Toward Improved Understanding and Management of Software Clones

by

Wei Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

The cloning of code is controversial as a development practice. Empirical studies on the long-term effects of cloning on software quality and maintainability have produced mixed results. Some studies have found that cloning has a negative impact on code readability, bug propagation, and the presence of cloning may indicate wider problems in software design and management. At the same time, other studies have found that cloned code is less likely to have defects, and thus is arguably more stable, better designed, and better maintained. These results suggest that the effect of cloning on software quality and maintainability may be determinable only on a case-by-case basis, and this only aggravates the challenge of establishing a principled framework of clone management and understanding.

This thesis aims to improve the understanding and management of clones within software systems. There are two main contributions.

First, we have conducted an empirical study on cloning in one of the major device drivers families of the Linux kernel. Different from many previous empirical studies on cloning, we incorporate the knowledge about the development style, and the architecture of the subject system into our study; our findings address the evolution of clones; we have also found that the presence of cloning is a strong predictor (87% accuracy) of one aspect of underlying hardware similarity when compared to a vendor-based model (55% accuracy) and a randomly chosen model (9% accuracy). The effectiveness of using the presence of cloning to infer high-level similarity suggests a new perspective of using cloning information to assist program comprehension, aspect mining, and software product-line engineering.

Second, we have devised a triage-oriented taxonomy of clones to aid developers in prioritizing which kinds of clones are most likely to be problematic and require attention; a preliminary validation of the utility of this taxonomy has been performed against a large open source system. The cloning-based software quality assurance (QA) framework based on our taxonomy adds a new dimension to traditional software QA processes; by exploiting the clone detection results within a guided framework, the developer is able to evaluate which instances of cloning are most likely to require urgent attention.

Acknowledgements

I would like to express my gratitude to my advisor Professor Michael Godfrey for his generous support during the last 2 years. His guidance, advice, suggestions and humor have made possible this research work.

I would also like to thank other two professors in the SWAG lab, Professor Ric Holt and Professor Reid Holmes. Both have offered their time to read this thesis and offered valuable suggestions. Professor Holt has offered his continuous suggestions about my work, particularly about communication skills. The passion for teaching and research I found in Professor Holmes is inspirational. I owe so much gratitude to my friend Dr. Ian Davis who helped my brainstorm ideas and invited me into so many interesting discussions about public affairs.

Thanks to all my peers in the SWAG lab. First I would like to thank Olga Baysal and Sarah Nadi for offering me helps that literally enabled me to survive in the lab, Kimiisa Oshikoji for his lunch and dinner times and his optimism, Raymond Nguyen for his support.

I have to thank the administrative staff in the school for their professional work, especially Wendy Rush and Paula Zister.

Special thanks to my parents.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software clones, that is, segments of source code that are highly similar to each other, are prevalent in software systems. Clones are usually introduced by "copy & paste" actions during software development.

Cloning is traditionally believed to be a negative sign (a bad "code smell"[20]) of overall software quality. Indeed, by duplicating code from one place to another, developers increase the maintenance risks in their software projects. Do they have a thorough understanding of the code being duplicated? Do they have good reasons for duplication instead of designing and writing code from scratch? Is seeking a common abstraction worthwhile? More importantly, do they have a clear intention of how to manage the evolution of a near-identical code fragments? Unsurprisingly, poor management of cloning has been found to cause problems such as the propagation of bugs after duplication, introducing dead code, and software bloat. Several empirical studies [36][8] have reported such instances in open source systems. In addition to causing maintenance headaches, cloning also carries the risk of infringing software licenses, sparking intellectual property disputes[4]. However, some other empirical studies [38][50] have found that for cloned code, the defect rate — the most important metric of software quality — is no higher than that of the non-cloned code.

Previous cloning researchers have mainly focused on developing automated clone detectors and quantitative analysis of clones in either proprietary or open-source software systems. The conventional yet untested notion about the harmfulness of cloning is often cited as the motivation of their research on automated clone detection. There have been a number of successful automated clone detectors[23][41][8], all of which can perform cloning analysis within a medium-to-large software systems.

Despite the supposed negative influence on software quality of clones, developers intentionally use cloning as a software development strategy. In particular, duplicating code can avoid development risks by reusing code from successful legacy systems [16]. This benefit is especially valuable for critical software systems, which are least tolerant to design or implementation mistakes. Kapser et al. [34] [35][33] further identified and summarized the patterns of utilizing cloning as a software development strategy in industry practices.

Whether cloning is harmful to software quality remains a question that is subject to empirical studies. And this question also influences how we manage existing clones: if cloning is proven to have negative influence on software quality, then we should emphasize on how to quickly remove cloning relations. There have been several tools that attempt to integrate clone detection results and subsequent clone removal strategies within software development environments [37][18] [49] [48]. However, little justification has been provided about the risks and benefits of this "hunt-and-kill" strategy. To our knowledge, there are no user studies on how these tools can actually improve the process of software maintenance, or the quality of software systems.

In this thesis, we address two aspects of cloning research: the understanding of clones and clone management. To advance our understanding of the role of cloning in software development, we first present a longitudinal study of cloning in a driver subsystem within the Linux kernel, where cloning is known to be a development strategy to overcome expensive and error-prone process of designing and writing new drivers when the underlying hardware changes only slightly. To establish a clone management framework, we propose a taxonomy of clones and profile clone instances by incorporating design knowledge and by crosschecking clone detection results with other software repositories. While we have not yet fully implemented all techniques in the proposed framework and have only conducted a partial evaluation, the results so far seem promising.

1.2 Contributions

This thesis makes two contributions: 1. We present a case study of clones in the Linux **S**mall **C**omputer **S**ystem **I**nterface (SCSI) driver system to clarify of the cause and other general features of clones in this system. 2. We proposed and partially evaluated a framework that utilizes cloning analysis to provide suggestions about clone management.

The first contribution involves a longitudinal study of clones in the Linux SCSI driver system. We analyzed absolute size, overall clone coverage, and average lifetime of clones within its layered architecture. We also discovered that a cloning relationship between two SCSI card drivers could effectively indicate high-level similarity among drivers. We further discuss how creating and deprecating drivers can affect cloning within this system.

Our second contribution involves a taxonomy-based framework that aids developers to manage clones. Our ultimate goal is to process the clone detection results and triage each item in the result list within the context of the underlying software system.

1.3 Thesis Organization

Chapter 2 presents the background and related work of cloning research, including the definition of clones and clone detection techniques. Other related research, such as software evolution, software feature analysis are also discussed.

One case study of clones is presented in Chapter 3. We discuss general background of our target system. Then we introduce our work on detecting clones, calculating clone-related metrics, and establishing a clone-based model to predict functional similarity among drivers in our target system. We further study the driving force behind the evolution of clones in the subject system.

Chapter 4 presents a framework of utilizing clone analysis results to improve software quality. We first introduce a taxonomy of clones and how we crosscheck cloning analysis results with software artifacts. Within the taxonomy, each category maps to a specific type of software quality issue. We further discuss the “top-10 list” techniques in our analysis. To investigate the effectiveness of our method, we have performed a partial evaluation of it on the open source Spring framework; we have also identified several bugs and other maintenance problems in this target system.

We summarize our contributions, and suggest future research directions in Chapter 5.

Chapter 2

An Overview of Cloning Research

This chapter provides an overview of research that is relevant to this thesis. We first provide definitions of important terms in this thesis. We then outline the major topics in the software cloning research, including automated clone detection and clone management. We further discuss related work in software evolution and software management.

2.1 Software Cloning: Definition and Detection

The term "software clones" may sound self-explanatory at first but the task of identifying the distinguishing characteristics of this term — a prerequisite of reaching a precise definition — has been clouded with unavoidable ambiguity. Researchers disagree on many aspects of a precise definition, including the minimal length of a duplicated code[8][32], the degree of similarity between clone pairs, and whether similarity in functions behind the code should be considered as a clone. A detailed discussion on these issues can be found in a survey on cloning analysis by Roy et al.[51].

Fortunately, a shared informal understanding on cloning has proven to be sufficient as a working definition for the research community. A typical working definition, proposed somewhat facetiously by Ira Baxter[8] , is:

> *"Code clones are snippets of code that are similar to each other according to some definition of similarity."*

This definition outlines two important features of cloning:

1. Software clones can occur not only in programs written in mainstream development languages such as Java or Python, but also in all kinds of programming languages, including mark-up languages, and machine languages.
2. Changes in criteria of similarity can alter the scope of software clones.

To address the problem of different degrees of textual similarity of software clones, the research community [9] typically categorizes clones as follows:

- **Type I Clones**:

  Type I clones have identical code fragments, ignoring differences in comments and white space formatting.

- **Type II Clones**:

  In addition to the differences in comments and white space formatting, Type II clones further allow differences in identifiers and explicit constant values.

- **Type III Clones**:

  In addition to all the allowed variances for Type II clones, Type III clones allows "gaps" among members of a clone

If we further loosen the constraints, code fragments with identical functionalities can be called **Type VI clones** or semantic clones.

The process of clone detection is often automated and typical clone detectors can discover at least Type I, II and III clones. The output of the clone detection consists of a set of *clone classes.*

Formally, a clone class C is a set of clone fragments f_1, f_2, f_3, ... , f_n where any pair of clone fragment (f_i,f_j) satisfies the definition of "is-a-clone-of" of a clone detector.

Many approaches of automated clone detections have been proposed and evaluated, including sequence-based techniques, graph- and tree-based techniques, hashing- and metrics-based techniques, as well as many hybrid methods. A typical clone detector usually begins the detection process by generating an internal representation of the source code, and then performs similarity comparison based on the internal representation. Previous case studies[51] have shown that it is common for 15–20% of the source code base to have a cloning relation with other code within the software system, although this depends on the system and the particular definition of is-a-clone-of.

2.2 Evolution of Clones

As Lehman [40] noted, software systems keep evolving to meet the ever changing external and internal driving forces. Cloned code snippets are no exception to this process.

Researchers address the problem of clone evolution from two perspectives. Several papers have performed quantitative analysis on aggregated data including overall clone coverage rate of a software system. For example, Livieri et al. [44] used the clone coverage rate as a metric to study the evolution trend over 136 versions of the Linux kernel.

Besides studying the overall trend of quantitative features of clones, researchers are also interested in tracing clone fragments and clone classes across versions. Kim et al.[37] traced clone classes over the evolution of an open source project and measured the lifespan of such clone classes. Göde et al.[23] integrated clone evolution analysis into a clone detection tool, called *iClones*, at the clone fragment level.

In this thesis, tracing clone fragment across versions is also automated by *iClones*. The definition of clone fragments is expressed by using syntactic information including content of the underlying source code, locations and the boundary of each clone fragment.

2.3 Management of Clones

With the untested notion of the supposed damages caused by clones to software systems, previous research on clone management mainly focused on automatic or semi-automatic refactoring to remove clones.

Toomim et al. [55] extended XEmacs to allow users to keep consistency among clone members by only editing one of the clone members. Baxter et al. [8] proposed a clone detection tool, called CloneDr, that suggested chances of elimination clones using in-line procedures or macros, which made CloneDr applicable to systems implemented in C or C++.

Not until recently, researchers have begun to evaluate the benefits and risks of this "hunt-and-kill" strategy to manage clones. As a result, recent researchers tend to provide computer-aided solutions for manual clone management instead of a fully automatic refactoring tool. Duala-Ekoko et al. [19] implemented a descriptive language within a development environment to help track clones over software evolution.

These solutions for clone management all suffer limitations in several aspects. First, many solutions attempt to solve the problem based on the a few unverified assumptions.

For example, adding extensions in development environments to enforce change consistency among clone members are driven by the assumption that all clones should have consistent changes. However, many clones are designed in a way that each clone member of one clone class should evolve independently. Besides these unverified assumptions, attempting to provide a unified solution for all clone instances often ignores the cause, length, location of individual clones. Clone management solutions with this problem inherently suffer from problem of overgeneralization, especially when users focus on managing a specific set of problems associated with software clones.

2.4 Experiment Setup

In this thesis, two subject systems are involved: the Linux SCSI driver system and the Spring Framework. The SCSI driver system is written in C (with a few chunks in assembly language) and the Spring Framework is primarily written in Java.

We use an automated clone detector, namely *iClones*[24], to identify and generate the clone detection results in all experiments in this thesis.

iClones detects clones of all three types and is capable of detecting clones in systems written in C, Java, or Ada. Details of *iClones* can be found in [24]. All clone detection results described in this thesis use 50 tokens as the minimal threshold in *iClones* as a candidate of a clone fragment; this threshold is widely considered appropriate as it exclude most accidental code duplication (for example, many “for loops” are identical with each other with a threshold of 15 tokens), while captures most real clones.

2.5 Summary

Although the basic idea of a software clone might seem simple enough at first glance, finding an acceptable common definition within the research community has proved difficult. Researchers try to capture the essence of this term to reach a working definition upon which we rely to develop clone detectors and exchange study results.

Regarding the ambiguity of this term, a three-level taxonomy of clone similarity is established, from the level of exact token match to the level that allows variance of one or more sentences among clone fragments within one clone class.

By tracking clones between consecutive snapshots of software systems, researchers can study the genealogy of clones. The consistency of changes among clone fragments in one clone class has been studied widely within the cloning research community.

The mainstream direction for clone management research was once to detect the clones and simply remove the relationship of duplication. Recently, researchers began to assess the risk introduced by this “hunt-and-kill” strategy and turned to a more conservative approach; that is, building tools to assist developers to manage clones within the context of each software project.

Chapter 3

Clone Evolution and Management in Linux SCSI Drivers

3.1 Introduction

Clones are most often the result of copy-paste actions. Cloning allows developers to reuse existing code-level designs within a new context rather than having to develop similar solutions from scratch [57][16]. This kind of reuse is especially convenient when a complex design space has been conquered once, and a similar space beckons.

However, the benefits of cloning come at a cost. Careless use of cloning may negatively impact the code quality of a system, especially with respect to maintainability. Along with desired functionality, bugs may also be migrated into the new space. And if bugs are eventually found, the resulting fixes may not be applied to all of the clones. The more complex the code, the harder it usually becomes to maintain these clones in parallel.

Reducing the study of code cloning to what is effectively an algorithms problem misses much of the rich context in which cloning is performed in practice. To understand the motivation, risks, and potential benefits of cloning requires that we study the background of the systems being examined; it is our experience that the project history, its organizational structure, the software development practices used, and the problem domain all bear strongly on how and why cloning may be employed. Previous work along these lines has included examining the perceived motivation and cause of cloning [16] [36], the construction of taxonomies of clone patterns [34][35][33], the relationship between software maintainability and cloning [47], and the impact of clones to software quality [52] [50].

This chapter presents a longitudinal case study of cloning as a development practice within the SCSI (Small Computer System Interface) drivers subsystem of the Linux kernel.

Specifically, this chapter addresses five research questions:

RQ1 *Do clones within the three architectural levels of the SCSI subsystem differ quantitatively?*

RQ2 *How do clones in the SCSI subsystem evolve between versions and over time?*

RQ3 *Can cloning be used as an effective predictor of hardware similarity?*

RQ4 *Does driver insertion/removal cause clone creation/removal? If so, to what agree?*

RQ5 *Are device-dependent cloned code chunks more or less likely to be duplicated within the SCSI subsystem?*

To measure cloning within the various architectural levels, we consider metrics such as absolute size of clones and clone coverage rates. To study the evolution of clones, we examine the ratio of clones over time, the lifespan of clones, and the kinds of changes encountered by clone classes over time. And to study the predictive power of cloning, we built three models — one based on cloning, one based on hardware vendor, and a random model — and evaluated how effective each was at predicting bus architecture compatibility.

There are four main contributions in this chapter:

1. We study how the software architecture of a system can affect and be affected by the practice of cloning.
2. We present a longitudinal empirical study of cloning in a large open source system, where cloning is known to have been used as an intentional development strategy.
3. We present a study of how studying cloning can be used as a proxy for determining hardware feature similarity among files that implement a similar interface.
4. By studying the proportion of many types of functions being cloned, we explore the likely motivations behind the decision to duplicate code.

3.2 Methodology

In this work, we have studied the subsystem of SCSI drivers of the Linux operating system. Unlike some previous studies, we have performed an architectural analysis, and measured the growth of the various components over time. We have also examined how cloning has been used as a development technique over time. We now provide some background information on the subject system, and discuss our methodology for performing these studies.

3.2.1 The Subject system: the Linux SCSI drivers

As mentioned above, the SCSI subsystem is a major class of device drivers of the Linux operating system. It provides support for peripheral devices that support the SCSI interface, which was designed for high-performance storage devices. The source code for the SCSI subsystem can be found in the `drivers/scsi` directory of the source code distribution. SCSI support has been present in Linux since the first official release of the kernel (in 1994). Like the rest of the kernel, the SCSI drivers are open source; however, most of the low-level drivers have been developed and then donated by the companies that developed the SCSI cards themselves. That is, unlike much of the core of the Linux kernel, the original development of most of the lower level driver code was performed within a company by its full-time developers, rather than the open source community.

To perform a longitudinal study of clones in the SCSI subsystem, we analyzed the source code of the kernel starting with version 1.0 (released in March 1994) and picking one version every six months ending at version 2.6.32.15 (released in June 2010). In total, we have analyzed 31 versions over the 16-year history of the Linux kernel.

One noteworthy feature of the SCSI subsystem is that it is designed and implemented as a strict three-level architecture: the top and middle levels provide a set of unified and consistent commands for the kernel to control various drivers, while the low-level drivers are concrete implementations of this functionality peculiar to the specifics of the particular hardware. Abstractly, the low-level drivers all implement the same functionality, such as hardware initialization, I/O handling, and error checking; they interact with the rest of the operating system only through the upper two levels.

