efforts.

Apart from the above references of vague and unclear definitions, I found that in many cases code clones are defined using different properties of code. Most of the definitions found in the literature are based on structural properties of snippets. However, clones have been defined based on similarities in tree structure, control flow in a code snippet or in some cases related to a developer’s cloning intent. Although I believe, definitions based on developer intent of cloning is an important factor to consider, proper benchmarking is required to define code clones over
textual, syntactic and semantic similarities. As previously attempted, a single definition might be
too broad or even not be possible; hence there should be an effort towards defining taxonomies. In
other words, concretely defined (i.e., no vagueness) groups of code clones that are distinct from
each other. It is critical to make sure that the groups are not overlapping and they should be
exhaustive (cover all possibilities). A future direction of my research is to put forward a concrete
and exhaustive taxonomy based on cloning intention of a developer.

3.1.4.2 Categorization of Code Clones

Keeping aside the idea of a categorization based on cloning intent, fundamentally code
clones are of two types: 1) Code fragments that are textually similar and 2) Code fragments that
are semantically similar. Textually similar code clones [Bellon et al., 2007; Koschke et al., 2006]
are the result of copy paste operations by a developer while coding, whereas, semantically similar
code fragments [Davey, Barson, Field, and Frank, 1995; Krinke, 2001; Mishne and Rijke, 2004]
are a result of imitating the functionality or the architecture of a certain piece of code. Further, the
textually similar code clones are commonly known as Type 1 and Type 2 while the semantically
similar clones are commonly known as Type 3 [Ali, Sulaiman, and Syed-Mohamad, 2011; Lavoie
and Merlo, 2011; Tiarks, Koschke, and Falke, 2011]. Bellon et al. [Bellon et al., 2007] put forward
this categorical distribution of clones and is as follows:
Type 1- “Textually identical clone fragments which can only differ in whitespaces and comments.

A normalization process of getting rid of white spaces and comments would render the two code fragments textually identical.”

Type 2- “Textually similar clone fragments with further modifications. Further modifications being modifications in identifiers, literals, added or removed statements etc.”

Type 3- “These are semantically similar code fragments which rather being textually similar, are structurally similar to each other and hence also referred to as structural clones. In other words these code fragments possess the same functionality or logic but have a different implementation.”

Type 1 clones are sometimes referred to as exact clones while Type 2 and Type 3 as near miss clones [Ali et al., 2011]. Based on observation, from the year 2009 some researchers started breaking Type 2 clones further into finer granularity. Gradually the research community started to conventionally accept the four different classifications of code clones namely, Type I, Type II, Type III and Type IV clones [Li and Thompson, 2009; Roy et al., 2009]. In the present scenario, the community has widely accepted and used the Type I-IV classification of code clones over the Type 1-3 classification.

I conjecture that the complexity of these clone types, with respect to clone detection, increases from Type I to Type IV. The Type I clones being exact textual copies are easier to detect. Type II and Type III clones are textually similar with modifications, which increases the complexity of detection from Type I. The Type IV clones, semantically similar clones, are the hardest to detect in a software system. It usually takes a graph based algorithm to detect these clones using the control flow of the code fragments concerned.
Although the code clone types discussed above make a fairly exhaustive list, the literature is full of similar matching terms. This confusion is primarily due to the vagueness and disagreement over the definitions of the types. Researchers have used specific terms for defining specific types of clones related to their studies. Table 3.3 lists all the terms that have been used along with their definitions, reference of the research paper and the way they map back to the conventionally accepted four types of clones.

Table 3.3 maps different terminologies to the respective clone type. Table 3.4 maps the four different types of clones to the respective terms. For example, a Type X clone (where X is a clone of Type I, II, III or IV) maps to terminology A, if and only if A is a subset of Type X.

Table 3.4 points to an interesting observation. There are no overlaps for Type IV clones, i.e. no two different terms are mapped to Type IV clones. Type IV clones are equivalent to ‘Structural Clones.’ They are just two different names of the same type of clones. We can conclude that the definition of a Type IV clone is concrete and needs no further investigation. There seems to be no confusion with the idea of a structural clone in the literature. Henceforth, because of the absence of ambiguity, Type IV clones can be conclusively defined as they have been in available literature:

"Type IV clones are two or more code fragments that perform the same computation but are implemented through different syntactic variants. This type of clones can represent a bigger replicated design level similarity and can be useful for reuse of code with a more robust architecture." [Basit et al., 2005a; Basit and Jarzabek, 2009; Bellon et al., 2007; Deissenboeck et al., 2008]

In terms of terminologies, it is apparent from Table 3.4 that there are overlaps between Type I and II. However, a close assessment revealed that the ambiguity lies in the terms ‘Simple
Table 3.3: Terminologies for Code Clones

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Definition</th>
<th>Mapping to 4 types of clones</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact Clones</td>
<td>Textually Identical clones</td>
<td>subset of Type I</td>
<td>[Lague et al., 1997]</td>
</tr>
<tr>
<td>Simple Clones</td>
<td>Textually identical copies or textual copies with modification.</td>
<td>Can be Type I, II or III</td>
<td>[Basit and Jarzabek, 2009]</td>
</tr>
<tr>
<td>Structural Clones</td>
<td>Can represent a bigger replicated design level similarity. Can be useful for reusable code with a more sturdy architecture.</td>
<td>Type IV</td>
<td>[Basit, Rajapakse, and Jarzabek, 2005a; Basit and Jarzabek, 2009; Deissenboeck et al., 2008]</td>
</tr>
<tr>
<td>Contextual Clones</td>
<td>Addition of referenced information rendering a part of the code understandable which was useless earlier can give rise to contextual clones</td>
<td>Subset of Type III</td>
<td>[Martin and Cordy, 2011]</td>
</tr>
<tr>
<td>Distinct Name Clones</td>
<td>The code fragments are identical on all metric values except for the names of the functions/methods. Essentially, any clone pair with identical body with a different identifier</td>
<td>Subset of Type II</td>
<td>[Lague et al., 1997]</td>
</tr>
<tr>
<td>Parameterized Clones</td>
<td>Similar to DistinctName where there is a systematic renaming of the code fragments</td>
<td>Subset of Type II</td>
<td>[Baker, 1995]</td>
</tr>
<tr>
<td>Gapped Clones</td>
<td>Partially identical and partially varying clones. The varying part is referred to as a gap</td>
<td>Type III</td>
<td>[Ueda et al., 2002]</td>
</tr>
<tr>
<td>Non-Contiguous Clones</td>
<td>Changes are allowed throughout the body of the cloned fragment</td>
<td>Subset of Type III</td>
<td>[Komondoor and Horwitz, 2001a]</td>
</tr>
<tr>
<td>Re-Ordered Clones</td>
<td>Essentially similar to each other but some lines might have been reordered to change the appearance and the structure</td>
<td>Subset of III</td>
<td>[Komondoor and Horwitz, 2001a]</td>
</tr>
<tr>
<td>Ubiquitous Clones</td>
<td>Repetitive clone fragments, usually small in size which are present in numerous locations</td>
<td>Can be Type I, II or III</td>
<td>[Kim et al., 2005]</td>
</tr>
</tbody>
</table>
Table 3.4: Type to Terminology Mapping

<table>
<thead>
<tr>
<th>Clone Type</th>
<th>Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type I</td>
<td>Identical clones, Ubiquitous clones, Simple Clones</td>
</tr>
<tr>
<td>Type II</td>
<td>Simple Clones, DistinctName Clones, Parameterized Clone, Ubiquitous clones</td>
</tr>
<tr>
<td>Type III</td>
<td>Simple Clones, Contextual Clones, Gapped Clones, Non-Contiguous Clones, Re-ordered Clones, Ubiquitous clones</td>
</tr>
<tr>
<td>Type IV</td>
<td>Structural Clones</td>
</tr>
</tbody>
</table>

*Clones*’ and ‘*Ubiquitous Clones*.’ Table 3.3 provides the definitions of these terms and clearly shows that they refer to any level of textual similarity without differentiating on the degree of similarity. Hence the terms ‘*Simple Clones*’ and ‘*Ubiquitous Clones*’ refer to clones with any degree (exact match, exact match with minor changes and match with further modifications) of textual similarity. Effectively the definition for Type I clones is the simplest and most clear cut. Hence, I safely say that the ambiguity between Type I and II is because of the ambiguously defined Type II clones. Following from this argument, I conjecture that Type I definition is distinct and clear cut and is as follows:

"*Type I code clones are textually identical fragments except for variations in whitespace, layout, and comments.*” [Bellon et al., 2007]

From the literature survey, it appears, there is ambiguity in Type II and Type III definitions. Figure 3.2 represents this overlap graphically. *Textual* clones and *Structural* clones divide code clones into two distinct broad groups based on textual similarity and semantic similarity. The structural clones over the years came to be known as *Type 3* and subsequently *Type IV* clones.
Textual clones were further divided into two groups based on similarity levels. First, the exactly similar clones which were called *Type 1* and subsequently *Type I*. Second, textual similar clones with various levels of differences were called *Type 2* clones. Researchers, in an attempt to reduce the ambiguity, further divided Type 2 into *Type II* and *Type III*. From Table 3.4 it is clear that there is an overlap of boundaries between the two. A simple solution to this problem is to further divide these two categories into multiple sub-categories to separate them out and define the boundaries more robustly. However, to derive such a classification, elaborate empirical studies are required aimed at mining software repositories and validate those findings empirically.

### 3.1.4.3 Threshold for Code Clone Sizes

Another factor that adds to the ambiguity of code clones is the minimum and maximum threshold sizes of code fragments that should be considered as code clones. The higher threshold
of sizes is not an issue usually because most clone detectors do not look across projects. Another
reason for less impact of larger clones is that they are generally fewer in number. However, the
lower threshold proves to be an important factor considering the frequency of formation. Kim et
al. in their study [Kim et al., 2005] observed developers while they wrote code. They scrutinized
the intent of the developer while copy pasting and thus giving rise to cloned code. Out of the total
candidates of code clones they studied, over 50% were mere copy paste of method/variable names.

Most of the clone related tools such as CC Finder [Kamiya et al., 2002] can accept the
variable threshold sizes of clones as an input. These tools accept threshold sizes either in terms of
number of tokens or number of lines of code. Some of the tool evaluation empirical studies [Kapser
and Godfrey, 2006b; Kim et al., 2005] standardized the minimum threshold for two code fragments
to qualify as code clones to be 30 tokens. Table 3.5 shows the minimum thresholds used in some
other tool evaluation studies.

<table>
<thead>
<tr>
<th>Study Citation</th>
<th>Minimum Threshold Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Kapser and Godfrey, 2006b; Kim et al., 2004]</td>
<td>30 Tokens</td>
</tr>
<tr>
<td>[Rajapakse and Jarzabek, 2005]</td>
<td>20 Tokens</td>
</tr>
<tr>
<td>[Li and Thompson, 2009]</td>
<td>30 Tokens</td>
</tr>
<tr>
<td>[Duala-Ekoko and Robillard, 2010]</td>
<td>50 Tokens</td>
</tr>
<tr>
<td>[Li et al., 2006]</td>
<td>30 Tokens</td>
</tr>
<tr>
<td>[Roy and Cordy, 2008b]</td>
<td>6 Loc</td>
</tr>
<tr>
<td>[Pham, Nguyen, Nguyen, Al-Kofahi, and Nguyen, 2009]</td>
<td>5 Loc</td>
</tr>
</tbody>
</table>

Minimum thresholds for clones are problematic to decide upon. High thresholds can pro-
voke a threat of leaving important clones undetected. On the contrary, low thresholds can lead to
detection of enormous number of code clones. For the sake of simplicity, lets name them naive
clones. Naive clones are small similar code snippets which essentially should not have been de-
tected as clones make detection results verbose. Higo et al. in their study discuss this problem of
choosing an appropriate lower threshold for clone sizes [Higo, Ueda, Kusumoto, and Inoue, 2007]. They ultimately fix the lower threshold to be 20 tokens. However, they experienced the problem of detecting a huge number of naive clones. Hence, there is clearly a trade off between detection of numerous naive clones and setting a threshold so as to make it manageable. However, the latter poses a possible threat of missing some of the important clones. In an academic research setting, it is usually viable to use the former approach as time is generally not a deciding factor. Diligently rejecting all naive clones would result in a better detection report where false positives are non-existent. However, the situation is not so ideal in an industrial setting. The two main practical reasons are, a) large sizes of industrial systems would render the reports containing an enormous number of naive clones, and b) time is an expensive resource which makes it important for the detection reports to be void of false positives. Based on the above argument, my suggestion would be to definitely keep the minimum threshold higher than 20 tokens.

Kim et al. showed in their study that developers do not consider very short snippets of code to be code clones [Kim et al., 2004]. One of the reasons for this observation could be that for a code snippet to be reasonably effective and to be critically affecting the system it would be bigger than 30 tokens at least. In another study Kim et al., based on their observation, fixed the minimum threshold size for clones to be 30 tokens [Kim et al., 2005]. Hence, increasing threshold sizes from 20 tokens to 30 tokens might not result in leaving out important clones but could certainly ensure reduction in detection of naive clones. In addition to the above argument, several experiments [Kamiya et al., 2002; Kapser and Godfrey, 2006b; Kim et al., 2005] have demonstrated that a minimum threshold size of 30 tokens is reasonable. However, further studies are required to prove this conjecture. Following from the above arguments, it appears that it is safe to assume the minimum threshold to be 30 tokens. It can be very indecisive to mark threshold based
on lines of code. Even after normalization, lines of code can vary depending upon writing styles of developers. Based on this argument and the evidence as demonstrated by some experiments [Pham et al., 2009; Roy and Cordy, 2008b] as discussed in Table 3.5, anything less than 6 lines of code should not be considered as code clones. Although the success of these experiments show this threshold size to be reasonable, there are experiments which have shown otherwise [Baker, 1995; Johnson, 1994]. However, in my opinion it is better to quantify them based on number of tokens as lines of codes depend highly on writing styles of developers and/or pre-processing used to normalize the source code.

Similarly for higher thresholds, though comparatively less critical because of less frequency, sizes should be fixed. More studies are required to conclusively fix the threshold sizes. Further studies are required to support the decisions made in the studies discussed above or the arguments provided in this section. Standardization of such critical parameters could lead to less confusion and better trustworthy validation.

3.2 Taxonomy Based on Cloning Intent

Section 3.1 discusses clone definitions and categorization solely based on code structure. In addition to being ambiguous in nature, these definitions and categorizations do not represent any relevant information regarding cloning intent and rationale. There is a need to understand the behavior of developers when creating clones. This section presents the method of qualitative developer interviews focused on understanding a developer’s reasons for cloning code. Identifying these reasons will lead to a clone categorization that focuses on developer intent rather than focusing on code structure.

There are two related works that address portions of this problem. First, Kapser et al. [Kapser and Godfrey, 2008] inferred a categorization of developer intent based upon a post hoc analysis
of large software systems. Such a retrospective analysis can reveal ‘what’ developers did but not ‘why.’ The intent, or motivation, for a maintenance task is equally crucial and helps to validate the observations from post hoc analysis of the system. Second, Zhang et al. [Zhang, Peng, Xing, and Zhao, 2012] used interviews to categorize developer intent when cloning code. However, they did not use their findings to categorize the code clones themselves. The following sections describe the study which overcomes the shortcoming of the two studies described here.

3.2.1 Study Overview

There were two primary objectives of this study. First, empirically validate the categorization suggested by Kasper et al. [Kapser and Godfrey, 2008] from their case studies of large software systems. Second, develop a taxonomy of code clones based on cloning intent and rationale. I interviewed developers to understand why they clone code or code structure. The aim was to evaluate the reasons why developers clone code, classify those reasons as either good or bad practices, and develop a taxonomy for types of code clone based on intent and rationale of code reuse.

3.2.1.1 Cloning Patterns

In their study, Kasper et al. [Kapser and Godfrey, 2008] described 11 high level patterns of code cloning. These 11 patterns were divided in four categories. The following are brief descriptions of the patterns.

1. **Forking:** Developers use large software artifacts as “springboards” for new development. The newer versions evolve independently.

   a) **Hardware Variations:** Older versions operate as springboards for latest hardware drivers.
b) **Platform Variations:** While porting software to new platforms by making platform specific changes.

c) **Experimental Variations:** Development of an experimental version of the old tested version for testing and other experimental purposes.

2. **Templating:** Developers reuse a known solution to a certain problem and modify according to the specific requirements.

   a) **Boiler-plating:** Reusing trusted code that is hard to abstract due to language constraints.

   b) **API/Library Protocols:** Reusing a set of procedure calls and modifying them to the requirement at hand.

   c) **Algorithmic Idioms:** Implementing certain algorithms and reusing them from their known solutions.

   d) **Parameterized code:** Reuse of a known solution by changing a few identifiers.

3. **Customization:** Modifying or extending similar known solutions because of a different requirement.

   a) **Bug Workarounds:** Work around ownership issues, sometimes overloading by fixing the copied buggy code.

   b) **Replicate and Specialize:** Similar known solution copied from elsewhere in the software system and modified based on requirements as the copied code was not the exact solution

4. **Exact Matches:** Reusing fragments by frequent and repeated use throughout the system
because either they are insignificant for abstraction or they cannot be used out of context of the place.

a) *Cross Cutting Concerns*: Semantic properties that cut across otherwise unrelated code.

b) *Verbatim Snippets*: Small repetitive logical fragments which do not have significant semantics. It is better reused multiple times than applied to a level of abstraction.

### 3.2.1.2 Developer Interviews

In order to empirically validate the cloning patterns discussed above, I developed interview questions for the developers (Table 3.6). These questions did not use the term ‘code clones’ in order not to bias the developers. This strategy made sure that developers did not base their responses on any preconceived notion of the term ‘code clone.’ Even though I did not specifically mention the term ‘code clone’, the interviewees may still have determined that the focus of the study was on code cloning. All participants were full-time software developers at the University of Alabama’s Center for Advanced Public Safety (CAPS\(^2\)). Some of these participants were also part-time Computer Science graduate students who likely heard about code cloning in their coursework. Hence, I cannot completely rule out any bias that may have occurred from preconceived ideas about code cloning.

The interviews took place in a conference room. Each interview session consisted of the participant completing an experience questionnaire followed by a one-on-one interview. I interviewed each participant for approximately 20 minutes. I took notes while interacting with the participant. Audio recording the interview sessions made sure that the notes taken during the in-

\(^2\) [http://caps.ua.edu/](http://caps.ua.edu/)
terview were consistent. I later destroyed the audio tapes in order to make the data anonymous and unrecognizable.

3.2.1.3 Pilot Study

Prior to the main study, I conducted a pilot study to ensure all questions were clear. I also used the pilot study to test that the interviews would take about 20 minutes. I conducted the pilot study in a similar environment to the actual study. The participant for the pilot study was a computer science graduate student who had more than 3 years of industrial experience as a developer. The pilot study resulted in a few minor wording changes to some of the questions. None of the changes were significant.

3.2.1.4 Participants

As I mentioned, the participants were full-time developers at CAPS. The range of experience based on number of years worked as a developer varied from 7 months to 13 years. Figure 3.3 plots the years of software development experience between x and y axes with a mean of approximately 60 months. Although the standard deviation for the plot is a bit high because of the variable range of years of experience, a mean of 5 years instills faith on the responses.

3.2.2 Data Analysis

The participants provided detailed responses in the form of qualitative data. I transcribed the audio recordings with the help of the notes taken during the interview sessions into text. A process of qualitative analysis method mapped the responses to Kapser’s classification. In the qualitative analysis process, the analyst organizes the qualitative data according to a coding scheme. Then the analyst maps the qualitative data to the existing codes. In case the data does not map to an existing code, the analyst creates a new code and adds it to the scheme. In this case, I derived the coding scheme from Kapser’s classification. The cases where the data did not map to the coding
## Table 3.6: Interview Questions

<table>
<thead>
<tr>
<th>Question</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Have you ever copied a large structural artifact (such as a class, package, or subsystem) and then evolved the copy independently of the original?</td>
<td>1.1 – If so, please describe the circumstances. 1.2 – Would you suggest such practices for developers you supervise?</td>
</tr>
<tr>
<td>2  Similar instance over a new platform?</td>
<td>2.1 – If so, please describe your strategy for developing the ported version and your rationale for using this strategy.</td>
</tr>
<tr>
<td>3  Have you ever developed a new implementation of an existing module (such as a class, package, or subsystem) using a different technology (e.g., API/library or programming language)?</td>
<td>3.1 – If so, please describe your strategy for developing the new implementation and your rationale for using this strategy.</td>
</tr>
<tr>
<td>4  Is there a programming idiom or construct that you use frequently or suggest developers you supervise to use? If so, please describe the benefits and drawbacks of this practice.</td>
<td></td>
</tr>
<tr>
<td>5  If it possible to use abstraction instead of using similar code snippets at various places, what strategy do you prefer?</td>
<td>5.1 – Any advantages and disadvantages that come to your mind?</td>
</tr>
<tr>
<td>6  Have you ever reused another programmer’s code via copy-paste? If so, please describe your motivation for doing so.</td>
<td></td>
</tr>
<tr>
<td>7  When reusing code via copy-paste, at what levels of granularity do you think is agreeable to copy-paste?</td>
<td>7.1 – few statements (e.g., a loop body), 7.2 – several statements (e.g., a method body), 7.3 – or multiple logical structures (e.g., a design pattern or structural template). 7.4 – Do you think any of the above is unacceptable?</td>
</tr>
<tr>
<td>8  To what extent would you prefer copy-paste another programmer’s code?</td>
<td></td>
</tr>
<tr>
<td>9  Any scenario that comes to mind where a bug was propagated because of code reuse practices?</td>
<td></td>
</tr>
<tr>
<td>10 In your opinion, what are key benefits and drawbacks of using code via copy-paste?</td>
<td></td>
</tr>
</tbody>
</table>
scheme formed an anomaly. There were three types of observations from the qualitative analysis of the interview data (⇒ means ‘this implies’).

