Toward an Understanding of Software Code Cloning as a Development Practice

by

Cory J. Kapser

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Cory J. Kapser

Abstract

Code cloning is the practice of duplicating existing source code for use elsewhere within a software system. Within the research community, conventional wisdom has asserted that code cloning is generally a bad practice, and that code clones should be removed or refactored where possible. While there is significant anecdotal evidence that code cloning can lead to a variety of maintenance headaches — such as code bloat, duplication of bugs, and inconsistent bug fixing — there has been little empirical study on the frequency, severity, and costs of code cloning with respect to software maintenance.

This dissertation seeks to improve our understanding of code cloning as a common development practice through the study of several widely adopted, medium-sized open source software systems. We have explored the motivations behind the use of code cloning as a development practice by addressing several fundamental questions: For what reasons do developers choose to clone code? Are there distinct identifiable patterns of cloning? What are the possible short- and long-term term risks of cloning? What management strategies are appropriate for the maintenance and evolution of clones? When is the "cure" (refactoring) likely to cause more harm than the "disease" (cloning)?

There are three major research contributions of this dissertation. First, we propose a set of requirements for an effective clone analysis tool based on our experiences in clone analysis of large software systems. These requirements are demonstrated in an example implementation which we used to perform the case studies prior to and included in this thesis. Second, we present an annotated catalogue of common code cloning patterns that we observed in our studies. Third, we present an empirical study of the relative frequencies and likely harmfulness of instances of these cloning patterns as observed in two medium-sized open source software systems, the Apache web server and the Gnumeric spreadsheet application. In summary, it appears that code cloning is often used as a principled engineering technique for a variety of reasons, and that as many as 71% of the clones in our study could be considered to have a positive impact on the maintainability of the software system. These results suggest that the conventional wisdom that code clones are generally harmful to the quality of a software system has been proven wrong.

Acknowledgements

I would like to thank my supervisor and friend, Dr. Michael Godfrey, for his continued support and guidance during this journey. I have learned a great deal about research and life throughout the course of his supervision. Thanks also go to my readers Giuliano Antoniol, Krzysztof Czarnecki, Ric Holt, Kostas Kontogiannis, and Steve McDonald. They have helped my research grow with their comments and suggestions.

The administrative staff at the University of Waterloo require special thanks, especially Wendy Rush whose enduring support has kept me sane. Thanks to Trevor Grove and Mike Patterson, without their technical knowledge and abilities many dissertations could never have been written.

Thanks to all of the members of the Software Architecture Group (SWAG) with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Mina Askari, John Champaign, Aseem Cheema, Jack Chi, Xinyi Dong, Alan Grosskurth, Ahmed Hassan, Abram Hindle, Davor Svetinovic, Jingwei Wu, and Lijie Zou. I have learned a great deal from all of them. I would also like to acknowledge Ric Holt as not just a reader, but someone who has been a leader and a mentor for many students in SWAG.

To all the members of *#beer* (you know who you are), thank you for sharing your thoughts, insights, and experiences. In particular, I would especially like to acknowledge Alma Juarez-Dominguez and Robert Warren. Sharing the "PhD experience" with them was a pleasure (or at least much more tolerable). I am lucky to have met such an excellent group of people.

I would like to acknowledge the guidance of the software engineering community, especially the WCRE and ICSM communities whose openness to discussion and review have helped shape my views and abilities as a researcher.

Most of all, I would like to thank my wife and best friend, Alina. Without her, this thesis may never have been completed. Her support and encouragement continue to make me a stronger and better person. I would also like to thank my mother, father, brother, and sister. They have been my biggest fans.

Invariably, acknowledgements always miss someone important. For those that I have not listed explicitly, thank you for being a part of this thesis and helping me grow as a person and a researcher.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This thesis examines the phenomenon of "code cloning" — the intentional or unintentional introduction of similar code segments — as it occurs in medium-sized to large software systems. Most software systems contain a significant amount of code cloning; typically 10–15% of the source code in large software systems is part of one or more code clones [51, 54]. While copy-and-paste is often used as the canonical example of code cloning activity, code cloning is not a result of this activity alone. Code clones may be introduced as programming idioms related to language or libraries, common usage of framework or library APIs, or even implementations based on common examples. Similarly, not all copy-and-paste activities should be considered code cloning. Copy-and-pasting trivial segments of code, such as boiler-plating of *for* loops or variable names, is not generally considered code cloning, as the resulting code segments typically share little interesting semantic content.

The effects of code cloning on the quality of source code are not well understood. In much of the literature on the topic, code cloning is considered detrimental to the quality of the source code, as it is generally believed that code clones can cause additional maintenance effort [7, 15, 26, 44, 47, 61, 74]. For example, changes to one segment of code may need to be propagated to several others, escalating maintenance costs [29]. Furthermore, locating and maintaining these code clones pose additional problems if the clones do not

evolve synchronously. Examples such as this have led Fowler to proclaim code clones as the most pernicious "bad smell" in source code [27]. With this in mind, methods for automatic refactoring have been suggested [9, 15], and tools specifically to aid developers in the manual refactoring of code clones have also been developed [38].

There is no doubt that code cloning can be an indication of sloppy design and in such cases should be considered to be a kind of development "bad smell". However, we have found that there are many instances where this is simply not the case. For example, cloning may be used to introduce experimental optimizations to core subsystems without negatively affecting the stability of the main code base [53, 55]. These experimental changes can be used as part of the production system, allowing end users to easily switch between experimental features and stable ones. Cloning can also be used to avoid overly complex code resulting from interleaving two or more segments of similar but non-identical code segments. It is examples such as these that motivated much of the work that is presented in this dissertation. The common view we found in the literature that code cloning is inevitably harmful to code quality and should be eradicated whenever possible — which we informally termed "cloning considered harmful" — seemed wrong to us, even "harmful" itself. Anecdotally, we had found that cloning could be employed in a number of ways and for a number of reasons, many of which seemed like principled engineering decisions to us. We therefore sought to systematically investigate the phenomenon of code cloning in real-world software systems. Using a tool we built, called the *Clone Interpretation and Classification System (CLICS)*, to aid in the detection, analysis, and categorization of code cloning we performed several exploratory case studies of medium-sized to large open source software systems that are in wide use. The results of these studies revealed code cloning patterns, which we have documented in an annotated catalogue. We examined the frequency and judged the "harmfulness" of these patterns within the studied systems and found that many code clones could be deemed to have beneficial impacts on the source code quality. Based on this work, the thesis statement of this dissertation can be given as:

> *Software cloning can be and is used as a principled design technique to achieve desired engineering goals.*

Or informally, "'Cloning considered harmful' considered harmful".

If we accept the proposition that not all clones are harmful, then a variety of concerns — such as stability, code ownership, and design clarity — need to be explicitly considered before refactoring is attempted; a software developer or maintainer should try to understand the reason behind creating the code clone before deciding what action (if any) to take. To aid these decisions, a catalogue describing common uses of code clones should be constructed, similar to the catalogues used to describe design patterns [28] or anti-patterns [17]. However, the current literature does not provide guidance for practitioners on how to manage code clones if they are to exist in the system for extended periods of time. This thesis lays the groundwork for such guidance, and provides evidence to suggest the stigma surrounding code cloning may not be well founded.

1.2 Thesis Overview

While clone detection is an area of active research, and several tools exist to facilitate code clone detection, until recently there has been relatively little empirical research on the types of clones that are found, where they are found within the design of the system, or how they are used in the context of building and maintaining software. Most research has explicitly or implicitly assumed that code cloning is harmful and has focused on methods for refactoring or removing code clones from the source code without considering the original decisions leading to the code clone.