We distinguish between "conceptual" low-level drivers and the files that implement them as follows: A conceptual low-level driver may provide support for a number of SCSI cards that share a similar hardware design. All drivers consist of at least two files: an implementation file (a "`.c`" file) and its associated header file (a "`.h`" file). However,

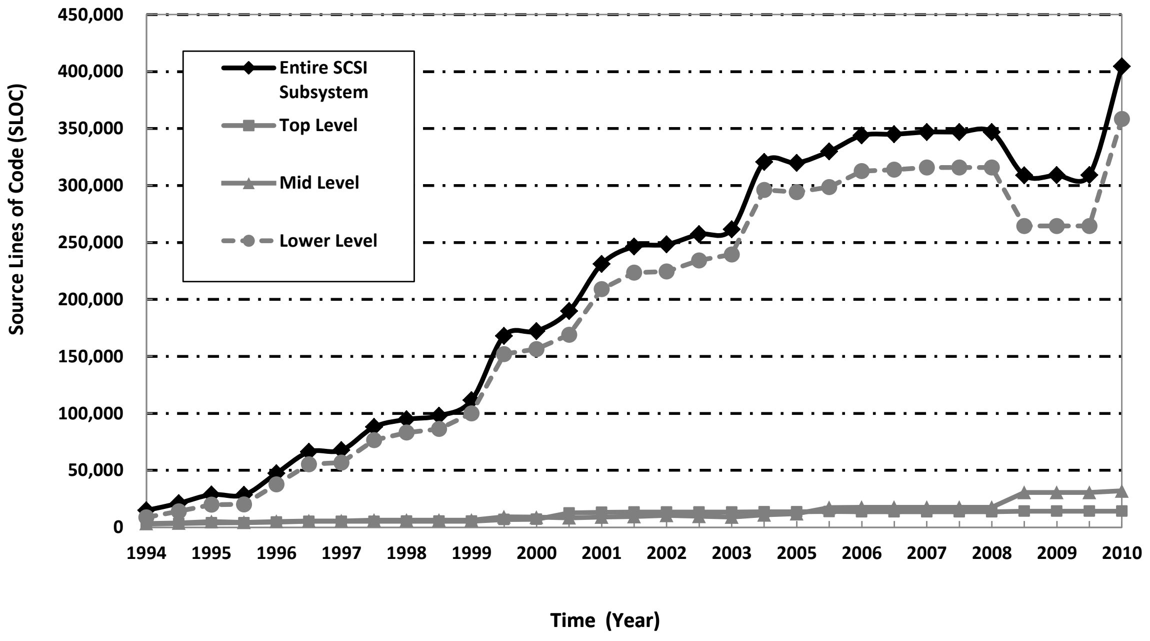

Figure 3.1: Growth in the number of Source Lines of Code (SLOC) for the SCSI subsystem as a whole as well as for each of the three architectural levels over all 31 snapshots, as measured by the open source tool *cloc*.

sometimes a single conceptual driver will span multiple implementation files, depending on the practices of the development team. These relationships are illustrated in Figure 3.2. Additionally, there are some `.h` files that are used by multiple conceptual drivers, due to the strong similarity between the cards.

In the most recent version (version 2.6.32.15) we studied, there are 96 conceptual drivers comprising over 319,000 SLOC[1] across 549 files. If we ignore shared `.h` files, we find that approximately 25% of the conceptual drivers are contained within a single source code file, about 50% are spread across two files, and the remaining 25% are spread across more than two files.

Since open source development allows for the reuse of existing code, it is not surprising that developers may choose to clone and then modify an existing lower level driver files when the underlying hardware of a new card is similar to an older one for which

[1]Source Lines of Code (SLOC) counts all physical lines that are not entirely comprised of white space or comments. We used the open source tool *cloc* to perform our measurements.

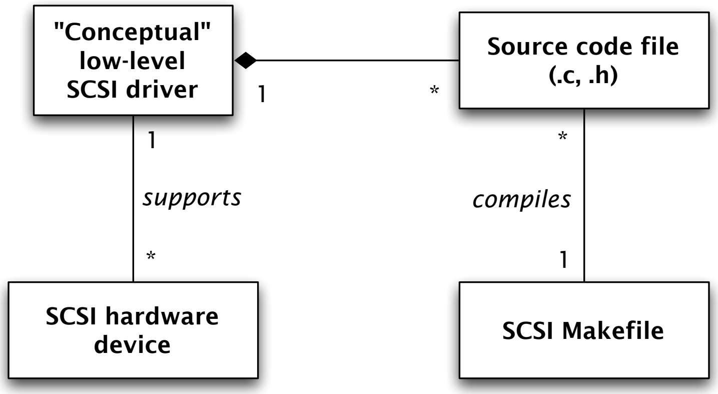

Figure 3.2: This UML class diagram illustrates the usual relationship between a SCSI device, its conceptual driver, and its source files. The composition relationship indicated by the solid diamond is slightly inaccurate as there are a few `.h` files that are shared among multiple conceptual drivers.

a driver already exists. Sometimes this cloning is explicitly mentioned in comments or documentation.

For example, consider this extract from the header comment for the Always IN-2000 card driver (file `in2000.c`, available in version no earlier than version 1.3.97 [1]):

> *I should also mention the driver written by Hamish Macdonald for the Amiga A2091 card [...] It gives me a good initial understanding of the proper way to run a WD33C93 chip, and I ended up stealing lots of code from it.*

The driver for the Amiga A2091 card driver (`wd33c93.c`) was first included in the Linux kernel in the version 1.3.94 (released in April, 1996), while the driver for the Always IN-2000 card first appeared in version 1.1.67 (released in November, 1994) but was later completely rewritten by another developer in version 1.3.97. Looking at the clone analysis results, it seems clear that the new developer for IN-2000 driver made significant use of the code for the Amiga A2091 driver. We found 42 code snippets in `in2000.c` that appeared to be “stolen from” the file `wd33c93.c`; these snippets comprised over a third of the code in the new driver (838 SLOC of cloned code out of a total of 2375 SLOC in the driver).

For each file of each version, we categorized it as belonging to one of the three levels using available evidence such as comments, documentation, static analysis results, configuration information, and build recipes.[2] Then for each of the 31 versions in our sample, we measured the size of each level as well as the SCSI subsystem as a whole in SLOC. Figure 3.1 shows the growth of the three levels and the SCSI subsystem as a whole over time.

We can see that initially the three levels are of comparable size; in version 1.0 the upper two levels comprise about 6200 SLOC, while the lowest level comprises about 8700 SLOC. However, over time the top two levels have shown relatively little growth while the lowest level has grown almost 50-fold, making up about 95% of the SCSI subsystem's 400,000 SLOC in the most recent version. Prima facie, these trends suggest that the SCSI subsystem is well engineered: The infrastructure has remained very stable over time, while the number of distinct "users" (concrete drivers) has shown steady and strong growth [26]
.

3.2.2 Clone detection

A clone detector automates the process of locating snippets of code with high similarity to each other. For this study, we have used the token-based incremental clone detection tool *iClones* from the University of Bremen [24]; its support for incremental detection made it particularly attractive to us for doing an evolutionary study.

Essentially, *iClones* takes multiple versions of a software system to be analyzed and computes the difference between each successive pair. Only the earliest version is processed with a conventional clone detection approach; clones in later versions are detected by textual differencing, combined with the detection results of the previous version.

Unsurprisingly, incremental detection using textual diffs is much faster than the naive approach of performing a clone detection across multiple versions of the full system's source code. Incremental detection has the additional significant benefit that it can track clone classes across versions automatically, since this is a simple side effect of the detection technique; to do so using the naive approach is likely to be significantly more effort and much less accurate.

As stated in Chapter 2, we used a threshold of 50 tokens as the minimum length for duplicated code to be considered as clones. The output of *iClones* detector reports a list of *clone classes* and each clone class contains a set of all *clone fragments* sharing the same duplicated code snippet.

[2]A detailed model of which file belongs to which layer can be found at Appendix A.

3.2.3 Results

In the following sections, we discuss the results of our study and examine each of the research questions in turn. We examine cloning within and across architectural levels over time, consistency of change between versions of clones, and cloning as a measure of feature similarity for low-level drivers.

3.3 Cloning and architecture

In this section, we address the first of our research questions:

RQ1 *Do clones within the three architectural levels of the SCSI subsystem differ quantitatively?*

We used *iClones* to performed a clone analysis on all 31 snapshots of the SCSI subsystem in our data set; this data set consists of snapshots at a six-month interval starting at version 1.0 in 1994 and continuing to the present. We also mapped each file and its constituent clones into its appropriate level within the three-level architecture of the SCSI subsystem, as described above. We wanted to track the amount of cloned code versus non-cloned code within each of the architectural levels. However, simply summing the LOC in each of the clone classes would give a misleading view, as snippets from different clone classes may physically overlap, and we did not want to count any given LOC more than once. Consequently, for each line of code in each file, we categorized it as either "clone" or "not a clone", based on whether it belonged to a snippet in any clone class.

3.3.1 Growth of clones in size of the SCSI subsystem

The three solid lines in Figure 3.3 shows (along the left y-axis scale) the absolute size of cloned code in each of the three architectural layers. It can be seen that the low-level drivers — which comprise most of the source code in the subsystem anyway — also comprise almost all of the cloned code in absolute terms. There are three dips in the absolute amount of clones in the lowest level; manual examination suggests that these dips were due to a number of deprecated drivers having been removed from the code base, rather than any large scale refactoring effort.

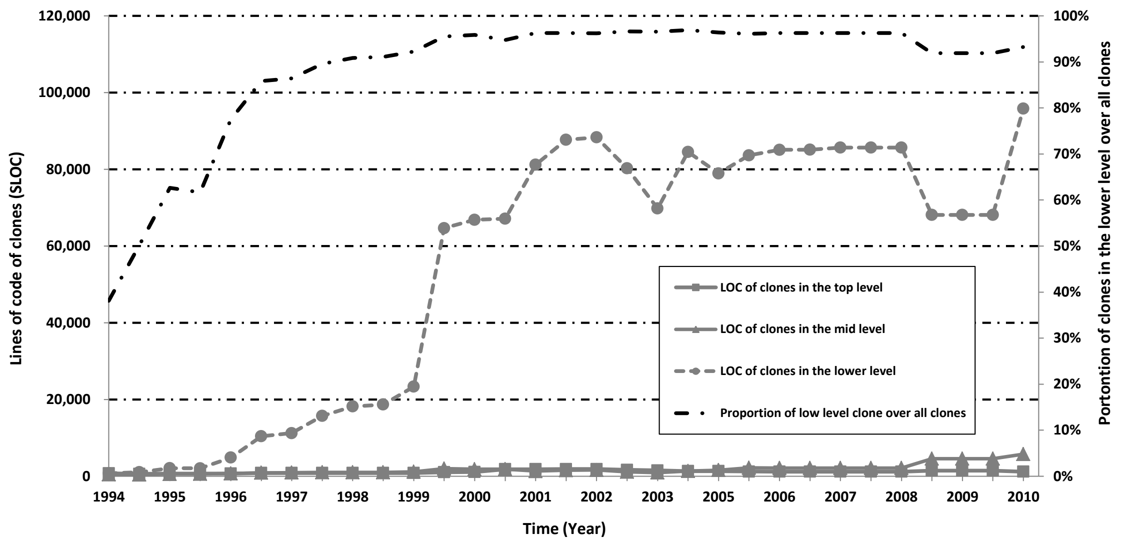

Figure 3.3: Absolute clone size in LOC for each level in the SCSI subsystem (solid lines, left y-axis), and the proportion of clones in the lower level over all clones (dashed line, right y-axis).

The dashed line in Figure 3.3 shows (along the right y-axis scale) the percentage of clones in the system that reside in the lowest level. It can be seen that after the first four years, 90% or more of the clones in the system reside in the lowest level. There is a mild dip over the last few years when there was a decrease in the amount of cloned code in the lowest level, combined with a mild increase in cloning in the other two levels.

3.3.2 Clone coverage rate of the SCSI subsystem

To get a more nuanced view of cloning within each of the architectural levels, we also computed the *clone coverage* rate, which is simply the proportion of clone code versus all code. That is, if a file (or a set of files) has a clone coverage rate of 0.2, then 20% of the lines of code in that file belong to a snippet that is a member of one or more clone classes.

Figure 3.4 shows that in early snapshots of the SCSI subsystem the clone coverage rates for all but the top level are relatively low; furthermore, and somewhat surprisingly, it is the low-level drivers that have the lowest clone coverage rate in this period. Over the long term, the top two levels have tended toward a rate of near 10%, although the top level has

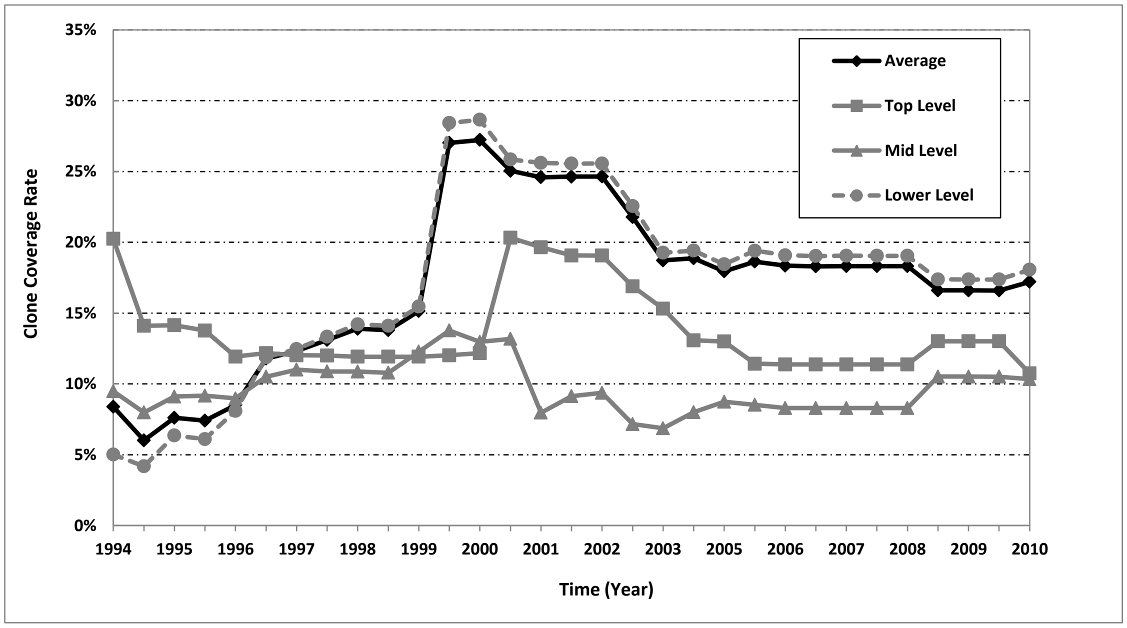

Figure 3.4: The clone coverage rates over time for each of the three architectural levels as well as the SCSI subsystem as a whole; this measures the proportion of code that is or has been cloned.

a large spike around 2000, which is immediately followed by a dip in the mid-level coverage rate.

The overall clone coverage rate has risen from just 6.0% to a maximum of 27.2% around 1999, and then fallen to just under 20% in recent years. It can be seen that the overall clone coverage rate closely matches that of the low-level drivers; this is because the low-level drivers come to overwhelmingly dominate the SCSI subsystem in terms of LOC, as Fig. 3.1 shows.

3.3.3 Discussion

We expected to see a significantly higher clone coverage rate among the low-level drivers compared to the rest of the system, since we had previously observed strong anecdotal evidence that cloning is used as a development strategy by SCSI driver developers [25]. We were surprised to find that, over the last several years, the clone coverage rate of the low-level drivers is almost double that of the two upper levels. A non-parametric Mann-Whitney U test revealed that the difference of the clone coverage rate results between the lower level and rest of the system is statistically significant with a confidence interval of over 99.99%.

Antoniol et al. studied clone ratios within and across subcomponents of the Linux kernel [5]. They found that clones were rarely scattered across subcomponents. In our study focusing on a single large subsystem of the Linux kernel, we also observed no clones spanning different architectural levels. Li et al. analyzed clones in the Linux kernel using data mining techniques, and suggested that SCSI concrete drivers for similar devices could be one of the major sources of high clone coverage rate of the drivers top-level directory over the rest of the Linux kernel modules [42], to some degree echoing earlier conjectures of Godfrey et al. [25]. Our analysis — which uses an entirely different clone detection technique from that of Li et al. — supports the hypothesis that low-level drivers contribute most of the clones, at least in the SCSI subsystem.

3.4 Clone evolution

In this section, we address the second of our research questions:

RQ2 *How do clones in the SCSI subsystem evolve between versions and over time?*

After studying how the clone class evolves as a whole, we also examine the evolution of the fragments contained by the various clone classes.

3.4.1 Change consistency of clones

- **Newly created** — None of the fragments of a clone class has a predecessor in the previous system snapshot.

- **Deleted** — none of the fragments of a clone class has a successor in the following system snapshot.

- **Consistently changed** — all fragments of a clone class change consistently from the previous snapshot to the next.