- Data mapped to coding scheme ⇒ Validated Kapser’s classification;

- Data did not map to coding scheme ⇒
  - Contradicts Kapser’s classification;
  - Add a new ‘developer intent’ based category to the taxonomy.

The developers provided a preference of resources they usually consult when they code. Table 3.7 shows the responses that the developers provided for the particular question. All but one participant, selected ‘self experience’ as their first preference. This participant was a new developer with an experience of a little over six months. All developers provided at least two choices based on their order of preference. Five developers provided three choices in order of preference. Figure 3.4
Table 3.7: Resources to Consult

When you are asked to write code to implement a new functionality, what resources do you consult? That is, do you...

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>Preference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rely on your experience/expertise and begin writing code immediately</td>
<td>9</td>
</tr>
<tr>
<td>Review code that you have written previously to solve similar problems</td>
<td>6</td>
</tr>
<tr>
<td>Review code that others have written to solve similar problems</td>
<td>3</td>
</tr>
<tr>
<td>Consult some other resource(s)</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 3.4: Preference of Resources
shows the breakdown of the responses based on first, second, and third preferences of resources that the developers consult.

One interesting observation that can be drawn from the figure is high dependency of developers on ‘Other Resources.’ Based on the responses, the ‘Other Resources’ in most cases was code available on external web resources such as Stack Overflow. This observation indicates that apart from the clone propagation that can occur between code fragments inside a system, code clones might be formed using the same code multiple times copied from sources external to the system. Such code clones can be completely disconnected and distant from each other especially in cases where different developers use the same external source. Such clones can be very hard to spot, especially without a clone detection tool. Hence, this type of cloning might pose greater threats to clone management because it will be hard to track them.

Table 3.8 presents the systematic qualitative analysis for the responses from the developers. The first column contains the interview questions. The second column presents the intention behind the specific actions that the developers took in various scenarios. The third column presents the number of responses that mapped to each intentions provided. The fourth column presents the classification of the intentions based on a forced or discretional rationale. All responses to the questions were scenario-based responses. The participants explained their intentions behind the choice they made for a particular action. If the choice they made was affected by any external factor, for example time constraints, the analyst marked the rationale as ‘forced.’ Any choice that the developer made solely based on the intention of writing the best code to his/her ability, the analyst marked the rationale as ‘discretional.’ The fifth column maps the intentions to the categories provided by Kapser et al. Finally, the sixth column presents the ‘developer’s perception’

3 http://stackoverflow.com/
in which the developers presented their intentions. Note that the analyst recognized developer’s perceptions solely based on the developer’s satisfaction level for the action s/he took related to the described scenario. 


⇒ highly positive, + ⇒ fairly positive, +⇒ neutral, - ⇒ fairly negative and – ⇒ highly negative. The analyst subjectively judged the developer’s perception into the five categories based on the tone and justification of the action s/he took. Note that the participants justified the choice of actions they made for each scenario during the interviews.

The intentions resulting from the systematic qualitative analysis mapped to each category from Kapser’s categorization except ‘Hardware Variations.’ The type of development the participants perform at CAPS could be a possible explanation for this observation. It is quite likely that interviewing developers who perform hardware related development would result in answers that map to the ‘Hardware Variations’ category. In Table 3.8, one of the intentions of cloning ‘in context of parts that do not change’, did not map to any of the Kapser et al.’s categories. As explained in Section 3.2.2, this outcome became the reason for formation of a new category of cloning. Hence, cloning in the parts that seldom change becomes the twelfth category of cloning based on intent. For the sake of simplicity, I will call this intent Persistent Modifications. Hence, qualitative analysis of the interview data was unable to validate one category and found one anomaly to Kapser’s intent based categorization. The next section presents the discussion based on the data analysis.

3.2.3 Discussion

The data analysis resulted in 12 intent-based categories of cloning. Eleven from Kapser’s taxonomy and one from the data not mapping to the coding scheme. Table 3.9 presents the categories that are associated to various intentions along with the developers’ perception. It is clear that any category related to intent ‘Time Constraint’ has a negative developers’ perception based
Table 3.8: Mapping of Developers’ Intentions to Kapser’s Taxonomy

<table>
<thead>
<tr>
<th>Question</th>
<th>Intent/Rationale based on experiences provided by the developers</th>
<th>Responses</th>
<th>Discretional / Forced</th>
<th>Mapping</th>
<th>Connotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Have you ever copied a large structural artifact (such as a class) and then evolved the copy independently of the original?</td>
<td>Sustaining robust code structure throughout a project</td>
<td>4</td>
<td>Discretional</td>
<td>Experimental Variations</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Avoid breaking the original working system and build on it</td>
<td>2</td>
<td>Forced</td>
<td>Experimental Variations</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Due to time constraints</td>
<td>3</td>
<td>Forced</td>
<td>Experimental Variations</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Cannot change original code because of lack of access or authorization</td>
<td>1</td>
<td>Forced</td>
<td>Boiler Plating</td>
<td>--</td>
</tr>
<tr>
<td>Have you ever ported a module to a new platform?</td>
<td>Copy conceptual idea while porting to a new platform</td>
<td>5</td>
<td>Discretional</td>
<td>Experimental Variations</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Reuse older system because of time constraints</td>
<td>3</td>
<td>Forced</td>
<td>Platform Variations</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Sustaining robust Code Structure throughout a project</td>
<td>4</td>
<td>Discretional</td>
<td>Platform Variations</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Did not reuse code</td>
<td>1</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Have you ever developed a new implementation of an existing module using a different technology?</td>
<td>Copy conceptual idea while porting to a new platform</td>
<td>4</td>
<td>Discretional</td>
<td>Experimental Variations</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Sustaining robust Code Structure throughout a project</td>
<td>6</td>
<td>Discretional</td>
<td>API/Library protocols</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Did not reuse code</td>
<td>3</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Is there a programming idiom or construct that you use frequently?</td>
<td>Certain code structures (loops/data structures/design patterns) which are better in quality than their counterparts</td>
<td>4</td>
<td>Discretional</td>
<td>General language or algorithmic Idioms</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>No use of specific code constructs</td>
<td>6</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>If it possible to use abstraction instead of using similar code snippets at various places, what strategy do you prefer?</td>
<td>Reuse code if logical distance is large</td>
<td>2</td>
<td>Discretional</td>
<td>Cross Cutting Concerns</td>
<td>+/-</td>
</tr>
<tr>
<td></td>
<td>Reuse code because of different versions/standards</td>
<td>1</td>
<td>Forced</td>
<td>Cross Cutting Concerns</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>To reuse similar code with modification in a different context</td>
<td>4</td>
<td>Discretional</td>
<td>Parameterized Code</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>In context of parts that do not change</td>
<td>3</td>
<td>Discretional</td>
<td>???</td>
<td>+/-</td>
</tr>
<tr>
<td>Do you ever use copy-paste to create multiple new instances of a code fragment?</td>
<td>To load resources locally</td>
<td>1</td>
<td>Discretional</td>
<td>Cross Cutting Concerns</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Due to time constraints</td>
<td>9</td>
<td>Forced</td>
<td>Replicate and Specialize</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>To access same module from various locations</td>
<td>1</td>
<td>Discretional</td>
<td>Verbatim Snippets</td>
<td>+</td>
</tr>
<tr>
<td>Have you ever reused another programmer’s code via copy-paste?</td>
<td>Written by experienced developer (trusted code)</td>
<td>2</td>
<td>Discretional</td>
<td>Replicate and Specialize</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Due to time constraints</td>
<td>5</td>
<td>Forced</td>
<td>Replicate and Specialize</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Sustaining robust code structure throughout a project</td>
<td>4</td>
<td>Discretional</td>
<td>Replicate and Specialize</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>While porting to a new standard</td>
<td>1</td>
<td>Forced</td>
<td>Replicate and Specialize</td>
<td>-</td>
</tr>
<tr>
<td>Have you propagated a bug via copy-paste?</td>
<td>While system maintenance to remove faults</td>
<td>3</td>
<td>Discretional</td>
<td>Bug Workaround</td>
<td>+</td>
</tr>
</tbody>
</table>

60
on the developers’ responses. This observation is a strong indication towards harmful effects of clones formed under such scenarios.

Table 3.9: Associated Categories to Intent

<table>
<thead>
<tr>
<th>Category</th>
<th>Intent and Rationale</th>
<th>Responses</th>
<th>Connotations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Variations</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Platform Variations</td>
<td>Time constraints</td>
<td>3</td>
<td>--</td>
</tr>
<tr>
<td></td>
<td>Sustaining robust code structure</td>
<td>4</td>
<td>++</td>
</tr>
<tr>
<td>Experimental Variations</td>
<td>Sustaining robust code structure</td>
<td>4</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Build on Original System</td>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Time Constraints</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>Copy Conceptual Idea</td>
<td>9</td>
<td>++</td>
</tr>
<tr>
<td>Boiler-plating</td>
<td>Cannot change original code</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>API/Library Protocols</td>
<td>Sustaining robust code structure</td>
<td>6</td>
<td>++</td>
</tr>
<tr>
<td>Algorithmic Idioms</td>
<td>Using better code structure based on preference</td>
<td>4</td>
<td>++</td>
</tr>
<tr>
<td>Parameterized code</td>
<td>Reuse with modification in a different context</td>
<td>4</td>
<td>++</td>
</tr>
<tr>
<td>Bug Workarounds</td>
<td>System maintenance</td>
<td>3</td>
<td>+</td>
</tr>
<tr>
<td>Replicate and Specialize</td>
<td>Reuse trusted code</td>
<td>2</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Sustaining robust code structure</td>
<td>4</td>
<td>++</td>
</tr>
<tr>
<td></td>
<td>Porting to new standard</td>
<td>1</td>
<td>+ -</td>
</tr>
<tr>
<td></td>
<td>Time Constraints</td>
<td>14</td>
<td>--</td>
</tr>
<tr>
<td>Cross Cutting Concerns</td>
<td>Large logical distance</td>
<td>2</td>
<td>+ -</td>
</tr>
<tr>
<td></td>
<td>Reuse for different version/standard</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Verbatim Snippets</td>
<td>To access same module from various locations</td>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>Persistent Modifications</td>
<td>In context of parts that do not change</td>
<td>3</td>
<td>+ -</td>
</tr>
</tbody>
</table>

In a real-world industrial setting, developers cannot avoid cloning. In an ideal setting developers should always track cloning. However, due to various reasons this practice might not be feasible. If developers have proper guidelines, they can at least flag the harmful clones they form based on a categorization. The categories from Table 3.9 together form an intent based taxonomy of code clones with a classification of harmful and non-harmful clones. Based on the results of this study, the following taxonomy can provide developers with guidelines to keep track of harmful clones and avoid the negative effects on the system. The following taxonomy attempts to categorize clones based on harmful effects related to intent and rationale of cloning.
3.2.3.1 Taxonomy Based on Cloning Intent

- **Harmful Clones:**

  - *Platform Variations Under Time Constraints* – Reusing code for platform specific changes for the intention of saving time. A properly working code on a platform might not work on a different platform.

  - *Experimental Variations Under Time Constraints* – Reusing experimental versions of old tested code for the intention of saving time. There could be risks that the modified code might be un-tested properly for deployment.

  - *Replicate and Specialize Under Time Constraints* – Modifying code for reuse for the intention of saving time. Without proper testing this activity has high risks involved of braking the system.

  - *Boiler Plating Without Access to Original Code* – Reusing code in a different context without having access to change and modify the original code might leave faults that can go unnoticed ans cause failure.

  - *Cross Cutting Concerns Over Multiple Versions* – Reusing unrelated code with similar semantic cross cutting properties over different versions. Propagation of a bug to an unrelated region of code might result in unexpected failure. Propagation of a bug to an unrelated region of code might result in unexpected failure.

- **Neutral Clones:**

  - *Replicate and Specialize While Porting to a New Standard* – Modifying existing code
for reuse in order to port the system to a new or different standard. This rationale might introduce unnecessary risks of fault injection.

– **Cross Cutting Concerns over Large Logical Distance** – Similar to above, faults might be hard to spot and might result in unexpected system breakdowns.

– **Persistent Modifications** – Such Cloning should be coupled with comprehensive testing to make sure that there are no faults. Since this code does not change much, it might be feasible to leave it simple.

Developers can use cloning of code based on any other intentions, such as sustaining robust code structure, reuse trusted code or other intentions that Table 3.9 presents, without high risks of failure or high maintenance costs. However, in ideal conditions it is always advisable for developers to keep a track of all reused code.

### 3.3 Summary

In the first part of this chapter the Section 3.1 summarized all the information that was available in the literature regarding definitions of code clones and their types. The SLR identified shortcomings of the presently available definitions and made recommendation for better and clear definitions. The main outcome of the SLR section was to emphasize the need for robust, concrete and of course clear definitions. Towards taxonomy of code clones, I discovered that the definitions of Type I and Type IV are concrete and clear. The overlap and vagueness is primarily due to the Type II and Type III clones. Note that clear definition and categorization of code clones are a secondary objective of this dissertation. Without proper definitions it would be challenging to estimate the effect of clones which leads to the primary objective of this dissertation. Also, the
SLR indicated that maximum and minimum threshold sizes for code clones are not very clear from the available literature.

To further the cause of understanding clones and their effects from a developer’s perspective, the second part of the chapter, Section 3.2, presented a taxonomy of code clones based on intent and rationale of cloning. This interview study validated a previously proposed categorization by Kapser et al. and proposed a taxonomy with a classification of harmful and non-harmful effects of cloning. If developers keep track of all the clones created due to harmful intentions, it can reduce the cost of efforts needed to fix issues related to cloning in software systems. This practice could be a low cost solution to the issues arising due harmful effects of clones. Note that the results of this study do not prove the harmful effects and future studies are required to investigate more on these lines and validate the findings.
This chapter presents a detailed synthesis of the second foundational study: Community Surveys. The two surveys gauge the specific areas that are open for further research. Following from the results of the SLR, the next step was to gather the opinions of code clone researchers across the world. This study asked important questions via an online survey. This chapter also presents a detailed methodology of community surveys and establishes its importance.

This chapter presents detailed synthesis of two community surveys aimed at gathering views of the code clone community researchers. The two surveys together constitute the second foundational study. The primary goal of the surveys was to gauge the gaps in code clone research and identify specific areas that are open for further research. This chapter also presents a detailed methodology of surveys and establishes its importance as a powerful method for identifying and documenting agreement and disagreement in a specific research area.

The code clone research community is well established and continues to grow. However, as a community grows (and becomes more diverse), the potential for divergent opinions within the community also grows. Consequently this effect diversifies topics such as the current state of knowledge, the most important research goals, and the roadmap for future research. Given that clones must be understood within the context of a particular software system or development scenario, the results of reported studies are sometimes inconsistent (or even contradictory). It appears that the code clone research community has likely reached that point. This observation is
evident from the fact that few claims regarding effects of clones on developer behavior have been validated with human studies.

An effective way to quickly gather the opinions of a large group of distributed experts is via survey. In this case, to begin to understand and document the level of agreement regarding various important topics, I along with two other researchers conducted two online surveys. The first survey focused on general topics related to clone research. This survey was partially propelled by the findings of the systematic literature review as discussed in the previous chapter. To understand and scrutinize issues in more detail, this survey was followed up by a second survey. The second survey was more focused on certain specifics aspects as compared to the more generalized latitudinally designed first survey. For the sake of clarity, in the remainder of the dissertation proposal ‘Survey 1’ will refer to the first survey and ‘Survey 2’ will refer to the second survey.

Over the years there has been valuable insight to the intention of developers giving rise to clones in a software system. While writing new code and throughout the iterative process of software maintenance, developers tend to use and reuse fragments. Reasons for reuse can be influenced by a number of factors. In general, most of the factors are related to causality between two different attributes. However, detailed verification is required to determine the presence of possible confounding attributes influencing such practices. It is widely accepted that all types of clones are not bad for the design of the system. It is very difficult to know the effect of all such intentions.

4.1 Research Topics

Based upon the literature survey and impressions from interacting with the community, three primary topics emerged for which a summary of community beliefs would be useful. The
following subsections each describe one of the topics in more detail and provide background literature that motivated its inclusion in the community survey.

4.1.1 Research Topic 1: General Clone Usage Information

This topic focuses on the definitions of clone types and on the use of clone information. For a community to progress, its members must establish concrete and agreed upon definitions for key terms. Chapter 3 discussed this issue in detail and ascertained the presence of ambiguity in the literature. In the survey, I wanted to gauge the community’s acceptance of the Bellon et al. commonly accepted classification of clone types [Bellon et al., 2007] which was reused by Roy et al [Roy et al., 2009] as stated in Chapter 1, Section 1.1.

The second aspect of this research topic concerns the ratio of cloned code to non-cloned code. In other words, this aspect tries to gauge if high clone ratio affects the software system in any particular fashion. The current status of community belief seems to be that code clones are not necessarily harmful, but do need to be tracked [Kapser and Godfrey, 2008; Kim et al., 2005; Lozano, Wermelinger, and Nuseibeh, 2007]. This would suggest that clone ratio does not have any necessary impact on the quality. Also, there are long-term risks associated with cloning, such as potential duplication of defects and possible loss of links among code fragments that must remain consistent [Göde and Koschke, 2013]. Therefore, we can hypothesize that code clones are important for system quality. Respondents answered questions about the effect of the clone ratio on code quality to clarify the point above.

4.1.2 Research Topic 2: Clones and Developer Behavior

It is accepted that clones impact maintenance. While there have been some studies focused on understanding developer behavior during maintenance tasks [Chatterji, Carver, Massengil, Oslin, and Kraft, 2011; Kamiya et al., 2002], there is not yet enough evidence to draw any
general conclusions about developer behavior regarding clone management or use of clone-aware tools. Developer adoption of clone tools is contingent on their expectations being met. However, there is a need for human-based empirical studies to determine their expectations.

In addition, understanding a developer’s intent when he or she clones code is important for identifying good and bad development practices. This information can provide insights into the types of clones that may be particularly problematic relative to quality and maintenance. Survey 2 focuses on this topic in more detail than Survey 1. Specifically, Survey 2 gathers community beliefs about how developers behave in different scenarios. The goal of the surveys is not to prove any claims, but rather to gauge the level of agreement within the community regarding claims and beliefs about developer behavior.

4.1.3 Research Topic 3: Clone Evolution

One of the new research directions relates to clone evolution, or how clones change over time. As clones change they exhibit various patterns and characteristics. An analysis of clone evolution can reveal, for example, which clones are change-prone and which clones are long-lived [Pate et al., 2013]. Developers may be able to use this information to better manage clones. In the survey, I wanted to understand the current beliefs about clones and their evolution.

4.2 Survey Design and Analysis Process

While each survey contained different questions, the process of survey design and qualitative data analysis remained the same. To encourage participation, I ensured that each survey would take no longer than 15 minutes to complete. In general, multiple choice questions take less time to answer than open-ended questions. However, due to the exploratory nature of the survey, some open-ended questions were necessary. Survey 1 contained 14 multiple choice questions (each with the option to provide an explanation of the answer), 8 open-ended questions and 8 questions that
required selection from a set of choices or a one-word answer). Using the open questions after analysis of Survey 1 data as motivation, we developed Survey 2 to focus primarily on research topic 2 (Developer Behavior). Survey 2 contained 12 multiple choice questions (each with the option to provide an explanation of the answer), 7 short answer questions and 1 question that required selection from a set of choices or a one-word answer.

Some questions did not receive adequate responses to provide useful insights, so I excluded them from discussion (six questions on Survey 1 and one question on Survey 2). Using the collection of papers on Robert Tairas’ website\(^1\), I generated a list of 71 clone experts to serve as the audience for surveys. The *12th International Workshop on Software Clones (IWSC 2012)* hosted the preliminary version, initial results of Survey 1 [Chatterji, Carver, and Kraft, 2012]. One of the other authors of this paper who presented the findings at this workshop, requested the attendees to complete Survey 2. Using this approach, Survey 1 had 22 responses and Survey 2 had 21 responses. The overall methodology is shown in Figure 4.1. The logistics of the surveys were as follows. For Survey 1, in the third week of November 2011 we sent emails to the list of potential participants inviting them to participate in the online survey. After sending out two reminders, we closed the survey in the second week of January 2012. The survey received 22 responses. We sent the survey request to the same email list as Survey 1. In addition, some participants of IWSC 2012 also participated. The authors sent the initial email out at the end of June, 2012. We closed the survey after two reminders. A total of 21 people responded to Survey 2.