With a focus on detecting and removing clones without first understanding how they are used, we are left with few means of evaluating the effectiveness or benefits of code clone removal on long-term maintenance. Additionally, without an understanding as to why and how developers use code clones, we cannot address the underlying issues that result in code cloning. Thus, we are led to three fundamental research questions raised in this thesis:

Question 1 *What are the common motivations for developers to use code clones?*

If clones are the result of lazy or careless programming practices then we would expect that the resulting code clones are likely to negatively impact the system. However, if developers intentionally create code clones for principled engineering reasons such as mitigating risk, supporting flexible evolution, or preventing fragmentation of concepts, then code cloning may have a positive impact on the system quality.

Whether developers are intentionally using cloning as a calculated part of software design or to avoid short term cognitive costs we must ask: in what scenarios are they using code cloning? Is there a regular or repeated use of code clones, both negative and positive, that can be discovered? This raises our second research question:

Question 2 *Are there common patterns of code cloning that occur in the development of software systems?*

Understanding the types of cloning that occur in software systems may help provide a better understanding towards the various types of maintenance challenges they present. For example, clones that are likely to be subject to different evolutionary forces, such as platform dependencies, are likely to require careful consideration when propagating changes. More concretely, bug fixes must be carefully considered. Bugs related to communicating with a specific platform are unlikely to appear throughout a set of code clones. However, bugs in a segment of code implementing interaction with the internal software system may be present in all or some of the code clones based on that code. In this case, the type of software bug and the type of code clone will dictate the maintenance tasks required.

If patterns can be discovered, can we measure the relative frequency with which they occur? As with any design practice, context will determine the appropriate actions. What can we learn about the harmfulness of cloning in software in relation to code cloning patterns? This leads us to our third question:

Question 3 *How are code cloning patterns used in practice, and to what extent is their use appropriate?*

Measuring the frequency or extent to which code cloning patterns appear in software can provide an indication of the relevancy of the clone patterns to software practitioners and code clone maintenance research. Investigating this question can also provide insight into the problems associated with using code cloning, even when using it in a principled way. If we discover that code clones are repeatedly used with good intent but lead to poor design over time, this would suggest that the well-intentioned use of code cloning is difficult and code cloning should simply be avoided.

Throughout our studies we have found that patterns of code cloning, such as the example of experimental forking described below (Section 1.4), are repeated in software systems. During recent case studies we observed several occurrences of code clones as useful artifacts for extending the features of a software system and hence they should not be removed [50, 51, 54]. Further analysis into cloning in software reveals that many of these "good" clones are created for similar reasons and in similar ways [53, 55]. While answering the first two questions, this thesis presents a set of patterns of code cloning as discovered in real software systems. Included in these patterns are the rationale for creating many of these clones and the corresponding development and maintenance trade-offs. These patterns are intended to be used to guide future maintenance activities involving cloned code. This guidance is aimed to help software practitioners make more efficient and effective decisions, in turn possibly contributing to a higher level of software quality.

To address the third question, this thesis presents the results of an evaluation of the use of code clones in two widely used, medium-sized software systems. By categorizing a sample of clones in two software systems and ranking their perceived harmfulness, we can obtain a measure of the overall quality of code clone use. These results show that a large number of code clones should not be viewed as harmful. The results in this thesis also suggest the percentage of good clones will vary with each software system studied.

1.3 A Definition of Code Cloning

Despite the growing research interest in the topic, a broadly accepted concrete definition of code cloning remains elusive. Even the general concept — code clones are similar segments of code that arise due to groups of problems that require the same or similar solutions — is not agreed upon. This shortcoming is even evident in the title of a recent workshop organized to bring together the experts on the topic: "Duplication, Redundancy, and Similarity in Software" [63]. Code cloning does not even appear in the title and it would seem code cloning is some form of conjunction of duplication, redundancy, and similarity in source code. Are code clones limited to intentionally duplicated code? Duplication implies a deliberate act, but some would suggest that duplication, such as copy-and-pasting, is only one way that code clones are produced. Is redundancy a criterion for a code clone? If it is,

that would imply that code that cannot be refactored (and is therefore not redundant) is not a code clone even though it will likely be subject to the same maintenance challenges code clones face. How is similar code related to code cloning? The building of user interfaces is often similar within a software system: the mechanism and order of adding buttons, text editors and displays, and event notification is restricted by the API provided by the toolkit being used. However, to what degree are these code segments code clones? Disappointingly, one conclusion of the before mentioned workshop is that "code clone" may not be a suitable term for our topic of research; however, no other term has achieved acceptance in the research community as of this writing [91].

A small survey of experts in the field indicated that agreement was low when deciding whether or not pairs of segments of code were code clones [48]. Because of this disagreement, many problems arise when one tries to precisely define the terms code clone and code cloning. What aspects of source code contribute to the definition of code clones? Given a small set of candidate code clones, code clone researchers were asked to decide if each candidate was in fact a code clone and to give a supporting rationale. Aspects of the segments of code that contributed to the decisions included:

- the degree of similarity (number and kinds of differences between the code segments),
- the size of the code segments,
- the type of code comprising the code segments,
- the mechanism for creating the possible code clone,
- the code structure,
- if the code segments were interesting or not,
- the difficulty in refactoring the code, and
- the context of the segments in the source code.

Disagreement on whether or not a candidate was a code clone tended to arise on two fronts: the importance of a particular attribute and the threshold of the aspect in determining whether or not a candidate is a code clone. When evaluating the importance of a certain

attribute some experts might deem the context or location of a candidate code clone more important than the changes made to it, while another expert might make the opposite decision. Concerning the threshold of an attribute, one expert might consider the segments in the candidate similar enough to be a code clone while another expert would claim they are too dissimilar. During the survey, decisions were made based on contradictory evaluations of the same attributes. This situation makes it particularly difficult to derive a precise and commonly accepted definition of code cloning.

With the conflicts in the decision making process, it is clear that we cannot provide a definition of the term code clone that will be inclusive of all or most of the prior work on the topic. However, we can provide our own interpretation of the general concept of code cloning to provide the context in which the results of this thesis were produced. In the most general sense, we consider a code clone to be two or more segments of code that are similar structurally, have a similar intended use or semantics, and represent the reuse/re-implementation of a recurring problem.

Definition 1 A *code clone (a.k.a. clone)* is two segments of code that are similar in both form and function, and represent the replication of a solution to a recurring problem.

Similarity of two segments of code can be measured from two perspectives: representation (form) and semantics (function) [92]. In the scope of the research presented in this thesis, representational similarity includes syntactic and lexical representation. If one ignores the interpretation of the meaning of the identifiers used to specify variables, constants, etc. syntax can be considered to be the structure of the code: if two segments of code have a one-to-one mapping of identifiers and their keywords, and if operators and separators are the same then these segments can be considered syntactically isomorphic. Lexical representation adds extra information about the code: the similarity of identifiers often has some correlation to the similarity or relationship of the code segments' intended interpretation. Formatting of code, such as indentation and line separation, can often be a matter of style for individual developers and, in the opinion of the author, should be ignored as part of the lexical representation. It is often the case that a developer will reformat a clone to match their own style preferences. While an uncommon style of coding that appears multiple times may be an indicator of cloning, we have considered this outside the scope of this thesis and our definitions.