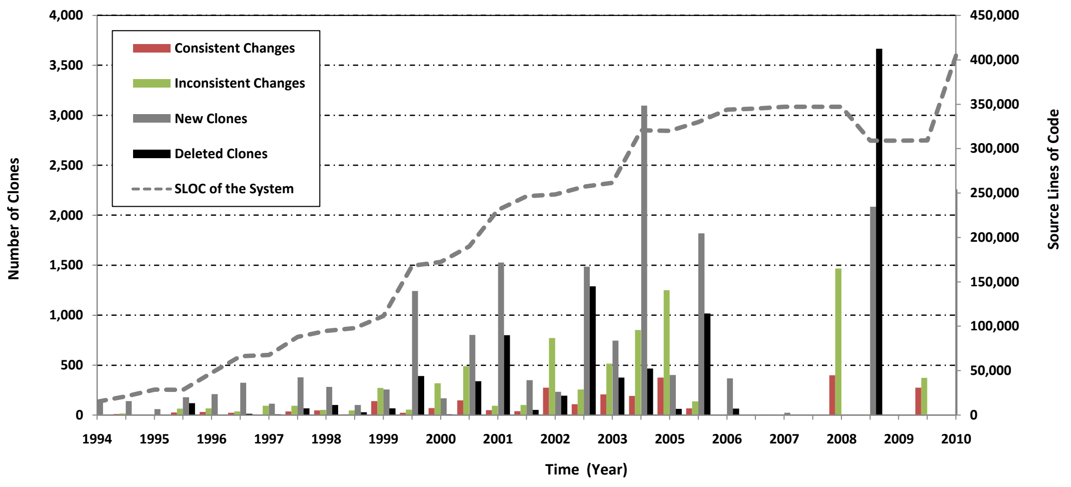

Figure 3.5: Clone change consistency and the number of created/deleted clones for each snapshot. The left y-axis represents the absolute number of clones and the right y-axis represents source code size in SLOC. Clones change during the entire evolution process we have studied. Inconsistent changes outnumber consistent changes for all versions of the SCSI subsystem we have studied.

- **Inconsistently changed** — at least one fragment of a clone class was modified in a way that is inconsistent with some other fragments in that clone class.

Two observations can immediately be drawn from Figure 3.5: First, inconsistently changed clone classes always outnumber consistently changed ones for all snapshots in our study where change was observed between versions. Second, new clones are being added to the SCSI subsystem continually over the time. We note that the unusual rise in the number of deleted clones instances in December 2008 (version 2.6.27.10) is due largely to the drop in the size of the subsystem, as can be observed in the dashed line plotted against the right y-axis.

3.4.2 Clone lifespan

To study the lifespan of code snippets of clones, we tracked all clone fragment traces over all 31 snapshots, and then calculated the absolute age of each fragment trace. Clone fragments traces are tracked across versions by using information such as content of source code, location, and boundary of each clone fragment; the tracing process is automated by

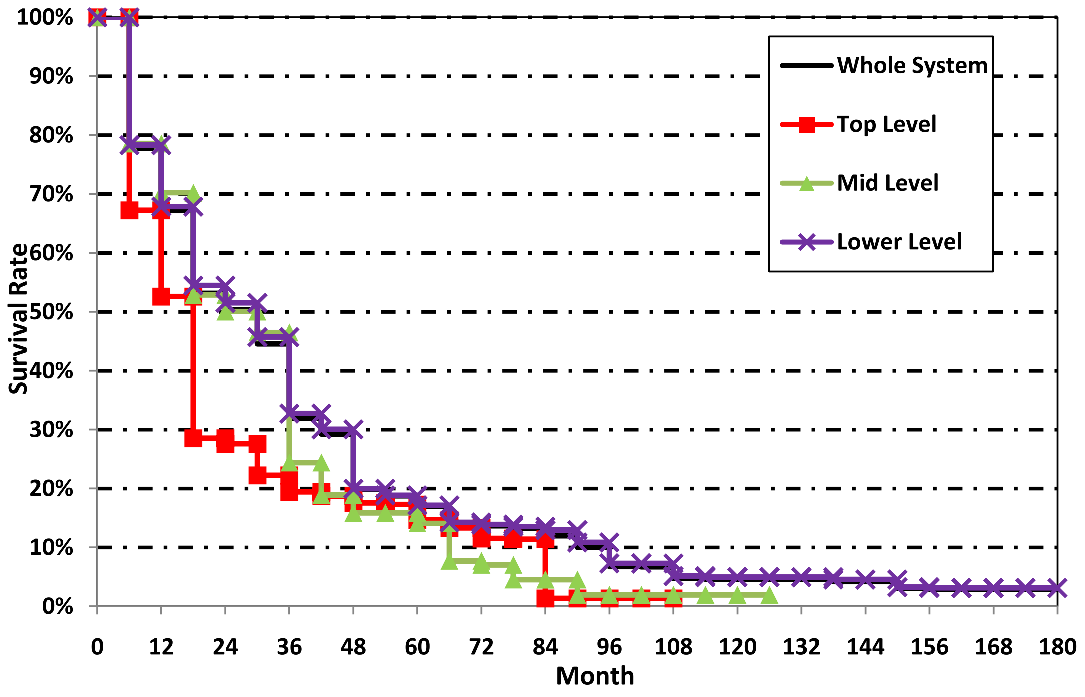

Figure 3.6: Kaplan-Meier survival curve for all clone fragments in the SCSI subsystem over all the snapshots we collected

the *iClones* detector. Note that a significant proportion of clone fragments have survived over the entire period of all snapshots we collected; in particular, recently created clone fragments are highly likely to persist in the most recent subsystem snapshot.

For example, for a clone fragment first observed at version 2.6.27.44 (released in January 2010) and still alive at the latest snapshot we collected (version 2.6.32.15, released in June 2011), it would be unrealistic to treat this fragment as if it had lifespan of only six months. It is also undesirable to exclude such survived clone fragments because, as shown in Table 3.2, over 30% of all clone fragments belong to "surviving" clones. To address this problem, we use the Kaplan-Meier estimator, where these fragments are called "right censored observations" in the survival analysis theory. The Kaplan-Meier estimator takes into account both censored and uncensored data instances.

Given a data-set of lifespan of clone fragments, let $t_i, i = 1, \ldots, I$ denote the ordered failure or censoring times, let d_i denote the number of clone fragments that disappear at

Level	**Unchanged**	**Consistent Changes**	**Inconsistent Changes**	Consistent and Inconsistent Changes	**Total**
Upper	518	161	632	65	1,376
Mid	641	87	493	19	1,240
Lower	16,121	2,933	10,490	503	30,047
Total	17,280	3,181	11,615	587	32,663

Table 3.1: The number of instances of clone fragment modifications

Level	**Censored**	**Died**	**Total**
Upper	344	1,951	2,295
Mid	682	1,104	1,786
Lower	12,898	34,278	44,121
Total	13,924	37,333	48,202

Table 3.2: The survival result of clone fragments in three levels of the SCSI subsystem.

time i, and let n_i denote the number of clone fragments (including censored fragments) that are still alive at time i. The survival function *S(t)* for the Kaplan-Meier estimator at time t is:

$$S(t) = \prod_{t_i \leq t} \frac{n_i - d_i}{n_i} \tag{3.1}$$

We calculated $S(t)$ for every 6 months for all clone fragment traces; the survival rate for each level of the SCSI subsystem is shown in Figure 3.6. Overall, we can see that clone fragments in the top-level have a shorter lifespan than those in the mid- and lower-level. The lifespans of clones in the latter two levels are similar to each other for clones up to about the age of four years; after that, clones in the lower-level have a much stronger survival rate. We note that even after 180 months, there are still a significant number of clones in the lower-level that have survived since the very first version.

The median lifespan of all clone fragments can be calculated by solving the equation $S(t) = 0.5$ or its approximate value when no survival function is available, as in our case.

The median lifespan of all clone fragments is 30 months, and for the top-, mid-, and lower-level the median lifespans are 18, 30, and 30 months respectively.

As discussed above, naively calculating the actual arithmetic mean or median value of all clone fragments, including the "right censored" (clone instances that terminated before the last snapshot we collected) ones, will unfairly underestimate the lifespan of all clone fragments. In our study, the naive arithmetic mean lifespan of all clone fragments is 23 months while the median is 15 months, which is much shorter than the median average lifespan measured by the Kaplan-Meier estimator.

To study the evolution of clones in the SCSI subsystem, we examined the proportion of different kinds of changes for each clone class of each snapshot. The clone detector in this study, *iClones*, can trace clone fragment across versions by comparing the similarity, location of clone fragments between two snapshots[23]; and this enabled us to backtrack all clones in the latest snapshot we collected.

One might expect that the upper- and mid-level would be less volatile than the lower-level, since changes in the higher levels might cause collateral changes to lower-level drivers. However, we have found no empirical evidence to support this notion. The proportion of unchanged clone fragments from the upper- and mid-level is not significantly lower than that in the lower levels. Furthermore, the median lifespan of clone fragments in the upper-level is even shorter than the overall median lifespan of all clone fragments.

Despite many differences in the systems being studied and other experiment configurations, we consider that it is useful to compare our results with findings from similar empirical studies on clone evolution. Kim et al. maps clone classes in different releases using location and textual comparisons [37]. They studied two systems; in the first, 37% of all genealogies lasted 41.7 days on average, and in the second 41% genealogies lasted 11.05 days on average. We note that the lifespan for both subject systems are shorter than observed in our system. We see two possible explanations for this. First, the systems studied may simply have different development patterns; second, we note that we study the lifespan of fragments while they studied the lifespan of clone classes.

Bettenburg et al. [10]reports a change consistency study on two open source projects — Apache Mina and JEdit — and found that the average lifespan for clone class genealogies are 4.59 releases (about 7.8 months) for Apache Mina and 9 releases (about 10.8 months) for JEdit.

The median lifetime in the SCSI subsystem is much smaller than the clone fragment lifespan observed in our study. It could be caused by the nature of different subject software systems and different ways of estimating lifespan information. Göde et al. also reported a study on the evolution of Type-I clones of a large system, Bauhaus, mainly written in

Ada [23][24][22]. Our findings about change consistency in the Linux SCSI subsystem support their conclusion that inconsistent changes comprise the majority of all clone class changes. With respect to the lifespan of clone fragments, they found the median age for clone fragments in their subject system to be 55 weeks. However, since they consider right censoring cases as indicating end-of-life, the median lifespan calculated using the Kaplan-Meier estimator might be higher.

3.5 Cloning as a predictor of hardware similarity

In this section, we address the third research question:

RQ3 *Can cloning be used as an effective predictor of hardware similarity?*

To evaluate this question, we built three models to predict bus type architecture compatibility: one based on cloning, one based on hardware vendors, and a random model. We now describe this work.

3.5.1 Background

In previous sections, we have shown that most clones in the SCSI subsystem are contributed by low-level drivers, and that the clone coverage rate within that architectural level is significantly higher than in the rest of the subsystem. While the Linux operating system supports a wide range of SCSI cards — there are over 90 "conceptual" drivers in the most recent version — each individual driver must still implement the same API within the three-level architecture specified by the Linux SCSI design documents. It is therefore not surprising that a developer creating a new driver might choose to copy and paste snippets of code from existing drivers, especially if there are commonalities in the underlying hardware, design philosophy, or if the concrete designs share cross-cutting concerns [36].

To find out how common this is — and specifically to find out how often cloning seems to be driven by similar hardware features rather than, say, laziness — we built a cloning-based model within the low-level drivers to test its effectiveness as a predictor of hardware similarity.

There are several dimensions along which one might organize a taxonomy of SCSI cards and their drivers, such as the generation of SCSI protocol being supported or the application domain of the device (e.g., storage device, CD-ROM drive, etc.). However,

```
1  config SCSI_IN2000
2        tristate "Always IN2000 SCSI support"
3        depends on ISA && SCSI
4        help
5          This is support for an ISA bus SCSI host
6          adapter.  You'll find more
7          information in <file:Documentation
8         /scsi/in2000.txt>. If it doesn't work out of
9         the box, you may have to change the jumpers
10        for IRQ or address  selection.
11
12        To compile this driver as a module, choose M
13        here: the module will be called in2000.
```

Figure 3.7: The configuration data for the Always IN-2000 device.

the availability and accuracy of information sources for many of these dimensions are problematic, especially for older devices. To overcome this, we use the computer bus type dependency extracted from the build configuration data (i.e., the "`Kconfig`" files) of the SCSI subsystem to measure the similarity between two drivers. Specifically, the `Kconfig` file shows the hard dependency of bus types for each driver (e.g., `ISA`, `PCI`, `EISA`) so that users can configure the build options to ensure the driver will run properly.

The configuration data for all low-level drivers in the SCSI subsystem consists of at least three mandatory entries: *name*, *dependency*, and *help text.*Figure 3.7 shows the complete configuration for the Always IN-2000 card: Line 1 gives the name of the configuration option, which corresponds to this driver as `SCSI_IN2000`, line 3 lists its required compilation dependencies, and lines 5–13 give the short, descriptive message intended to help the user understand what device support is being enabled by this option. The dependency entry lists the other configuration options that are required to be enabled to include support for the this card: `SCSI` and `ISA`.

There are several advantages of using the bus type dependency entry of SCSI card drivers as the variable to compare similarity:

1. Bus type dependency information is released with the `Kconfig` file of the Linux kernel to enforce correct configuration for a driver. That is, it is checked and maintained.

2. Bus type support is a fundamental feature of a hardware card, and driver developers must conform to a set of communication protocols determined by bus type support of the hardware. That is, it has essential semantic value to the driver.

3. Bus type dependency entries are represented as logical expressions which makes comparisons easier, automatable, and less error prone than natural language descriptions.

For these reasons, we consider using bus type dependency as the variable to compare similarity to be a reasonable choice.[3]

3.5.2 Methodology

To compare bus type dependency entries, we first convert each logical expression into a disjunctive normal form (DNF) formula. For a pair of DNF formulas, we define three levels of matching:

- Two entries are considered to be a *match* if there is at least one common conjunctive term that appears in both formulas.
- Two entries are considered to be a *partial match* if there is a conjunctive term in one formula can be expressed by concatenating extra variables in one conjunctive term in another formula.
- Two entries that are not a match or partial match are considered to be a *mismatch*.

For example, the DNF formula for bus type dependency for the IN-2000 card is "`ISA && SCSI`". This formula matches the formula "`(ISA && SCSI) ||(PCI && SCSI)`" because the formula for IN-2000 appears in one of the conjunctive term in the latter formula. It is also a partial match of DNF formula "`ISA && SCSI && PCI`" since the latter can be formed by concatenating extra variables onto the formula for the IN-2000. The formula "`SCSI && X86_32`" is a mismatch with IN-2000 because there is no way to make a match or a partial match. We built an automated tool to perform the matching evaluation.

After establishing a similarity metric for conceptual drivers based on bus type dependency, we decided to evaluate how strong an indicator cloning is for similarity in the underlying hardware. We built three models: one based on cloning information, one based on hardware vendor, and a random model. We then evaluated for each pair of entries in each model the strength of the bus type dependency match.

For the cloning-based model, we first mapped each low-level driver file to the conceptual driver that it implements, and then "lifted" the cloning information to the level of conceptual drivers. That is, suppose that we found a cloning relationship between two files,

[3]We note that the `Kconfig` entries model other information, including annotations, including annotations for the Linux kernel development process, such as "EXPERIMENTAL" or "BROKEN", but we have discarded all except the bus architecture data in our study.

`alpha.c` and `bravo.c`, and that `alpha.c` and `bravo.c` are part of the Delta and Foxtrot conceptual drivers, respectively. Then we would infer a cloning relationship between Delta and Foxtrot.

We then asked for each pair of conceptual drivers that have a cloning relationship: What is the strength of the match of the corresponding bus type dependency formula? For the vendor-based model, we took pairs of cards from the same hardware vendor and evaluated the strength of the match of the corresponding bus type dependency formula. We took the vendor information from the `Kconfig` entry for each conceptual driver.

Finally, we created a randomized model to test if the results of the other models are meaningful. We generated this model by randomly selecting 100 pairs of drivers and comparing their bus type similarity. However, not all conceptual drivers specify their bus type information in the `Kconfig` file; consequently, to make the comparisons fairer we limited ourselves only to those drivers that explicitly specified this information.

3.5.3 Discussion

As shown in Fig. 3.8, the cloning-based model has the highest accuracy rate of the three. Out of 110 pairs of conceptual drivers that share clone code snippets with each other, only 8 have a bus type mismatch while about 90 pairs match each other, and 12 are partial matches. We were surprised how much stronger these results were compared to the random model, where 76% of the pairs were a mismatch and only 9% were a match. This suggests that the existence of a cloning relationship between two conceptual drivers is a strong indicator of hardware similarity for bus architectures.

The cloning-based model also fared significantly better than the vendor-based model, which we found surprising. Although the vendor-based model performed much better than the random model, having the same vendor resulted in a bus architecture match only 54% of the time, with about a third being mismatches. This suggests that the vendor-based model is a good but not strong indicator of bus type compatibility.

Finally, we note that for the random model, 76% of the randomly chosen pairs were mismatches. This suggests that bus type compatibility is a non-trivial relationship, and that the results of the other two models are meaningful. To summarize, we have presented strong evidence that cloning relationships are a good predictor of bus architectures in SCSI cards in the Linux kernel operating system. We feel that this opens up broader questions of the predictive power of cloning on other kinds of relationships, such as other kinds of similarity in the underlying hardware or feature sets of classes of software components that implement similar functionality; however, we do not address them here further.