Regarding the process used to analyze the qualitative data produced by the open-ended questions on both surveys, I used a systematic qualitative data analysis process called Constant Comparison [Glaser, 1965]. First, I developed a coding scheme, which is a method for organizing

\(^1\) http://students.cis.uab.edu/tairasr/clones/literature/
qualitative data to make it easier to analyze. To develop a coding scheme, the researcher goes through each qualitative response and summarizes the main point or points. Then the researcher examines the existing codes to determine if this summary matches one of them. If it does not, then he creates a new code. In other words, this process creates a set of multiple-choice answers \textit{a posteriori} rather than \textit{a priori}. With the help of this process, I reduced the qualitative data to a relatively small number of items along with counts of how many respondents provided that answer. These items were fairly easy to comprehend and analyze. To create the coding scheme, another researcher and I worked independently to code the responses. Then we met to resolve any discrepancies and arrived at a final set of codes.
4.3 Respondent Demographics

These demographics provide some context for the later results by indicating the distribution of respondents across a number of important factors. Because both surveys targeted the relatively small community of clone researchers, it was expected there would be overlap in the samples for the two surveys. To determine this overlap, Survey 2 asked the respondents whether they participated in Survey 1. Eight of the twenty-one respondents to Survey 2 indicated they were also respondents to Survey 1, nine did not participate in Survey 1 and four could not remember. Because both surveys were anonymous, it is not possible for us to link the responses of the eight who took both surveys.

Figure 4.2 shows the survey respondents’ primary and secondary research interests. Interestingly, in both surveys, the respondents were heavily focused on detection research and had little or no focus on visualization research. To provide a more detailed understanding of the respondents’ knowledge, each respondent indicated whether they were familiar with each of the following topics drawn from the IWSC 2012 Call for Papers:
A. Causes and effects of clones;

B. Effect of clones on system complexity and quality;

C. Applications of clone analysis;

D. Tools and systems for detecting and analyzing software clones;

E. Techniques and algorithms for clone detection, analysis, and management;

F. Clone and clone pattern visualization;

G. Clone evolution and variation;

H. Evaluation and benchmarking of clone detection methods;

I. Role of clones in software system evolution;

J. Clone management;

K. Clone analysis in families of similar systems;

L. Refactoring through clone analysis;

M. Clone-aware software design and development;

N. Others.

Figure 4.3 shows the results of these answers. Analysis showed that topics A, B, and C are more general than the other topics. Therefore, there was a general expectation that each respondent would select one or more of them. For Survey 1 all but one respondent selected at least one of the three general topics, with just over 50% selecting all three topics. However, for Survey 2, all but
Figure 4.3: Familiarity Distribution

four respondents select at least one of the three general topics, with only 2 selecting all three. Consistent with their response to the first question (that clone detection was their primary research interest), 80% of the Survey 1 respondents and 70% of the Survey 2 respondents selected both topics D and E, which are related to code clone detection tools and techniques.

The remaining demographic questions (shown in Table 4.1) help to characterize the respondents to provide context for the analysis discussed in Section 4.4. The responses to question D1 showed that in both surveys, most respondents work at Universities (86% for Survey 1 and 75% for Survey 2), while the balance between Research Labs (9% for Survey 1 and 10% for Survey 2) and Industry (5% for Survey 1 and 15% for Survey 2) was inverted in the two surveys. For Question D2, I argue that the responses could be ordered as follows based on years of experience (in decreasing order of credibility): Professor > Researcher > Post Doc > Graduate Student > Undergraduate Student. Figure 4.4 shows that the distribution of responses is similar for both surveys and is skewed towards the higher credibility end of the spectrum. Similarly, for question D3, more years of experience should generally translate into more credibility. Figure 4.5 shows that respon-
dents to Survey 1 appeared to be a bit more experienced than those from Survey 2. The answers to Questions D2 and D3 indicate that the respondent pool had high credibility and therefore likely provided answers that are trustworthy. Finally, Figure 4.6 shows that the survey respondents were diverse relative to country (D4), with Survey 2’s respondents being even more diverse than Survey 1’s respondents.

The following sections discuss the details of the two surveys separately in detail.

4.4 Survey 1

The main goal of Survey 1 was to determine the level of agreement among community members regarding unvalidated claims and beliefs in the code clone research area. Section 4.1
Figure 4.5: D3: Years of Experience

Figure 4.6: D4: Countries
describes the three research topics that organized the survey. Each of the following subsections provides the questions and analyses for one of those research topics. Throughout I mention the questions asked to the respondents for each survey. The survey questions have 3-level numbering scheme. For example, 1.2.3 means Survey 1, topic 2 and question number 3. Similarly 2.3.1 means Survey 2, topic 3 and question number 1 and so on.

4.4.1 General Clone Usage Information

Table 4.2 shows the five questions related to general clone usage information. Question 1.1.1 focused on whether clone ratio, which is defined as the ratio of lines of cloned code to the total number of lines of code, is an indicator of system quality. The respondents were fairly evenly divided (45% - no vs. 55% - yes) on this question. As this result does not lead to any concrete conclusions, further study is required to determine whether clone ratio is a useful measure of quality.

<table>
<thead>
<tr>
<th></th>
<th>GENERAL QUESTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1</td>
<td>Do you think that the clone ratio in a software system can be a measure of the quality of the system? Please explain briefly.</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Type 1: Code fragments are identical except for variations in whitespace, layout, and comments. Do you agree? If no, give your definition?</td>
</tr>
<tr>
<td>1.1.3</td>
<td>Type 2: Code fragments are structurally and syntactically identical except for variations in identifiers, literals, types, layout and comments. Do you agree? If no, give your definition?</td>
</tr>
<tr>
<td>1.1.4</td>
<td>Type 3: Code fragments are copies with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments. Do you agree? If no, give your definition?</td>
</tr>
<tr>
<td>1.1.5</td>
<td>Type 4: Two or more code fragments perform the same computation but are implemented through different syntactic variants. Do you agree? If no, give your definition?</td>
</tr>
</tbody>
</table>

Questions 1.1.2-1.1.5 focused on whether respondents agreed with the standard definitions
of Type 1 - Type 4 clones (as given in the questions). For Type 1 clones, while all respondents agreed with the definition, one preferred the term “identical clones” to the term “Type 1 clones”. For Type 2 clones, all but three respondents agreed. The three respondents who disagreed with the definition generally expressed concern with the definition’s ambiguity compared to the definitions of Type 1 and Type 2 clones. For Type 3 clones, 5 respondents out of 22 disagreed. The respondents who disagreed with the definition had issues with the ambiguity of the boundaries between Type 2, Type 3 and Type 4 clones. One respondent thought that Type 2 and 3 clones should be merged, two said that Type 2 clones are more like Type 4 clones, and the rest of the negative respondents were concerned about the definition of terms like ‘further modifications.’ Finally, for Type 4 clones, six respondents disagreed with the definition. One respondent suggested that Type 1 clones are the only ones that should be called “code clones” because the other types were mostly identical code. Another respondent suggested that the definition of Type 4 clones was too broad. The other four expressed confusion about the boundary between Type 3 and Type 4 clones. Figure 4.7 shows the distribution of the answers for all four clone types.
The community, apparent from the responses from the survey, were fairly divided in most of the general clone usage details such as, whether or not code clones are a good measure of system quality. There was also a fair amount of disagreement for concrete definitions of critical terms.

4.4.2 Developer Behavior

This section contained two types of questions: eight questions about specific developer actions/expectations and one question about literature claims regarding the maintenance of code clones. The former type was intended to gather the beliefs of community members while the later was intended to gather community opinions about claims. All the observations about developer behavior discussed in this section are hypotheses at this point. Additional human-based empirical studies, focused specifically on these questions, can provide the data required to validate or refute these claims. Table 4.3 shows the open-ended questions (Questions 1.2.2-1.2.9) related to this research topic and their responses.

Question 1.2.1 asked the respondents to indicate when developers address clones. The answer choices, along with the number of responses, were: while programming - 7, at the end of day’s work - 2, periodically (i.e., every week or two) - 1, at the end of a module or project - 3 or other - 8. Most respondents that answered ‘other’ indicated that it depends upon the task at hand. Interestingly enough, two respondents specifically indicated that this question needs a study to validate it. The majority of the respondents who gave an answer besides ‘other’ believed that developers should be addressing clones while programming. The results for this question indicate that there is a level of confusion in the community regarding this question. This issue also needs to be empirically validated by human-based studies.

I used analysis process described in Section 4.2 to analyze the open-ended responses to Questions 1.2.2 through 1.2.9. Table 4.3 shows the results of this analysis. The first column pro-
vides the text of the question. The second column describes the reason for including the question on the survey. The third column presents the elements of the coding scheme that resulted from the analysis. Finally, the fourth column shows the number responses for each category. Question 1.2.9 relates to a claim from the literature [Cai and Kim, 2011]. We asked the respondents for their opinion on this claim.

The responses for Questions 1.2.3 and 1.2.4 suggest that maintenance tasks which have a broad impact or affect long-term system qualities should be assisted with clone evolution information. Whereas, short term or relatively minor types of maintenance, such as defect fixing or adding modular functionality, do not require expensive information provided by the evolution tools. However, these results are only suggestions; proper studies need to be performed to collect evidence.

From the responses to the Questions 1.2.5 to 1.2.8 we derived the following observations. A majority of the respondents said that removing clone groups provides long-term benefits to quality. An open question is: What are those long-term benefits? Additionally, the responses indicate that it is better to leave cloned fragments if there is a risk involved that refactoring might render a part of the system or the whole system not to function the way it should. The respondents thought it is acceptable to independently evolve clone fragments that occur in different contexts. The respondents also thought that developers consistently propagate clones of which they are aware. Although there were quite a few responses related to this research topic, there also seemed to be a lack of consistency in the researchers’ expectations about developer behavior when exposed to clone information. This observation suggests that there is uncertainty among the community regarding issues related to developer behavior.

These responses suggest that maintenance tasks which have a broad impact or affect the
<table>
<thead>
<tr>
<th>Question</th>
<th>Intention</th>
<th>Coding</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.2. Describe a maintenance scenario or task when developers track code clones.</td>
<td>Estimate the situations in which developers track code clones so that the tools built to assist the developers can be fine-tuned to certain scenarios.</td>
<td>Refactoring</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixing bugs</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Making a change in multiple locations</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Performing a quality assessment</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>2</td>
</tr>
<tr>
<td>1.2.3. Describe a maintenance scenario or task where static clone information from the current version of the system is substantial.</td>
<td>Estimate the situations in which clone information from the current version is substantial.</td>
<td>Fixing bugs</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Refactoring</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Performing a quality assessment</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ensuring consistent propagation of clones in a group</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>0</td>
</tr>
<tr>
<td>1.2.4. Describe a maintenance scenario or task where clone evolution information over a limited history of the system is useful.</td>
<td>Estimate the situations where more detailed clone information over the multiple versions of the system might be required.</td>
<td>Judge the evolution of the system through version in terms of increase or decrease in the number of clones or other propagation related issues.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Determine history of a ghost fragment that may have diverged out of a clone group in some previous version.</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fixing bugs</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Tracking the appearance or disappearance of clones</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Deciding whether to refactor based on identification of changes that might break the system or affect its quality</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>0</td>
</tr>
<tr>
<td>1.2.5. Describe a maintenance scenario or task where you would remove a clone/clone group via refactoring.</td>
<td>To judge scenarios where developers get rid of clones from the system, thus lowering the clone ratio over a period of time.</td>
<td>Clones that can be merged into a parameterized function</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When removing clones might help improve the quality of the system giving long term benefits.</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Clones which are identical or nearly identical.</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Buggy code fragments that get affected in a similar way.</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>0</td>
</tr>
<tr>
<td>1.2.6. Describe a maintenance scenario or task where you would leave a clone/clone group untouched.</td>
<td>To judge scenarios where developers would not touch or refactor clones in order to change system quality.</td>
<td>A change might harm the system</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cloned parts that never change or are never refactored</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In case the clones have separately evolved</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not being sure of what to change</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>3</td>
</tr>
<tr>
<td>1.2.7. Describe a maintenance scenario or task where it is ok to make an inconsistent change to clones within a clone group.</td>
<td>To judge the reasons of inconsistent propagation of clones. A situation that might lead to ghost fragments.</td>
<td>Planned independent evolution to a part where the contexts of the cloned parts are different</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Planned independent evolution where the purpose varies to change a certain cloned fragment</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Planned independent evolution to add a new functionality to a particular clone in a group</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>2</td>
</tr>
<tr>
<td>1.2.8. Describe a maintenance scenario or task in which you would make a consistent change to a clone/clone group.</td>
<td>To judge a scenario where a developer would find all the cloned fragments in a clone group to evolve them consistently.</td>
<td>Fixing bugs</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Cannot be refactored or changed due to constraints</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Identical code designed such that a change one part requires a change in others</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>3</td>
</tr>
<tr>
<td>1.2.9. Clones located in different files are more likely to be refactored than the clones in the same file [Bellon 2002]</td>
<td>To estimate if a developer necessarily tries to find the code fragments or is this action causal on the ease of finding the cloned fragment.</td>
<td>Proximity enables easy identification and refactoring</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Code refactoring is easier to perform at different places in the same file than in different files.</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Depends on the situation and the level of coupling</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>0</td>
</tr>
</tbody>
</table>
long-term system qualities should be assisted using clone evolution information. Whereas, short
term or relatively minor maintenance (e.g., such as defect fixing or adding modular functionality)
do not require expensive information provided by the evolution tools. However, these are mere
indications; proper studies can be performed to collect evidence. A majority of the respondents
said that it will be better to remove clone groups for long terms benefits to the system quality. A
question that needs to be asked in future studies is: what can those long term benefits be? Further,
the responses point out that it is better to leave cloned fragments alone if there is a risk involved that
might break the system. The respondents pointed out that it is okay to execute planned independent
evolution if the cloned fragments have different context. Also, from the responses it is clear that the
survey respondents think that developers consistently propagate clones provided they are aware of
them. All of the above mentioned indications towards developer behavior are yet to be ascertained
and future planned empirical studies can be performed to prove them.

4.4.3 Evolution of Clones

The questions related to this topic focused on clone evolution including: late propagation,
consistent but asynchronous changes to a clone group [Pate et al., 2013], and the impact of system
age on clones. Only three questions received enough responses to properly analyze. Because these
questions were all open-ended, we followed the analysis process described in Section 4.2. Ta-
ble 4.4 shows the results of the analysis, using the same columns as in Table 4.3. A majority of the
respondents indicated that clone evolution information could be useful for program comprehen-
sion, though they did not specify development/maintenance tasks that might require a developer to
apply the knowledge gained.

Version aware tools or tools which track history of a software system, require more re-
sources and hence the information obtained from them is more expensive as compared to a snap-
Table 4.4: Systematic Qualitative Analysis of Evolution Related Section

<table>
<thead>
<tr>
<th>Questions</th>
<th>Intention</th>
<th>Coding</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.1. Is clone evolution information useful to developers? Why or why not?</td>
<td>To check what has happened to a clone or a clone group over a period of time</td>
<td>Old and large systems are more prone to code reuse.</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Older systems will have more inconsistencies.</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>3</td>
</tr>
<tr>
<td>1.3.2. Do you think the clone evolution pattern can be impacted by how long a system has existed (long-lived systems vs. newly developed systems)?</td>
<td>To estimate the effects of long lived systems.</td>
<td>If they are aware of the clones</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other answers</td>
<td>1</td>
</tr>
<tr>
<td>1.3.3. Developers tend to consistently propagate clone changes immediately where needed [Göde and Koschke, 2010].</td>
<td>To judge developer behavior regarding consistent propagation.</td>
<td>Other answers</td>
<td>2</td>
</tr>
</tbody>
</table>

shot tool. To estimate if the extra resources spent in capturing clone information over multiple versions of the system is worth the extra effort we wanted to ask the questions to the community to estimate the usefulness of this information. A majority of the respondents indicated that clone evolution information could be useful for some specific tasks. The respondents also agreed that the age of the system would impact cloning patterns.

4.4.4 Follow-up Survey

The results show that in some cases, such as the definition of a Type I clone, there is general agreement, while in other cases, such as the effect of clone ratio on a system, there is strong disagreement. These results indicate that there is a need for empirical work to begin providing insight into some of these questions via data that can be collected from human participants. The results of Survey 1 leave a number of open questions related to each of the research topics. The list below presents seven such research ideas.

First, it is clear that there is a general lack of consensus about the appropriate differentiation among the clone types. In order to make any significant progress in an area, consistent, agreed-upon definitions are crucial. Therefore, an open question in need of further research is:
1. *How should clones be classified so they provide differentiation that is useful for other important research questions?*

With regard to developer behavior during maintenance, there is not widespread agreement on the use of clone information in some key development activities that developers face during maintenance. Currently, many beliefs are based upon anecdotal evidence rather than objective empirical evidence. Empirical evidence from appropriate studies could help eliminate some of the disagreement. Some specific questions in need of further research include:

2. *How does when a developer addresses clones affect how they address clones?*

3. *How does the type of maintenance (i.e., broad vs. localized) affect the importance of clone information?*

4. *When is it beneficial to refactor a clone group? What benefits can be realized by this action?*

5. *In what ways can visualization of clone information help developers?*

With regard to clone evolution, there is a trade-off between the expense of keeping track of evolution information and the importance of that information for version-sensitive maintenance tasks. There is a need to understand how best to use clone evolution information. Specific research questions include:

6. *How do the lifespan and evolution pattern of a clone lineage affect developer behavior with regard to the lineage?*

7. *Can we classify clone evolution patterns according to their relationship to or impact on specific maintenance tasks?*
The largest set of unanswered questions relate to developer behavior during maintenance. This topic is also the one that was of most interest to us. Feedback from Survey 1 also indicated that providing more concrete questions would be helpful. Therefore, we developed Survey 2 to complement Survey 1 by investigating some of the open questions described above. To make the survey as concrete as possible, we included questions that asked respondents to focus on what information a developer would expect or how the developer would behave in specific maintenance scenarios. The following section provides the details of Survey 2.

4.5 Survey 2

The primary goal of Survey 2 was to gather community opinions regarding developer behavior in the face of clone-related activities and tools. Rather than covering all three research topics, as with Survey 1, this survey focused primarily on Research Topic 2 - Developer Behavior. While most questions on Survey 1 asked respondents to describe specific scenarios in which a developer would perform a given action, Survey 2 provided the respondents with specific scenarios and asked the respondents to give their opinion on which action a developer should take in that scenario. The goal of this type of question was to get more focused feedback from the respondents, as opposed to the open-ended questions on Survey 1.

More specifically, Survey 2 asked respondents to provide their opinion about the usefulness of clone-aware tools for various maintenance scenarios. It also contained questions geared towards understanding the most pressing issues and the most important studies the community would like to see. Survey 2 also followed-up on the lack of consensus about clone definitions uncovered in Survey 1. Overall, the survey covered five sub-topics: General Developer Behavior, Benefits of Clone-Aware Tools, When to Refactor, Effects of Clone Distance on Maintenance, and Clone Definitions. Each sub-topic is analyzed in a separate sub-section. The first sub-section discusses
general questions. The remaining four sub-sections, namely, benefit for developers from Clone aware tools, type of clones that should be refactored, maintenance of clones based on location and distance of the fragments and problems with the current widely accepted taxonomy of code clones, focus on specific questions that arose out of Survey 1.

4.5.1 General Developer Behavior

The goal of the first set of questions was to gather the views of the community regarding developer behavior in general (as opposed to the more specific questions that follow in later subsections). The survey contained the following three questions:

Question 2.1.1. What do you think is the most pressing research question related to developer behavior and code clones?

Question 2.1.2. Briefly describe the most important clone-related developer study that needs to be conducted (not necessarily by you)

Question 2.1.3. During which software maintenance/evolution task(s) do you believe that clone information is most useful?

Questions 2.1.1 and 2.1.2 sought to identify aspects of developer behavior that were in need of additional empirical research. Unlike the other survey questions that are aimed at determining level of agreement, these questions aim at generating a list of important issues and specific studies that are needed.

The qualitative analysis on the responses to Question 2.1.1 resulted in seven research questions. We then matched each study described in the answer to Question 2.1.2 to one of these seven research questions. We did this matching independent of how the respondent answered Question 2.1.1; i.e., we did not necessarily match the study described in Question 2.1.2 from a particular respondent to the research question that the respondent gave in Question 2.1.1, if we through the
study fit more closely to a different research question. Table 4.5 summarizes the responses and the mapping. The two “Responses” columns provide the number of responses matched to each item in the column immediately to the left.

If each respondent described a study in Question 2.1.2 that was related to the research question described in Question 2.1.1, then the total number of responses for the Related Studies should equal the number of responses for the corresponding issue. For example, items 1a-1f should total 8 because Issue 1 had 8 responses. Upon examination of the table, we noted some discrepancies, meaning that some respondents described a study that was not related to their most pressing research question. Further analysis of the data showed that five respondents fell into this category. Specifically, three respondents that listed Issue 1 as most important described studies related to other issues (i.e., studies 2b, 4a and 5a). Similarly, two respondents that listed Issue 6 as the most important described studies related to other issues (i.e., studies 1f and 3a respectively).