Semantic similarity is a measure of the likeness of the intended interpretation of the meaning of two or more code segments. The interpretation of the meaning of a code segment could come in the form of an analysis of input and output, the operations comprising the code segment, the meaning of the lexicons used to encode the identifiers, or a combination of all three. For example, the two sorting algorithms Quicksort and Heapsort produce similar output given the same input. Both sorting algorithms could be used interchangeably in specific scenarios and therefore might be considered to have similar semantics externally. Quicksort and Heapsort behave quite differently internally. Differences in internal operations, data structures, and worst case complexity can make choosing one algorithm over the other essential, implying that, in some instances, their internal behaviours are an important aspect of measuring their semantic similarity. The lexical representation, including comments and identifiers, can provide insight into the intended usage of a code segment which in turn adds to its semantics. If two sorting algorithms are implemented to sort *BucketsOfFish* and *GrainsOfSand*, one might use their lexical contents to predict the characteristics of the intended input of each function. We can expect *GrainsOfSand* will be an extremely large input and as such one would want to avoid the $O(n^2)$ worst-case of Quicksort. While programmatic interpretation of the semantics of two code segments is likely difficult, Walenstein et al. suggest that a measure of semantic similarity can be approximated using several program representation analysis techniques such as execution curve similarity, abstraction equivalence distance, program dependence graph similarity, and Levenshtein distance on the segments' operations [92].

According to the above definition, a code clone requires both representational and semantic similarity. This definition excludes the re-implementation of the same concept in two very different ways. For instance, two implementations of Quicksort, one using recursive calls, and another using a queue, would not be considered a code clone even though they are implementing the same concept. This may be considered a form of redundancy, but was not considered a code clone in the scope of the research presented here.

Structurally similar code that does not share a semantic relationship is also excluded by the above definition. Examples of such code occur in and around (but are not restricted to) language features that have a limited form of common usages (`switch` statements in C/C++ are classic examples of this). Structurally similar but semantically distinct

segments of code are not considered code clones in this thesis because they are artifacts of the language rather than reuse/re-implementation of a specific problem and solution.

The distinction between segments of code that share only one form of similarity and those that share both is important. Segments of code that are similar semantically but not representationally are unlikely to manifest the same development and maintenance concerns as segments that are similar both semantically and representationally. It is more likely to be the case that segments of code that share both forms of similarity were developed, intentionally or not, with the same underlying assumptions and intended behaviours. For the duration of time that these code segments maintain similarity there will be shared maintenance concerns such as propagating changes occurring in one segment to other segments in the code clone relationship.

A related definition of cloning was described by Bellon et al., who defined three types of code clones based on the degree and type of similarities [16]. Type 1 clones are segments of code that are lexically identical. Type 2 clones are segments of code that are lexically identical if one ignores identifier names. Type 3 clones are adjacent sets of type 1 and type 2 clones, separated by lines that are not syntactically isomorphic (i.e., "gapped" clones). These definitions of clones are restricted to the representational similarity of code clones, and are more closely tied to clone detection results than the definition of code clone described here.

The cardinality of the code clone relationship is based on the classical definition of code clones used throughout literature on the topic [3, 6, 15, 4, 26, 44, 43, 47, 60, 61, 74]. However, groups of code clones can also be formed. To group or cluster code clones, the code clone relationship is often considered to be an equivalence relation, especially in the case of sub-string matching algorithms. In this case, rules of symmetry, transitivity and reflexivity are assumed to hold. Using this definition of the clone relationship groups of clones, often called *clone classes*, can be formed [74], also referred to as clone groups, or more recently clone multiplications [11]. This relation does not always hold for all types of clones. Segments of code that are considered clones may have large changes, such as insertions of statements, and these changes may break transitivity without affecting symmetry and reflexivity.

Definition 2 A *clone class* is a group of code clones formed according to the transitive closure of the code clone relationship, which is assumed to be an equivalence relation.

Definition 1 does not restrict the size or importance of the code segments in question, only that the segments provide a solution to a recurring problem. This implies that even the most simple tasks could be considered code clones, including programming idioms such as the allocation of memory and initialization of memory on the heap. In our investigation of code clones in the Apache httpd web server, we observed several clones of this type with many variations on the overall implementation. While these clones were very simple, perhaps even trivially so, they comprised several instances of an idiom with incomplete implementations. Examples such as these underscore the importance of a definition of code clone that does not exclude even the most basic code fragments.

The above definition is purposely imprecise in its description of similarity. The field of code clone detection and analysis has several open questions that have yet to be solved. Evaluating similarity, at the current state-of-the-art of code clone detection, and determining whether or not two code fragments are code clones remains a subjective process that relies on an individual's knowledge about the software system under study as well as software development in general. Measuring lexical and syntactic similarity can be performed programmatically — algorithms such Levenshtein distance and parametric string matching were developed for this purpose — but the evaluation of the importance of individual similarities and differences is not well understood. For example, there is no generally accepted definition of how lexically similar two segments of code should be before being deemed code clones. Further, should operators, separators, and keywords be included in the edit distance metric? Or more generally, what attributes of the source code can provide us with a metric, in a reasonable amount of time, of the representational and semantic similarity of two pieces of code?

Many techniques have been proposed to address the problem of measuring similarity to find code clones. These techniques, described in Chapter 2.4, can be referred to as code clone detection. For the purpose of this thesis, we now define code clone detection as follows.

Definition 3 *Code clone detection* is a process of identifying similar segments of code within a body of source code according to a precise, technique-specific, definition of

similarity, and reporting these detected segments as *candidate* code clones.

Definitions 1 and 3 have an important technical difference. It is often the case that two segments of code are code clones if they are detected by a code clone detector. Definition 1 defines a code clone using an unspecified measure of similarity but maintains the requirement of both semantic and representational similarity. On the other hand, Definition 3 does not require both semantic and representational similarity, but instead imposes a specific measure of similarity that depends on the particular detection technique employed. Ideally, code clone detection techniques would use criteria that encompass both forms of similarity. However, as stated above, this area of research is still developing. Code clone detection techniques try to approximate a measure of similarity through a variety of textual or graph-based methods that primarily measure representational similarity. This difference in definitions leads to both false positives — the incorrect identification of segments of code as code clones — and false negatives — the omission of code clones from the set of reported code clones. Because these techniques do not identify code clones based on a generally accepted definition of a code clone, they only present *candidate* code clones. The verification of the correctness of each set of similar segments of code is left to the investigator.

Finally, we must define what it means to create a code clone.

Definition 4 *Code cloning* is the intentional or unintentional replication of the solution to a recurring problem in the form of a code clone.

Code cloning is not restricted to the canonical example of copy-and-pasting code to reuse a pre-existing implementation of a solution. Other ways code clones can arise might be implementing solutions based on common examples, using a common API, or even using a mental conception of a solution formed by solving a similar problem in the past [15]. This notion of allowing both intentional and unintentional actions to form code clones allows us to study a group of artifacts that are likely to have common maintenance concerns regardless of whether or not the developers are aware of the artifacts.

1.4 Examples of Cloning

To motivate our study, we now present several examples of what we consider to be benign code clones. When a portion of source code within a software system is required to support several deployment environments, such as operating systems or hardware devices, the system architect often has two basic architectural choices: maintain a single version that can accommodate all possible environments, or find a common interface with the rest of the software system and fork several versions of the deployment-specific code. When the latter approach is taken, as is often the case for many long-lived successful systems, behaviour of the individual deployment-specific code bases will be similar to each other at a high level; however, complex variations are often also necessary. In other words, the interaction between the cloned code and the rest of the software system will be similar but the interaction with the external environment will be different. Completely separating these internal and external interactions can often result in complex, difficult to maintain code. Alternatively, the system designer may "fork" existing code by first duplicating it and then specializing it for the particular environment. If these environments evolve separately, changes to the cloned code are less likely to affect the operation of the original code. In this way the testing and maintenance risks associated with continual development of the independent interfaces is decoupled. If these code clones are refactored, changes made to support the evolution of one interface may affect the compatibility with the other interfaces, coupling the testing and maintenance risks. This pattern of code cloning is common when supporting multiple operating systems or external software entities such as database management systems. In these cases the interfaces are expected to evolve independently as well as impose unique requirements on the communication. Examples of this are found throughout the Apache httpd web server [54] where, for example, concepts such as process synchronization require operating system dependent components such as process or thread creation, semaphores, etc.