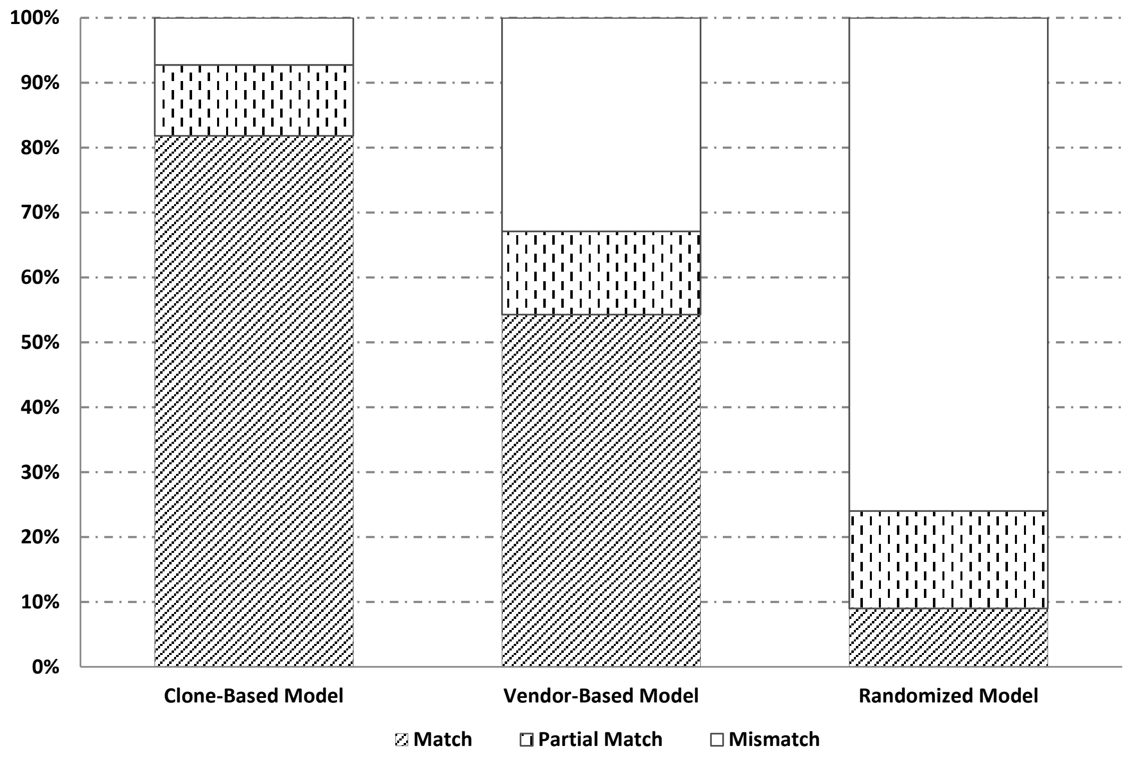

Figure 3.8: Accuracy of predicting bus type similarity using the cloning-based model, the vendor name-based model, and the randomized model. The numbers on the bars show the absolute number of pairs in that category; the y-axis shows the percentage of each level of match.

3.6 Clone evolution driven by driver development

In the previous sections, we have examined the evolution of clones by tracking clone traces along its life cycle and calculated the lifespan of each clone trace. We have also provided quantitative evidence regarding the absolute number of created/deleted clone fragments for every snapshot we have collected.

Many researchers have studied the likely reason behine code duplicating in software development yet we can only at best approximate the answers by recording software development plus on-site interviewing to developers[36]. Source code is usually the primary development artifact that is available for most cloning research; however, a source code repository alone is insufficient to recover motivation and the process behind clone-related activities. This study is as well constrained by this limitation. However, knowing the development model of the SCSI drivers, particularly the facts of when each low-level driver was inserted and removed from the system, we can study whether and how clone evolution is influenced by the life cycle of low-level drivers.

With the source code and the version control repository, we can recover an accurate time-line for each case of driver insertion and removal. With this time-line, we can address the fourth research question:

RQ4 *Does driver insertion/removal cause clone creation/removal? If so, to what agree?*

The goal of investigating this research question is to best recover the motivation and features behind copy-paste activity, especially from the perspective of driver developers. This empirical study can provide new insights about copy & paste practices, especially for driver development.

In previous sections, we have discussed the evolution of clones, including scenarios of introducing new clone fragments, removal of clone fragments.

3.6.1 Driver removals as the cause of clone removals

We use *iClones* to analyze clone evolution. A clone is considered “removed” by *iClones* if it cannot be traced in the following snapshot of the SCSI subsystem. In addition to the clone chunks that have been actually removed, code chunks that have been modified may also be considered “removed” by *iClones* if the clone detector cannot trace the modified code chunk because of the changes of the modified code chunk.

In Figure 3.5, it can be observed that the number of deleted clone fragments fluctuated dramatically — the number peaked in December 2008 (as of version 2.6.27.10) , with 3665 clone fragments being removed — while other versions have far fewer removal instances. In order to pinpoint of cause of clone fragments removal and to find out the reason of such fluctuation, we analyze all clone traces collected over time and study all the traces that are found to be terminated. For a clone trace that has been reported to be terminated in one of the versions, we ask whether the source file that contains the removed clone snippet is also removed.

We found 3101 cases of clone removal that were directly caused by deprecating old drivers over time, which accounts for 31.8% of all removal cases for clone fragments. Particularly, in the version 2.6.27.10, 81.2% (1790 out of 2202 instances) of all clone fragment

Fragment	File	Begin Line	End Line	Length
1	`scsi/cyberstormII.c`	76	56	121
2	`scsi/fastlane.c`	95	120	121
3	`scsi/blz1230.c`	85	110	121
4	`scsi/cyberstorm.c`	84	109	121

Table 3.3: Source location of a clone which has been inserted into multiple drivers in the SCSI subsystem.

removal cases are caused file-level deletion. This correlates well with the number of deprecated SCSI drivers in this version: there were 58 SCSI drivers being deprecated in the same version according to the official Git notes of Linux kernel.

From the above result, we can conclude that developers for the SCSI drivers do not often deliberately refactor clone snippets to remove them. In fact, file removal is the major driving force of removing cloned code from the system. The sudden rise of clone removal cases in the version 2.6.27.10 simply reflects a one-time deprecation of many old drivers.

3.6.2 Introducing new clones when inserting new low-level driers

In the previous sections, we have shown that cloning has been used to aid the development of new low-level drivers. The most conceivable scenario of introducing new clone fragments is duplicating code snippets from an existing driver to a new driver. However, there are a number of alternative situations that can introduce new clones that span two or multiple concrete drivers.

The example of driver IN2000 (refer to section 3.2.1) stands as an example. The driver file for IN2000, `in2000.c`, first appeared before `wd33c93.c` (the driver for the A2091 card). However in2000.c borrowed 42 code snippets from `wd33c93.c` when the file `in2000.c` was refactored.

To assist discussion, one instance of such clone is shown in Figure 3.9.

The same piece of code snippet is inserted into four concrete drivers: “cyberstormII”, “fastlane”, “blz1230” and “cyberstorm”.

This cloned code snippet permits the motherboard to detect the underlying SCSI device and then register it within the Linux kernel. Because of the commonality of design within

```
if(eregs->esp_cfg1 != (ESP_CONFIG1_PENABLE | 7))
         return 0; /* Bail out if address did not hold data */
      esp = esp_allocate(tpnt, (void *) esp_dev);

      /* Do command transfer with programmed I/O */
      esp->do_pio_cmds = 1;

      /* Required functions */
      esp->dma_bytes_sent = &dma_bytes_sent;
      esp->dma_can_transfer = &dma_can_transfer;
      esp->dma_dump_state = &dma_dump_state;
      esp->dma_init_read = &dma_init_read;
      esp->dma_init_write = &dma_init_write;
      esp->dma_ints_off = &dma_ints_off;
      esp->dma_ints_on = &dma_ints_on;
      esp->dma_irq_p = &dma_irq_p;
      esp->dma_ports_p = &dma_ports_p;
      esp->dma_setup = &dma_setup;

      /* Optional functions */
      esp->dma_barrier = 0;
      esp->dma_drain = 0;
      esp->dma_invalidate = 0;
      esp->dma_irq_entry = 0
```

Figure 3.9: Source code of a Type I clone that was inserted into multiple SCSI drivers.

the 4 drivers, a similar approach to implement SCSI card detection in drivers can be simply reused by copying & pasting, without any minor customization within the new source code destination.

We first study the number of low-level driver files that are involved in clones with other low-level driver files. This question is important because it shows how often a new low-level driver reuses code snippets from other drivers. Note that we are looking at the file level instead of driver level. That is because one header file can be potentially shared by multiple drivers (however such header files usually do not have duplication with other files); as a result, identifying the ownership of header files is difficult. Moreover, most low-level files have a one-to-one relationship with the corresponding conceptual driver (as shown in Figure 3.2).

To study this question, we process the detection results generated by the *iClones* detector. We examined each file containing code for a low-level driver, and then determined whether it ever contained clones in other low-level files associated with other drivers.

There are 786 files with distinct file names that ever appear as low-level driver files within the SCSI system, of which we found 443 files to be associated with clones. To answer the question about clones spanning multiple drivers, there are 301 files that contain clones with other low-level driver files. In other words, every time a developer writes a new low-level SCSI file (one file represents one driver in most cases), she has a chance of 38% (301 files over 786 files) of duplicating code snippets from another concrete driver file within the SCSI subsystem. This fact again supports the notion that SCSI driver developers often use cloning as a development strategy to write new drivers.

Another important scenario of introducing new clones is to when one code snippet is inserted into several different files at the same time. In this scenario, developers would introduce clones — mainly by inserting code snippets with same content — to several driver files. Because we analyze snapshots of the SCSI system at a six-month interval, any new clones that are introduced within the six-month window could be potentially considered as positive cases even if they are not introduced at the precisely same time. However, we consider that the six-month window provides a reasonable granularity to approximate the fact of this scenario.

To study this scenario, we tuned the *iClones* detector to report clone classes that only contain newly created clone fragments at one snapshot. After tuning the *iClones* detector, we have also calculated the number of files that contain such clones over the entire history of the SCSI subsystem to show the prevalence of such phenomena within the SCSI subsystem.

We found there are 3624 clone classes that belong to this scenario over time and 297 files out of 433 files contains clones under this scenario. It shows how often developers introduce clones in the scenario where developers of SCSI drivers copy & paste one piece of code snippet to implement functions that are common spanning multiple drivers. Another interesting phenomenon is that every major peak for new clone fragment shown in Figure 3.5 correlates well with peaks observed in this scenario. Top three instances of such peak of newly created clone fragments all contain a large number of files under this scenario. This suggests copy-and-paste single code fragment into multiple drivers is an important contributing force of sudden rise of newly created clone fragments for many snapshots.

3.7 Proportion of routines in cloned code snippets

Concrete device drivers serve to support SCSI cards by invoking predefined interfaces provided by the upper levels in the SCSI subsystem. Therefore most SCSI driver developers follow a well-structured programming paradigm to write their own drivers — such as device initialization, port mapping with firmware and I/O related operations. Some kinds of source code, such as port mapping, are more device specific while others are independent of device specifications. Recall that we have established a cloning-based model which can effectively predict bus type similarity between two SCSI drivers. A follow-up question is:

RQ5 *Are device-dependent cloned code chunks more or less likely to be duplicated within the SCSI subsystem?*

In this section, we examine the proportion of clones within the SCSI lower level drivers. We first propose taxonomy of source code in SCSI drivers according to the official development guide for SCSI drivers; we further examine the percentage of the proportion of each category for clones that span SCSI drivers (referred as diff-driver clones later in this thesis). We also report the same percentage data for clones that exist only within one driver (same-driver clones).

We categorize source code into 6 major categories:

- initialization routine
- device open/close routine
- function building routine
- sense code-related routine
- configuration routine
- others routines

The dimension of this taxonomy is established according to the official development guide of SCSI drivers [2] [13]. The taxonomy attempts to capture fundamental features of essential components required to write a fully-functioning driver from the perspective of developers. This taxonomy also allows for manual triage by identifying a set of syntactic signatures of each category.

Category	Feature
Initialization Routines	Invoking specific methods from upper levels to enable initialization
Device Open/Close Routines	Invoking device open/clones implementations from upper levels
Function Building Routines	Functions other than initialization and device open/close, usually error handling.
Sensor/Port-related Operation	Reading and writing data from ports or sensors-related arrays that describes the status of SCSI cards
Configuration Routines	Reading and writing data from ports or sensors-related arrays of the underlying SCSI cards
Other Routines	Code snippets that cannot be put into any other categories above.

Table 3.4: Types of routines in the lower level drivers of the SCSI driver system.

For example, initialization routines must contain method invocations from upper architectural levels concerning device initialization with a specific interruption handler. For device open/close routines, they must also invoke generic device open/close interfaces from upper levels. Sensor code-related routines must access and read data from ports or sensors of the underlying SCSI cards. Configuration routines bear the feature of long code snippets about accessing predefined C structures that determines the status of the driver. Function building routines include important functions except — such as error handling functions or showing device status to users — are categorized as function building routines.

In our manual triage process, we also use possible comments around the clone fragment to quickly determine the purpose of every code snippets. With these features identified from the clone snippets, most clone fragments can fit into one of the category. With the above signatures, we can also minimize the involvement of subjective factors when classifying. There are still a small percentage of clone fragments that are hard to put into

	Initialization Routines	Device Open/close Routines	Function Building Routines	Sensor/Port-related Operation	Configuration	Others	Sum
Diff-driver clones	13 (13.7%)	14 (14.7%)	36 (37.9%)	25 (26.0%)	6(6.3%)	1 (1.0%)	95
Same-driver clones	13 (5.6%)	37 (16.0%)	89 (38.5%)	32 (13.9%)	45 (19.5%)	15 (6.5%)	231

Table 3.5: Proportion of routines in respect to SLOC in the SCSI driver system.

any of the categories and we list them as others in our taxonomy. Description of typical routines can be found in the official guide [2] [13] of SCSI driver programming in the Linux kernel.

In our study, we sample 20% of diff-driver clones and 5% of the clone classes from same-driver clones. After collecting samples from both groups, we ask what category fits best for the cloned area in each clone class. Given that the categories under our taxonomy are mutually exclusive, one clone class is only assigned to belong to only one category.

The result is shown in Table 3.5. It can be observed from the Table 3.5 that function building routines take up the largest proportion for both categories. With two groups under comparison, one noticeable distinction is that the proportion of initialization routines in the same-driver group is smaller than in the diff-driver group. The possible explanation is that a single driver may not need multiple initialization routines within itself while duplicating initialization routines from an old driver to a new one can assist writing quick driver development if the new driver shares similar initialization process with the old one. A similar reason is also applicable to the phenomena that less port-related operation routines are cloned when comparing same-driver group to diff-driver group.

Another observation based on this experiment is that diff-driver clones contain more code snippets that are hardware-dependent, such as device initialization, open/close routine and sensor/port-mapping operations. Results of this experiment also support the notion that functional-level similarity among hardware drivers is the major force that caused developers to copy & paste code snippets from one concrete driver to other drivers.

3.8 Summary

In this chapter, we describe a longitudinal study of the SCSI driver system in the Linux Kernel. We found that both the absolute number and coverage rate of clones among lower level drivers are significantly higher than among files in the upper two architectural level.

We observed many more inconsistent changes than consistent changes across clone classes over time. We also found that a spike in deleted clones was usually accompanied by a spike in new clones, suggesting that significant refactoring was occurring whenever old drivers were deprecated. And we found that new clones were continually being added to the system, even in the absence of observed refactoring.

We further used the Kaplan-Meier Estimator to study lifespan of all clones and found that the median lifespan of all clones is approximately 30 months.

We found that if two conceptual drivers had a cloning relationship, then there was an 82% chance that they had compatible bus architectures. This was a much better predictor of compatibility than having the same vendor or being chosen randomly; consequently, we conclude that cloning can sometimes be used as a predictor of hardware similarity. Of course, we have performed only one study measuring one possible dimension from one domain, so much work remains to be done to address the broader questions implied by RQ3.

We further examine two another research problems: clone evolution from the perspective of design and what kinds of code snippets contribute most clones in lower level drivers. By tuning clone detector and crosschecking clone detection result at file-level evolution, we conclude that a fair proportion of clones are removed simply because the driver file that the driver is deleted as deprecated drivers.

We also establish taxonomy of cloned code along the dimension of functionalities that are instrumental to support a driver. It shows that the most hardware-dependent kind of code are sensor/port-related routines contribute more percentage of clones to diff-driver classes than same-driver classes while other code routines, such as configuration routines, reports higher proportion in same-driver clones.

Chapter 4

Cloning Analysis as a Part of Software Quality Assurance

4.1 Overview

Software quality can be affected by many activities of software developers, and copying & pasting source code is certainly one of them. Cloning was once widely believed to have negative effects on the overall software quality. However, recent empirical studies on this topic have provided some contradictory evidence[10][50]. Furthermore, cloning is a software development strategy that allows reuse of valuable software design and implementation, and recent studies have provided evidence that cloning can be used in a principled way as a design tool [34][35][33][16].

In this chapter, we present a preliminary study that propose a framework that utilizes cloning analysis as part of software quality assurance (QA). The ultimate goal of this framework is to help a QA team to effectively address cloning-related quality issues. We first identify the key challenges in using cloning analysis; then we discuss how a new taxonomy of clones can reduce the candidate set of QA-related clone classes. A preliminary experiment on a large open-source system reveals several quality problems in this software.

We start our study by identifying the obstacles to utilizing cloning analysis to improve software quality. Our proposed framework crosschecks the cloning analysis results with other software artifacts, such as bug repositories. The result of crosschecking highlights a subset of clone detection result so that QA teams can focus on clones that rank highly according to specific “interestingness” criteria.

We note that the work described in this chapter is at a early stage. We have 1. proposed a framework of utilizing cloning analysis for software QA, 2. implemented part of it, and 3. performed a preliminary study using it on a large open source system. Our work has also a few external limitations, especially the lack of metrics for a QA framework, which prevent us from comparing our framework from existing approaches. In this chapter, we list the findings by applying our framework to a large open-source software, to explore the feasibility of our framework.