To summarize how a developer can feel overwhelmed by a lot of clone information, one respondent stated: “Clones are very much prevalent in all code. And since the clone results are really huge and unmanageable, there needs a mechanism to manage these results and fix duplication in code.” The responses to Question 2.1.2, described in Table 4.5, detail a large number of potential studies regarding the connection between clones and developer behavior. Specifically, there are a large number of respondents that believe a basic understanding of developer behavior with clone-aware tools needs to be studied further. This observation is in line with our own personal views about the need for additional empirical study about developer behavior and clones.

Table 4.6 summarizes the opinions of the community members regarding the usefulness of clone information during specific tasks. Code refactoring and Defect Tracing/Removal stand out. This observation suggests that most of the community believes that clone information is most
<table>
<thead>
<tr>
<th>Table 4.5: Issues and Related Studies</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Issues (2.1.1)</strong></td>
</tr>
<tr>
<td>How useful are clone aware tools to developers and in what ways can clone aware tools benefit developers?</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>What is the intent of cloning and what are the risks, benefits and trade-offs related to it?</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Do developers care about code clones?</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>When do developers address code clones?</td>
</tr>
<tr>
<td>How can clone evolution information further the cause of clone management?</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>What are the needs of the industry?</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>How can the harmfulness of code clones be judged?</td>
</tr>
</tbody>
</table>
useful in cases of refactoring and tracing of cloned code for purposes such as defect removal or in case of other refactoring needs, such as adding a functionality.

<table>
<thead>
<tr>
<th>2.1.3. During which software maintenance/evolution task(s) do you believe that clone information is most useful?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug tracing/removal</td>
</tr>
<tr>
<td>Code refactoring</td>
</tr>
<tr>
<td>Program comprehension/familiarization</td>
</tr>
<tr>
<td>Authorship/Plagiarism/Copyright issues</td>
</tr>
<tr>
<td>Managerial review process</td>
</tr>
<tr>
<td>During transition to a newer version</td>
</tr>
<tr>
<td>Reusing existing code</td>
</tr>
</tbody>
</table>

4.5.2 How Can Developers Benefit from Clone Aware Tools?

The goal of the next four questions was to understand the types of information developers expect clone-aware tools to provide during maintenance scenarios:

Question 2.2.1. How do you think software developers most benefit from Clone Detection Tools or Methods?

Question 2.2.2. How do you think software developers most benefit from Clone Analysis Tools or Methods?

Question 2.2.3. How do you think software developers most benefit from Clone Evolution Tools or Methods?

Question 2.2.4. How do you think software developers most benefit from Clone Visualization Tools or Methods?

Table 4.7 shows the results of the qualitative data analysis for each question. The first column lists the codes identified during the qualitative analysis. The other columns show the
### Table 4.7: Clone Aware Tools

<table>
<thead>
<tr>
<th>Coding</th>
<th>2.2.1 Detection Tools/Methods</th>
<th>2.2.2 Analysis Tools/Methods</th>
<th>2.2.3 Evolution Tools/Methods</th>
<th>2.2.4 Visualization Tools/Methods</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Locating/fixing related bugs.</td>
<td>5</td>
<td>-</td>
<td>4</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Understanding the evolution of the system (trend of evolution)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Tracing the history of a cloned fragment for further reference</td>
<td>-</td>
<td>-</td>
<td>5</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>Refactor duplicates</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>Impact analysis</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Understanding high level structure for detecting relations</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Focus selection for maintenance</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Reuse code/code structure</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Identify similar code structure (code comprehension) implementing a specific function, idiom, pattern</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>While planning large scale maintenance</td>
<td>4</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>Overview about redundancy in the system (harmfulness check)</td>
<td>4</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>Assessing quality using cloning as a parameter</td>
<td>2</td>
<td>1</td>
<td>-</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>None or very little benefit for developers</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>Not sure</td>
<td>-</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>Total Number of Responses</td>
<td>27</td>
<td>19</td>
<td>24</td>
<td>22</td>
<td>92</td>
</tr>
</tbody>
</table>
number of responses for each question. The last column and last row show the total number of responses across each row and each column respectively.

The results show a fair percentage of responses, 19 out of 92, indicate that the tools are of no use or very little use to developers (19% for clone detection tools, 16% for clone analysis tools, 21% for clone evolution tools and 30% for clone visualization tools). This observation contradicts our original expectation. Given that the clone research community has produced an abundance of clone aware tools, we expected that most respondents would be of the opinion that clone aware tools offer substantial benefits to developers. Thus, the responses summarized in Table 4.7 motivate research that will result in tools that target specific developer needs.

In addition to the above observations from the table, the data for this section had some interesting comments from the respondents. These questions point to industrial applications of the tools. Some of these points have been raised earlier in the discussion of question 2.1.3. One of the respondents pointed out the use of clone aware tools for higher level management. The respondent wrote “These tools are good for quality assessment to some extent, e.g., by detecting hot spot areas (parts with high clone density) or compliance violations (e.g., code should not leave a certain subsystem but has been copied). Hence, it is especially good for stakeholders on the management level (e.g., project manager, quality manager)”.

Another respondent pointed out that the clone visualization tools should always be used in conjunction with detection tools to visualize the clones on a higher grained level.

Table 4.6 and Table 4.7 point out an interesting observation. We asked the respondents about the use of clone information and use of clone aware tools. The results for both questions show that the most number of respondents believe that clone aware tools and the information from
them is most useful in the case of ‘Refactoring of duplicates’ and ‘Locating and fixing defects.’
The consistency of these responses hint at the most important application area of the tools.

4.5.3 What Type of Clones should be Refactored?

The next set of questions focused on judging whether there was a consensus regarding allowing clones to remain intact vs. refactoring them. These questions considered only “exact” (Type 1) and “near miss” (Type 2 or 3) clones, as the results of Survey 1 indicated that Type 4 clones are more confusing (and perhaps less interesting) to community members than are Type 1 - 3 clones. The questions included in this set were:

Question 2.3.1. *Do you agree that exact clones should always be refactored?*

Question 2.3.2. *Do you agree that Nearly exact (near-miss) clones should always be refactored?*

Table 4.8 summarizes the data for these two questions. The responses are grouped by whether the respondent generally agreed with the question or generally disagreed with the question. The “coding” column lists the reasons given for agreement or disagreement. The last two columns provide a count of how many respondents gave each answer.

One respondent raised an interesting question: “*Do the clones have to be maintained coherently?*” Whenever the answer to this question is ‘no,’ the clones should be refactored. Also, as mentioned by a respondent, refactoring requires naming and placing the code fragment away from its use context, which might also be a reason not to refactor out the clones.

Out of the varied responses as shown in Table 4.8 most of the respondents thought that the decision whether to refactor clones depends on the situation. There can be many reasons not to refactor a clone. However, after considering the responses from the survey, we propose the need to identify distinct scenarios that arise during development and require that the refactoring of clones be addressed.
<table>
<thead>
<tr>
<th>Coding</th>
<th>2.3.1</th>
<th>2.3.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Exact clones should always be refactored.</td>
<td>Nearly exact (near-miss) clones should always be refactored.</td>
</tr>
<tr>
<td>Agree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>They mostly indicate design errors</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>They are potential variants</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Sometimes and it depends</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>If it works do not disturb it</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Should not be refactored in case of Purposeful cloning</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Should not be refactored if they are coherent in nature</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Unrelated clones should not be refactored</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Difficulty/Risk in refactoring is a reason not to refactor them</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Disagree</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Refactoring sometime can introduce artificial dependency</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Refactoring sometimes may introduce a bug</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Clones are low level features and hence not worth the effort specially smaller clones</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>No Opinion</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>
4.5.4 Maintenance of Clones based on Location and Distance of the Fragments from Each Other

The goal of the next set of questions was to understand the specific issues that arise because of the distance between cloned fragments. Distance between fragments refers to the distance in location; for example, cloned fragments in the same file, in different files or even further apart in different directories. Two similar fragments could have independently evolved and merged into a clone group and have no interdependency or coupling. As a result of which, irrespective of the distance of the fragments, there might be cases where the clones are totally independent of each other. The questions included in this set were:
Question 2.4.1. Are clones across files difficult to maintain and why?
Question 2.4.2. Are clones across directories difficult to maintain and why?
Question 2.4.3. Are clones across subsystems difficult to maintain and why?

Table 4.9 summarizes the responses to these questions. The responses are grouped by whether the respondent thought the different boundaries introduced difficulty in clone maintenance. The “coding” column lists the reasons given for agreement or disagreement. The last three columns provide a count of how many respondents gave each answer.

Clone detection tools should help address one of the reasons that respondents found it ‘harder to spot and easier to forget’ clones across boundaries. One respondent put it nicely: “Without a clone detection tool, I think that even clones within a same file are too difficult to detect and to maintain coherently. Once you use a clone detection tool, their maintenance should be equally easier.” This quote clearly points out the need for more developer-friendly clone-aware tools.

For the question related to clones across subsystems, one of the participants said, “Not sure if the clones across subsystems are even useful.” Similarly another participant wrote “Additionally, it could be useful to avoid refactoring clones in different subsystems. They could evolve separately.”
In cases where different subsystems are designed by separate teams, it would appear that any clones identified were likely created and evolved independently. These observations lead to an important open question regarding the scope of a clone group: Should similar fragments across subsystem boundaries be clones? This question points to the need for a better and concrete definition for code clones.

Figure 4.8 illustrates the total number of responses for Questions 2.4.1, 2.4.2, and 2.4.3 regarding the difficulty of handling clones across various boundaries. The results show that most respondents believe that clones that cross a boundary are harder to maintain. This observation contradicts other research that says the distance between cloned fragments has no effect. There is a need to study this issue empirically to better understand the discrepancies.

4.5.5 Problems with the Current Widely Accepted Taxonomy of Code Clones

Because this survey was primarily focused on developer behavior, we revisited the definitions of clones in that context. The two questions in this set were:
Question 2.5.1. What is the primary benefit of differentiating types of clones?

Question 2.5.2. What are the key limitations of this clone taxonomy?

Presently, the commonly accepted clone taxonomy is based on the structural architecture of the clone. One of the respondents wrote, “The taxonomy doesn’t help the developer who is faced with the clones. For them, it would be better to identify what can (or cannot) be done with the code: refactoring etc.” Table 4.10 and 4.11 summarize the data for each question respectively.

Regarding Question 2.5.1, many respondents thought that the clone taxonomy should include qualities such as clone severity, management strategies and intentions of cloning. This type of information is different than structural information that is the basis of the most commonly accepted taxonomy. Interestingly, some respondents thought the structural clone taxonomy was not very useful. Some sample quotes regarding this issue:

- “I am not convinced that this is a useful distinction in practice ... should be part of a study to find out.”

- “...once we hit type III it is less than useful.”
2.5.1. What is the primary benefit of differentiating types of clones?

<table>
<thead>
<tr>
<th>Coding</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>It groups the clones based on different characteristics and structural commonalities.</td>
<td>4</td>
</tr>
<tr>
<td>Taxonomy based on different techniques used in clone detection that helps to determine corresponding techniques and types (which kind of clone is detected by which technique or tool)</td>
<td>6</td>
</tr>
<tr>
<td>Taxonomy based on severity of clones hence provides a sense of complexity as we go higher up the degree of clones.</td>
<td>1</td>
</tr>
<tr>
<td>Taxonomy based on Refactoring and clone management strategies</td>
<td>3</td>
</tr>
<tr>
<td>Academic purposes (communicate the problem being addressed)</td>
<td>3</td>
</tr>
<tr>
<td>No benefit or not useful</td>
<td>2</td>
</tr>
</tbody>
</table>

2.5.2. What are the key limitations of this clone taxonomy?

<table>
<thead>
<tr>
<th>Coding</th>
<th>Responses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Does not model detection techniques well enough</td>
<td>1</td>
</tr>
<tr>
<td>Does not address intent/rationale OR not based on how developers perceive code clones.</td>
<td>4</td>
</tr>
<tr>
<td>Hard to differentiate based on context of the cloned fragment of code</td>
<td>1</td>
</tr>
<tr>
<td>Does not consider harmfulness as a factor to incorporate with the taxonomy</td>
<td>1</td>
</tr>
<tr>
<td>It’s not an exclusive taxonomy</td>
<td>4</td>
</tr>
<tr>
<td>Too coarse grained</td>
<td>5</td>
</tr>
<tr>
<td>No idea</td>
<td>1</td>
</tr>
</tbody>
</table>
• “Calling Type IV a form of cloning is a bit counter-intuitive.”

The responses to Question 2.5.2 indicated that the respondents found many limitations to the current taxonomy. Apart from the limitations mentioned earlier regarding the granularity of the taxonomy or the few outlying types which should be included, there is need for a taxonomy based on the developer’s intent when creating a clone. Such a taxonomy can be used as a guide for pointing out good and bad development practices and can be helpful for developers while performing maintenance tasks. The respondents provided some interesting quotes that help summarize the key issues here:

• “...but does it correspond to the way in which programmers think about clones?”

• “The taxonomy doesn’t help the developer who is faced with the clones. For them, it would be better to identify what can (or cannot) be done with the code: refactoring etc.”

• “The current taxonomy only helps analyze the kind of changes that are made to the two pieces of code that still remains cloned. However, it does not help in any way to decide on which clones to be fixed first or which clones are impacting the quality of the code more.”

• “It does not reflect any intent or reason of change.”

This discussion indicates that there is a pressing need for new clone taxonomies based on the intent of cloning and pointing out the harmfulness of clones formed by certain specific practices.

4.6 Discussion of Results Across Survey 1 and 2

This section combines the results of the two surveys to provide some overall insights. As mentioned earlier, there were eight respondents that completed both surveys. Because the surveys
were anonymous, we could not pool those results. In this section, we make general observations across the results of both surveys. This discussion focuses on three topics that were common across the two surveys and had similar data that can be analyzed together.

4.6.1 Code Clone Taxonomy

The results of Survey 1, shown in Figure 4.7, indicate that, although the taxonomy described earlier is widely accepted, many respondents do not agree with it. The disagreements seemed to focus on the granularity of the definitions, the ambiguity of the boundaries, and whether the taxonomy was complete. Survey 2 investigated these disagreements in more detail. Results from Survey 2 showed that respondents believe that a code clone taxonomy should include qualities such as clone severity, management strategies. A respondent also pointed out the need for a taxonomy based on developer intent of cloning.

![Figure 4.9: Key Limitations to Current Taxonomy](image)

- **P**: Does not model detection techniques well enough
- **Q**: Does not address intent of cloning
- **R**: Not differentiable based on context
- **S**: Does not consider harmfulness of clones
- **T**: Not exclusive
- **U**: Too coarsely grained

Figure 4.9: Key Limitations to Current Taxonomy
The results of both surveys lead to the following conclusion. Achieving an ideal taxonomy is quite difficult. The most widely-accepted taxonomy is primarily based on structural information. Figure 4.9, which contains data from Question 2.5.2, indicates that the respondents desire a finer grained taxonomy that includes other types of clones. These results support the need for a taxonomy that is based on a developer’s intent when cloning code and classifying the harmfulness of such development practices.

4.6.2 Refactoring of Clones

The results of both surveys provide a number of reasons why developers do not refactor clones, including:

1. Refactoring might increase the risk of introducing artificial dependency due to parameterization.

2. Refactoring might involve a system breakdown.

3. Coherency between two fragments might also be a reason sometimes to leave them as they are.

4. In cases where the clones are not significant, it would be a waste of resources and time to refactor them.

The results of both surveys indicated that the decision to eliminate clones is a critical one because of the existing dependencies between the fragments and risks involved. Survey 1 respondents thought it was acceptable to refactor clones if they could be merged into a parameterized function. Conversely, Survey 2 respondents suggest that creating parameterized functions might introduce artificial dependencies with negative effects. These conflicting results do not allow for
a definite conclusion on this point. Rather, as many respondents stated, the decision whether to refactor depends upon the circumstances.

Additionally, many respondents thought clones should be left alone in cases where they were purposefully created. Similarly, many Survey 1 respondents thought it was better to leave clones alone if they evolved independently. If cloning is performed in lieu of imitating a robust code structure or a good design pattern, it should not be considered harmful. For instance if a novice developer imitates a similar design used by another expert developer, apart from the learning opportunities for the young developer it also involves a robust structure being reused. Regardless of the reason for the clone, documentation should always be provided to ease maintenance. These are mere conjectures and the need for empirical validation remains.

4.6.3 Maintenance of Clones based on Location

The respondents to both surveys indicated that increasing the distance between clones also increases the complexity of managing those clones. Figure 4.10 shows the results from Question 1.2.9. on Survey 1 and Questions 2.4.1, 2.4.2 and 2.4.3 on Survey 2 (mean of the three answers). There were many reasons given by the respondents, including:

1. the chances of the fragments to differ in context are greater;
2. proximity enables easy identification so further they are, harder to spot; and

3. authorship issues, where chances are that different authors might have written the code as in case of independent evolution.

Many of these situations could be addressed with a clone detection tool and educating developers on its use.

4.7 Conclusion

Across the two surveys, we identified and analyzed community perspectives with Survey 1 focusing on general topics and Survey 2 focusing on developer behavior. Based on those results, here we lay out a roadmap for clone research to help resolve the inconsistent views and inconclusive results regarding clones and developer behavior. Even though Survey 2 probed further into some of the open questions from Survey 1, there are still a number of open questions that are unvalidated and can serve as a starting point for further research. In addition, the Survey 2 respondents identified a set of specific questions in need of research. Using all of this information, we identified a focused and objective set of open questions. These questions touch on two different phases of the clone lifecycle: formation and management. The open questions are:

1. **Formation of Clones:** This set of questions focuses on the way clones are formed during development.

   (a) What are the specific reasons for purposeful cloning and how do purposeful clones affect system qualities?

   (b) What are the good and bad intentions of cloning and what are the advantages and risks involved in such practices?
(c) How can we differentiate clones based on their harmfulness and the intent of cloning and what is the relation between them?

2. **Management of Clones:** This set of questions focuses on the way developers handle clones and clone information.

   (a) What are the expectations of a developer from a clone-aware tool during management and maintenance tasks?

   (b) What are specific scenarios where clones should be refactored?

   (c) Does the distance between clones matter in regards to the impact factor and the ease of management of clones?

This study will hopefully encourage the clone community to perform various empirical research to provide insight into these questions. To further this cause, this chapter proposes the following roadmap towards achieving the goal to understand the causes and effects of clones and clone management:

**Step I)** Develop a taxonomy of clones based on intent of cloning.

**Step II)** Identify the types of clones that are harmful and those that are not.

**Step III)** Develop tools that can detect clones that might negatively affect the system.

**Step IV)** Train and educate developers on how to avoid the bad practices of cloning and using the tools in order to remove such harmful clones.

The four steps above impact the formation and maintenance of code clones. Reducing the bad effects of cloning by reducing harmful practices of development and knowing what types of
clones negatively affect system qualities would certainly be able to bring the maintenance costs down. Substantial research efforts would be needed to observe developers and validate the taxonomy and the tools that can detect clones in lieu of this newly proposed paradigm.

The results from the foundational studies laid the foundation for the empirical studies that will be explained in the following chapter. Considering the width of the steps above, it will be challenging to address all of them.

Across the two surveys, I identified and analyzed community perspectives on a number of important clone topics. As explained in Chapter 1 and as shown in Figure 1.6, these three fundamental studies lay the foundation for further empirical investigation which will be discussed in Chapter 5.
EMPIRICAL HUMAN-BASED VALIDATION OF CLONE MAINTENANCE

Researchers have recently developed techniques for clone detection and tools to implement those techniques. However, additional data about the clones in the system are needed to assist developers who must make decisions regarding these clones during software maintenance tasks. For example, when a change is to be made to one clone fragment in a clone group (collection of clone fragments), a developer must determine whether the change should be made to all other clone fragments in that clone group. Such decisions are critical, because if the developer makes an incorrect decision, then a defect could be introduced. A developer’s analysis and decision-making process regarding code clones can be complicated further by the sheer volume of clones that are present in a large software system. The primary objective for the human-based empirical studies was to observe software developers in real-life maintenance tasks with respect to code clones.

While code clones are not inherently harmful [Cordy et al., 2004; Geiger, Fluri, Gall, and Pinzger, 2006; Kapser and Godfrey, 2008; Rahman et al., 2010; Thummalapenta et al., 2010], they are associated with long-term risks to software quality and maintainability. Relative to quality, if code with a defect is cloned, the defects are also cloned. Relative to maintainability, defect fixes and other changes must be propagated to all cloned fragments potentially resulting in increased maintenance effort. In addition, explicit or implicit links among cloned code fragments must be maintained when those fragments need to remain consistent.