Another example of benign code cloning is cloning to avoid complex abstractions. Often complex abstractions are required to consolidate the variability in two similar solutions. In these cases, code is cloned and parameters are changed and/or statements are added or removed. When the added or removed statements are distributed throughout the cloned code, the control flow of a refactored solution can become increasingly complicated. Lan-

guage or paradigm limitations may be a contributing factor to this complex abstraction, especially in the cases where polymorphism cannot be used.

There are also cases where code is copied and then modified to closely tie an implemented solution to a complex concept such as mathematical formulae. A common form of code cloning found in Gnumeric, a spreadsheet application, is *templating*. In this case, the only changes made are to the identifiers for variables and functions, while the variable and parameter types are left unchanged. While the code clone in many cases can be easily removed using an obvious abstraction, such as function pointers, the developers choose to duplicate the code in order to achieve traceability of the code to the external concept. We consider this to be a benign form of code cloning because the resulting code is more comprehensible and maintainable. It is a principled use of code cloning to achieve a clearer and more maintainable design.

1.5 Thesis Contributions

This thesis introduces the notion of cataloguing patterns of code cloning in a similar fashion to the cataloguing of design patterns [28] or anti-patterns [17]. There are several benefits that can be gained from this characterization of code cloning. First, it provides a flexible framework on top of which we can document our knowledge about how and why code clones occur in software. This documentation crystallizes a vocabulary that researchers and practitioners can use to communicate about cloning.

This vocabulary partially addresses the problem of forming a generally accepted definition of code clones. As has been previously noted by Walenstein [93] and the author [48], there is no general agreement within the research community of what exactly constitutes a code clone. While the general idea of what constitutes a clone is simple and non-controversial — i.e., a code clone is two similar segments of code — there is little agreement on a concrete definition. Often factors such as the rationale behind creating the similar code, future maintenance strategies, and a specific definition of similarity play a role in an individual's definition of what a code clone is.

Using a very general definition of a code clone, we construct patterns of code cloning that encompass many of these varying interpretations of the general definition. By con-

structing the patterns of different types of code clones, this nascent description framework is expandable, allowing the accommodation of new patterns of code cloning, resulting in a growing vocabulary that can be used to discuss code clones in a clear and objective manner [53, 55].

This documentation of code cloning patterns is a first step towards formally defining these patterns to aid in automated detection and classification. These classifications can then be used to define metrics concerning code quality and maintenance effort associated with code clones, enabling developers to make informed decisions about how code clone maintenance should be carried out, including whether a particular code clone is still useful or should be refactored. Automatic classifications may also provide us with better ways to measure code cloning in software systems and better ways to gauge the severity of the problem in general.

Another contribution of this thesis is the insight it provides into how and why developers create code clones rather than forming higher level abstractions. These insights will have applications in the clone detection and analysis research community. A better understanding of the types of code clones that exist in source code will help focus work on improving clone detection methodologies. For example, we may be able to provide insight into the types of changes that are made when forming different types of code clones. Understanding common ways that developers use code clones in software systems will also aid in developing tools and methods to better manage code clones as they are created. If we can understand when it is deemed necessary to create code clones, we may be able to provide insights into improved languages, development environments, and API design.

Throughout the investigation of cloning in software we gained insights into the problem of analyzing code clones in software [31, 50, 49, 51, 52, 53, 54, 55]. These insights led to a high level description of the requirements of an effective clone analysis environment and the development of a prototype [52, 54]. In this work, an additional categorization of cloning is introduced. This categorization is a taxonomy of code clones in software systems that takes into account the location of the code clones within the software architecture, the type of code the code clones reside in, and the similarity of the regions the code clones reside in. This work is novel and has been an important step toward fully understanding the problems involved with the management of code clones in software systems. Our clone

navigation tool, the *Clone Interpretation and Classification System (CLICS)*, incorporates the taxonomy as a way of displaying code clones. The taxonomy also provides a method of classification and filtering of code clones.

1.6 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the terminology, background, related work, and our own innovations in the topic of clone detection and analysis. Chapter 3 provides an overview of the motivating research questions and the tools and methodologies that were used to carry out the studies for this thesis. Chapter 4 describes a set of code cloning patterns discovered through in-depth investigations of code cloning in software. Chapter 5 presents a case study evaluating code cloning in software and the use of the patterns described in Chapter 4. Chapter 6 discusses the answers to the three questions of this thesis and describes a possible application of code cloning patterns in software maintenance. Chapter 7 summarizes the contributions of this work.

Chapter 2

An Overview of Clone Detection and Analysis

2.1 Introduction

Existing literature on the topic of code cloning gives us little understanding of how and why code cloning is used as part of software development. Code cloning research has, until recently, focused primarily on detection and removal. While those works are important they may not help developers make day-to-day decisions on how to write better, more maintainable software. The focus of this dissertation, and the works leading up to it, is to analyze and document code clones in real software to further our understanding of how and why code cloning is used as a software development practice. To identify and motivate the questions of this dissertation, this chapter outlines code clone detection techniques and studies of code cloning in software, including our own, leading to the results presented in Chapter 4 and Chapter 5.

Before proceeding further, however, we must distinguish between code clone *detection* and code clone *analysis*. As defined in Chapter 1, code clone *detection* is the process of locating segments of similar source code, according to a precise definition of similarity, within a software system. Code clone *analysis* uses those results to examine code cloning in a software system. The goal of clone analysis is understanding the use of code cloning, either macroscopically, studying code cloning throughout the whole software system, or

microscopically, studying individual code clones. Until recently, most research has focused on the former rather than the latter. The work described in this thesis focuses on clone *analysis* rather than *detection*. A survey of clone detection techniques is provided in Section 2.4, and a survey of clone analysis techniques is provided in Section 2.6.

The rest of this chapter is organized as follows. Section 2.2 discusses current views on code cloning as a development practice and defines the terminology that has been used to describe this phenomenon. Section 2.3 provides an overview of the process of detecting and analyzing code clones in a software system. Section 2.4 describes, at a high level, the various techniques available to detect code clones and the characteristics of the code clones they can find. Section 2.5 describes the various kinds of postprocessing that have been used to filter and analyze the candidate code clone sets. Section 2.6 describes the tools and approaches that have been used to analyze code cloning in software systems. Section 2.7 summarizes the current methods proposed to manage clones in software. Section 2.8 describes the work that has been done to further understand code cloning in software systems. Section 2.9 discusses the open problems that remain to be analyzed in the field of code cloning in software systems.

2.2 Code Cloning — Use and Abuse

The literature on the topic has described many situations that can lead to the introduction of code clones within a software system [7, 15, 44, 47, 61, 74]. Many of these can be considered harmful uses of code cloning. For example, developers may copy-and-paste code because the short term cost of forming the proper abstractions may outweigh the cost of copying code. Developers may also clone code when they do not fully understand the problem or the solution, but are aware of code that can provide some or all of the required functionality. These examples attribute code cloning to programmer laziness. Code clones can also be introduced as a side effect of programmers' memories; programmers may repeat a common solution, unconsciously introducing code clones into the software system [15].