4.2 Obstacles to utilizing cloning analysis within a software quality assurance process

As an non-mainstream way of design and implementing software functions, duplicating code snippets carries many risks with respect to software quality. The presumed harmfulness of cloning has also been used to motivate many cloning research projects, which often promise that a better understanding of cloning would will aid in mitigating the harmfulness of cloning. However, to our knowledge, cloning analysis has not proven to effectively improve software quality for conventional industry practice. We believe the possible reasons include:

1. **Lengthy cloning detection reports**

   A typical software system usually contains a large amount of clone classes which takes up 15%–20% of the entire code base, regardless of what systems are being analyzed [51]. As a result, processing a clone report demands unrealistic resources from a software team. The fact that most clone detection reports are unranked and unstructured only makes cloning detection report even less useful for software quality assurance.

2. **Low sensitivity of cloning as an indicator of software quality issues**

   Clones might be a symptom of sloppy software design, but the ratio of "problematic clones" is low, in terms of bugs[50]. As a result, the overall clone detection results may be insensitive to many of the software quality problems.

3. **Uncertain and illusive benefits of clone management**

   Many aspects of software quality issues, such as maintainability, lack accurate and broadly accepted metrics. As a consequence, the effectiveness of addressing clone-related issues also suffers from the difficulty of measuring changes in software quality by addressing clone-related issues.

Due to the limitation of cloning analysis for software QA, an effective software quality assurance solution cannot be achieved solely through cloning analysis. Cloning analysis can at best complement other QA methods. However, cloning has proven to be caused or associated with a wide range of software quality issues, including bug propagation, license infringement; and these quality issues may have multiple copies due to the nature of cloning.

4.3 Taxonomy of Clones for Studying Software Quality Assurance

Software quality is the degree of conformance of one software system to software product specification [53] [17]. Functional requirements of a software system — usually specified by a software product specification — can be measured by the failure rate of running test cases against the software system. Non-functional requirements (NFRs) of a software product — such as usability, understandability or robustness — can be difficult to state unambiguously, let alone the establishment of measurements[56].

A number of metrics have been established to measure NFRs of a wide range of software systems. For instance, cyclomatic complexity of source code is often employed as a metric of overall code quality, yet arguably this metric suffers from the limitation of only addressing problems that are associated with overly complicated control flows.

Cloning analysis has been applied to assess software quality in previous research, mainly focusing on software defects rate (defects per software repository unit) in clone snippets. Monden et al. [47] evaluated qualitative evidence of the impact of clones to software defect rate and revision frequency, concluding that software modules containing clones with long snippets (200 SLOC as the minimal threshold as “long snippets”) can cause problems in software reliability (measured by the defect rate) and maintainability (measured by the revision frequency). Li et al.[42][41] concentrated on bugs that are caused by renaming of identifiers in duplicated clones.

Despite many important facts and insights provided by previous research, most related papers address only a limited range of software quality features affected by cloning. The number of software defects — found in various phases of testing — is the primary metric of quality assurance. However, it is difficult to correlate some NFRs with quantitative measurements. For example, there is no generally accepted metric for software extensibility, which is a key attribute for virtually all software systems. Consequently, most of previous research on software quality does not capture the whole picture of software quality by just studying the correlation of defect rates with bugs that have been found. However, cloning imposes a wide range of software quality issues, including:

1. **Bug propagation**

   Bugs in the cloned code are propagated to all instances of its siblings as clone fragments in the same clone class.

2. **Introducing new bugs**

   Besides bug propagation, cloning could create new bugs even if the clone code is correct in its original location, especially for code snippets with complicated control flow structures, or subtle environmental assumptions.

3. **Program comprehension difficulties**

   In cloned code chunks, naming conventions, comment styles, and context of the cloned code might be incompatible to that of the surrounding code of the cloned code area, which often negatively affect readability; that is, even if no new bugs are introduced, cloning can inhibit comprehension.

4. **License infringement**

   If cloning occurs across systems with incompatible licences, copyright and other legal disputes may arise [4]. Some techniques [15], [43] that are applicable to clone detection has been used to find license infringement cases.

We propose a comprehensive approach of using cloning analysis to assess and then improve software quality. Our approach addresses a wide range of quality problems, including software defects, software maintainability headaches, architectural design flaws, and software license infringement. Furthermore, our proposed framework can also aid in software quality assurance by assessing the cost-effectiveness of addressing each problem under our approach.

To allow focusing on specific software issues, we categorize clone classes along the dimension of categories of other software artifacts to generate the list of clones. There are four kinds of data sources in our study: lexical analysis results, other program analysis techniques (advanced static source code analysis), revision data, and bug repository data. The taxonomy is built based on the four data sources for our analysis.

In Table 4.1, we list all categories, sub-categories, the potential impact on software quality of each sub-category, and the benefit to address each sub-category.

4.4 The "Top-10 List" for Each Category of Cloning-related Issues

Table 4.1 shows a list of categories and subcategories concerning quality issues that may be affected by the presence of clones in software projects. The list is based on existing works of code clone research, our project experience and feedback from our industry collaborators. Researchers, developers, and maintainers can each use this checklist on software projects to help identify cloning-related software quality issues.

Table 4.1 explains the potential risks each subcategory causes in projects as we perceive them to be. The analysis process can be automated for all subcategories. Subcategories are not mutually exclusive; for example, it is possible for a clone class with the same authorship to fit in several categories listed in the table.

Even after the clone detection results are broken down by each subcategory, our method is still burdened by potentially long list within each subcategory. To address this issue, we generate the top-10 list of each subcategory by ranking one or more key factors of the underlying subcategory.

The technique of "top-10 list" has already been used in software risk management by Boehm[12]. Boehm listed the top 10 primary sources of software risks so that software managers can focus on these risks rather than blindly allocate resources on all risks. The strategy behind the "top-10 list" — highlighting the most critical entities to generate a prioritized task list — can also be applied to discover most error-prone software subsystems. Hassan et al.[28] used heuristics to predict the top 10 error-prone subsystems in large software systems so that project managers can allocate testing resources more appropriately.

In this chapter, the "top-10 list" of each subcategory also allows task prioritization by highlighting the most critical instances under each subcategory. For example, the top-10 list of the sub-category "clones spanning different subsystems" can be generated by ranking the number of subsystems each clone class spans. By doing this, the top-10 list of each sub-category includes clone classes that capture the most prominent features of spanning different subsystems thus concentrate the maintenance attention on highly hopeful cases rather than focusing on rest potentially low priority clone instances.

Table 4.1: The taxonomy, with its associated potential impact on software quality of each category and the estimated potential benefits of address each category in a cloning-aware software quality assurance process.

Required artifacts	*Criterion*	*Impact on software quality*	*Automated detection available?*	*Likely benefit if improved*
(1) Based solely on source code and lexical analysis result	1(a). Clones spanning different subsystems	Difficult to find without clone detection tools	✓	Low
	1(b). Whole function clones	Refactoring opportunities	✓	Low
	1(c). Whole file clones	Indicate potential design improvement point	✓	Medium
	1(d). Design level structural clones	Reveal design of the SW system	✓	Low
	1(e). Clones with long code snippet	Difficult to fully understand the cloned area; error prone after code migration	✓	Medium
	1(f). Clones with many members	Share implementation across various software component; Potential cross-cutting concerns	✓	Low

continued on next page

Required artifacts	*Criterion*	*Impact on software quality*	*Automated approach available?*	*Likely benefit if improved*
	1(g). Clones spanning third party code and the main system	Software license breaches	partial	High
(2)Requires program analysis techniques other than lexical analysis	2(a). Collateral clones	requires collateral changes	X	Low
	2(b). Clones in dead code	Spurious code	X	Low
(3) Require revision data	3(a). Clones which all members of one clone are introduced by the same author	Development style of certain developers and potential cloning abuse	✓	Low
	3(b). Clones with inconsistent changes	Potential bugs and error prone	✓	Medium
	3(c). Volatile clones	Cause extra burdens to maintain evolution consistency of clones	✓	Low
(4) Require bug repository data	4(a).clones containing causes of bugs	Do other members of the clone also contain the (same) bug yet remain unfixed?	✓	High

continued on next page

Required artifacts	*Criterion*	*Impact on software quality*	*Automated approach available?*	*Likely benefit if improved*

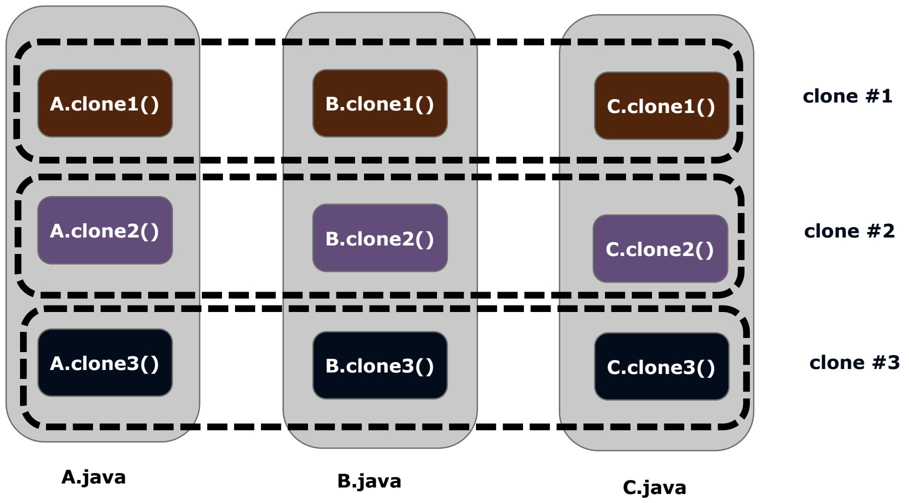

Figure 4.1: An example of a design-level structural clone. Three clone classes that all spanning the same three Java classes, resulting in one design-level structural clone.

In the Table 4.1, we summarize how we divide clones into each subcategory. We list all categories of clones, which is based on the software artifacts required for analysis. We have also include the estimated benefit of resolving the underlying issues in each subcategory. Clearly, some categories represent design flaws of a system while other categories may only indicate features of a system. For example, as shown in Table 4.1, some items such as "clones spanning different components" are intended to reveal design decisions rather than uncover imminent software flaws. By studying clones under these subcategories, software maintainers can collect the design information behind them to aid further decision making process about software maintenance in the relevant software entities. Other items with higher potential harmfulness — such as clones containing causes of bugs — are generally the ones that requires higher attention to quickly resolve them.

The four categories and each subcategory under these are explained in detail below:

1. **Based solely on source code and lexical analysis**

   This category uses only lexical analysis to generate clones of each sub-category.

   (a) **Clones spanning different subsystems**

   A clone class under this subcategory contains clone fragments that spans different subsystems inside the subject software.

(b) **Whole function clones**

A clone class under this subcategory contains at least one clone fragment that covers one or more function boundary in source code.

(c) **Whole file clones**

A clone class under this subcategory contains at least one clone fragment that covers the entire source file.

(d) **Design-level structural clones**

This sub-category describes a pattern of certain combination of multiple clone classes, where these multiple clone classes covers the same set of source files. One example of design-level structural clone is shown in Figure 4.1. To support the cloned code snippet about "clone1" in three files, developers are forced to duplicate other two supporting functions "clone2()" and "clone3()" to all files. These high-level clones may reveal dependencies between clone classes that impose more maintenance headaches than a single clone class.

We transform the problem of automated detection of design-level structural clones into a frequent itemset generation problem[54]. In data mining, frequent itemset generation address the challenge of association among objects. For instance, in online shopping, retailers use the frequent itemset generation to detect "buy-together" relationships in transactions so that recommendations can be made for customers that buy one of the items in a "buy-together" relationship. Likewise, in cloning, a clone class is a "transaction" which contains filenames associated with a clone member and if a fixed pattern of filenames appears multiple times in clone classes, the frequent itemset generator reports all relevant clone classes as a design-level structural clone.

(e) **Clones with long code snippets**

A clone class under this subcategory contains clone fragments with code snippets longer than a threshold (usually in LOC or SLOC).

(f) **Clones with many members**

A clone class under this subcategory contains more than k clone fragments, for some user-specified value of k; we used the value k=3 in our study.

(g) **Clones with third party code**

A clone class under this category spans the subject system and third party code.

2. **Requires program analysis techniques other than lexical analysis**

   (a) **Collateral clones**

   Similar with design-level structural clones, collateral clones describes a combination of clones where a clone fragment of clone class A invokes a function defined in a clone fragment of clone class B. Detecting collateral clones require call graph analysis.

   (b) **Clones in dead code**

   A clone class under this category contains dead code in at least one of its clone fragment. Copying-and-pasting may result in certain control path inaccessible in the new area.

3. **Requires revision data**

   (a) **Clones of which all members of a clone class are introduced by the same author**

   (b) **Clones with inconsistent changes**

   A clone class under this category contains at least two clone fragments which had inconsistent changes. Clones with inconsistent changes often implies clone management risks.

   (c) **Volatile clones**

   Volatile clones refer to clone code chunks that has been changed for an excessive amount of times

4. **Require bug repository data**

   Clones containing causes of bugs

   A clone class under this category contain causes of bugs can be found in at least one snapshots of the subject software system.

The overall process of utilizing cloning analysis for software QA can be decomposed into three phases: 1. select the single subcategory and crosscheck the clone detection report with the required repository; 2. rank the result and select the top 10 instances 3. study the clones in the "top-10 list" for the QA process.

```
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
            // Invocation on ConnectionProxy interface coming in...

            if (method.getName().equals("equals")) {
                // Only considered as equal when proxies are identical.
                return (proxy == args[0] ? Boolean.TRUE : Boolean.FALSE);
            }
            else if (method.getName().equals("hashCode")) {
                // Use hashCode of Connection proxy.
                return new Integer(System.identityHashCode(proxy));
```

Figure 4.2: An example of a cloned source code snippet which spans multiple software components.

4.5 Case Study

4.5.1 Experiment design

After discussing the details of the taxonomy of clones to aid clone management, we conduct an experiment to evaluate it. We are still in the preliminary phase of this study, so that the experiment represents a proof-of-concept, rather than a complete evaluation.

There are a few external limitations in this experiment. The most important one is the lack of measurement of the effectiveness of our approach. As discussed in Section 4.2, few metrics are available to evaluate our tool. Therefore, we can only list the number of offenders of each subcategory found by our framework, and discuss lessons we have learnt by studying instances of offenders.

Due to this limitation, the goal of this study is to explore the feasibility of our approach; instead of proving whether our approach is superior to other QA methods.

The Spring Framework is the subject system of this experiment. In essence, the Spring Framework provides package solution for enterprise application development. It employs many features to allow fast changes and provide easy implementation of dependency injection. The Spring Framework contains multiple modules, among which are server scheduling, code instrumentation, web services and interfaces to mainstream database systems.

We collected 13 snapshots of the Spring Framework (from version 1.0 to 2.5.6) and used *iClones* to detect clones and its evolution information of the 13 snapshots. Then we follow the procedures of our framework to, to perform an exploratory study of the effectiveness of our proposed framework. Despite many desirable features of the Spring Framework as a subject system, this open source system still lacks a few necessary resources for analysis of several subcategories under our framework: the bug repository of Spring is poorly structured thus cannot be crosschecked with the clone detection results. The source distribution of the Spring Framework also does not contain code for any of the third-party libraries that it uses, so we have ignored these for now.Therefore, we do not study clones under this subcategory that requires bug repository or source code from the third party.

The full details of our study on the Spring Framework can be found in the Appendices. In this chapter, we discuss instances that are typical under some categories to demonstrate the likely kind of results that our approaches would yield.

In this subsection, we describe the result of our findings of partially applying the proposed framework on the Spring Enterprise System. In this chapter, we only discuss instances under each subcategory, and the detailed location of our findings is described in Appendix B. Result of each category is shown in a simple template:

Subcategory

Name of the subcategory. Describes the name of the subcategory and its numbering in Table 4.1.

Process of extraction. Describes the procedure of finding clones under each subcategory.

QA-related findings. Describes one instance of the result.

In the framework, we have examined the result of subcategory 1(a),1(d),1(e), and 1(f).

4.5.2 Experiment result

Clones spanning different subsystem (Subcategory 1(a))

Process of extraction:

We chose the last snapshot (version 2.5.6) that was collected from the repository. To find out such clones, we compare the location of every clone fragment in one clone class, if there are more than two directories, we categorize it into this class.

QA-related findings:

In the last version, there are in total 65 such clone class instances in this category. To assist discussion, we show a typical clone class under from category; the source code of this clone class is shown in Figure 4.2, and the location of all clone fragments of this clone class is listed in Table 4.2.

This clone class is a Type I clone (exact match) that span 15 files in 4 components. In the source code shown in Figure 4.2, the duplicated area concerns delegating commands by implementing the same interface by matching the name of the object ("`method`") against two special instances ("`equals`" and "`hashcode`"). What follows the cloned code is the source code that implements functions that are specific to each file.

It has been long believed that clones may indicate high-level similarity or even crosscutting concerns, especially when the clone class spans multiple software components[14]. By studying this code, new programmers can have immediate understanding of the purpose of the interface and two typical cases ("`equals`" and "`hashcode`") that are used spanning the Spring Framework. We can also identify patterns of implementing of the interface and potentially crosscutting concerns for better design.

Design level structural (Subcategory 1(d))

Process of extraction:

As discussed above, design-level structural clones are identified by applying the frequent item-set mining technique, that is, finding the clone classes that all contain clones from the same set of source files.

QA-related findings:

Let us look an example of 4 clone classes (see Figure 4.4), involving only two files: `RmiClientInterceptor.java` and `JndiClinetInterceptor.java`. Both files implement the same interface "`MethodInterceptor`", which describes an aspect-oriented programming (AOP) style method, invoking all necessary methods to support remote connection, with stub of either RMI or JNDI. Both files provide the concrete implementation of the method defined in the interface "`MethodInterceptor`", `Object doInvoke(MethodInvocation invocation, Remote stub)`.