Researchers have used retrospective analysis of software repositories to study the effects
of code clones on quality and maintainability [Aversano, Cerulo, and Di Penta, 2007; Bettenburg et al., 2009; Harder and Göde, 2012; Juergens, Deissenboeck, Hummel, and Wagner, 2009; Krinke, 2007, 2011; Mondal, Roy, Rahman, Saha, Krinke, and Schneider, 2012; Monden, Nakae, Kamiya, Sato, and Matsumoto, 2002; Thummalapenta et al., 2010]. These post hoc studies have focused on qualities such as genealogical traits [Bettenburg et al., 2009; Thummalapenta et al., 2010] and clone propagation [Krinke, 2007]. While post hoc studies can reveal important information, they do have limitations. For example, these studies examine snapshots of the source code, but are not able to understand the actions that occurred in between the snapshots. Hence, these studies may miss valuable information that could provide insight into the maintenance process. Therefore, to understand the effects of code clones on software maintainability, there is a need to actively observe developers as they maintain code. There is currently a dearth of this type of human-based empirical studies to complement the retrospective studies of code repositories.

This chapter describes the completed empirical validation studies of this dissertation work thus far. The objective of the first study was to determine the effects of clones and use of clone information in a defect localization task. The objective of the second study was to summarize the challenges faced by developers in a cloned-defect refactoring task. The participants of the two observational studies mentioned in Section 5.1 and 5.2 were students undertaking software maintenance tasks. To validate the findings of these studies, in a third study, I interviewed full time software developers. Section 5.3 presents the

Section 5.1 and Section 5.2 present the Defect-Localization and the Defect-Refactoring studies respectively. The Defect-Localization study was published in the International Symposium on Empirical Software Engineering and Measurement (ESEM), 2011 [Chatterji et al., 2011].
The Defect-Refactoring study was published at 20th Working Conference on Reverse Engineering (WCRE), 2013 [Chatterji, Carver, Kraft, and Harder, 2013].

5.1 Defect-Localization (DL) Study

Due to the lack of empirical studies in the literature on code clone maintenance, it appears that the creators of the clone detection approaches assume that providing a developer the report from a code clone detection tool would ensure its usability. In searching the literature, we were unable to find any human-based empirical studies that verify the usefulness of the information provided in the code clone reports that are output from the tools. Therefore, the main motivation behind this study was to observe how developers use the output from a code clone tool to perform the maintenance task of localization of defects, without being given specific instructions on how to use the tool output. Often, in an industrial setting, developers use new tools without the benefit of detailed training. Therefor, our experimental setup mirrors that reality.

The following research questions drive the study:

**DL-RQ1. How do developers use the information from a clone report produced by a code clone detection tool while performing a defect-localization task?**

**DL-RQ1.1 Is the information in a clone report useful for finding defects?**

**DL-RQ1.2 Does the information from the clone report lead developers to identify false-positives?**

**DL-RQ2. Do novice and professional developers use the information from the clone report differently? If so, how?**
5.1.1 Study Overview

The goal of this study was to understand how developers use the information generated by a code clone detection tool to support a standard maintenance task. To make the study tractable, the maintenance task was defect-localization (i.e., identifying the source code elements that must be modified to correct a defect) rather than repairing the defect. This choice of tasks is logical because clone information is more useful in identifying cloned code that must change than it is in actually changing the code.

5.1.1.1 Software System: Apache Ant 1.6.5

Because the difficulty of the defect-localization task is highly dependent upon the software system being examined, we determined that the software used in this study should:

• be large enough to make defect identification a nontrivial task;

• be small enough that participants could perform the task in one hour; and

• contain both a cloned defect and non-cloned defect to control for the placebo effect.

To meet these requirements we chose Apache Ant version 1.6.5. Apache Ant automates the software build process, similar to Make. Version 1.6.5 contained two distinct defects that met the requirements. Below is a detailed description of the two defects.

We retrieved the two reports for the defects from the Apache Bugzilla repository. The IDs for the defects are 38175 (affected Copy.doFileOperations and Copy.doResourceOperations) and 38082 (affected Scp.parseUri). Both defects were reported in version 1.6.5. However, the first defect was actually introduced into the trunk after the release of version 1.6.5. Thus, revision 367315 of the trunk of the Ant Subversion repository was used in this study. This revision
was the last in which both defects existed. In particular, the defects in \texttt{Copy.doFileOperations}, \texttt{Copy.doResourceOperations}, and \texttt{Scp.parseUri} were repaired in revisions 367316, 367342, and 417590, respectively. The Bugzilla repository only provides the reproduction rules for the Windows platform. However, because the participants in our study worked on a Unix platform, we also provided the reproduction rules for the Unix platform. The descriptions of the defects and the reproduction rules are as follows.
Defect 1: The failonerror="no" doesn’t work for locked file. When we try to perform a recursive copy in a directory that contains a locked file, the copy fails before the end of the whole copy, even if I have the attribute failonerror set to "no".

Reproduction: If Mozilla Thunderbird is open, there is a locked file in the profile directory (parent.lock). Figure. 5.1 shows the reproduction of Defect 1.

Defect 2: SCP Task password with special characters. The scp task does not handle password with special characters like "@". Figure. 5.2 shows the reproduction of Defect 2.

The first defect, Defect 1 appeared in two clone fragments within the same clone group (and within different methods of the same class). We use the term fault to refer to an instance of a defect. In this case, there were two faults related to Defect 1. Defect 2, the second defect appeared in only one code fragment (i.e., no clones). There was only one fault related to Defect 2. Using the defect identification form in Figure. 5.6, each participant had the opportunity to identify three faults (both instances of Defect 1 and one instance of Defect 2). The two defects were actual, user-reported defects available in the Apache Ant Bugzilla repository. We used solutions posted in Bugzilla to verify the correctness of the participants’ solutions. Because the solutions were available on the web, the participants were not allowed access to the internet during the study.

5.1.1.2 The Clone Report

We used CCFinder version 10.2.7 to detect the clones. CCFinder records clone pairs, to which it assigns unique identifiers, in a plain text file. In addition, for each cloned code fragment, it records token range in a collection of plain text files (one file for each input file). CCFinder provides an industrial-strength GUI to analyze and visualize the detected clones. However, to counter the need for training the participants to use this complex GUI, we post-processed the plain text files and provided a simplified report, described below. The post-processing included merging
related clone pairs to form clone groups and converting the token ranges to literal source code fragments.

**Code Clone Report Format:** The forms of clone group entries and source code are shown in Figure 5.3. The format was as follows: clone group entries (separated by blank lines), a horizontal rule (60 dashes), and source code entries (separated by two horizontal rules (40 dashes).

5.1.2 Study Design

This section describes the participants, the training and the task performed.

**Participants:** Forty three subjects from graduate software courses participated in the study, 13 from the University of Alabama (UA) and 30 from the University of Alabama in Huntsville (UAH). Some participants were novices (i.e. they had little or no industrial experience) while others had industrial experience.

**Training:** In both courses, the course instructors gave the participants a brief lecture about code clones and their impact on software development and maintenance. These lectures helped the
participants become familiar with the terms code clone and code clone report. The lectures did not give the participants specific guidance on how to use the clone information to perform various software engineering tasks. This information was not included in the training because the goal of the study was to test the efficacy of the clone report for the average developer, not one with detailed knowledge about code clones and the use of code clone information. After the training, the participants received an overview document containing the following information needed to complete the experimental tasks.

1. A brief introduction to code clones, an overview of the Apache Ant system and definitions of important terms;

2. The directory structure of Apache Ant 1.6.5; and

3. Information on how to reproduce the two defects.

The participants also received the Ant documentation, the j2dsk-1.4.2 documentation, and the clone report.

The Task: Based on the information provided during training, the participants were instructed to identify the specific location (file, method and line number) that should be modified to correct each defect. They were told that they should identify all portions of the code that must be modified to correct the defect. The participants were not expected to actually make the repair. All tasks were conducted in a Linux environment. The participants were allowed to use the standard Linux utilities "find" and "grep" within the terminal or were allowed to use the search dialog box provided by Ubuntu. All analysis was performed statically (i.e., the participants could not compile or execute the code). We used this approach to avoid introducing any bias due to the participants’
familiarity with the development environment (e.g. the compiler or the execution environment). We wanted to ensure we focused the participants on the clone information rather than the functionalities of the development tools. A similar approach is used in the study by Fry et al. [Fry and Weimer, 2010]. Once a participant identified a defect, he recorded his findings on the defect identification sheet described in the next section.

5.1.3 Data Collection

We collected two types of data: data from naturalistic observation and self-reported data.

5.1.3.1 Naturalistic Observation:

During the Defect-Localization tasks, two of the researchers stood behind the participants and watched over their shoulders (i.e., as passive observers). In the study by Kim et al. [Kim et al., 2004], the researchers began by using naturalistic observation. They later shifted to automated data collection when they noticed that the observers were creating an unnecessary disturbance to the participants. In our case, the observers did not interfere with or disturb the activities of the participants. In addition, the observers were able to answer the participants’ questions about the conduct of the study.

Figure 5.4 illustrates the experimental setting. The participants performed the study in
groups of four. To observe all four participants simultaneously, the two observers had to observe multiple participants as follows. Each observer was responsible for two participants at a time. Every two minutes, they alternated between observing each of their two participants. After 30 minutes (one-half of the experimental session), the observers switched positions and observed the other two participants for the next 30 minutes.

For example, referring to Figure. 5.4, Observer 1 began by observing Participant 1 for two minutes while Observer 2 observed Participant 3. After two minutes, Observer 1 observed Participant 2, while Observer 2 observed Participant 4. After two minutes, Observer 1 and 2 returned to observing Participants 1 and 3, respectively. This pattern continued for 30 minutes. At that point, Observer 1 began observing Participants 3 and 4 while Observer 2 began observing Participants 1 and 2, switching every two minutes just as before.

The motivation behind this somewhat complicated data collection method was to help ensure the accuracy and consistency of data collected. The observation interval was set at two minutes so that it was long enough for the observers to observe what the participants were doing while being short enough for making as many iterations as possible to provide good insight into each participant’s process. Also, by having the observers switch places after 30 minutes, we helped ensure that data was gathered consistently regardless of the observer. The pilot study, discussed later in this section, proved to be useful in finalizing the length of the observation interval. In sessions with less than four participants, the observers observed the participants for two minutes, then did not observe them for two minutes to ensure that the same data was collected throughout the study. For example, if there were only 3 participants, Observer 2 would observe Participant 3 for two minutes and then observe no one for two minutes.

The observers maintained a ‘fly on the wall’ perspective during the experiment session un-
less the participants specifically asked questions. The questions asked by the participants included: "What exactly are we supposed to do here?"; "How do I read the clone report?"; "Can I execute the code?"; "Can I access the internet?"; and "I am not able to find the defect, what should I do?". The observers kept their answers as brief as possible and did not answer certain types of questions that would have unfairly aided a participant in completing the task. In this way the observers strove to not introduce any type of bias into the study. During the naturalistic observation, the observers recorded the following information:

- the resources used (i.e. the code, the code clone report, the report form or the overview sheet);
- actions performed; and
- subjective notes about the participant’s actions.

To record this information, the observers used the Observer’s Data Recording Form shown in Figure. 5.5.
5.1.3.2 Self-Reported Data

Participants self-reported data on two data collection forms: the defect identification form and the post-observation questionnaire. The participants used the defect identification form (Figure 5.6) to report the identified faults. The form included places to record a time stamp, location, description, steps taken to locate the defect and how they would fix the defect. Even though only Defect 1 had two associated faults (the original and the clone), so as not to bias the results, the defect form provided space to report two faults for each defect. Such a form allowed for the possibility of three correct faults and one false positive. The information from this form helped us calculate the number of faults detected, the number of false positives, if any, and the time required to find them.

On the post-observation questionnaire the participants reported their level of experience with software development, defect-localization, code clones and the CCFinder tool. The participants also could give their opinion of the usefulness of the clone report for defect-localization.

5.1.4 Pilot Study

We conducted a pilot study to validate and debug the study design. Four graduate students, who were not going to participate in the main study, participated in the pilot study. The pilot study

![Figure 5.6: Defect Identification Form](image)
took place in a similar environment as the main study. The participants performed the same task on the same system. The results of the pilot study motivated some changes to the originally planned study design.

First, during the pilot study, the observers used a one minute observation interval. We determined that one minute was not long enough to observe and record the necessary information. As a result, we used two minute intervals in the main study. This two minute interval ensured that the observer could look closely at the work of the participants and record more accurate data.

Second, the pilot study participants suggested some other minor study design changes. These changes were mostly concerned with the overview document and the defect identification sheet to arrive at the versions used in the main study. For example, we rearranged the sections of the overview document and rephrased some questions on the defect identification sheet.

5.1.5 Data Analysis

This section is organized around the research questions presented in Section 5.1. I used an alpha value of 0.05 for judging statistical significance in all tests. Before presenting the analyses, we first give an overview of the data.

5.1.5.1 Overview of the Data

Before getting into analysis and results, it is important to understand the overview of the data and the classification of participants based on experience. Each participant can be characterized using two variables. The first variable, experience came from the post-observation questionnaire. The second variable, clone report usage strategy, came from the observational data. For the experience variable, on the post-observation questionnaire each participant used a five-point scale to report his level of experience. Participants who had either no experience or only classroom
experience were characterized as novices (23 participants). Those who had industrial experience were characterized as professionals (20 participants).

For the clone report usage strategy variable the observers noticed that each participant seemed to employ one of two strategies when using the clone report. The first strategy, called after, was to use the code clone information to search for cloned code only after first identifying a fault. The second strategy, called before, was to use the code clone information before identifying a fault. Because many participants were unfamiliar with code clone information, during the first 10 minutes a participant was allowed to familiarize himself with the clone report and still be characterized as using the after strategy.

Figure 5.7 illustrates each participant based on the number of faults found and whether s/he reported the false positive. The vertical axis represents the participant’s clone report usage strategy.
(before or after) and the horizontal axis represents the participant’s level of experience. Hence, all participants on the left of Y-axis are novices (represented by triangles) and on the right are experts (represented by squares). Similarly, all participants below the X-axis used the before strategy and above the axis used after strategy. These two axes partition the space into four quadrants based on these two variables. Each of the concentric circles contains data points representing participants who found a specific number of faults. Within each circle, the distance from the center point is meaningless. For example, the participants represented by the two data points lying in the lower right quadrant of the outermost circle each found three faults. One of those participants also reported the false positive. The largest group of participants was those who found zero faults.

Prior to this study, most participants had little experience identifying defects in large software systems. The data showed that participants, who had some software development experience, as reported in the post observation questionnaire, were more confident and found more defects than those without such experience. The few participants who found all three faults had all worked on multiple industrial software projects. Another observation from this data is that in spite of having a detailed clone report; several participants were able to locate the initial fault but did not find the second fault for Defect 1 (i.e. the clone).

5.1.5.2 DL-RQ1.1 - Is Clone Information Useful?

On the post observation questionnaire the participants used a 5-point scale (1-not effective at all to 5-highly effective) to report the efficacy of the clone report for the defect-localization task. The scatterplot in Figure. 5.8 shows the relationship between this efficacy value and the number of faults found by each participant. Because neither variable is normally distributed (based on the Shapiro-Wilk test) we conducted a Kendall’s tau-b correlation. The correlation was .196 and was
not significant (p=.13). Hence the results do not show a significant causality between efficacy of report and success in the task.

We also wanted to see if there is a significant difference in success between participants who found the clone report useful and them who did not find it useful. We split the participants into two groups: those that had a positive impression of the clone report (4 and 5 on the scale) and those that had a negative impression (3 or lower). We then performed a non-parametric Mann-Whitney U test to compare the average number of faults found by the participants in each group. The participants who had a positive impression of the clone report were more effective than those with a neutral or negative impression of the clone report (1.45 faults vs. .78 faults on average). This difference was not significant. Hence we cannot conclusively say that participants with a positive impression of the clone report performed better. However, the trends do indicate that.

We suspected that the analysis may be skewed by the large number of participants who found no defects. Therefore, to further analyze these results, the 19 participants who found no defects were excluded and the data reanalyzed. Figure 5.9 shows a scatter plot of the data. In this

Figure 5.8: Positive Relation Between Number of Clones Found and the Clone Efficacy
case the Kendall’s tau-b coefficient increased to .468 and was significant \((p=.009)\). In addition, a non-parametric Mann-Whitney U test showed that those who found the clone report useful were significantly more effective in locating the two defects Defect 1 and Defect 2 (2.29 faults vs. 1.47 faults) \((p = .049)\), where Defect 1 had two faults, than those who did not. This observation indicates that participants who achieved some success in the task did find the report useful.

The data does not allow us to draw any conclusions about causality. We are not sure whether 1) finding the clone report useful caused the increase in defects found, 2) finding more faults resulted in a more positive view of the clone report, or 3) some other variable influenced the results. Further study is required.

As a second analysis to address this question, we investigated whether the clone report usage strategy impacted effectiveness. If a participant fully understood how to use the clone report, he would realize that the clone report was only useful after identifying a fault to help search for clones of the problematic code. Because the clone report is merely a list of cloned code, it is not useful for identifying the original fault. Figure 5.10 and 5.11 illustrate the overall trend that
participants who used the clone report after finding a fault tended to identify more real faults and fewer false positives than those who used it before finding a fault. The qualitative observations indicated that many of the 35 participants who used the clone information before finding a fault appeared to be attempting to use the clone information to locate the initial faults. This observation suggests that developers need at least a small amount of training on how to use clone information in order to use it effectively.

Following on this discussion of clone report usage strategy, we conducted two analyses. First, we tested whether either strategy made the participants more effective. Those who used the clone report after finding a fault found an average of 1.62 faults compared with 0.8 faults (out of a possible 3) for those who used the clone information before finding a fault. A non-parametric Mann-Whitney U test showed that this difference was not significant (\( p = .059 \)). Second, we tested whether the participants who used the clone information after finding a fault were more effective in finding the cloned fault related to the defect Defect 1. Twenty-five percent of the participants who used the clone information after first finding a fault did find the cloned fault whereas only 5.71%
Figure 5.11: Correct vs. Incorrect Use of Clone Report

of the participants who used the clone information before finding a fault were successful in finding the cloned fault. However, this difference was not significant ($p = .094$).

Additionally, we computed the percentage of time each participant spent using each of the three resources provided to them (i.e., the clone report, the code base and the documentation). During each observation, the observer noted which resource(s) the participant used. If a participant used more than one resource, the time was divided accordingly. For example if during an observation a participant used only the clone report, 100% of the time was assigned to the clone report. If a participant used both the clone report and the documentation then each resource was assigned 50% of the time. Similarly, if the participant used all three resources, each resource was assigned 33% of the time. We made an assumption that the distribution of time between the resources would be the same for the time that each participant was not being observed.

Based on the number of faults found, we divided the sample into three groups: those who found no faults, those who found at least one fault, but not the cloned fault, and those who found all three faults, including the cloned fault. Table 5.1 shows the average percentage of time the par-
participants from each group used the resources. The participants who found no faults used the clone information the most, followed by those who actually found the cloned fault. It is not surprising that the participants who found the cloned fault used the clone information more than those who did not find it. It is surprising to see that the participants who found no faults used the clone report more than anyone else.

<table>
<thead>
<tr>
<th>Faults Found</th>
<th>Clone Report</th>
<th>Code Base</th>
<th>Documentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No faults found (19)</td>
<td>27.95</td>
<td>54.16</td>
<td>17.89</td>
</tr>
<tr>
<td>One or two faults found (20)</td>
<td>14.5</td>
<td>74.5</td>
<td>10.7</td>
</tr>
<tr>
<td>All three faults found (4)</td>
<td>22.75</td>
<td>71</td>
<td>6.25</td>
</tr>
</tbody>
</table>

A chi-square test indicated that the distributions among the three groups was significantly different ($\chi^2_{24} = 13.727; p = .02$), which is consistent to the earlier observations. At this point we cannot conclusively explain this result. However, the consistency of observations do indicate certain trends to be further researched upon. It seems to indicate that those who found no faults were unsure of how to use the clone report and therefore were using it incorrectly. Logically, if a participant did not find any faults, they should not have had little use for the clone information, as it is most useful in finding cloned faults.

5.1.5.3 DL-RQ 1.2 - Does the information from the clone report lead developers to identify false-positives?

This research question evaluated whether the clone report misleads a developer, resulting in the reporting of false positives while looking for the clone of a defect. None of the eight participants who used the after strategy reported a false positive. All five participants who did report a false positive used the before strategy. While a Mann-Whitney U test showed that this result was not significant (likely due to small number of participants who used the after strategy), the fact that no
one who used the *after* strategy reported a false positive suggests that there may be an important phenomenon here.

5.1.5.4 **DL-RQ 2 - Novice vs. Professionals**

To determine the effects of the participants previous programming experience, we compared the effectiveness of novices and professionals. On average, the professionals identified more faults than the novices (1.53 vs. .43 respectively). This difference was significant according to a non-parametric Mann-Whitney U test (p < .001). In addition, all four participants who correctly identified the cloned fault were professionals. The professionals also tended to employ the *after* strategy of using the clone report more often than the novices did. Table 5.2 is the contingency table showing the distributions of these two variables. Even though a chi-square test did not show a significant result ($X^2 = 1.010; p=.315$), there is a positive trend.

<table>
<thead>
<tr>
<th>Novice</th>
<th>Before</th>
<th>After</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>3</td>
<td>23</td>
</tr>
<tr>
<td>Professional</td>
<td>15</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>35</td>
<td>8</td>
<td>43</td>
</tr>
</tbody>
</table>

5.1.6 Threats to Validity

This section presents the threats to validity of the study.