Code clones can also be introduced with good engineering intentions. In particular, there are four categories of motivations that can be readily identified: improving code understandability, improving code evolvability, technology limitations, and external business

forces. Code clones created for improved code understandability will be created to enhance readability, conceptual cohesion/coupling, and traceability. Code cloning can, in some situations, be used to keep software architectures clean and understandable. Code clones can also be used to keep unreadable, complicated abstractions from entering the system.

Improving code evolvability concerns the difficulty of introducing changes to existing code to address evolving requirements. Code that is abstracted to address two or more similar but separately evolving requirements may be difficult to modify. For example, a virtualization layer that is used to interact with several similar operating systems will need to maintain compatibility with each operating system as it evolves. As these operating systems evolve, the compatibility requirements may diverge or even conflict. Changes made to address one set of requirements may affect the code's fitness with the other sets of requirements. This kind of evolutionary force can lead to the *forking* clones described in Chapter 4. Clones of this type may be used for change decoupling to limit the scope of the impact of changes.

Technology limitations affecting developers' ability to reuse code through encapsulation or modularization often appear in the form of limited or cumbersome tools for abstraction, in some cases caused by lack of expressiveness of a programming language. In these cases, limitations of a given programming language may lead to the use of "boiler-plated" solutions for particular problems [93], or even source code generation. This kind of technique is common in COBOL development, for example, and can lead to *templating* clones, described in Chapter 4. In these cases, the use of code cloning is typically well understood by the developers, and the aim is to prevent errors by re-using trusted solutions in new contexts.

External business forces may necessitate the use of code cloning. Cordy notes that financial institutions consider code quality the most important concern when maintaining software because the cost of errors in software can dwarf software maintenance costs [22]. Fixing or modifying an abstraction can introduce risks of breaking existing code and requires that any dependent code, code calling directly and indirectly the changed code, be extensively tested, a process that is both costly and time consuming. Code cloning is a common method of risk minimization used by financial institutions that allows code to be maintained and modified separately, containing the risk of introducing errors to a single

system or module. Another external business force is time-to-market and opportunity cost. In some cases the long term cost of maintaining source code may be greatly outweighed by the short term opportunity cost of a lengthy time-to-market, especially in emerging markets where technology adopters may be difficult to attract once they have invested in a competing implementation. Code cloning is a practice that can be used to rapidly develop similar yet distinct sets of features. This motivation is one possible factor in the relatively high levels of code cloning found in web-based applications [78, 79].

Several software maintenance problems have been associated with the use of code cloning. In the long term, clones can unintentionally diverge if not carefully managed [72]. Code cloning can also lead to an unnecessary increase in code size [7, 44]. Code cloning code can lead to unused, or "dead", code in the system when the desired solution does not require all of the functionality provided by the clone (for example, not all branches of the original code may be used in the code clone's new context). Left unchecked, this unused code can cause problems with code comprehensibility, readability, and maintainability over the life time of the software system [44]. These long-term maintenance problems require tools and processes to track and manage cloned software entities over the evolution of a software system.

There are other maintenance risks associated with the use of code cloning. If a bug is identified within code that has been cloned, then care must be taken to ensure that the bug is fixed in every clone instance. This may be both time consuming and risky: in addition to the extra effort required to fix the same bug several times, the location of the clones may not have been recorded explicitly, and differing contexts may make it hard to simply copy-and-paste the fix.

When code cloning is performed without a solid understanding of the original code and its context, bugs can be introduced. For example, variables may be shared and modified unknowingly [44]. Program comprehensibility can be negatively effected by the need to understand the differences between the code clones.

However, despite these known problems, we have found that developers can and do use code cloning as a design tool when they judge that the benefits outweigh the risks; that is, these developers believe that the use of code cloning can improve the design of the code. For example, aggressive refactoring can sometimes create abstractions that are complex, overly

subtle, and non-intuitive; in this case, modified code clones may be easier to understand and maintain than a solution that employs abstraction, as the study performed by Toomim et al. suggests [87].

When clones are used to minimize exposure to risk or support alternative external requirements, the scope of the impact of a change is reduced, improving modifiability and testability attributes. Rajapakse et al. found that reducing code clones in a web application not only had negative effects on the modifiability of an application — after significantly reducing the size of the source code a single change required testing of a vastly larger portion of the system — and also reported that avoiding code cloning during initial development could contribute to a significant overhead [79]. Code cloning can, in specific cases, enable faster time-to-market which may have significant market share benefits. These characteristics of clones are also useful in exploratory development, where the reuse of behaviour can be used to fast track development of a new feature but the eventual path of evolution is too uncertain to be able to anticipate the appropriate abstractions.

Evaluating the likely positive and negative effects of code cloning is a continuous balancing act. Code clones that improve comprehensibility (and thereby improve maintainability) may also increase the amount of effort required to extend or change code (thereby decreasing maintainability). Similar to many development decisions, developers must assess the overall cost of code cloning and decide on an individual basis the overall expected gain versus cost. The patterns detailed in Section 4 are intended to provide some guidance, and to enable developers to make decisions based on qualities of the problem domain, the development and deployment environments, and the code itself.

2.3 Clone Detection and Analysis

Code clone detection and analysis can be modelled as a three step process, as shown in Figure 2.1: generating a list of candidate clones, post-processing the results, and analyzing the clones. Clone detection has been extensively studied and is described in detail in Section 2.4. A high level view of the clone detection process is depicted in the box "Clone Detection" in Figure 2.1. The first step is to transform the source code into a representation that will be processed by the code clone detection algorithm. The transformation