A method named "`prepare()`" initializes the stub so it is ready to process and ensure the correct startup necessary.

All four methods in this example share design between two files, and the method `invoke()` calls the method `getStub()`, which makes the duplication of the method `getStub()` a must if developers are to reuse the method `invoke()` by copying & pasting. As described in the table, the so-called "structural clones" may be harder to maintain than a simple clone.

Clones with long code snippet (Subcategory 1(e))

Process of extraction:

We identify the top-10 clone classes under this category by ranking all clone classes according to the Source Lines of Code (SLOC) of the clone fragments of each clone class.

QA-related findings:

Figure 4.3 shows a member of a clone class whose members are relatively long. The location of the code is listed in Table 4.3. The code snippet of each member of the clone class is 42 LOC long. Within the code snippet, there are seven control flow statements, in which three control flow statements are nested with each other. At first glance, it is worrisome that such a complex piece of code has been copied multiple times: Did the developers fully understand all of the logic? Were there any non-obvious contextual constraints in the original? Answering these questions would require project-specific knowledge that is usually available in a QA process.

Clones with many members (Subcategory 1(f))

Process of extraction:

To identify clones under this category, we select clone classes that contains more than k clone fragments (k=3 in this study).

QA-related findings:

The clone fragment shown in Figure 4.2 is an example under this category. This clone class contains 15 clone fragments. If changes are required within the code, identifying all copies of the code scattering over the system escalates software maintenance effort, not to mention examining and prescribing an appropriate solution for each member of this clone class.

4.6 Summary

In this chapter, we present an approach of using software clone detection results for software quality assurance. Raw clone detection report — which is processed to highlight

Location of clone fragment	Start Line : End Line
src/org/springframework/jdbc/datasource/TransactionAwareDataSourceProxy.java	177:188
src/org/springframework/orm/jpa/AbstractEntityManagerFactoryBean.java	418:427
src/org/springframework/orm/jdo/TransactionAwarePersistenceManagerFactoryProxy.java	146:157
src/org/springframework/orm/jdo/TransactionAwarePersistenceManagerFactoryProxy.java	192:203
src/org/springframework/jca/cci/connection/TransactionAwareConnectionFactoryProxy.java	129:140
src/org/springframework/jdbc/datasource/LazyConnectionDataSourceProxy.java	275:289
src/org/springframework/jca/cci/connection/SingleConnectionFactory.java	1275:1286
src/org/springframework/orm/toplink/AbstractSessionFactory.java	131:142
src/org/springframework/orm/toplink/AbstractSessionFactory.java	179:190
src/org/springframework/jdbc/datasource/SingleConnectionDataSource.java	314:325
src/org/springframework/orm/jpa/JpaTemplate.java	386:397
src/org/springframework/orm/jdo/JdoTemplate.java	588:599
src/org/springframework/jms/connection/SingleConnectionFactory.java	473:482
src/org/springframework/jms/connection/TransactionAwareConnectionFactoryProxy.java	293:304

Table 4.2: One clone class spanning multiple components in the Spring Framework version 2.5.6.

the problem each clone reflects — often shows poor sensitivity as an indicator for many problems in software quality, yet cloning is associated with a specific sets of problems for software. In this study, we propose a two-pronged approach: first, we narrowing down the candidate set for each software quality problem by adopting a taxonomy-based framework for software QA; second, we combining cloning detection results with other related software artifacts to raise the sensitivity of clones to specific quality problems.

Location of clone fragment	Start Line : End Line
src/org/springframework/beans/factory/ support/ConstructorResolver.java	188 : 230
src/org/springframework/beans/factory/ support/ConstructorResolver.java	366 : 408

Table 4.3: A clone class with long code snippets in the Spring Framework

Our framework proposes a taxonomy-based approach to investigate how cloning affects software quality. Extra software repositories, such as software design documents, source code repositories, and bug tracking system, are integrated with clone detection results to pinpointing the software problem reflected by each clone. For every category in the taxonomy, we also list the possible solution to it, and the estimated benefit if we solve it.

To perform a preliminary investigation into the likely effectiveness of our framework as a software QA method, we use the Spring Enterprise Framework and its associated change history as the subject system. We first identify clones under each subcategory and then list the “top-10 offenders” of each sub-category. Several software quality problems, including poor code styles, unnecessary dependency have been reported and discussed in this chapter. Instances under many subcategories have been discussed in this chapter. All these instances provide design knowledge (mostly domain similarity) in the Spring Framework and some of them directly reveal quality problems in the system. With the knowledge provided from our framework, a QA team can integrate out proposed framework into existing QA techniques and further utilize the domain knowledge of their underlying software system to improve the QA process.

```
  if (resolvedValues != null) {
                // Try to resolve arguments for current constructor.
                try {
                    args = createArgumentArray(
                          beanName, mbd, resolvedValues, bw, paramTypes, candidate,
autowiring); }
                catch (UnsatisfiedDependencyException ex) {
                    if (this.beanFactory.logger.isTraceEnabled()) {
                       this.beanFactory.logger.trace(
                             "Ignoring constructor [" + candidate + "] of bean '" +
beanName + "': " + ex);
                    }
                    if (i == candidates.length - 1 && constructorToUse == null) {
                       if (causes != null) {
                          for (Iterator it = causes.iterator(); it.hasNext();) {
                             this.beanFactory.onSuppressedException((Exception)
it.next());
                          }
                       }
                       throw ex;
                    }
                    else {
                       // Swallow and try next constructor.
                       if (causes == null) {
                          causes = new LinkedList();
                       }
                       causes.add(ex);
                       continue;
                    }
                }
             }
             else {
                // Explicit arguments given -> arguments length must match exactly.
                if (paramTypes.length != explicitArgs.length) {
                    continue;
                }
                args = new ArgumentsHolder(explicitArgs);
             }

             int typeDiffWeight = args.getTypeDifferenceWeight(paramTypes);
             // Choose this constructor if it represents the closest match.
```

Figure 4.3: An example of a cloned source code snippet which is long in SLOC.

```
   protected  Object  doInvoke(MethodInvocation  invocation,  Remote  stub)  throws
Throwable {
      if (stub instanceof RmiInvocationHandler) {
         // RMI invoker
         try {
            return doInvoke(invocation, (RmiInvocationHandler) stub);
         }
         catch (RemoteException ex) {
            throw RmiClientInterceptorUtils.convertRmiAccessException(
               invocation.getMethod(), ex, isConnectFailure(ex), getServiceUrl());
         }
         catch (InvocationTargetException ex) {
            Throwable exToThrow = ex.getTargetException();
            RemoteInvocationUtils.fillInClientStackTraceIfPossible(exToThrow);
            throw exToThrow;
         }
         catch (Throwable ex) {
            throw new RemoteInvocationFailureException("Invocation of method [" +
invocation.getMethod() +
                  "] failed in RMI service [" + getServiceUrl() + "]", ex);
         }
      }
      else {
         // traditional RMI stub
         try {
            return      RmiClientInterceptorUtils.invokeRemoteMethod(invocation,
stub);
         }
         catch (InvocationTargetException ex) {
            Throwable targetEx = ex.getTargetException();
            if (targetEx instanceof RemoteException) {
               RemoteException rex = (RemoteException) targetEx;
               throw RmiClientInterceptorUtils.convertRmiAccessException(
                     invocation.getMethod(),     rex,     isConnectFailure(rex),
getServiceUrl());
            }
            else {
               throw targetEx;
            }
         }
      }
   }
```

Figure 4.4: An example of a cloned source code snippet which is involved in a design-level structural clone.

Chapter 5

Conclusions and Future Work

In this thesis, we have presented two studies on cloning to advance the understanding and management of software clones.

Our first study is a longitudinal study of clones in the SCSI driver system of the Linux kernel. After deciding the architecture layer of each source file, we have studied four aspects of clones of the SCSI driver system: the clone ratio of each architecture layer, the evolution of clones, the effectiveness of cloning as a predictor of high-level similarity, and the driving force of clone evolution of this subject system. We have analyzed the absolute size and the proportion of clones over time, found that the clone coverage rate of the concrete driver layer is significantly higher than that of the interface layers, suggesting the use of cloning as a development strategy to reuse design among concrete SCSI drivers. We have also successfully established a model that used the presence of cloning relationship between drivers to predict the functional similarity between them. The cloning-based model had strong predictive power in guessing the underlying hardware bus architecture of the card; the presence of cloning is a superior predictor, compared with a random selection or sharing the same manufacturer. This suggests that cloning may sometimes be an indicator of features similarity within a problem domain. This discovery opens up broader questions of how to use the presence of cloning to predict and understand domain similarity in many dimensions within software systems. By studying clone evolution in the analytical level of single driver, we conclude that inserting and deprecating low-level drivers, rather than other ways of refactoring, is the major cause of clone evolution.

The second study of this thesis presents a clone-based framework for software quality assurance. Duplication of code can lead to a specific set of problems in software quality yet the sensitivity of clones in diagnosing most of the quality problems is prohibitively low, according to study results[50] and our experience. To solve this problem, we first establish taxonomy of clones. The taxonomy is established based upon the additional software artifact that is combined in the analysis; they are: syntactical analysis results, bug repositories, version control systems, and software documentation. Each category may have several subcategories which maps to one specific type of software quality issue, allowing users to tailor the clone-based QA framework. Second, by crosschecking clone detection reports with other software artifacts, our proposed framework filters out "uninteresting" clones for one category so that the sensitivity of clones under every category is raised. Third, we use the "top-10 list" technique to make a short list of hopeful candidates of problematic clones. Each category reflects software problems with different degree of severity, from the most severe (clones that are likely bugs) to benign ones(clones that only suggest similar design). We performed a preliminary study of our framework on the Spring Enterprise Framework. Our study result is encouraging: each sub-category contains expected software quality issues.

5.1 Future Work

5.1.1 Application of Cloning for software comprehension

Our study in Section 3.5 shows that among Linux SCSI drivers, the presence of cloning between two drivers is a strong indicator of compatible bus type architectures in the respective SCSI cards. We feel that this opens up several research questions about the use of cloning as an indicator of similarity of aspects of the underlying problem domain, and of the development processes employed. Does cloning indicate that there is an emergent cross-cutting concern that is highly similar in its implementation across products? Are some "aspects" of the problem domain — such as hardware or user-visible features — more likely to involve cloning than others? If so, why? Are these aspects maintained in parallel over time, or are they prone to inconsistent maintenance? Further study across other systems and problem domains can help to shed light on these questions.

Our taxonomy presented in Chapter 4 is a work in progress. Our ultimate goal is to provide a means of using cloning analysis results as part of an overall QA process. However, our evaluation of the taxonomy and its use is partial and preliminary: only some of the criteria have been implemented, and our study on the Spring framework as a target system represents a proof-of-concept rather than a detailed empirical finding. As future work, we intend to fully implement the taxonomy, and to study the results of applying it on several target systems. In particular, we seek to understand which taxonomy elements provide the best diagnostic value in evaluating software quality and development risk.

APPENDIX A
Source Files Categorization within The Three-Level Architecture of the SCSI Subsystem

Appendix A demonstrate the full result of categorization of the source files according to its architectural level in the SCSI subsystem (version 2.6.32.15). The categorization is based upon associated artifacts of the source files, including comments, documentation, static analysis results, configuration setups, and build recipes.

Table 1: The categorization of source files according to the architecture level of the SCSI subsystem.

The Categorization of source files according to the architecture level of the SCSI subsystem		
Upper Level		
osst.c	osst.h	sd.c
sd.h	sg.c	sr.c
sr.h	st.c	st.h

Mid Level		
constants.c	device_handler/scsi_dh.c	device_handler/scsi_dh_alua.c
device_handler/scsi_dh_emc.c	device_handler/scsi_dh_hp_sw.c	device_handler/scsi_dh_rdac.c
hosts.c	iscsi_tcp.c	iscsi_tcp.h
libiscsi.c	libiscsi_tcp.c	libsas/sas_ata.c
libsas/sas_discover.c	libsas/sas_dump.c	libsas/sas_dump.h
libsas/sas_event.c	libsas/sas_expander.c	libsas/sas_host_smp.c
libsas/sas_init.c	libsas/sas_internal.h	libsas/sas_phy.c
libsas/sas_port.c	libsas/sas_scsi_host.c	libsas/sas_task.c
scsi.c	scsi.h	scsicam.c
scsi_debug.c	scsi_devinfo.c	scsi_error.c
scsi_ioctl.c	scsi_lib.c	scsi_lib_dma.c
scsi_logging.h	scsi_module.c	scsi_netlink.c
scsi_priv.h	scsi_proc.c	scsi_sas_internal.h
scsi_scan.c	scsi_sysctl.c	scsi_sysfs.c
scsi_tgt_if.c	scsi_tgt_lib.c	scsi_tgt_priv.h

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem		
Lower Level		
scsi_transport_api.h	scsi_transport_fc.c	scsi_transport_fc_internal.h
scsi_transport_iscsi.c	scsi_transport_sas.c	scsi_transport_spi.c
scsi_transport_srp.c	scsi_transport_srp_internal.h	scsi_typedefs.h
scsi_wait_scan.c	sd_dif.c	ses.c
sr_ioctl.c	sr_vendor.c	st_options.h
3w-9xxx.c	3w-9xxx.h	3w-xxxx.c
3w-xxxx.h	53c700.c	53c700.h
a100u2w.c	a100u2w.h	a2091.c
a2091.h	a3000.c	a3000.h
a4000t.c	aacraid/aachba.c	aacraid/aacraid.h
aacraid/commctrl.c	aacraid/comminit.c	aacraid/commsup.c
aacraid/dpcsup.c	aacraid/linit.c	aacraid/nark.c
aacraid/rkt.c	aacraid/rx.c	aacraid/sa.c
advansys.c	aha152x.c	aha152x.h
aha1542.c	aha1542.h	aha1740.c
aha1740.h	aic7xxx/aic7770.c	aic7xxx/aic7770_osm.c
aic7xxx/aic79xx.h	aic7xxx/aic79xx_core.c	aic7xxx/aic79xx_inline.h
aic7xxx/aic79xx_osm.c	aic7xxx/aic79xx_osm.h	aic7xxx/aic79xx_osm_pci.c
aic7xxx/aic79xx_pci.c	aic7xxx/aic79xx_pci.h	aic7xxx/aic79xx_proc.c
aic7xxx/aic7xxx.h	aic7xxx/aic7xxx_93cx6.c	aic7xxx/aic7xxx_93cx6.h
aic7xxx/aic7xxx_core.c	aic7xxx/aic7xxx_inline.h	aic7xxx/aic7xxx_osm.c
aic7xxx/aic7xxx_osm.h	aic7xxx/aic7xxx_osm_pci.c	aic7xxx/aic7xxx_pci.c
aic7xxx/aic7xxx_pci.h	aic7xxx/aic7xxx_proc.c	aic7xxx/aicasm/aicasm.c
aic7xxx/aicasm/aicasm.h	aic7xxx/aicasm/aicasm_insformat.h	aic7xxx/aicasm/aicasm_symbol.c
aic7xxx/aicasm/aicasm_symbol.h	aic7xxx/aiclib.c	aic7xxx/aiclib.h
aic7xxx/cam.h	aic7xxx/queue.h	aic7xxx/scsi_iu.h
aic7xxx/scsi_message.h	aic7xxx_old/aic7xxx.h	aic7xxx_old/aic7xxx_proc.c

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem

Lower Level		
aic7xxx_old/aic7xxx_reg.h	aic7xxx_old/aic7xxx_seq.c	aic7xxx_old/scsi_message.h
aic7xxx_old/sequencer.h	aic7xxx_old.c	aic94xx/aic94xx.h
aic94xx/aic94xx_dev.c	aic94xx/aic94xx_dump.c	aic94xx/aic94xx_dump.h
aic94xx/aic94xx_hwi.c	aic94xx/aic94xx_hwi.h	aic94xx/aic94xx_init.c
aic94xx/aic94xx_reg.c	aic94xx/aic94xx_reg.h	aic94xx/aic94xx_reg_def.h
aic94xx/aic94xx_sas.h	aic94xx/aic94xx_scb.c	aic94xx/aic94xx_sds.c
aic94xx/aic94xx_sds.h	aic94xx/aic94xx_seq.c	aic94xx/aic94xx_seq.h
aic94xx/aic94xx_task.c	aic94xx/aic94xx_tmf.c	arcmsr/arcmsr.h
arcmsr/arcmsr_attr.c	arcmsr/arcmsr_hba.c	arm/acornscsi.c
arm/acornscsi.h	arm/arxescsi.c	arm/cumana_1.c
arm/cumana_2.c	arm/eesox.c	arm/fas216.c
arm/fas216.h	arm/msgqueue.c	arm/msgqueue.h
arm/oak.c	arm/powertec.c	arm/queue.c
arm/queue.h	arm/scsi.h	atari_NCR5380.c
atari_scsi.c	atari_scsi.h	atp870u.c
atp870u.h	be2iscsi/be.h	be2iscsi/be_cmds.c
be2iscsi/be_cmds.h	be2iscsi/be_iscsi.c	be2iscsi/be_iscsi.h
be2iscsi/be_main.c	be2iscsi/be_main.h	be2iscsi/be_mgmt.c
be2iscsi/be_mgmt.h	bfa/bfad.c	bfa/bfad_attr.c
bfa/bfad_attr.h	bfa/bfad_drv.h	bfa/bfad_fwimg.c
bfa/bfad_im.c	bfa/bfad_im.h	bfa/bfad_im_compat.h
bfa/bfad_intr.c	bfa/bfad_ipfc.h	bfa/bfad_os.c
bfa/bfad_tm.h	bfa/bfad_trcmod.h	bfa/bfa_callback_priv.h
bfa/bfa_cb_ioim_macros.h	bfa/bfa_cee.c	bfa/bfa_core.c
bfa/bfa_csdebug.c	bfa/bfa_fcpim.c	bfa/bfa_fcpim_priv.h
bfa/bfa_fcport.c	bfa/bfa_fcs.c	bfa/bfa_fcs_lport.c
bfa/bfa_fcs_port.c	bfa/bfa_fcs_uf.c	bfa/bfa_fcxp.c