5.1.6.1 **Construct Validity**

We took great care to make sure our observational process recorded data correctly. However, since there was a two minute gap between observations, a threat remains for the observers to miss some information. The estimations made from these observations could be a threat to the construct validity. The repeating observational pattern reduced this threat to some extent, as the intervals were shortened by increasing frequency of time periods.
5.1.6.2 Internal Validity

There was potentially a subject expectancy threat that could have arisen from the fact that we gave the participants a clone report. The participants could have assumed that the clone report was supposed to be used for the defect-localization task without fully understanding how to properly use it. Choice of the programming language could also have been an internal validity threat. We chose a system written in Java because we assumed most students would be familiar with that language. If someone was not familiar with Java, it could bias the results. Ideally the participants should have had an option to select the programming language with which they were most familiar. The participants performed the task in a Linux environment which could be a validity threat if they were not familiar with Linux and the search features it provides. Although nothing major, since some participants could have performed better than they did, there might have been a small impact on the results. Finally, to prevent a threat to validity of the participants being able to locate the solutions on the web, we did not provide internet access. Similarly, the participants were not allowed to execute the code. This approach reduced a threat to internal validity that the results may be caused by participants’ familiarity with the development and execution tools.

5.1.6.3 External Validity

The participant population consisted of both novices and professionals, reducing a threat to external validity. Conversely, the task was performed in isolation rather than as part of a complete maintenance process, which may introduce a threat to external validity. The task was a realistic task (i.e. defect-localization), but it was not complete (i.e. the participants did not fix the defect). Also, because we increased internal validity by preventing the participants from compiling or executing the code, the way a developer would typically work, there is a threat to external validity. Further,
the participants were not trained in using the clone report. Although this lack of training could introduce a threat to validity, one of our study goals was to see how people would use the report without training. Finally, the fact that the study was done as a part of a classroom course introduces an important threat to the external validity of the study.

5.1.7 Conclusion from this Study

This exploratory study provides insights into how developers use the information from a clone report. Some concrete conclusions from the study are:

- Initial evidence shows that, without training, most participants employed the before strategy of clone report usage, which appears to be less useful than the after strategy. Some researchers argue that clone detection tools are needed to support the maintenance process because those tasks are often assigned to entry level developers. However, if those developers are not able to effectively use the clone report, such clone detection tools are of little use to them.

- There is a relationship between effectiveness and employing the after strategy of clone report use, although we have yet to establish a causal relationship.

- The clone report does help developers locate clones of a defect as participants using the after strategy were more successful than those using the before strategy.

- Use of the clone report may also reduce reporting of false positives.
5.2 Defect-Refactoring (DR) Study

This section describes the Defect-Refactoring study in detail. As mentioned earlier, this study was a combined effort from me, Dr. J. Carver, Dr. N. Kraft and Jan Harder [Chatterji et al., 2013].

In the Defect-localization study [Chatterji et al., 2011] presented in the previous section, developers used a clone report to help them locate defects in cloned code, but did not actually repair the defects. The results of this study showed that proper use of clone information was helpful. While the results of the Defect-localization study provided insights into the behavior, efficiency, and effectiveness of developers tasked with understanding and locating defects in cloned code, it had limitations. Another study by one of the authors of this (Defect-refactoring) study [Harder and Tiarks, 2012], observed developers fixing defects in cloned code, but were not provided with clone information. This study did not find any significant difference in the difficulty of fixing cloned defects compared with non-cloned defects.

To better understand the factors that may have affected the results in these prior studies, to provide additional evidence regarding the effects of code clones on software maintainability, and to understand the impact of providing developers with clone information during a debugging task, I, Dr. Carver and Dr. Kraft got together with Jan Harder to design a new study to replicate the defect removal study by Harder and Tiarks [Harder and Tiarks, 2012].

Replication of experiments in different environments with different populations is necessary for validation and generalization of results [Andersson, 2007; Juristo and Vegas, 2009; Shull, Basili, Carver, Maldonado, Travassos, Mendonça, and Fabbri, 2002]. Although the level of involvement of the original experimenters in the replication is a debated topic [Juristo and Vegas,
we (researchers’ group at the University of Alabama) opted to collaborate with Harder to ensure the fidelity of the replication and to draw the most benefit from his experiences conducting the original study. During the replication, Harder acted as a consultant, helping in the setup of the study design and the data analysis. He shared the laboratory package used for the original study which consisted of the archives of the Eclipse plugin used for this study and the source code for the software systems along with an instruction manual.

5.2.1 Study Design

The overall goal of this study was to investigate the effects of cloned code on software maintainability via a replicated experiment. In this case, the replication was an extended replication, which has two parts:

- The Replication - a strict replication of the original study [Harder and Tiarks, 2012] to confirm (or refute) the results of the original study, and
- The Extension - an extension of the replication to determine whether developers were more effective in removing defects when provided with code clone information.

Harder and Tiarks developed a laboratory package for the original study that investigated the effects of cloned code on the performance of developers performing debugging tasks [Harder and Tiarks, 2012]. We used this laboratory package to conduct the Replication and an augmented version of it (which included clone information about the subject software systems) to conduct the Extension. We used the same study design for both the Replication and the Extension, with the exception of providing the participants with clone information during the Extension. Compared to
the original study design of Harder and Tiarks, the design of the replication introduced a few minor changes.

First, in the original study, each participant performed the study tasks during one of several sessions along with a small number of other participants, whereas in the replication, each participant performed the study tasks during two consecutive, 75-minute class meetings along with his/her classmates. Second, in the replication, each task was treated as a classroom assignment, and participants were given credit for attempting the tasks (rather than for completing the tasks successfully). Students in three distinct courses participated in the study.

5.2.1.1 Research Objectives

The study addressed three research questions:

**DR-RQ1**: Does the time needed for a defect removal increase when the defect is cloned?

**DR-RQ2**: Does the probability of incorrect defect removals increase when the defect is cloned?

**DR-RQ3**: Does the performance of participants improve when they are provided with clone related information?

The *Replication* addressed *DR-RQ1* and *DR-RQ2*, which we adopted from the original study. The *Extension* addressed *DR-RQ3*. In particular, we formulated and tested three null hypotheses, each of which is motivated by one of the research questions:
\( H^\text{time}_0 \) The time needed for removal of a cloned defect is \textit{less than or equal to} the time needed for removal of a non-cloned defect.

\( H^\text{corr}_0 \) The probability of a correct removal of a cloned defect is \textit{greater than or equal to} the probability of a correct removal of a non-cloned defect.

\( H^\text{cinf}_0 \) The probability of a correct removal of a defect without clone information provided is \textit{greater than or equal to} the probability of a correct removal of a defect with clone information provided.

The corresponding alternative hypotheses are:

\( H^\text{time}_A \) The time needed for removal of a cloned defect is \textit{greater than} the time needed for removal of a non-cloned defect.

\( H^\text{corr}_A \) The probability of a correct removal of a cloned defect is \textit{less than} the probability of a correct removal of a non-cloned defect.

\( H^\text{cinf}_A \) The probability of a correct removal of a defect without clone information provided is \textit{less than} the probability of a correct removal of a defect with clone information provided.

5.2.1.2 Variables

The variables are the same as in the original study.

• \textbf{Independent Variables}

A. Programs — We used two software systems for the maintenance tasks: Frozen Bubble and Pacman (described in more detail in Section 5.2.1.4).

B. Versions — Each software system had two different versions: one containing a multiple-instance (cloned) defect and one containing a single-instance (non-cloned) defect.
• **Dependent Variables**

A. Time — Seconds needed by a participant to finish the tasks.

B. Correctness — Three levels describing the removal of a defect:

1. *Addressed*: For a cloned defect, the participant removed at least one, but not all, of the instances of the defect.

2. *Complete*: The participant removed all instances of the defect. (Note that for the non-cloned defects, this level is equivalent to the *Addressed* level)

3. *Incomplete*: The participant was unable to remove any instances of a defect, resulting in his or her data being excluded from the study.

5.2.1.3 **Subject Selection**

In the original study, the participants included 21 computer science students from the University of Bremen and 12 attendees of Dagstuhl Seminar 12071, which hosted researchers from the code clone community. All student participants were required to have taken a Java course before participating in the study. The original experimenters differentiated between novice and expert programmers. The student participants were deemed to be novice programmers. The Dagstuhl participants were deemed to be expert programmers.

In the replication, the 47 participants included 23 students enrolled in a junior-level software engineering course, 8 students enrolled in a senior-level software engineering course, and 16 students enrolled in a graduate-level software engineering course (all at the University of Alabama). All participants had previously taken at least one programming course. Similar to the original study, we assumed that we could split the participants into a novice group (undergraduate students) and an expert group (graduate students). In reality it was not possible to make
this distinction based on the data collected from the background surveys. The participants’ mean self-evaluation of Java familiarity on a scale of 1 (not familiar) to 100 (expert) was 42.96 for the undergraduates and 48 for the graduates. The difference of 5.04 between the groups was not large enough to treat them as separate groups as was done in the original study with the students and the Dagshtul attendees. The level of expertise of the participants in the replication is lower than that of the participants in the original study (i.e. in which participants not familiar with Java were excluded). Given the small sizes of the software systems and the low complexity of the fixes, we did not consider lack of familiarity with Java to be a threat to validity.

Another difference between the replication participants and the original participants is that only 2/3 of the replication participants were familiar with Eclipse while all of the original participants were familiar with Eclipse.

All participants took part in both the Replication and the Extension. After collecting the data, we excluded 10 participants from the Replication and the Extension along with 2 additional participants from the Extension only due to one of the following reasons:

- Corrupt XML files in the workspace,
- Corrupt Eclipse log files (due to improper shutdown), or
- Incomplete tasks

Thus, we analyzed data for 37 participants in the Replication and 35 in the Extension.

5.2.1.4 Instrumentation

This section describes the software systems and the plug-ins that the participants used to perform the tasks.
Software Systems: FrozenBubble (FB) and Pacman (PC) are both open source games with fewer than 3,000 lines of code. The choice of systems ensured that the defect removal tasks would be neither too difficult, nor too easy for the participants. In addition, we chose games for their visual appeal and their potential to engage the participants in removing the defects.

FB is a Tetris-like game that uses a launcher to shoot bubbles of different colors. A group of three or more of the same colored bubbles can be eliminated by shooting them with another bubble of that color. The aim of the game is to eliminate all bubbles. The defect represented a visual glitch. In both versions with defects (cloned and non-cloned), there are only grey bubbles available. Because all bubbles are the same color, one shot eliminates all of the bubbles. Figure 5.12 shows the pseudocode for the original code snippet from the *FrozenGame* class. A copy of the snippet shown in Figure 5.12 existed in class *LaunchBubbleSprite*, which was similar except the third array was missing. For the cloned version with defect of FB, the researchers inserted the defect by changing \( i+1 \) to 1 in lines 6 through 8. For the non-cloned version, we removed the copy in the class *LaunchBubbleSprite*. For this version, the *FrozenGame* class had the sole responsibility to load the bubbles and pass them to *LaunchBubbleSprite* upon instantiation. The following report describes the problem with the defect: *There are eight differently colored bubbles in the game. When a level starts, only gray bubbles appear on top. The launcher at the bottom will also fire only gray bubbles. Obviously not all available bubbles are loaded.*

In PC, the two characters (Pacman and Ghost) are governed by the direction of movement and an invisible grid to check for collisions. Change of the grid movements for up and left directions introduced the defect. The visual effect of this defect made the characters jump in a flickering motion back and forth. Figure 5.13 presents the pseudo code for the original movement function. There were copies of this function in the classes *Player* and *Ghost*. The researchers from the origi-
inal study created the cloned version of PM by changing subtractions to additions in lines 8 and 38 in both the clones. To create the non-cloned version, they abstracted the switch structure that was cloned in both classes to the Actor class using proper post processing steps. The following report describes the problem with the defect: *For all game characters, the movement up and left does not work correctly. Instead of moving up or left the characters move in the opposite direction in a flickering motion. Moving down and right works fine for all characters.*

We designed these two systems to emulate real-life defect-fixing scenarios. Defect reports and expected behaviors were provided. The defects were small and localized, allowing them to be solved within the time constraints. For the cloned defects, a similar defect was injected in two existing clones in the system. It is to be noted that the code clones in both the software systems were of near-miss type and not exact clones. To produce the non-cloned defects, these clones were abstracted to a single defective entity. The defects produced visual behavioral symptoms rather than complete system crashes, making it easier for the participants to reproduce them. In case of the cloned defects, the two different instances created distinct visual symptoms. Total removal of visual symptoms was possible only after fixing both the instances of the defect.

**Eclipse Plugin:** The original experimenters extended Eclipse with a plug-in that displayed step-by-step instructions (e.g., the task descriptions) to guide the participants through the experiment. The plug-in also logged participant data including the time required to complete tasks and the usage of features like search. Finally, the plug-in administered the study surveys.

The environment for performing the study tasks was distributed to participants via USB drives. Each key had pre-configured Eclipse environments for Linux, Mac OS X, and Windows. The subjects unpacked the appropriate archive, ran Eclipse, and loaded the provided workspace. The participants were instructed to save any code changes in the workspace to allow the researchers
Table 5.3: Assignment of Groups to Task

<table>
<thead>
<tr>
<th>Sessions</th>
<th>Task #</th>
<th>Group A</th>
<th>Group B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replication</td>
<td>1</td>
<td>Frozen Bubble w/non-cloned defect (FBnc)</td>
<td>Frozen Bubble w/cloned defect (FBc)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Pacman w/cloned defect (PMc)</td>
<td>Pacman defect w/non-cloned defect (PMnc)</td>
</tr>
<tr>
<td>Extension</td>
<td>1</td>
<td>Pacman w/non-cloned defect (PMnc)</td>
<td>Pacman w/cloned defect (PMc)</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>Frozen Bubble w/cloned defect (FBc)</td>
<td>Frozen Bubble w/non-cloned defect (FBnc)</td>
</tr>
</tbody>
</table>