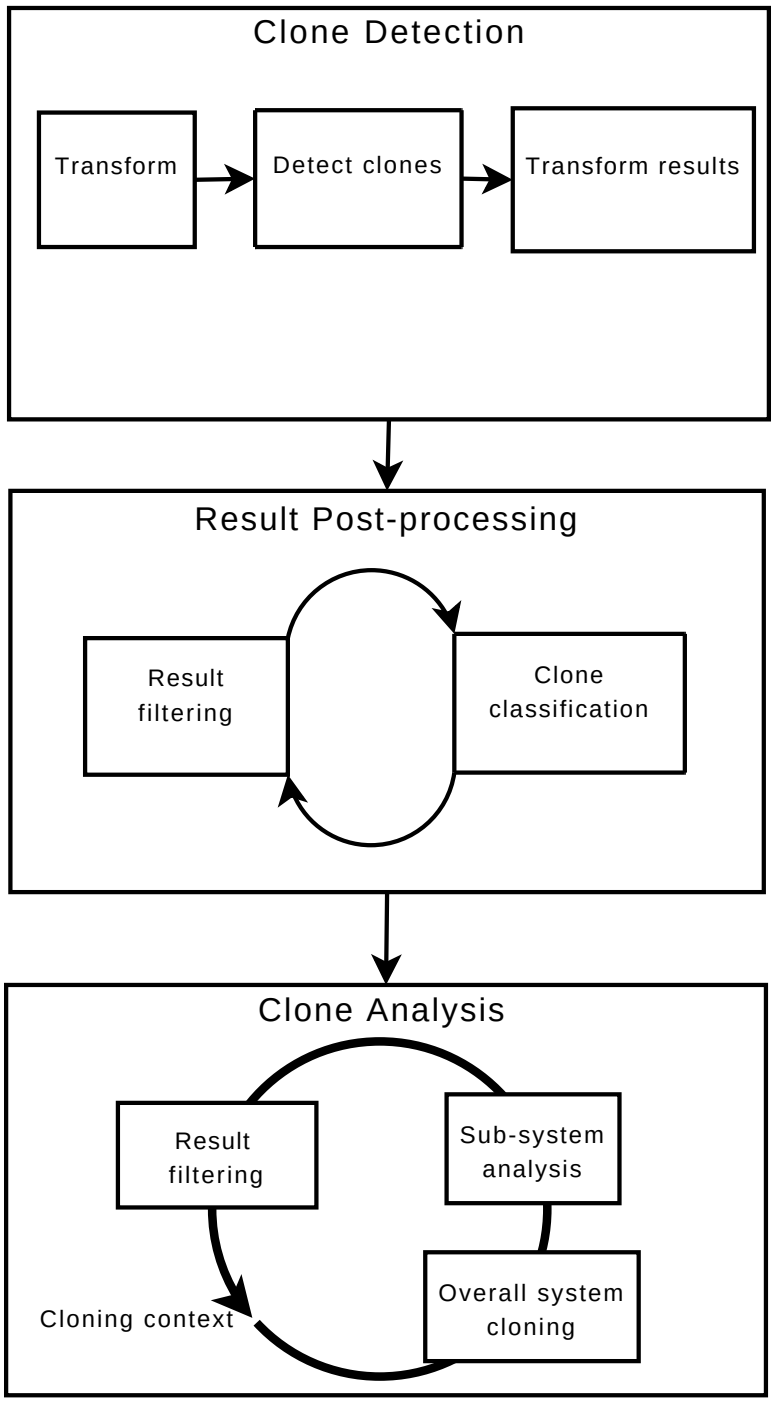

Figure 2.1: Process of clone detection and analysis

is intentionally lossy: details that are thought to be unimportant or a hindrance to clone detection are removed or ignored. For example, detection methods that use token streams typically keep only operators, non-white space separators, identifiers, and keywords. Detection of the clones within the transformed code is then performed using a variety of data mining techniques. The specific method to do this is dependent of the source code representation (further discussion can be found in Section 2.4). After detection is performed, the results are transformed to directly reference the original source code.

After candidate code clones are identified, the results are post-processed to remove clones that are not of interest, and the remaining code clones are further analyzed and categorized. Filtering may be done for a variety of reasons including removal of false positives, trivial code clones, or code clones that are not similar within a specified threshold. Categorization can be done using constraints [8, 51, 62, 74] and pattern-matching [13]. These steps are shown as a loop in Figure 2.1 because categorization can feed back into filtering. For example, different regions of code tend to have different types of false positives. We have found in our case studies [50, 51, 52, 54] that categorization is a useful tool for applying clone-type specific filters because these filters can be more strict when applied to specific types of source code.

The final step of the process is to analyze the remaining candidate code clones. Analysis can be comprised of automatic high-level metrics gathering, semi-automatic identification of interesting clones, and manual analysis of cloning within the software system. The process is shown in the box "Clone Analysis" in Figure 2.1. The clone analysis shown in the figure more closely reflects manual analysis as we describe it in [52, 54]. In-depth analysis of code cloning in a software system is usually iterative in nature. Information about the overall cloning in the system, such as frequency of clone types and location of clones, contributes to the context of the clones being analyzed. This context provides a guide for the investigator to study the details of cloning within the subsystems of the source code. It enables the investigator to make informed decisions about filtering the view of the clones in the analysis. As the investigator continues the analysis, more information is added to the cloning context contributing to an overall understanding of code cloning in the software system. Details on the various approaches to clone analysis are provided in Section 2.6.

2.4 Code Clone Detection

This section summarizes the current literature on clone detection methodologies in detail. Clone detection has been an active topic of research since the early 1980s. In the early 1980s, this research tried to find ways to detect plagiarism in students' assignments [35, 37, 40, 73]. In this application, similarity detection methodologies used techniques such as comparing metrics of the overall program (common examples are use of operators, variable access, and order of procedure calls) [35, 37] or comparing the static call graphs of the two programs [40].

In the early 1990s, focus shifted toward finding code clones within software systems as a software engineering problem. A variety of new techniques are now available, ranging from language independent approaches such as line-by-line comparisons [26, 43, 44] to more language dependent approaches such as program metrics comparison [60, 74]. These code clone detection methods can be compared based on their source representation resulting from the transformation shown in Figure 2.1. These representations include abstract syntax trees (ASTs) [15, 61, 74], program dependence graphs (PDGs) [59, 64], normalized lines of code [26, 43, 44], and parameterized token streams [7, 6, 47]. All of these approaches to clone detection primarily use representational similarity as a basis for comparison. The key difference between these techniques is the level of textual and syntactic analysis that is used to detect the clones. ASTs and PDGs use deeper syntactic analysis of the source code. Normalized lines of code use only textual analysis. Parameterized token streams use mostly textual analysis with some syntactic analysis. The following subsections describes these code clone detection categories in more detail.

Throughout this section two key terms from the field of information retrieval will be used when discussing the characteristics of the candidate code clones returned by a clone detector: *precision* and *recall*. Precision refers to the quality of the candidates returned by the detection method: high precision indicates the candidate code clones are mostly correctly identified as code clones (i.e., there are few false positives) and low precision indicates the candidate code clones contain many candidates that are not actual code clones (i.e., there are many false positives). Recall refers to the overall percentage of artifacts that exist in the source code that have been detected by the clone detector: high recall indicates most of the code clones in the source code have been found (i.e., there

are few false negatives), low recall indicates most of the code clones in the source code have not been found (i.e., there are many false negatives). When comparing code clone detection techniques, precision and recall are often referenced as measures of the accuracy and completeness of the candidate code clones. It should be noted that classifying a candidate code clone as being correctly or incorrectly identified (as a true or false positive) is a highly subjective task, making accurate measures of precision and recall difficult to obtain. Recall is particularly hard to estimate as it requires that we already know the complete set of code clones in a software system. In Section 2.4.5 we discuss the studies that have measured and compared the precision and recall of code clone detectors.

2.4.1 Normalized Lines of Code

The goal of normalizing source code is to remove trivial or irrelevant elements of source code to enable direct comparison of the relevant portions of source lines of code. Formatting, typically in the form of indentation and other white space, is used to make the code more readable but has no formal semantic value. Comments serve a similar purpose. Formatting code to reflect nesting depth or developer preference is often done when copying source code but reformatted code clones cannot be found when directly comparing lines of code. Normalizing the source code by removing formatting and commenting ensures that syntactically identical lines of code will have the same textual representation. As described in this section, there are several methods that can be used to find clones using this simple data structure.

Methods

Ducasse et al. describe a clone detection algorithm with two steps [26]. The first step is to transform the code. To maintain maximum language independence, only a very simple transformation is made: only white space and comments are removed. While removing comments is language dependent, the process is usually trivial and poses no serious challenges. For example:

```
if (CAR_IS_FAST && (brakes == NULL))    you_are_toast = 1;
```

transforms to:

```
if(CAR_IS_FAST&&(brakes==NULL))you_are_toast=1;
```

Further normalization was considered by Ducasse et al. [25]. Constants, identifiers, and function names can also be normalized. As one might expect, as the amount of normalization increases, the precision decreases. In their study, Ducasse et al. found that these forms of normalization dropped precision from 94% to 70% in one case study and from 42% to 11.5% in another. On the other hand, this normalization improved recall by as much as 20% [25].

The next step compares each non-empty line of the source code to every other non-empty line. The comparison value is stored in a matrix where the coordinates are the line position of each compared line. The complexity of such an algorithm with an input of n lines is $\Omega(n^2)$. To improve the performance of their algorithm, Ducasse et al. hash the strings into B buckets and compared within the buckets, reducing the runtime by a factor of B.

An alternative to this matching approach is to generate a sequence of hash values for the program by hashing each line of code. Using this sequence of hashes, a suffix tree can be built. The advantage of this approach over that described by Ducasse et al. is that suffix trees can be built in $O(n)$ time, and all maximal matches, matches that would break if they were extended one character in either direction, can be found in $O(n + z)$ where n is the number of lines and z is the number matching pairs of sequences of lines.