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem

Lower Level		
bfa/bfa_fcxp_priv.h	bfa/bfa_fwimg_priv.h	bfa/bfa_hw_cb.c
bfa/bfa_hw_ct.c	bfa/bfa_intr.c	bfa/bfa_intr_priv.h
bfa/bfa_ioc.c	bfa/bfa_ioc.h	bfa/bfa_iocfc.c
bfa/bfa_iocfc.h	bfa/bfa_iocfc_q.c	bfa/bfa_ioim.c
bfa/bfa_itnim.c	bfa/bfa_log.c	bfa/bfa_log_module.c
bfa/bfa_lps.c	bfa/bfa_lps_priv.h	bfa/bfa_module.c
bfa/bfa_modules_priv.h	bfa/bfa_os_inc.h	bfa/bfa_port.c
bfa/bfa_port_priv.h	bfa/bfa_priv.h	bfa/bfa_rport.c
bfa/bfa_rport_priv.h	bfa/bfa_sgpg.c	bfa/bfa_sgpg_priv.h
bfa/bfa_sm.c	bfa/bfa_timer.c	bfa/bfa_trcmod_priv.h
bfa/bfa_tskim.c	bfa/bfa_uf.c	bfa/bfa_uf_priv.h
bfa/fab.c	bfa/fabric.c	bfa/fcbuild.c
bfa/fcbuild.h	bfa/fcpim.c	bfa/fcptm.c
bfa/fcs.h	bfa/fcs_auth.h	bfa/fcs_fabric.h
bfa/fcs_fcpim.h	bfa/fcs_fcptm.h	bfa/fcs_fcxp.h
bfa/fcs_lport.h	bfa/fcs_ms.h	bfa/fcs_port.h
bfa/fcs_rport.h	bfa/fcs_trcmod.h	bfa/fcs_uf.h
bfa/fcs_vport.h	bfa/fdmi.c	bfa/include/aen/bfa_aen.h
bfa/include/aen/bfa_aen_adapter.h	bfa/include/aen/bfa_aen_audit.h	bfa/include/aen/bfa_aen_ethport.h
bfa/include/aen/bfa_aen_ioc.h	bfa/include/aen/bfa_aen_itnim.h	bfa/include/aen/bfa_aen_lport.h
bfa/include/aen/bfa_aen_port.h	bfa/include/aen/bfa_aen_rport.h	bfa/include/bfa.h
bfa/include/bfa_fcpim.h	bfa/include/bfa_fcptm.h	bfa/include/bfa_svc.h
bfa/include/bfa_timer.h	bfa/include/bfi/bfi.h	bfa/include/bfi/bfi_boot.h
bfa/include/bfi/bfi_cbreg.h	bfa/include/bfi/bfi_cee.h	bfa/include/bfi/bfi_ctreg.h
bfa/include/bfi/bfi_fabric.h	bfa/include/bfi/bfi_fcpim.h	bfa/include/bfi/bfi_fcxp.h
bfa/include/bfi/bfi_ioc.h	bfa/include/bfi/bfi_iocfc.h	bfa/include/bfi/bfi_lport.h
bfa/include/bfi/bfi_lps.h	bfa/include/bfi/bfi_port.h	bfa/include/bfi/bfi_pport.h

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem

Lower Level		
bfa/include/bfi/bfi_rport.h	bfa/include/bfi/bfi_uf.h	bfa/include/cna/bfa_cna_trcmod.h
bfa/include/cna/cee/bfa_cee.h	bfa/include/cna/port/bfa_port.h	bfa/include/cna/pstats/ethport_defs.h
bfa/include/cna/pstats/phyport_defs.h	bfa/include/cs/bfa_checksum.h	bfa/include/cs/bfa_debug.h
bfa/include/cs/bfa_log.h	bfa/include/cs/bfa_perf.h	bfa/include/cs/bfa_plog.h
bfa/include/cs/bfa_q.h	bfa/include/cs/bfa_sm.h	bfa/include/cs/bfa_trc.h
bfa/include/cs/bfa_wc.h	bfa/include/defs/bfa_defs_adapter.h	bfa/include/defs/bfa_defs_aen.h
bfa/include/defs/bfa_defs_audit.h	bfa/include/defs/bfa_defs_auth.h	bfa/include/defs/bfa_defs_boot.h
bfa/include/defs/bfa_defs_cee.h	bfa/include/defs/bfa_defs_driver.h	bfa/include/defs/bfa_defs_ethport.h
bfa/include/defs/bfa_defs_fcpim.h	bfa/include/defs/bfa_defs_im_common.h	bfa/include/defs/bfa_defs_im_team.h
bfa/include/defs/bfa_defs_ioc.h	bfa/include/defs/bfa_defs_iocfc.h	bfa/include/defs/bfa_defs_ipfc.h
bfa/include/defs/bfa_defs_itnim.h	bfa/include/defs/bfa_defs_led.h	bfa/include/defs/bfa_defs_lport.h
bfa/include/defs/bfa_defs_mfg.h	bfa/include/defs/bfa_defs_pci.h	bfa/include/defs/bfa_defs_pm.h
bfa/include/defs/bfa_defs_pom.h	bfa/include/defs/bfa_defs_port.h	bfa/include/defs/bfa_defs_pport.h
bfa/include/defs/bfa_defs_qos.h	bfa/include/defs/bfa_defs_rport.h	bfa/include/defs/bfa_defs_status.h
bfa/include/defs/bfa_defs_tin.h	bfa/include/defs/bfa_defs_tsensor.h	bfa/include/defs/bfa_defs_types.h
bfa/include/defs/bfa_defs_version.h	bfa/include/defs/bfa_defs_vf.h	bfa/include/defs/bfa_defs_vport.h
bfa/include/fcb/bfa_fcb.h	bfa/include/fcb/bfa_fcb_fcpim.h	bfa/include/fcb/bfa_fcb_port.h
bfa/include/fcb/bfa_fcb_rport.h	bfa/include/fcb/bfa_fcb_vf.h	bfa/include/fcb/bfa_fcb_vport.h
bfa/include/fcs/bfa_fcs.h	bfa/include/fcs/bfa_fcs_auth.h	bfa/include/fcs/bfa_fcs_fabric.h
bfa/include/fcs/bfa_fcs_fcpim.h	bfa/include/fcs/bfa_fcs_fdmi.h	bfa/include/fcs/bfa_fcs_lport.h
bfa/include/fcs/bfa_fcs_rport.h	bfa/include/fcs/bfa_fcs_vport.h	bfa/include/log/bfa_log_fcs.h
bfa/include/log/bfa_log_hal.h	bfa/include/log/bfa_log_linux.h	bfa/include/log/bfa_log_wdrv.h
bfa/include/protocol/ct.h	bfa/include/protocol/fc.h	bfa/include/protocol/fcp.h
bfa/include/protocol/fc_sp.h	bfa/include/protocol/fdmi.h	bfa/include/protocol/pcifw.h
bfa/include/protocol/scsi.h	bfa/include/protocol/types.h	bfa/loop.c
bfa/lport_api.c	bfa/lport_priv.h	bfa/ms.c
bfa/n2n.c	bfa/ns.c	bfa/plog.c

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem

Lower Level		
bfa/rport_c	bfa/rport_api.c	bfa/rport_ftrs.c
bfa/scn.c	bfa/vfapi.c	bfa/vport.c
bnx2i/57xx_iscsi_constants.h	bnx2i/57xx_iscsi_hsi.h	bnx2i/bnx2i.h
bnx2i/bnx2i_hwi.c	bnx2i/bnx2i_init.c	bnx2i/bnx2i_iscsi.c
bnx2i/bnx2i_sysfs.c	BusLogic.c	BusLogic.h
bvme6000_scsi.c	ch.c	cxgb3i/cxgb3i.h
cxgb3i/cxgb3i_ddp.c	cxgb3i/cxgb3i_ddp.h	cxgb3i/cxgb3i_init.c
cxgb3i/cxgb3i_iscsi.c	cxgb3i/cxgb3i_offload.c	cxgb3i/cxgb3i_offload.h
cxgb3i/cxgb3i_pdu.c	cxgb3i/cxgb3i_pdu.h	dc395x.c
dc395x.h	dmx3191d.c	dpt/dpti_i2o.h
dpt/dpti_ioctl.h	dpt/dptsig.h	dpt/osd_defs.h
dpt/osd_util.h	dpt/sys_info.h	dpti.h
dpt_i2o.c	dtc.c	dtc.h
eata.c	eata_generic.h	eata_pio.c
eata_pio.h	esp_scsi.c	esp_scsi.h
fcoe/fcoe.c	fcoe/fcoe.h	fcoe/libfcoe.c
fdomain.c	fdomain.h	fd_mcs.c
FlashPoint.c	fnic/cq_desc.h	fnic/cq_enet_desc.h
fnic/cq_exch_desc.h	fnic/fcpio.h	fnic/fnic.h
fnic/fnic_attrs.c	fnic/fnic_fcs.c	fnic/fnic_io.h
fnic/fnic_isr.c	fnic/fnic_main.c	fnic/fnic_res.c
fnic/fnic_res.h	fnic/fnic_scsi.c	fnic/rq_enet_desc.h
fnic/vnic_cq.c	fnic/vnic_cq.h	fnic/vnic_cq_copy.h
fnic/vnic_dev.c	fnic/vnic_dev.h	fnic/vnic_devcmd.h
fnic/vnic_intr.c	fnic/vnic_intr.h	fnic/vnic_nic.h
fnic/vnic_resource.h	fnic/vnic_rq.c	fnic/vnic_rq.h
fnic/vnic_scsi.h	fnic/vnic_stats.h	fnic/vnic_wq.c

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem

Lower Level		
fnic/vnic_wq.h	fnic/vnic_wq_copy.c	fnic/vnic_wq_copy.h
fnic/wq_enet_desc.h	gdth.c	gdth.h
gdth_ioctl.h	gdth_proc.c	gdth_proc.h
gvp11.c	gvp11.h	g_NCR5380.c
g_NCR5380.h	g_NCR5380_mmio.c	hptiop.c
hptiop.h	ibmmca.c	ibmvscsi/ibmvfc.c
ibmvscsi/ibmvfc.h	ibmvscsi/ibmvscsi.c	ibmvscsi/ibmvscsi.h
ibmvscsi/ibmvstgt.c	ibmvscsi/iseries_vscsi.c	ibmvscsi/rpa_vscsi.c
ibmvscsi/viosrp.h	imm.c	imm.h
in2000.c	in2000.h	initio.c
initio.h	ipr.c	ipr.h
ips.c	ips.h	jazz_esp.c
lasi700.c	libfc/fc_disc.c	libfc/fc_elsct.c
libfc/fc_exch.c	libfc/fc_fcp.c	libfc/fc_frame.c
libfc/fc_lport.c	libfc/fc_rport.c	libsrp.c
lpfc/lpfc.h	lpfc/lpfc_attr.c	lpfc/lpfc_bsg.c
lpfc/lpfc_compat.h	lpfc/lpfc_crtn.h	lpfc/lpfc_ct.c
lpfc/lpfc_debugfs.c	lpfc/lpfc_debugfs.h	lpfc/lpfc_disc.h
lpfc/lpfc_els.c	lpfc/lpfc_hbadisc.c	lpfc/lpfc_hw.h
lpfc/lpfc_hw4.h	lpfc/lpfc_init.c	lpfc/lpfc_logmsg.h
lpfc/lpfc_mbox.c	lpfc/lpfc_mem.c	lpfc/lpfc_nl.h
lpfc/lpfc_nportdisc.c	lpfc/lpfc_scsi.c	lpfc/lpfc_scsi.h
lpfc/lpfc_sli.c	lpfc/lpfc_sli.h	lpfc/lpfc_sli4.h
lpfc/lpfc_version.h	lpfc/lpfc_vport.c	lpfc/lpfc_vport.h
mac53c94.c	mac53c94.h	mac_esp.c
mac_scsi.c	mac_scsi.h	megaraid/mbox_defs.h
megaraid/megaraid_ioctl.h	megaraid/megaraid_mbox.c	megaraid/megaraid_mbox.h

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem

Lower Level		
megaraid/megaraid_mm.c	megaraid/megaraid_mm.h	megaraid/megaraid_sas.c
megaraid/megaraid_sas.h	megaraid/mega_common.h	megaraid.c
megaraid.h	mesh.c	mesh.h
mpt2sas/mpi/mpi2.h	mpt2sas/mpi/mpi2_cnfg.h	mpt2sas/mpi/mpi2_init.h
mpt2sas/mpi/mpi2_ioc.h	mpt2sas/mpi/mpi2_raid.h	mpt2sas/mpi/mpi2_sas.h
mpt2sas/mpi/mpi2_tool.h	mpt2sas/mpi/mpi2_type.h	mpt2sas/mpt2sas_base.c
mpt2sas/mpt2sas_base.h	mpt2sas/mpt2sas_config.c	mpt2sas/mpt2sas_ctl.c
mpt2sas/mpt2sas_ctl.h	mpt2sas/mpt2sas_debug.h	mpt2sas/mpt2sas_scsih.c
mpt2sas/mpt2sas_transport.c	mvme147.c	mvme147.h
mvme16x_scsi.c	mvsas/mv_64xx.c	mvsas/mv_64xx.h
mvsas/mv_94xx.c	mvsas/mv_94xx.h	mvsas/mv_chips.h
mvsas/mv_defs.h	mvsas/mv_init.c	mvsas/mv_sas.c
mvsas/mv_sas.h	NCR5380.c	NCR5380.h
NCR53c406a.c	ncr53c8xx.c	ncr53c8xx.h
NCR_D700.c	NCR_D700.h	NCR_Q720.c
NCR_Q720.h	nsp32.c	nsp32.h
nsp32_debug.c	nsp32_io.h	osd/osd_debug.h
osd/osd_initiator.c	osd/osd_uld.c	osst_detect.h
osst_options.h	pas16.c	pas16.h
pcmcia/aha152x_core.c	pcmcia/aha152x_stub.c	pcmcia/fdomain_core.c
pcmcia/fdomain_stub.c	pcmcia/nsp_cs.c	pcmcia/nsp_cs.h
pcmcia/nsp_debug.c	pcmcia/nsp_io.h	pcmcia/nsp_message.c
pcmcia/qlogic_stub.c	pcmcia/sym53c500_cs.c	pmcraid.c
pmcraid.h	ppa.c	ppa.h
ps3rom.c	qla1280.c	qla1280.h
qla2xxx/qla_attr.c	qla2xxx/qla_dbg.c	qla2xxx/qla_dbg.h
qla2xxx/qla_def.h	qla2xxx/qla_devtbl.h	qla2xxx/qla_dfs.c

continued on next page

continued from previous page

The Categorization of source files according to the architecture level of the SCSI subsystem		
Lower Level		
qla2xxx/qla_fw.h	qla2xxx/qla_gbl.h	qla2xxx/qla_gs.c
qla2xxx/qla_init.c	qla2xxx/qla_inline.h	qla2xxx/qla_iocb.c
qla2xxx/qla_isr.c	qla2xxx/qla_mbx.c	qla2xxx/qla_mid.c
qla2xxx/qla_os.c	qla2xxx/qla_settings.h	qla2xxx/qla_sup.c
qla2xxx/qla_version.h	qla4xxx/ql4_dbg.c	qla4xxx/ql4_dbg.h
qla4xxx/ql4_def.h	qla4xxx/ql4_fw.h	qla4xxx/ql4_glbl.h
qla4xxx/ql4_init.c	qla4xxx/ql4_inline.h	qla4xxx/ql4_iocb.c
qla4xxx/ql4_isr.c	qla4xxx/ql4_mbx.c	qla4xxx/ql4_nvram.c
qla4xxx/ql4_nvram.h	qla4xxx/ql4_os.c	qla4xxx/ql4_version.h
qlogicfas.c	qlogicfas408.c	qlogicfas408.h
qlogicpti.c	qlogicpti.h	raid_class.c
sgiwd93.c	sim710.c	sni_53c710.c
stex.c	sun3x_esp.c	sun3_NCR5380.c
sun3_scsi.c	sun3_scsi.h	sun3_scsi_vme.c
sun_esp.c	sym53c416.c	sym53c416.h
sym53c8xx_2/sym53c8xx.h	sym53c8xx_2/sym_defs.h	sym53c8xx_2/sym_fw.c
sym53c8xx_2/sym_fw.h	sym53c8xx_2/sym_fw1.h	sym53c8xx_2/sym_fw2.h
sym53c8xx_2/sym_glue.c	sym53c8xx_2/sym_glue.h	sym53c8xx_2/sym_hipd.c
sym53c8xx_2/sym_hipd.h	sym53c8xx_2/sym_malloc.c	sym53c8xx_2/sym_misc.h
sym53c8xx_2/sym_nvram.c	sym53c8xx_2/sym_nvram.h	t128.c
t128.h	tmscsim.c	tmscsim.h
u14-34f.c	ultrastor.c	ultrastor.h
wd33c93.c	wd33c93.h	wd7000.c
zalon.c		

APPENDIX B
Results of Clones under Each "Interestingness" Criterion

Tables of software quality issue under each subcategory in the clone-aware framework discussed in Chapter 4. The intention of this appendix is to show the effectiveness of our framework while avoid disrupting reading the main content. Information listed in this appendix also serves to encourage open discussion of methods of our framework. All locations listed in the following table are based on the source code of the Spring Framework version 2.5.6 provided by SpringSource.org[3].