Figure 5.12: Pseudo Code for FB Code Snippet

```java
to later analyze them for correctness. I supervised the sessions and was available to help setup Eclipse. Each participant received the appropriate USB drive for his or her assigned group (see Section 5.2.1.5).

5.2.1.5 Experimental Operation

We used the two variants of each of the two software systems from in the original study. For clarity we refer to these variants with the same terms as used in the original study, namely: $FB_{nc}$ and $FB_{c}$ (the non-cloned and cloned versions of FB) and $PM_{nc}$ and $PM_{c}$ (non-cloned and cloned versions of PC).

Similar to the original study, we divided the participants randomly into Group A and Group B. The objective of partitioning participants into two groups was to counterbalance the study design to remove any potential ordering or interaction effects related to the artifacts. For each of the four tasks (two for the Replication and two for the Extension), the participants received a version of
```java
switch currentDirection
  case up
    if canMoveTo(gridX, gridY - 1) then
      deltaX = 0
      deltaY = deltaY - speed
      if 1deltaY1 <= CELL_SIZE then
        deltaY = 0
        moveTo(gridX, gridY - 1)
      end
    end
    break;
  case right
    if canMoveTo(gridX + 1, gridY) then
      deltaX = deltaX + speed
      deltaY = 0
      if 1deltaX1 <= CELL_SIZE then
        deltaX = 0
        moveTo(gridX + 1, gridY)
      end
    end
    break
  case down
    if canMoveTo(gridX, gridY + 1) then
      deltaX = 0
      deltaY = deltaY + speed
      if 1deltaY1 <= CELL_SIZE then
        deltaY = 0
        moveTo(gridX, gridY + 1)
      end
    end
    break
  case left
    if canMoveTo(gridX - 1, gridY) then
      deltaX = deltaX - speed
      deltaY = 0
      if 1deltaX1 <= CELL_SIZE then
        deltaX = 0
        moveTo(gridX - 1, gridY)
      end
    end
    break
end
```

Figure 5.13: Pseudo Code for PM Code Snippet

Clone Group: UID
FILE0: Lines (X0-X1)
FILE1: Lines (y0-y1)
...
FILEn: Lines (n0-n1)

Figure 5.14: Clone Report Template
the system that included one or more defects. During the *Replication* portion, the participants located and repaired the defects without any additional documentation aids. However, during the *Extension*, we provided clone reports to the participants to aid in defect-localization and repair. Table 5.3 illustrates which system the participants in each group used for each of the four tasks.

At the end of each session, the Eclipse plug-in created a zip file and saved it back to the USB drive. The zip file contained the four workspaces from the four tasks. Each workspace included: the modified source code, log files and responses to short surveys. We modified the analysis scripts from the original study to work properly on both the *Replication* and the *Extension*.

### 5.2.1.6 Extended Design

The *Extension* added two Defect-repair tasks, this time with the benefit of a clone report. This section describes the clone report and requisite training required to use the report.

**Clone Reports:** We provided the participants with text-based clone reports similar to those used in our earlier study [Chatterji et al., 2011]. The clone reports listed clone groups within the
systems as illustrated in the template in Figure 5.14. Figures 5.15 and 5.16 show the specific clone groups that contained the defects in the $FB_c$ and $PM_C$ systems respectively.

**Training:** To avoid biasing the *Replication* portion of the study, we did not inform the participants the subject of the study was code clones. Prior to the *Extension*, we provided a brief background on code clones and how to read a clone report. To prevent biasing the results of the *Extension* portion, we took care only to describe how to read a clone report, now how to use the report for defect removal.

5.2.2 Original Study Results

The authors tested to see if it took the participants more time to remove the cloned defects than the non-cloned defects. Correctness was tested based on completeness of the solution. Based on the level of correctness, participants were classified in three categories: (a) those who solved a single defect, (b) those who solved at least one instance of the cloned defect, and (c) those who solved all instances of the cloned defect. The results of the study were unable to answer the research questions (RQ1 and RQ2). There was no significant relation found between correctness and time taken to the type of the defect. However, the results did point out to trends that were motivating for further investigation.

The data obtained in the original study did not provide statistical evidence to reject $H_0^{time}$ or $H_0^{corr}$. Nevertheless, the results indicated promising trends. Regarding $H_0^{time}$ (DR-RQ1), the participants took more time to fix the cloned defect than they did to fix the non-cloned defect, in most cases. The only exception was that the expert group on average solved $PM_c$ more quickly than they solved $PM_{nc}$. The authors concluded that such a difference may not have a considerable effect in practice.

Regarding $H_0^{corr}$ (DR-RQ2), most participants succeeded in fixing at least one defect occur-
However, for the cloned defect, many participants submitted incomplete solutions. Less than half of the students succeeded in fixing both instances of the cloned defect for FB. For PM, one third of the solutions for the cloned defect were incomplete. Surprisingly, some of the experts also failed to fix the cloned defect. For $PM_c$, one third did not fix both defect occurrences and one expert missed the cloned defect in $FB_c$. In general, the experts performed better than the students, but the experts participated in the study within a cloning seminar and may have expected clones. The fact that these expert participants provided several incomplete solutions, although they should have been aware of clones, illustrates the risk potential of incomplete fixes of cloned defects. The authors of the original study concluded that although the results were not statistically significant they indicate that there may be a high risk of such incomplete solutions. These findings encouraged us to replicate the study to further investigate the effect.

5.2.3 Replication

This section describes the results of the *Replication* portion of the replication in three sections: Descriptive Analysis, Timing & Effort, and Correctness.
### Table 5.4: Average Times and p-values for Mann-Whitney U Test

<table>
<thead>
<tr>
<th></th>
<th>Frozen Bubble</th>
<th></th>
<th>Pacman</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cloned</td>
<td>Non-Cloned</td>
<td>Cloned</td>
<td>Non-Cloned</td>
</tr>
<tr>
<td><strong>Addressed Participant Times (in seconds)</strong></td>
<td>Mean</td>
<td>&gt;</td>
<td>1243.7</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td>&gt;</td>
<td>1135</td>
<td>&gt;</td>
</tr>
<tr>
<td>p-value</td>
<td>0.212</td>
<td></td>
<td>0.432</td>
<td></td>
</tr>
<tr>
<td><strong>Complete Participant Times (in seconds)</strong></td>
<td>Mean</td>
<td>&gt;</td>
<td>1337.25</td>
<td>&gt;</td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td>&gt;</td>
<td>1155</td>
<td>&gt;</td>
</tr>
<tr>
<td>p-value</td>
<td>0.238</td>
<td></td>
<td>0.44</td>
<td></td>
</tr>
</tbody>
</table>

5.2.3.1 Descriptive Analysis

We divided the participants randomly in two groups Group A and Group B. Data for the final analysis came from 37 participants for the *Replication*. Exclusion occurred when a participant was unable to remove even one instance of a defect or when a participant’s provided corrupted data. We assigned each participant a level of correctness (*Addressed*, *Complete*, or *Incomplete*) for each task based on his or her success in solving that task. Only data from participants that were in the *Addressed* or *Complete* level of correctness remained in the analysis. The Venn diagram in Figure 5.17 illustrates the distribution of participants. The set “Cloned defect — *Addressed*” represents the participants who solved at least one instance of the cloned defect. This set has a proper subset “Cloned defect — *Complete*” that represents the participants who fixed both defect instances. Similarly, the set “Non-Cloned defect — *Complete*” represents the participants who fixed the non-cloned defect.

5.2.3.2 Timing & Effort

We used different strategies to measure time and effort. We retrieved these log files to calculate the timing measurements. As there was no direct way to measure effort, we chose to use the Task Load Index (TLX) [Hart and Staveland, 1988]. TLX is an assessment technique proposed by NASA to assess perceived workload for a particular task. TLX is computed over four different parameters: mental demand (MD), performance (PF), effort (EF) and frustration (FR).
Figure 5.18: Task Load Index

Figure 5.19: Times for All Participants (In Seconds)
Higher values imply more demanding tasks. We used surveys to gather the following information from participants relative to each TLX parameter:

- mental demand needed to solve the defect,
- how successful were they performing the task,
- how much effort was needed, and
- how frustrated were they while performing the task.

The results for each artifact, illustrated in Figure 5.18, show conflicting results. For FB, the TLX scores are higher for the cloned version than for the non-cloned version \([\text{TLX}_{FBnc} (25.56) < \text{TLX}_{FBc} (39.47)}\)]. We expected this result as fixing two defects should be more demanding than fixing one. Conversely, for PM, the TLX scores are higher for the non-cloned version than for the cloned version \([\text{TLX}_{PMnc} (26.56) > \text{TLX}_{PMc} (22.02)}\)]. We did not expect this result. Because \(PM_c\) had two instances of the defect, we expected the TLX to be higher than for \(PM_{nc}\). We hypothesize that this result may be due to the use of a more complex abstraction to produce the non-cloned version. Further studies are required to check whether there is a definitive reason for this behavior or whether the result is merely an outlier.

To calculate the timing measurements we evaluated the recorded times in the respective workspaces. The Box-and-Whisker plot in Figure 5.19 illustrates the distribution of times for the replication. Table 5.4 provides the mean and median times for the participants. The Addressed Participant Times are the times for the participants who fixed at least one defect. The Complete Participant Times are the times for participants who found and fixed all the instances of the defect.
Table 5.5: Correctness Evaluation

<table>
<thead>
<tr>
<th></th>
<th>Fisher (1-tailed)</th>
<th>Bernard (1-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>&lt; 0.01</td>
<td>&lt; 0.01</td>
</tr>
<tr>
<td>PM</td>
<td>&lt; 0.01</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 5.4 shows that the mean and median times were greater for the cloned versions in all cases. The median time to completely fix the cloned defect was 38% greater than the mean time to completely fix the non-cloned in the FB program and 11% longer in the PC program. These differences translate into about 5 minutes for the FB program and just under 1 minute for the PC program. We applied the Mann-Whitney U test to assess whether the independent samples (times for cloned and non-cloned version) were significantly different than each other. Similar to the results of the original study, the differences were not significant (p-value rows in Table 5.4). Hence, though we cannot reject the null hypotheses $H_{0}^{\text{time}}$, the results do support the same trend that was evident in the original study that the cloned defect took more time to fix than the non-cloned defect.

5.2.3.3 Correctness

To judge the correctness of the each participant’s solution, we manually checked his or her workspace for defect fixes. First, we executed the programs to check if all the visual symptoms of the defect were removed. Second, we used a diff to inspect the changes to the code. For the Replication, 20/37 achieved a Complete solution for the cloned defect, with 30/37 achieving a Complete solution for the non-cloned defect. Fourteen additional participants achieved an Addressed solution for the cloned defect (i.e., they were able to fix only one instance of the defect).

To evaluate $H_{0}^{\text{corr}}$, we used Barnard’s exact test and Fisher’s test to test whether the participants were able to obtain a Complete solution to the non-cloned defect more often than they were able to obtain a Complete solution to the cloned defect. These statistical tests are suitable
for use on small data sets in the form of contingency tables. The results of these tests, shown in Table 5.5, illustrate that the participants were significantly more likely to obtain a Correct solution to the non-cloned defect. Thus, we can reject the null hypothesis $H_0^{corr}$ and accept the alternate hypothesis $H_A^{corr}$. This result provides more strength to the results of the original study which showed a promising trend, but not a significant result.

5.2.4 Extension

The second part of the replication, the Extension, studied the effect of providing developers with information about the clones present in the code being maintained. The Extension was motivated by DR-RQ3 and tested hypothesis $H_0^{cinf}$ to see whether providing developers clone reports helped them more effectively fix cloned defects than without clone reports. The Venn Diagram in Figure 5.20 illustrates the results of the Extension.

Similar to the analysis for the Replication portion of the experiment, we tested the workspaces
collected from each participant for correctness. As shown in Table 5.6, in this case, all 35 participants Addressed the cloned defect, with 29 providing a Complete solution.

The results from our previous work showed that, if used correctly, clone reports can help developers locate multiple instances of a defect [Chatterji et al., 2011]. During the training, we suggested that the participants should employ the correct strategy when using the clone reports, i.e. first locate a defect then use the clone report to determine if there are other instances of that defect. The results shown in Table 5.6 show that the participants in the Extension were more successful in locating the cloned defect and obtaining a Complete solution. In the Extension, 83% of the participants who fixed at least one instance of the defect (Addressed) were successful in fixing the cloned instance (Complete). This result was an improvement over the Replication, where the ratio Addressed:Complete was only 59%. A Chi-Square test on this data showed that the ratio of Addressed:Complete was significantly different between the Replication and the Extension ($X^2_{21} = 4.267; p = .039$). These results allow us to successfully reject the null hypothesis $H_0^{cinf}$ and accept the alternate hypothesis $H_A^{cinf}$. Therefore, these results indicate that the proper use of clone information can help developers maintain cloned code.

### 5.2.5 Comparison and Discussion of Results

To gain the most insight from the replication, this section compares the results from the original study to the results from the Replication part of the replication study with regards to the time and correctness variables.

#### 5.2.5.1 Time

Table 5.7 summarizes the results for the time variable across both studies. In general, it took developers more time to fix the cloned defects than it did to fix the non-cloned defects, although these differences were rather small (5 minutes and 1 minute, respectively) and not statistically sig-
Table 5.7: Addressed Vs. Complete Average Times in Seconds

<table>
<thead>
<tr>
<th></th>
<th>FBc</th>
<th>FBnc</th>
<th>p-value</th>
<th>PMc</th>
<th>PMnc</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Students</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Original Study)</td>
<td>Addressed</td>
<td>Complete</td>
<td></td>
<td>Addressed</td>
<td>Complete</td>
<td></td>
</tr>
<tr>
<td>Students</td>
<td>1,615</td>
<td>&gt;</td>
<td>812</td>
<td>0.008</td>
<td>&gt;</td>
<td>490</td>
</tr>
<tr>
<td>Experts</td>
<td>934</td>
<td>&gt;</td>
<td>812</td>
<td>0.22</td>
<td>842</td>
<td>&gt;</td>
</tr>
<tr>
<td>(Original Study)</td>
<td>Addressed</td>
<td>Complete</td>
<td></td>
<td>Addressed</td>
<td>Complete</td>
<td></td>
</tr>
<tr>
<td>Experts</td>
<td>923</td>
<td>&lt;</td>
<td>1,276</td>
<td>0.91</td>
<td>838</td>
<td>&gt;</td>
</tr>
<tr>
<td>(Original Study)</td>
<td>Addressed</td>
<td>Complete</td>
<td></td>
<td>Addressed</td>
<td>Complete</td>
<td></td>
</tr>
<tr>
<td>Experts</td>
<td>516</td>
<td>&lt;</td>
<td>1,276</td>
<td>0.985</td>
<td>825</td>
<td>&gt;</td>
</tr>
<tr>
<td>Replication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Addressed</td>
<td>1,244</td>
<td>&gt;</td>
<td>1,045</td>
<td>0.106</td>
<td>&gt;</td>
<td>460</td>
</tr>
<tr>
<td>Complete</td>
<td>1,337</td>
<td>&gt;</td>
<td>1,045</td>
<td>0.119</td>
<td>&gt;</td>
<td>460</td>
</tr>
</tbody>
</table>

Table 5.8: Addressed Vs. Complete for Cloned Defect

<table>
<thead>
<tr>
<th></th>
<th>Total</th>
<th>Complete</th>
<th>%</th>
<th>Fisher (1-tailed)</th>
<th>Barnard (1-tailed)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FB</td>
<td>PM</td>
</tr>
<tr>
<td>Students</td>
<td>21</td>
<td>12</td>
<td>57%</td>
<td>0.0085</td>
<td>0.0902</td>
</tr>
<tr>
<td>(Original Study)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Experts</td>
<td>12</td>
<td>9</td>
<td>75%</td>
<td>0.5000</td>
<td>0.5000</td>
</tr>
<tr>
<td>(Original Study)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replication</td>
<td>37</td>
<td>20</td>
<td>59%</td>
<td>&lt;0.001</td>
<td>&lt;0.001</td>
</tr>
</tbody>
</table>

significant. Across both studies, there was only one case where this result did not hold (experts from the original study working on FB). The authors of the original study did not provide a concrete explanation for why experts behaved differently on the FB problem. They argued that the unexpected behavior may have been an artifact of the small sample size or an uneven distribution of expertise. Our initial hypothesis for this result was that the process used to refactor the clone may have been more complex in the FB program. But, after analyzing the code, the refactoring approach was similar in both systems and the clones seemed to be of similar complexity across both systems.

While cloned defects tended to take longer to fix, the difference was only significant in one case. Therefore, we are not yet able to draw a definitive conclusion. While some previous research has suggested that clones can be problematic for maintenance, this view is not universal. The results of these two studies seem to confirm that the impact of clones on Defect-repair still needs further study.
5.2.5.2 Correctness

Table 5.8 summarizes the correctness evaluation for the complete solutions along with the p-values. The results show that for the students in the original study and for the participants in the Replication, correctness was significantly worse for the FB program when clones were present. Because the participants in the Replication are most similar to the students in the original study, this result is consistent. The results relative to the PC program were inconsistent with a significant result in the Replication and a non-significant result in the original study. At this point, we have no concrete explanation for this difference.

Overall, the expert population seemed to behave differently than the students. This result is not surprising because the experts were drawn from the participants in a workshop on code clone research. Due to this characteristic, it is possible that these participants were expecting the system to contain clones. Such an expectation might introduce a participation bias where the traits of the participants can affect the outcome of the study. The students in the original study and the participants in the Replication were not aware that the study was focused on code clones, therefore they were likely not biased to look for clones.

In general, the results of the replication are consistent with the results of the original study, with regards to the novice participants. It appears that cloned defects are more difficult to completely fix correctly.

5.2.6 Summary of the Study

The results of the original study showed some trends, but most results were not significant. Specifically, the original study was not able to validate:

1. whether it takes more time to fix a cloned defect than a non-cloned defect, or
2. whether developers are more likely to correctly remove a non-cloned defect than a cloned defect.

While the results of the replication study were also unable to support (1), a similar trend was observed. Conversely, the results of the replication did show that it was significantly more difficult to completely maintain cloned defects.

In the *Extension* part of the study, we verified that providing developers with clone information and training them on how to use it, helps in the maintenance of cloned defects. We found that the participants performed significantly better in completely fixing the clone defects when they had clone information than when they did not. Overall, the evidence from this extended replication indicates that it is more difficult to maintain cloned code than non-cloned code. However, given appropriate clone information and the proper training this difficulty can be reduced.

5.3 Outcomes of the Human-Based Studies and Validation

To summarize the two human-based empirical studies, it is clear that cloned code is harder to maintain than non-cloned code. The developers showed improvement with proper awareness and training. Based on this finding, providing software developers with clone-aware information can surely simplify the task of code clone related maintenance and management. In the localization study, many developers failed to locate the cloned defects even after being provided a clone report. In the Defect-Refactoring study many found it hard to fix an abstracted defect. This result might be an indication of situations where abstraction of a cloned defect was not the best strategy. Understanding developer intentions and effects of cloning can be critical in order to answer the maintenance related problems.

Because these studies were conducted on university students, there is a need to validate
the results with professional developers. To perform that validation, I interviewed professional developers who work full time at CAPS\(^1\). The primary objectives of interviewing developers were:

a) Validate findings related to developer behavior during clone maintenance; and

b) Understand challenges related to clone maintenance faced by developers in a real industrial scenario.

The study design and the qualitative data analysis process was similar to what was explained in Chapter 3, Section 3.2.

Figure 5.21 shows the awareness of the participants towards code cloning. Although all developers but one were aware of the term, they rarely take any conscious efforts towards tracking or management of clones. On being asked if they are familiar with the term ‘code clone’, one of the developers said, “*Heard from manager and colleagues. There is a little awareness about code clones in my workplace.*” On being asked if they clone code, all the participants said “*yes.*” Out of the 10 participants who were working as full time developers, two had supervising responsibilities. Their interviews revealed that there is hardly any awareness about code cloning in the work environment.

However, the developers seemed to be aware of relevant risks involved with code cloning as all of them hinted that they either avoid or would like to avoid cloning code, given an ideal situation. For example one of the developers responded, “*I prefer not to, even for the sake of ease. I would create a class or function to handle it.*” Note that, in this context, my intention is not to typecast code cloning as an avoidable action. However, randomly cloning code without keeping track has its share of risks tied to it. In the absence of substantial clone aware resources, as in the

\(^1\) http://caps.ua.edu/
case of CAPS, these un-tracked clones can propagate faults and defects that might go unnoticed and cause system failure. On being asked if they track code clones, the responses were heavily inclined towards “No.” Two developers responded affirmatively. One of them responded, “Yes. For example I copied a class from a web resource, so I put a reference. But usually I don’t.” Multiple copies from the same web resources can give rise to unintentional clones. While making another copy of code from the same external resource, it is not necessary that the developer would remember where s/he copied the first time. Also, there can be cases where multiple developers use the same external resource. In such cases it is very unlikely for the developers to track the clones simultaneously. Hence, in such cases it is very important to document the reason for reusing external code. Another participant responded, “Always put xml comments pointing back to the parent fragment.” Wide use of such a strategy could prove helpful for clone management.

The results from Section 5.1 and Section 5.2 showed that it was harder for the students to maintain cloned bugs than non-cloned bugs. Since the developers at CAPS do not perform any required clone maintenance, it was futile to ask the participants their experience with maintenance of cloned code. The next question I asked them regarded the granularity at which they clone
Figure 5.22: Granularity of Code Cloning

code. Since the clones used for the BR and BL studies were method/function level clones, it was important to understand if that is the case in a real life scenario. The interviewer asked the participants about the level of granularity for reusing code. Figure 5.22 shows the responses from the participants.

It is clear from the figure that the majority of the developers give rise to clones on a method level. Hence, it is safe to say that the results from the studies BL and BR can be generalized in a real industrial scenario. For example, a participant said, “Not copy bigger structure rather copy small fragments. Even in case of robust structures it is better not to copy bigger chunks. There is a risk of unknown defects.” However, some of the participants thought differently. One such participant said, “Bigger chunks. Try not to copy smaller fragments.” Although majority of the participants were inclined towards copying at the method level, the results from this study can be mere indications. Substantial future work is required to prove this finding. Another indication from the above observation can be the confusion and lack of awareness by developers regarding ways
to handle and manage code clones. In my studies, the results showed that little training can assist developers in managing clones better. Hence, the indications suggest that the issues related to clones can be avoided by moderately training developers about clone management. Further studies are required to gather more data about this conclusion.

Apart from training and monitoring, there can be other simple practices that can help avoid problematic issues. For example a developer responded, “We do a code review process within our group where everybody can critique others’ code and defend their code by explaining their actions.” If code cloning review is also made a part of such code review meetings, it can possibly provide motivation for developers to do a better job at managing code in general and also make sure that their intention of cloning is clear.

The DL study showed that proper use of clone reports help in localizing cloned defects. The DR study showed that cloned code is harder to maintain. Combining the results above, the evidence suggests that developers can benefit from training and access to clone aware information in all maintenance tasks. However, in a real industrial scenario there are other factors that might render this suggestion challenging to implement. The participants who had supervising responsibilities made it very clear that, developers work under intense time pressures. Most of the times ideal situations are far from the real scenario.
6

SUMMARY

This dissertation proposed a ‘Four Pillar Architecture’ in Chapter 1 and identified a set of research questions for investigation. The four pillars represent four crucial research concentrations in the area of code clone research. The primary objective of this dissertation was to fill the void due to lack of empirical validation in the area of code clone research. The fast pace of growth in this research area has led to some confusions and ambiguity. With this rate of growth in research, the gaps are bound to widen. This dissertation is an effort towards taking a step back to solidify the foundation and support growth.

Another primary motivation for this dissertation was to understand the applicability of academic code clone research in an industrial scenario. This motivation led to the focus on understanding a developers’ behavior and perspective when exposed to a code clone maintenance scenario. Code clones can have substantial impact on maintenance. Maintenance is the most expensive part of the software life cycle, by far. This fact makes code clones a crucial factor to study and provides a great degree of research opportunity. The origin of a clone can be traced back to a developer’s intention to reuse code. Hence, it is important to understand the behavioral traits of developers that lead to formation of clones and later relate to clone maintenance.

Based on these motivations, this dissertation achieved some degree of success in better understanding the fundamental issues, act upon the research motivations to find answers, provide