As a final step in the approach described by Ducasse et al. [26], pattern matching is used to post-process the results to find sequences of matches with breaks. This can happen when a line of code is changed after a code clone is created. The pattern matcher looks for diagonal lines in the matrix with a predefined threshold of percentage of breaks in the line. Wettel et al. [94] details a simple pattern matcher for chaining together smaller matches. Another post processing step is to remove accidental clones that arise from programming idioms or recurring program structures, such as the *break* statement.

Johnson describes an approach to finding matching substrings within source using substring fingerprints [43]. A nearly identical approach was later described by Schleimer et al. [83] in their description of the MOSS system for source code plagiarism detection. The process starts by normalizing the source code and then generating fingerprints for each substring. An ideal fingerprint is a function that maps data to a set of fingerprints such

that $f(x) = f(y)$ only if $x = y$ and $f(x) \neq f(y)$ only if $x \neq y$. Using integer values as fingerprints reduces the space and time requirements of comparing lines of code in large software systems. In this approach the fingerprint of all possible substrings of size l are generated with a maximum of M characters and a minimum of m characters. Substrings are only chosen at the start and end of lines.

Johnson uses the fingerprinting algorithm described by Karp and Rabin [84], where each substring is treated as numeral N encoded in some base r (r is 256 in ASCII text). The fingerprint is then calculated as $N \bmod p$ and can be encoded in $\lceil log_2 p \rceil$ bits. The constant p must be decided at the start of the execution. Discussion on choosing p can be found in detail in [43].

To reduce the number of fingerprints to be compared at a later stage, some fingerprints are culled during their computation. For any given segment of size c characters, a substring is chosen to represent it. In Johnson's work the substring that covers the segment and has the largest fingerprint is chosen. This culling will lose small matches but matches of size $2 * M - c$ are guaranteed to contain at least one matching fingerprint. Setting c to 1 causes the maximum amount of culling while setting it to M causes no culling to be done.

After all fingerprints are computed, they are sorted and grouped. A group of matches is a set of substrings with the same fingerprint. To reduce the size of the results matching groups can be combined. A set of groups that occur consecutively or overlap can be joined as one group.

Another variation using normalized lines of code is the island grammar [77] approach introduced by Cordy et al. [23]. The method begins by extracting interesting syntactic structures from the source code, called *islands*. Non-interesting code is consider *water*. In their validation Cordy et al. examined tables and forms in HTML source code for interesting syntactic structures [23]. In imperative programming languages such as C or Java islands could be procedures. Extracted *islands* are pretty printed and stored in text files. The final detection process is line-based, so the pretty printing aims to separate meaningful programming structures on separate lines. In many ways this is analogous to creating a token stream except that here more than one token could be considered as a single unit. Finally, after pretty printing, the *islands* are compared using the Unix program diff. *Islands* are considered to be clones if they overlap by a certain threshold (this threshold may vary

according to the specific project or programming language [23]). The number of potential comparisons is quadratic but the problem space can be greatly reduced by intelligently selecting the files to compare. For example, *islands* must be of similar size, within 50%, to be likely matches. As another optimization, once a pair of *islands* are considered to be clones, one of the *islands* is used as an exemplar and the other is not used for further comparison. These two optimizations have the potential to reduce the recall of the method but are necessary to make the approach feasible.

Discussion

Methods using normalized lines of code have the key advantage of being largely language independent and lightweight. This flexibility makes them easy to apply in large, multi-language software systems. However, due to the simplicity of the source code transformations only exact or near-exact code clones are likely to be found, while modifications to identifiers or literals can cause matches to be missed. Splitting or merging lines of code also poses a particular problem. Our own investigations have shown that a large number of code clones do not fit into the simple formula of similarity these methods are targeted toward.

While the approach of Cordy et al. is not strictly normalized lines of code, its simple transformation is essentially normalizing code and removing newlines characters. This extra transformation enables their method to overcome missed clones caused by changes on multiple lines. However, their method of comparison is close to quadratic, both in the number of islands to compare and the method of comparing the islands. The second optimization assumes that the cloning relationship is transitive, but in the case of inserted or deleted lines this is not the case. It is conceivable that an exemplar A may match an island B but not match island C which B could match based on their definition of a match.

2.4.2 Parameterized String Matching

During the reformatting of code, it is common that lines will be merged and split or variables will be renamed. For example, a developer using the C programming language may prefer to have an opening brace alone on a new line or on the same line as the statement

the block will be dependent on. When this type of reformatting operation is performed on a code clone, line-based detection methods will not detect the match. Parameterized string matching methods use token streams as their underlying data structure making them insensitive to reformatting. By replacing identifiers and literals with placeholders, methods using this data structure are largely insensitive to changes in variable names and literals.

Methods

Parameterized string matching was first introduced by Baker [6] and further improved by Kamiya et al. [47]. In the course of our research we have primarily used the tool CCFinder, developed by Kamiya et al. [47] and have now developed our own parameterized string matching tool.

The tool CCFinder, which is in wide use within the cloning research community, begins by performing a lexical analysis of the source code, resulting in the creation of a list of tokens based on the syntax of the given programming language. The tokens of all the files are concatenated into a single array of tokens. As part of the code transformation, all white space and comments are ignored. Next, several language specific transformation rules are applied [47]. Then type, variable, and constant identifiers are replaced by a special identifier (such as `$P` [47] or a unique ID for each identifier type [70]).

Once the source code has been transformed into this abstract token stream, an exact match algorithm is performed to find maximal matching strings within the transformed code. This is done by constructing a suffix tree, which has $O(n)$ complexity, and locating matching substrings within the tree, as proposed by Baker [7, 6].

After the exact matches have been found, parameter matching is performed to filter the results [47]. That is, starting from the beginning of a pair of exactly matched transformed strings, CCFinder begins one-to-one matching of the identifiers. As the identifiers are mapped, if a conflict is found but a sufficiently large number of tokens have been matched the current match up to this point is reported, and parameter matching begins again.

CP-Miner, created by Li et al. [70] uses frequent sub-sequence mining techniques from data mining to find code clones. The process begins by scanning the source code and generating a generalized token stream where all identifiers and constants of the same type are given the same placeholder. Then, each program statement is hashed and the pro-

gram is turned into a sequence of hash values. A sequence database is created using the hashes. As a performance optimization, the sequences that are recorded are limited to basic programming blocks rather than entire files. Using the CloSpan algorithm, common sub-sequences are mined. Because these sub-sequences can have arbitrarily large gaps that would be meaningless in the context of clone detection Li et al. modify CloSpan to mine only sequences with a user-specified maximum gapping distance.

After mining the common sub-sequences, the candidate clones are filtered in a similar way to the one-to-one mapping used by CCFinder. In the case of CP-Miner, however, the one-to-one mapping does not need to be exact. The user can specify a threshold that allows for a certain percentage of mismatches. Overlapping and small clones are also filtered from the results. After this filtering is performed, the remaining clones are grouped or joined to their neighbouring clones if possible. The newly formed clones must pass the filtering requirements of the smaller clones otherwise the smaller clones are kept ungrouped. The major advantage of this method over suffix tree methods is that it easily overcomes insertion and deletion of statements.

Discussion

Tokenization and parameterization is considerably less computationally intensive than complete parsing and computing the AST of a software system. Clone detection based on parameterized strings uses only structural information, making it flexible enough to handle small changes introduced in the code clones. Sub-sequence matching is more flexible in handling line insertions and deletions, something substring matching cannot easily do.