Table 2: The top-10 list of clone classes that span multiple software components.

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
1	Type I Clone	src/org/springframework/jdbc/ datasource/TransactionAware-DataSourceProxy.java	177:188
		src/org/springframework /orm/jpa/AbstractEntity-ManagerFactoryBean.java	418:427

continued on next page

continued from previous page

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
		src/org/springframework /orm/jdo/TransactionAware-PersistenceManagerFactoryProxy.java	146:157
		src/org/springframework /orm/jdo/Transaction-AwarePersistence-ManagerFactoryProxy.java	192:203
		src/org/springframework /jca/cci/connection/Transaction-AwareConnection-FactoryProxy.java	129:140
		src/org/springframework /jdbc/datasource/LazyConnection-DataSourceProxy.java	275:289
		src/org/springframework /jca/cci/connection/Single-ConnectionFactory.java	1275:1286
		src/org/springframework /orm/toplink/Abstract-SessionFactory.java	131:142
		src/org/springframework /orm/toplink/Abstract-SessionFactory.java	179:190
		src/org/springframework /jdbc/datasource/Single-ConnectionDataSource.java	314:325
		src/org/springframework/orm/jpa/ JpaTemplate.java	386:397
		src/org/springframework/orm/jdo/ JdoTemplate.java	588:599
		src/org/springframework /jms/connection/Single-ConnectionFactory.java	473:482

continued on next page

continued from previous page

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
		src/org/springframework /jms/connection/Transaction-AwareConnectionFactoryProxy.java	293:304
2	Type I Clone	src/org/springframework /orm/jpa/support/OpenEntity-ManagerInViewInterceptor.java	71:78
		src/org/springframework /orm/hibernate3/support/Open-SessionInViewInterceptor.java	144:151
		src/org/springframework/ orm/jdo/support/OpenPersistence-ManagerInViewInterceptor.java	93:100
3	Type I Clone	src/org/springframework/ orm/hibernate3/support/Blob-SerializableType.java	148:162
		src/org/springframework/ orm/ibatis/support/Blob-SerializableTypeHandler.java	67:81
4	Type II Clone	src/org/springframework/orm/ jdo/Jdo-TransactionManager.java	314:339
		src/org/springframework/orm/ jpa/JpaTransactionManager.java	330:355
5	Type I Clone	src/org/springframework/ejb/ access/ SimpleRemoteSlsbInvokerIn-terceptor.java	101:108
		src/org/springframework/remoting/ rmi/RmiClientInterceptor.java	362:369
6	Type I Clone	src/org/springframework/aop/config/ SpringConfigured-BeanDefinitionParser.java	53:58

continued on next page

continued from previous page

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
		src/org/springframework/context/config/SpringConfigured-BeanDefinitionParser.java	46:53

Table 3: The top-10 list of clone classes with many members.

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
		src/org/springframework/ jdbc/datasource/Transaction-AwareDataSourceProxy.java	177:188
		src/org/springframework/ orm/jpa/AbstractEntity-ManagerFactoryBean.java	418:427
		src/org/springframework/ orm/jdo/TransactionAwarePersis-tenceManagerFactoryProxy.java	146:157
		src/org/springframework/ orm/jdo/TransactionAware-PersistenceManager-FactoryProxy.java	192:203
		src/org/springframework/ jca/cci/connection/Transaction-AwareConnectionFactoryProxy.java	129:140
		src/org/springframework/ jdbc/datasource/Lazy-ConnectionDataSourceProxy.java	275:289
		src/org/springframework/ jca/cci/connection/Single-ConnectionFactory.java	227:236
		src/org/springframework/orm/ hibernate3/Hibernate-Template.java	1275:1286

continued on next page

continued from previous page

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
1	Type I Clone	src/org/springframework/ orm/toplink/Abstract-SessionFactory.java	131:142
		src/org/springframework/ orm/toplink/Abstract-SessionFactory.java	179:190
		src/org/springframework/ jdbc/datasource/Single-ConnectionDataSource.java	314:325
		src/org/springframework/orm/jpa/ JpaTemplate.java	386:397
		src/org/springframework/orm/jdo/ JdoTemplate.java	588:599
		src/org/springframework/ jms/connection/Single-ConnectionFactory.java	473:482
		src/org/springframework/ jms/connection/Transaction-AwareConnectionFactoryProxy.java	293:304
2	Type III Clone	src/org/springframework/orm/ hibernate3/HibernateTemplate.java	921:933
		src/org/springframework/ orm/hibernate3/Hibernate-Template.java	947:959
		src/org/springframework/ orm/hibernate3/Hibernate-Template.java	988 : 1000
		src/org/springframework/ orm/hibernate3/Hibernate-Template.java	1015 :1027

continued on next page

continued from previous page

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
3	Type III Clone	src/org/springframework/ context/support/Reloadable-ResourceBundleMessageSource.java	273:277
		src/org/springframework/ context/support/Reloadable-ResourceBundleMessageSource.java	301:305
		src/org/springframework/ context/support/Reloadable-ResourceBundleMessageSource.java	333:337
4	Type I Clone	src/org/springframework/util/Object-Utils.java	560:576
		src/org/springframework/util/Object-Utils.java	591:608
		src/org/springframework/util/Object-Utils.java	623:639
		src/org/springframework/util/Object-Utils.java	654:670
		src/org/springframework/util/Object-Utils.java	685 : 702
		src/org/springframework/util/Object-Utils.java	717 : 734
		src/org/springframework/util/Object-Utils.java	749 : 765
		src/org/springframework/util/Object-Utils.java	780:796
		src/org/springframework/util/Object-Utils.java	811:827
		src/org/springframework/beans/ factory/annotation/ AutowiredAnnotationBean-PostProcessor.java	248:256

continued on next page

continued from previous page

Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
5	Type II Clone	src/org/springframework/ orm/jpa/support/Persistence-AnnotationBeanPostProcessor.java	316:324
		src/org/springframework/ context/annotation/Common-AnnotationBeanPostProcessor.java	297:305
6	Type I Clone	src/org/springframework/ jms/listener/serversession/ ServerSessionMessage-ListenerContainer102.java	83:90
		src/org/springframework/ jm-s/core/JmsTemplate102.java	177:184
		src/org/springframework/ jms/listener/SimpleMessage-ListenerContainer102.java	59:66
		src/org/springframework/ jms/listener/DefaultMessage-ListenerContainer102.java	77:84
7	Type I Clone	src/org/springframework/ jdbc/support/incrementer/SqlServer-MaxValueIncrementer.java	76:91
		src/org/springframework/ jd-bc/support/incrementer/ Sybase-MaxValueIncrementer.java	76 : 91
		src/org/springframework/ jdbc/support/incrementer/HsqlMax-ValueIncrementer.java	87 : 102
		src/org/springframework/ jd-bc/support/incrementer/ Derby-MaxValueIncrementer.java	129 : 144

continued on next page

continued from previous page			
Clone Class	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*

Table 4: The top-10 list of clones with long code snippets.

Clone Class	*SLOC*	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
1	42	Type II Clone	src/org/springframework/ bean-s/factory/support/ ConstructorRe-solver.java	188 : 230
			src/org/springframework/ bean-s/factory/support/ ConstructorRe-solver.java	366 : 408
2	41	Type III Clone	src/org/springframework/ jdbc/support/incrementer/SqlServer-MaxValueIncrementer.java	76 : 117
			src/org/springframework/ jdbc/support/incrementer/Sybase-MaxValueIncrementer.java	76 : 117
			src/org/springframework/ jdbc/support/incrementer/HsqlMax-ValueIncrementer.java	87 : 128
			src/org/springframework /jd-bc/support/incrementer/ Derby-MaxValueIncrementer.java	129 : 170
3	41	Type III Clone	src/org/springframework/ jdbc/support/incrementer/HsqlMax-ValueIncrementer.java	87 : 128
			src/org/springframework/ jd-bc/support/incrementer/ Derby-MaxValueIncrementer.java	129 : 170
			src/org/springframework/ context/support/Abstract-ApplicationContext.java	506 : 543

continued on next page

continued from previous page

Clone Class	*SLOC*	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
4	37	Type II Clone	src/org/springframework/ context/support/Abstract-ApplicationContext.java	569 : 606
5	32	Type III Clone	src/org/springframework/ orm/hibernate3/Hibernate-Template.java	1278 : 1310
			src/org/springframework/ jd-bc/core/JdbcTemplate.java	1277 : 1306
			src/org/springframework/ or-m/jdo/JdoTemplate.java	591 : 619
6	28	Type II Clone	src/org/springframework/ core/NestedChecked-Exception.java	105 : 133
			src/org/springframework/ core/NestedRuntime-Exception.java	106 : 134
7	29	Type II Clone	src/org/springframework/ web/jsf/el/WebApplication-ContextFacesELResolver.java	70 : 99
			src/org/springframework/ web/jsf/el/WebApplication-ContextFacesELResolver.java	106 : 135
8	27	Type I Clone	src/org/springframework/ schedul-ing/backportconcurrent/ Thread-PoolTaskExecutor.java	71 : 98
			src/org/springframework/ scheduling/concurrent/ Thread-PoolTaskExecutor.java	69 : 96
			src/org/springframework/ orm/hibernate3/Hibernate-Template.java	1275 : 1299

continued on next page

continued from previous page				
Clone Class	*SLOC*	*Type of Clone Class*	*Location of Source*	*Start Line : End Line*
9	24	Type I Clone	src/org/springframework/ or-m/jdo/JdoTemplate.java	588 : 611
10	21	Type III Clone	src/org/springframework/ u-til/AntPathMatcher.java	270 : 291
			src/org/springframework/ u-til/AntPathMatcher.java	291 : 313

References

[1] Always in2000 driver linux kernel source code. `http://www.cs.fsu.edu/~baker/devices/lxr/http/source/linux/drivers/scsi/in2000.c`.

[2] The linux scsi programming howto. accessed in November, 2011. `http://tldp.org/HOWTO/archived/SCSI-Programming-HOWTO/`.

[3] Springsource.org. accessed in November, 2011. `http://www.springsource.org`.

[4] Google says oracle seeking 2 billion in android dispute. accessed in October, 2011. `http://www.businessweek.com/news/2011-09-21/google-says-oracle-seeking-2-billion-in-android-dispute.html`.

[5] G. Antoniol, U. Villano, E. Merlo, and M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755 – 765, 2002.

[6] Brenda S. Baker. On finding duplication and near-duplication in large software system. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE '95)*, pages 86–95, Toronto, Ontario, Canada, July 1995. IEEE Computer Society.

[7] Hamid Abdul Basit and Stan Jarzabek. A data mining approach for detecting higher-level clones in software. *IEEE Transactions on Software Engineering*, 35:497–514, 2009.

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

[9] Stefen Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33:577–591, 2007.

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

[11] Andrew Binstock. In praise of small code. accessed in November, 2011. `http://www.informationweek.com/news/development/architecture-design/231000038`.

[12] Barry W. Boehm. Software risk managment: principles and practices. *IEEE Software*, 8 (1):32–41, 1991.

[13] James Bottemley. Scsi interfaces guide. accessed in November, 2011. `http://www.kernel.org/doc/htmldocs/scsi.html`.

[14] Magiel Bruntink, Student Member, Ieee Computer Society, Arie Van Deursen, Remco Van Engelen, and Tom Tourw. On the use of clone detection for identifying crosscutting concern code. *IEEE Transactions on Software Engineering*, 31:804–818, 2005.

[15] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, and Amit Seker. Shared information and program plagiarism detection. *IEEE TRANS. INFORMORMATION THEORY*, 50:1545–1551.

[16] James R. Cordy. Comprehending reality – practical barriers to industrial adoption of software maintenance automation. *International Conference on Program Comprehension*, 0:196, 2003.

[17] Philip B. Crosby. *Quality is Free*. McGraw-Hill, 1979.

[18] Michiel de Wit, Andy Zaidman, and Arie van Deursen. Managing code clones using dynamic change tracking and resolution. In *Proceedings of 25th IEEE International Conference on Software Maintenance*, Edmonton, Canada, 2009. IEEE Computer Society.

[19] Ekwa Duala-Ekoko and Martin P. Robillard. Clone region descriptors: Representing and tracking duplication in source code. In *ACM Transactions on Software Engineering and Methodology, 20(1)*, pages 1–31. ACM, June 2010.

[20] Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison-Wesley, 1990.

[21] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, pages 1–39, 1993.

[22] Nils Gde and Rainer Koschke. Studying clone evolution using incremental clone detection. *Journal of Software Maintenance and Evolution: Research and Practice*, 2010.

[23] Nils Göde. Evolution of type-1 clones. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 77–86, Washington, DC, USA, 2009. IEEE Computer Society.

[24] Nils Göde and Rainer Koschke. Incremental clone detection. In *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, pages 219–228, Washington, DC, USA, 2009. IEEE Computer Society.

[25] Michael Godfrey, Davor Svetinovic, and Qiang Tu. Evolution, growth, and cloning in the linux: A case study. In *2000 CASCON workshop on 'Detecting deplicated the near duplicated structures in large software systems: methods and applications'*, 2000. `http://plg.uwaterloo.ca/~migod/papers/2000/cascon00-linuxcloning.pdf`.

[26] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*, pages 131–142.

[27] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *Proceedings of the 17th international conference on Software engineering*, ICSE '95, pages 186–195, New York, NY, USA, 1995. ACM.

[28] Ahmed E. Hassan and Richard C. Holt. The top ten list: Dynamic fault prediction. In *Proceedings of the 21st International Conference on Software Maintenance*, Budapest, Hungary, Sept. 2005. IEEE Computer Society.

[29] J Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceeding of the 1993 Conference of the Centre for Advanced Studies Conference (CASCON'93)*, pages 171–183, Toronto, Canada, October, 1993.

[30] J Howard Johnson. Navigating the textual redundancy web in legacy source. In *Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON'96)*, pages 7–16, Toronto, Canada, October 1996.

[31] John Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the 10th International Conference on Software Maintenance*, pages 120–126, Victoria, British Columbia, September 1994. IEEE Computer Society.

[32] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering*, pages Vol. 28(7): 654– 674. IEEE Computer Society, July 2002.

[33] Cory Kapser. *Toward an Understanding of Software Code Cloning as a Development Practice.* PhD thesis, School of Computer Science, University of Waterloo, 2009.

[34] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[35] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, December 2008.

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

[37] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 187–196, New York, NY, USA, 2005. ACM.

[38] Denis Kozlov, Jussi Koskinen, Markku Sakkinen, and Jouni Markkula. Exploratory analysis of the relations between code cloning and open source software quality. In *Proceedings of the 2010 Seventh International Conference on the Quality of Information and Communications Technology (QUATIC '10)*, Washington DC, USA, 2010. IEEE Computer Society.

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

[40] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W.M. Turski. Metrics and laws of software evolution — the nineties view. In *Proceedings of the Fourth International Software Metrics Symposium*, Albuquerque, New Mexico, United States, 1997. IEEE Computer Society.

[41] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, Berkeley, CA, USA, 2004. USENIX Association.

[42] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32:176–192, 2006.

[43] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. Gplag: Detection of software plagiarism by program dependence graph analysis. In *In the Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD06*, pages 872–881. ACM Press, 2006.

[44] Simone Livieri, Yoshiki Higo, Makoto Matsushita, , and Katsuro Inoue. Analysis of the linux kernel evolution using code clone coverage. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 22–31, Washington, DC, USA, 2007. IEEE Computer Society.

[45] Simone Livieri, Yoshiki Higo, Makoto Matsushita, , and Katsuro Inoue. Analysis of the linux kernel evolution using code clone coverage. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, Washington, DC, USA, 2007. IEEE Computer Society.

[46] Jean Mayrand, Claude Leblanc, and Ettore Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th International Conference on Software Maintenance (ICSM'96)*, pages 244–253, Monterey, CA, USA, November 1996. IEEE Computer Society.

[47] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and Ken-ichi Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of the 8th International Symposium on Software Metrics*, METRICS '02, Washington, DC, USA, 2002. IEEE Computer Society.

[48] Tung Thanh Nguyen, H.A. Nguyen, N.H. Pham, J.M. Al-Kofahi, and T.N. Nguyen. Cleman: Comprehensive clone group evolution management. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, L'Aquila, Italy, 2008. IEEE Computer Society.

[49] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Clone-aware configuration management. In *Proceedings of the 24th*

IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, 2009. IEEE Computer Society.

[50] Foyzur Rahman, Christian Bird, and Premkumar Devanbu. Clones: What *is* that Smell? In *Proceedings of the Seventh Working Conference on Mining Software Repositories.* IEEE Computer Society, 2010.

[51] Chanchal Kumar Roy and James R. Cordy. A survey on software clone detection research. *SCHOOL OF COMPUTING TR 2007-54*, 115, 2007.

[52] Gehan M.K. Selim, Liliane Barbour, Weiyi Shang, Bram Adams, Ahmed E. Hassan, and Ying Zou. Studying the impact of clones on software defects. *Working Conference on Reverse Engineering*, 0:13–21, 2010.

[53] Ian Sommerville. *Software Engineering. 9th edition.* Addison-Wesley, 2011.

[54] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Addison-Wesley, 2006.

[55] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC '04)*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.

[56] Axel van Lamsweerde. Requirements engineering: from craft to discipline. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 238–249, New York, NY, USA, 2008. ACM.

[57] Wei Wang and Michael W. Godfrey. A study of cloning in the linux scsi drivers. In *Proceedings of the 11th Working Conference on Source Code Analysis and Manipulation*, Williamsburg, Virgina, United States, September 2011. IEEE Computer Society.