substantial indication towards answers for the fundamental issues and finally, lay a solid foundation for fueling further research.

6.1 The Four Pillar Architecture

This section revisits the research questions related to each fundamental pillar and summarizes the findings. Each subsection discusses the motivation for the pillar, a brief summary of study results, and the high-level contribution to code clone research.

6.1.1 Consistent Definitions

The code clone literature lacks consistent and concrete definitions. There is a substantial amount of ambiguity and vagueness in the manner code clones are defined and categorized. However, it is crucial to understand the fundamental definitions of code clones because without having concrete and clear cut definition it is difficult to gauge their effect on maintenance.

6.1.1.1 Research Questions

The research questions related to this pillar were:

RQ1 What are the various definitions of code clones and how are code clones categorized?

RQ2 What type of clones are considered harmful to software quality?

RQ3 What are upper and lower bounds on the size of two fragments that can be considered as clones?

6.1.1.2 Results

The systematic literature review pointed out the overlap between Type II and Type III clones. Similarly, the community surveys indicated a high level of confusion about the structural definitions of clones. The results also detected a level of ambiguity in the upper and lower size thresholds for code clones. However, the results indicated that 30 tokens or 6 lines of code...
are reasonable threshold sizes. Tracing the formation of clones to the intention of developers resulted in classification of the positive or negative effects of cloning. Developer interviews focused on understanding cloning intent provided indications towards the classification of harmful/helpful code clones. Based on the intent-based taxonomy, whenever a developer creates a clone based on a forced intention, the clone appears to be associated with a negative perception. Further studies can build upon these findings to further validate the classification of harmful and helpful clones.

6.1.1.3 Contribution

The intent-based taxonomy will help reduce the indecisiveness for the decisions to be taken at the clone formation stage. Pointing out the ambiguity related to structural definitions of clones should fuel further research towards removing the shortcomings and defining concrete boundaries to eliminate the overlap. This dissertation is a starting point for future empirical studies that can answer fundamental questions such as clearer definitions, effects on software quality and intention of cloning.

6.1.2 Causes and Effects of Code Clones

It is important to understand the reasons for clone formation. Capturing the intention of developers can spread some light on this issue. In most cases, the intentions of developers for reusing code is responsible for the effects of code clones on the system. Sometimes these effects can be harmful, giving rise to maintenance related issues. To assist developers in clone-related maintenance, it is important to understand the reason for the creation of the clone. This information can not only lead to low maintenance cost, but can also guide developers to follow better practices and produce higher quality code.

6.1.2.1 Research Questions

The research questions related to this pillar were:
RQ4  Why do developers reuse code?

RQ5  How are clones helpful/harmful?

RQ6  What preventive measures can minimize bad effects arising from harmful practices?

6.1.2.2 Results

This dissertation proposed an intent and rationale-based taxonomy of code clones. The taxonomy also classified clones as harmful and helpful. The taxonomy indicated that avoiding the harmful intentions of cloning could act as a preventive measure to minimize the bad effects resulting from code clones.

6.1.2.3 Contribution

The community has been deliberating on the harmful and helpful effects of clones for a long time. There were no clear-cut answers to suggest whether clones are harmful or helpful for system quality and/or maintenance. The classification of clones into helpful and harmful clones fills this gap in the code clone research. Also, this dissertation indicated that cloned code is more complicated to maintain than non-cloned code. This revelation makes the process of tracking clones more important to assist in the process of maintenance. Using the taxonomy based on intent, developers can avoid the negative effects and benefit from the positive effects of cloned code to write more maintainable code.

6.1.3 Clone Awareness

The code clone literature is abundant with clone detection research. All the clone detection tools provide a variety of clone aware information. However, there has been hardly any studies to gauge the efficacy of the clone aware information. A better approach towards understanding how developers can use this information to their benefit could help reduce the cost of maintenance.
6.1.3.1 Research Questions

The research questions related to this pillar were:

RQ7 What assistance can be provided with clone aware information?

RQ8 How can developers use clone aware information to their maximum benefit?

6.1.3.2 Results

The survey results showed that most of the clone aware information available has no or little use for developers working on maintenance tasks. However, the maintenance studies did show that clone information could be helpful. The Bug Localization study and the Bug Refactoring study indicated that the use of clone aware information helped participants to perform significantly better at the clone maintenance tasks. The results also indicated that a little training can further assist the developers to perform better at such tasks.

6.1.3.3 Contribution

This dissertation provided some evidence regarding the use of clone aware information and ways to make it useful for system maintenance. This dissertation also showed that providing developers with clone aware information and training them on how to use it can improve their performance. This could be an incentive for the software development companies to adopt the process of code clone maintenance using clone aware information. Using this dissertation as a starting point of future research, researchers can devise ways to use the huge variety of clone aware information in an industrial scenario.

6.1.4 Clone Management

This pillar focuses on the decisions that developers have to make in clone maintenance scenarios. There is a lot of confusion about what decisions developers must make after they are
aware of the clones in the system. One strategy is to leave them untouched and unaltered. Another strategy is to refactor all code clones using abstraction. Harmfulness of code clones can be a deciding factor for what clones to keep in the system and which ones to be refactored. Hence, just keeping track of all the clones is not an answer to all the maintenance related issues.

6.1.4.1 Research Questions

The research questions related to this pillar were:

RQ9  *How to manage clones after developers are aware of their existence?*

RQ10  *Which clones should be refactored, kept or encouraged?*

6.1.4.2 Results

The human-based clone maintenance studies showed that cloned code is more complicated than non-cloned code. Hence, trying to purposely refactor the clones out of the system by processes like abstraction might be a complicated and a time consuming process. Therefore, it is advisable in most cases to just keep track of clones rather than trying to refactor them. However, the results from the Developer Interview study found a trade-off for the above observation. The study results showed a classification of good and bad clones based on developer intent. It might be advisable to refactor out the harmful clones and just keep a track of the good ones.

6.1.4.3 Contribution

The findings of this dissertation, related to this pillar, should be able to shift the focus of the community towards the management of code clones. The research community is focused more on detecting clones in the system. However, there is not much research to indicate solutions to issues beyond the point of detection. More focus is required to answer questions related to management issues like the ones raised in this section. The results from this dissertation point out
a few strategies which can prove helpful in clone management. However, a lot more research is required to provide substantial indications towards finding solutions to these issues.

6.2 Conclusion and Future Work

The surveys pointed out the degree of confusion among the community researchers regarding important fundamental issues and identified gaps in the area of code clone research. This dissertation also proposed a roadmap for future studies which could lead to solutions related to critical issues. Further, this dissertation was able to gauge the amount of confusion and ambiguity regarding fundamental issues related to code cloning. The systematic literature review pointed out the shortcomings in the literature regarding definitions of code clones. The ambiguity lies in the area of Type II and Type III code clones and further studies are required to clear this ambiguity related to structural definitions of clones.

The results indicate that intent and rationale of cloning is an important factor that lead to harmful effects of cloning. Software development environments can reduce the effects by employing simple measures to tackle harmful bad intentions of cloning. This dissertation pointed out that it might not always be feasible to avoid the harmful intentions because of factors such as time constraints. In such cases simple measures like tracking and documenting the harmful clones can help reduce the problems at a later stage of maintenance. Code clone maintenance is a complicated process and there is help that can be provided to developers in the form of clone aware information. However the issue here is to understand what kind of information will be most useful in specific scenarios, which might vary from industry to industry. The professional developers work under intense time pressures and hence they are most likely to continue cloning code. Rather than futile efforts towards trying to eradicate the root cause, it is better to make the process of cloning smoother for developers by creating awareness about do’s and don’ts of cloning.
Although the results of this dissertations indicate towards important answers, they are far from being proved. Apart from my continued efforts, I would like to urge the community of researchers to focus more interest towards the applicability of code clone research in industrial scenarios. I would like to focus my future work on the clone maintenance issues to find answers to issues of clone visualization, clone aware information for developers and code clone maintenance. Another future direction of research I would like to pursue would be in the area of clone visualization. This area focuses on useful presentation of the clone-aware information. More research in this area can provide indications for providing more effective ways of representing clone-aware information for developers during maintenance tasks.

6.3 Publications

Below is a list of my publications in chronological order.


Planned Journal Paper Submissions:

8 Definitions of Code Clones - Ambiguous Structural Definitions Vs. Cloning Intent Based Definitions (Debarshi Chatterji, Jeffrey C. Carver) at the *Information and Software Technology Journal* – Chapter 3 of the dissertation and expected to be submitted in July 2014.

9 Clone Research Community Beliefs about Code Clones and Developer Behavior - Two Surveys (Debarshi Chatterji, Jeffrey C. Carver, Nicholas A. Kraft) at the *Journal of Systems and Software* – Chapter 4 of the dissertation and and expected to be submitted in June 2014.

10 Empirical Human-Based Validation of Cloned Code Maintenance (Debarshi Chatterji, Jeffrey C. Carver, Nicholas A. Kraft) at the *Empirical Software Engineering Journal* – Chapter 5 of the dissertation and expected to be submitted in July 2014.
REFERENCES


Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995). Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley Longman Publishing Co., Inc.


Appendices
June 14, 2010

Nicholas Kraft, PhD
Department of Computer Science
College of Engineering
Box 870290

Re: IRB # EX-09-CM-053 (Revision) “SHF: Small Collaborative Research: Improved Code Clone Categorization”

Dear Dr. Kraft:

The University of Alabama Institutional Review Board has reviewed the revision to your previously approved exempt protocol. The board has determined that the change does not affect the exempt status of your protocol.

Should you need to submit any further correspondence regarding this proposal, please include the assigned IRB application number.

Good luck with your research.

Sincerely,

Stuart Usdan, PhD
Chair, Non-Medical Institutional Review Board
University of Alabama
UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

<table>
<thead>
<tr>
<th>Principal Investigator</th>
<th>Second Investigator</th>
<th>Third Investigator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names: Nicholas Kraft</td>
<td>Jeffrey Carver</td>
<td></td>
</tr>
<tr>
<td>Department: Computer Science</td>
<td>Computer Science</td>
<td>Engineering</td>
</tr>
<tr>
<td>College: Engineering</td>
<td></td>
<td></td>
</tr>
<tr>
<td>University: Alabama</td>
<td>Alabama</td>
<td></td>
</tr>
<tr>
<td>Address: Box 870290</td>
<td>Box 870290</td>
<td></td>
</tr>
<tr>
<td>Telephone: 8-4740</td>
<td>8-9829</td>
<td></td>
</tr>
<tr>
<td>FAX: 8-0219</td>
<td>8-0219</td>
<td><a href="mailto:carver@cs.ua.edu">carver@cs.ua.edu</a></td>
</tr>
</tbody>
</table>

Title of Research Project: SHF: Small: Collaborative Research: Improved Code Clone Categorization

Date Submitted: December 17, 2009
Funding Source: National Science Foundation

Type of Proposal
- [ ] New
- [x] Revision
- [ ] Renewal
  - Please attach a renewal application
- [ ] Completed
- [ ] Exempt
  - Please attach a continuing review of studies form
  - Please enter the original IRB # at the top of the page

UA faculty or staff member signature: [Redacted]

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: ______ Full board ______ Expedited

IRB Action:
- [x] Approved-this proposal complies with University and federal regulations for the protection of human subjects.
  - Approval is effective until the following date: 8/1/2010
  - Items approved:
    - [x] Research protocol (dated ____________)
    - [x] Informed consent (dated ____________)
    - [x] Recruitment materials (dated ____________)
    - [x] Other add staff (3) (dated ____________)

Approval signature: [Redacted] Date 6-15-10
November 15, 2011

Jeffrey C. Carver, Ph.D.
Department of Computer Science
College of Engineering
The University of Alabama

Re: IRB # 11-OR-328 “Survey on Code Clone Research”

Dear Dr. Carver:

The University of Alabama Institutional Review Board has granted approval for your proposed research

Your application has been given expedited approval according to 45 CFR part 46. You have also been granted the requested waiver of documentation of informed consent. Approval has been given under expedited review category 7 as outlined below:

(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.

Your application will expire on November 14, 2012. If your research will continue beyond this date, complete the relevant portions of the IRB Renewal Application. If you wish to modify the application, complete the Modification of an Approved Protocol Form. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants. When the study closes, complete the appropriate portions of the IRB Study Closure Form.

Should you need to submit any further correspondence regarding this proposal, please include the above application number.

Good luck with your research.

Sincerely,

[Redacted]
Director & Research Compliance Officer
Office for Research Compliance
The University of Alabama
UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

<table>
<thead>
<tr>
<th>Principal Investigator</th>
<th>Second Investigator</th>
<th>Third Investigator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names: Jeffrey Carver</td>
<td>Nicholas A. Kraft</td>
<td>Debarshi Chatterji</td>
</tr>
<tr>
<td>Department: Computer Science</td>
<td>Computer Science</td>
<td>Computer Science</td>
</tr>
<tr>
<td>College: Engineering</td>
<td>Engineering</td>
<td>Engineering</td>
</tr>
<tr>
<td>University: Alabama</td>
<td>Alabama</td>
<td>Alabama</td>
</tr>
<tr>
<td>Address: Box 870290</td>
<td>Box 870290</td>
<td>Box 870290</td>
</tr>
<tr>
<td>Telephone: 8-9829</td>
<td>8-4740</td>
<td>8-1528</td>
</tr>
<tr>
<td>FAX: 8-0219</td>
<td>8-0219</td>
<td>8-0219</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:carver@cs.ua.edu">carver@cs.ua.edu</a></td>
<td><a href="mailto:nkraft@cs.ua.edu">nkraft@cs.ua.edu</a></td>
<td><a href="mailto:dchatterji@crimson.ua.edu">dchatterji@crimson.ua.edu</a></td>
</tr>
</tbody>
</table>

Title of Research Project: Survey on Code Clone Research

Date Submitted: October 25, 2011
Funding Source: National Science Foundation

Type of Proposal: [ ] New [ ] Revision [ ] Renewal [ ] Completed [ ] Exempt

Please attach a renewal application
Please enter the original IRB # at the top of the page

Please attach a continuing review of studies form

UA faculty or staff member signature: [Redacted]

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):

Type of Review: [ ] Full board [ ] Expedited

IRB Action:
[ ] Rejected
[ ] Tabled Pending Revisions
[ ] Approved Pending Revisions
[ ] Approved—this proposal complies with University and federal regulations for the protection of human subjects.

Date: [11/4/12]

Items approved: [ ] Research protocol
[ ] Informed consent
[ ] Recruitment materials

Approval is effective until the following date: 11/15/2014

Approval signature: [Redacted]
October 30, 2012

Jeffrey Carver, PhD
Department of Computer Science
College of Engineering
Box 870290

Re: IRB#: 11-OR-328-R1 “Survey on Code Clone Research”

Dear Dr. Carver:

The University of Alabama Institutional Review Board has granted approval for your renewal application.

Your protocol has been given expedited approval according to 45 CFR part 46. Approval has been given under expedited review category 7 as outlined below:

(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.

Your application will expire on October 29, 2013. If your research will continue beyond this date, complete the relevant portions of the IRB Renewal Application. If you wish to modify the application, complete the Modification of an Approved Protocol Form. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants. When the study closes, complete the appropriate portions of the IRB Request for Study Closure Form.

Should you need to submit any further correspondence regarding this proposal, please include the above application number.

Good luck with your research.

Sincerely,

[Redacted]

Director & Research Compliance Officer
Office of Research Compliance
The University of Alabama
UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

Principal Investigator: Jeffrey Carver
Second Investigator: Nicholas A. Kraft
Third Investigator: Debarshi Chatterji

Department: Computer Science
College: Engineering
University: Alabama
Address: Box 870290
Telephone: 8-9829
FAX: 8-0219
E-mail: carver@cs.ua.edu

Department: Computer Science
College: Engineering
University: Alabama
Address: Box 870290
Telephone: 8-4740
FAX: 8-0219
E-mail: nkraft@cs.ua.edu

Department: Computer Science
College: Engineering
University: Alabama
Address: Box 870290
Telephone: 8-1528
FAX: 8-0219
E-mail: dchatterji@crimson.ua.edu

Title of Research Project: Survey on Code Clone Research

Date Submitted: October 25, 2011
Funding Source: National Science Foundation

Type of Proposal: □ New □ Revision □ Renewal □ Completed □ Exempt
Please attach a renewal application
Please attach a continuing review of studies form
Please enter the original IRB # at the top of the page

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: _______ Full board □ Expedited

IRB Action:
□ Rejected Date: _______
□ Tabled Pending Revisions Date: _______
□ Approved Pending Revisions Date: _______
□ Approved-this proposal complies with University and federal regulations for the protection of human subjects.

Approval is effective until the following date: 10/29/2013
Items approved:
□ Research protocol (dated _______
□ Informed consent (dated _______
□ Recruitment materials (dated _______

Approval signature: __________________ Date: 10/30/2012
January 17, 2013

Debarshi Chatterji
Department of Computer Science
College of Engineering
The University of Alabama


Dear Mr. Chatterji:

The University of Alabama Institutional Review Board has reviewed the revision to your previously approved expedited protocol. The board has approved the change in your protocol.

Please remember that your approval period expires one year from the date of your original approval, August 22, 2012, not the date of this revision approval.

Should you need to submit any further correspondence regarding this proposal, please include the assigned IRB application number. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants.

Good luck with your research.

Sincerely,

[Signature]

Carpathian T. Myles, MSM, CM
Director & Research Compliance Officer
Office for Research Compliance
The University of Alabama
UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

<table>
<thead>
<tr>
<th>Principal Investigator</th>
<th>Second Investigator</th>
<th>Third Investigator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Debarshi Chatterji</td>
<td>Jeffrey Carver</td>
<td>Nicholas A. Kraft</td>
</tr>
<tr>
<td>Department:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Computer Science</td>
<td>Computer Science</td>
<td>Computer Science</td>
</tr>
<tr>
<td>College:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Engineering</td>
<td>Engineering</td>
<td>Engineering</td>
</tr>
<tr>
<td>University:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alabama</td>
<td>Alabama</td>
<td>Alabama</td>
</tr>
<tr>
<td>Address:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Box 860290</td>
<td>Box 870290</td>
<td>Box 870290</td>
</tr>
<tr>
<td>Telephone:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-1528</td>
<td>8-9829</td>
<td>8-4740</td>
</tr>
<tr>
<td>FAX:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8-0219</td>
<td>8-0219</td>
<td>8-0219</td>
</tr>
<tr>
<td>E-mail:</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="mailto:dchatterji@crimson.ua.edu">dchatterji@crimson.ua.edu</a></td>
<td><a href="mailto:carver@cs.ua.edu">carver@cs.ua.edu</a></td>
<td><a href="mailto:nkraft@cs.ua.edu">nkraft@cs.ua.edu</a></td>
</tr>
</tbody>
</table>

Title of Research Project: Evaluating the Effect of Code Structure on Software Maintenance

Date Submitted: 01/10/12
Funding Source: National Science Foundation

Type of Proposal: New Revision

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: Full board Expedited

IRB Action:
- Approved-this proposal complies with University and federal regulations for the protection of human subjects.
- Approval is effective until the following date: 8/21/13

Items approved:
- Research protocol (dated
- Informed consent (dated
- Recruitment materials (dated
- Other (dated

Approval signature: [Redacted] Date 1/17/2013
February 8, 2013

Debarshi Chatterji  
Department of Computer Science  
College of Engineering  
Box 870290

Re: IRB # 13-OR-045: “Code Clone Taxonomy Based on Cloning Intent”

Dear Debarshi,

The University of Alabama Institutional Review Board has granted approval for your proposed research.

Your application has been given expedited approval according to 45 CFR part 46. Approval has been given under expedited review category 7 as outlined below:

(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.

Your application will expire on February 7, 2014. If the study continues beyond that date, you must complete the IRB Renewal Application. If you modify the application, please complete the Modification of an Approved Protocol form. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants. When the study closes, please complete the Request for Study Closure (Investigator) form.

Please use reproductions of the IRB-stamped consent form.

Should you need to submit any further correspondence regarding this application, please include the assigned IRB application number.

Good luck with your research.

Sincerely,

[Redacted]

Office for Research Compliance  
The University of Alabama
UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

<table>
<thead>
<tr>
<th>Principal Investigator</th>
<th>Second Investigator</th>
<th>Third Investigator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Names: Debarshi Chatterji</td>
<td>Jeffrey Carver</td>
<td>Nicholas A. Kraft</td>
</tr>
<tr>
<td>Department: Computer Science</td>
<td>Computer Science</td>
<td>Computer Science</td>
</tr>
<tr>
<td>College: Engineering</td>
<td>Engineering</td>
<td>Engineering</td>
</tr>
<tr>
<td>University: Alabama</td>
<td>Alabama</td>
<td>Alabama</td>
</tr>
<tr>
<td>Address: Box 860290</td>
<td>Box 870290</td>
<td>Box 870290</td>
</tr>
<tr>
<td>Telephone: 8-1528</td>
<td>8-9829</td>
<td>8-4740</td>
</tr>
<tr>
<td>FAX: 8-0219</td>
<td>8-0219</td>
<td>8-0219</td>
</tr>
<tr>
<td>E-mail: <a href="mailto:dchatterji@crimson.ua.edu">dchatterji@crimson.ua.edu</a></td>
<td><a href="mailto:carver@cs.ua.edu">carver@cs.ua.edu</a></td>
<td><a href="mailto:nkraft@cs.ua.edu">nkraft@cs.ua.edu</a></td>
</tr>
</tbody>
</table>

Title of Research Project: Code Clone Taxonomy based on Cloning Intent.

Date Submitted: 01/15/13
Funding Source: National Science Foundation

Type of Proposal: ☑ New

☐ Revision | ☐ Renewal application

☐ Completed | ☐ Exempt

Please attach a continuing review of studies form

Please enter the original IRB # at the top of the page

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: ☑ Full board ☑ Expedited

IRB Action:
☑ Approved - this proposal complies with University and federal regulations for the protection of human subjects.

Approval is effective until the following date: 2/7/2014

Items approved: ☑ Research protocol (dated )
☑ Informed consent (dated )
☑ Recruitment materials (dated )

☐ Other: 

Approval signature: ______________ Date: 3/8/13
March 4, 2014

Debarshi Chatterji
Department of Computer Science
College of Engineering
The University of Alabama

Re: IRB # 13-OR-045-R1 “Code Clone Taxonomy based on Cloning Intent”

Dear Mr. Chatterji:

The University of Alabama Institutional Review Board has granted approval for your proposed research.

Your renewal application has been given expedited approval according to 45 CFR part 46. Approval has been given under expedited review category 7 as outlined below:

(7) Research on individual or group characteristics or behavior (including, but not limited to, research on perception, cognition, motivation, identity, language, communication, cultural beliefs or practices, and social behavior) or research employing survey, interview, oral history, focus group, program evaluation, human factors evaluation, or quality assurance methodologies.

Your application will expire on March 3, 2015. If your research will continue beyond this date, complete the relevant portions of the IRB Renewal Application. If you wish to modify the application, complete the Modification of an Approved Protocol Form. Changes in this study cannot be initiated without IRB approval, except when necessary to eliminate apparent immediate hazards to participants. When the study closes, complete the appropriate portions of the IRB Study Closure Form.

Please use reproductions of the IRB approved informed consent form to obtain consent from your participants.

Should you need to submit any further correspondence regarding this proposal, please include the above application number.

Good luck with your research.

Sincerely,

[Redacted]

Office for Research Compliance
The University of Alabama
UNIVERSITY OF ALABAMA
INSTITUTIONAL REVIEW BOARD FOR THE PROTECTION OF HUMAN SUBJECTS
REQUEST FOR APPROVAL OF RESEARCH INVOLVING HUMAN SUBJECTS

I. Identifying information

Principal Investigator
Names: Debarshi Chatterji
Second Investigator
Names: Jeffrey Carver
Third Investigator
Names: Nicholas A. Kraft
Department: Computer Science
College: Engineering
University: Alabama
Address: Box 860290
Telephone: 8-528
FAX: 8-0219
E-mail: dchatterji@crimson.ua.edu

Second Investigator
Names: Computer Science
College: Engineering
University: Alabama
Address: Box 870290
Telephone: 8-9829
FAX: 8-0219
E-mail: carver@cs.ua.edu

Third Investigator
Names: Computer Science
College: Engineering
University: Alabama
Address: Box 870290
Telephone: 8-4740
FAX: 8-0219
E-mail: nkraft@cs.ua.edu

Title of Research Project: Code Clone Taxonomy based on Cloning Intent.

Date Submitted: 02/18/14
Funding Source:

Type of Proposal □ New □ Revision □ Renewal □ Completed □ Exempt
Please attach a renewal application
Please enter the original IRB # at the top of the page

Please attach a continuing review of studies form

II. NOTIFICATION OF IRB ACTION (to be completed by IRB):
Type of Review: _____ Full board □ Expedited

IRB Action:
□ Rejected Date: 
□ Tabled Pending Revisions Date: 
□ Approved Pending Revisions Date: 
□ Approved—this proposal complies with University and federal regulations for the protection of human subjects.

Approval is effective until the following date: 3-3-55.
Items approved: ___ Research protocol (dated ___)
___ Informed consent (dated ___)
___ Recruitment materials (dated ___)

Approval signature __________________________
Date 3/4/2014