Semantic analysis can be approximated when identifiers are mapped. In many cases, if a sequence of identifiers can be mapped one-to-one, a similar operation is being performed in each code segment. However, many false positives slip though even this filter. For example,

```
int x = z; x = pow(x, 4) + x;
```

and

```
long factor = INIT_VAL; factor = recursive_call(y, depth) + y;
```

both transform to

```
$P $P = $P; $P = $P($P, $P) + $P;
```

and have a one-to-one mapping. Therefore the above code snippets match according to the definition of parametric matching but it is very likely these segments are not code clones. Our own experience has shown that false positives occur in areas of source code comprised of a simple structure, such as initialization lists and *switch* statements. Many of these false positives can be filtered using strict filters on clones found in specific regions of code [52, 54]. To avoid the need for this type of filter, our own implementation of parameterized string matching does not parameterize source code that resides outside of procedure bodies. This source code is most often comprised of data type definitions, procedure declarations, basic macro definitions, and other simple, repetitive structures that lead to false positives.

2.4.3 Abstract Syntax Tree Methods

An abstract syntax tree (AST) is an abstract representation of the program source commonly used as an intermediate data structure in a compiler and is a common structure used in program analysis. It represents the abstract syntax of the language and includes only elements of the language that affect the program semantics. For example, the parentheses that determine the order of operations are removed as this order is encoded into the structure of the tree. This representation is insensitive to formatting and less sensitive to coding styles such as variable naming, particularly when the structure of the tree is the focus of the detection process.

Methods

This group of clone detection techniques can be divided into two sub-groups: sub-tree hash comparison and sub-tree node comparison. Sub-tree hashing first appeared as programming block metrics comparison. Metrics-based clone detection methods use sets of metrics to generate "fingerprints" for each code block or function in the source code. These metrics are often gathered using both the program source, as in the case of number of lines of comments, and from an AST, as in the case of cyclomatic complexity. Metrics-based clone

Scale	**Name**	**Layout**	**Expression**	**Control Flow**
ExactCopy	=	=	=	=
DistinctName	≠	=	=	=
SimilarLayout	X	∼	=	=
DistinctLayout	X	≠	=	=
SimilarExpression	X	X	∼	=
DistinctExpression	X	X	≠	=
SimilarControlFlow	X	X	X	∼
DistinctControlFlow	X	X	X	≠

Table 2.1: Mayrand et al. categorization

detection was introduced by Mayrand et al. [74] and Kontogiannis et al. [61], and further studies have used function metrics as a basis of clone detection [3, 4, 8, 10, 9, 60, 88].

In general, metrics-based approaches begin with gathering metrics for blocks of code. The granularity for the blocks of code is either statements residing between opening and closing separators, or whole functions. The choice of metrics that are used may vary, but generally metrics relating to layout, control flow, interface, and expressions are used so as to get better coverage of orthogonal properties of the code. These metrics become the fingerprint of the block, and the basis for comparison.

Mayrand et al. [74] uses these metrics to classify program fragments as code clones on an ordinal scale ranging from *Exact Copy* to *DistinctControlFlow*, shown in Table 2.1. They define four orthogonal categories of comparison and describe metrics to be used in each category: the name of the function, the layout, the expressions, and the control flow. Each metric of comparison has a threshold delta associated with it that specifies the maximum difference for two metric values to be considered similar. Metric values that exceed such a delta are considered distinct. These deltas are used in the ranking of function similarity, as summarized in Table 2.1. In the table "=" signifies all metrics match for that point, "∼" signifies at least one metric value pair is not equal but is similar, "≠" signifies that at least on metric value pair is distinct, and "*X*" signifies that this comparison point is not considered for this clone type.

Kontogiannis et al. describes an AST-based approach that uses a metrics matcher to find a set of candidate clones and filters these results by measuring the similarity of the code segments using a dynamic programming algorithm [61]. Their metrics matcher uses two alternative approaches to grouping possible cloned blocks. The first uses the Euclidean distance of the metrics of two code blocks. The second clusters all functions according to each computed metric and iteratively builds clusters of cloned functions by taking the intersections of each cluster. The clustering is done by grouping functions whose $metric_i$ differ by less than d_i where d_i is a parameter specified by the user, similar to the approach used by Mayrand et al. [74].

Once a set of candidate code blocks is selected using the metrics matching algorithm, a similarity measure is computed using dynamic programming. The similarity measure is computed by measuring the optimal alignment of the two code fragments and computing the number of insertions and deletions of statements required to make one fragment the same as the other. This is very similar to computing the Edit Distance of a string [69]. However, the elements of the string are statements rather than characters and the comparison is done on a feature vector computed using the AST. Three possible feature vectors were suggested: definitions and uses of variables and literals, definitions and uses of data structures, and metrics used in the metrics matching phase. The method was further refined by Balazinska et al. to use tokens rather than statements [8].

Jiang et al. have recently introduced a parse-tree based method similar to the AST-based methods previously described [42]. A parse tree represents the rules used to parse a segment of code. As opposed to an AST whose nodes are the meaningful syntactic units found in the source code, a parse tree's internal nodes represent the non-terminal rules of the parsing grammar and the terminal (leaf) nodes represent the syntactic unit. As a consequence, a parse tree usually includes all tokens found in the source. Parse trees may be used as an intermediate step to producing an AST, and generally have a lower computational cost to produce. For each sub-tree, a *characteristic vector* $v = \langle v_1, v_2, ..., v_n \rangle$ of integers is computed. Each element v_i is the count of the number of the i^{th} *atomic tree pattern* found in the sub-tree. Jiang et al. describe a *q-level atomic tree pattern* as a complete labelled binary tree with labels from set $\pounds$ (there are $|\pounds|^{2^{|\pounds|-1}}$ unique trees). Jiang et al. use 1-level atomic tree patterns where the trees are node types considered

relevant to code clone detection. Relevant nodes are those that have semantic value or, in other words, nodes that appear in an AST. Irrelevant nodes are nodes such as parentheses and braces. Vectors are not generated for all possible sub-trees, only for those sub-tree types considered relevant. This can be configured by the user.

Once vectors are generated for the sub-trees, merged sub-tree vectors are formed. This is done to find larger clones that may cover more than one sub-tree. Sibling sub-tree forests are serialized and a moving window of size k nodes creates a *characteristic vector* for each set of possible merged sub-trees by summing their *characteristic vectors*.

To detect clones, the vectors are clustered according to their Euclidean distance. To avoid the quadratic problem of pairwise comparison of vectors, the problem can be formulated as a nearest neighbour problem. Using *Locality Sensitive Hashing (LSH)* [42] clone clusters can be found efficiently. Given (p_1, p_2, r, c), LSH generates a family of hashes for the vectors such that if the distance between a pair of vectors v_1 and v_2 is less than r, the hash of v_1 will equal the hash of v_2 with a probability greater than p_1. If their distance is greater than cr, the hash will be equal with a probability less than p_2. Using LSH hash tables, all clone clusters can be found in in $O(dn^{\rho+1} \log n)$ where d is the dimension of the vectors (the number of *q-level atomic tree patterns*), n is the number of vectors, and $\rho = log_{p_2} p_1 < \frac{1}{c}$. To allow for larger difference between larger clones, vectors are placed in overlapping groups according to size. LSH is then applied on each group with appropriate constraints according to size and the final results are merged.

The approach of Jiang et al. bears some similarity to that of Kontogiannis et al.. In both approaches feature vectors are used to compare sub-trees. The original approach suggested by Kontogiannis et al. allows for a variety of feature vectors to be used. However, the approach by Jiang et al. introduces the simple metric of the number *q-level atomic tree patterns* and an efficient approach to clustering according to Euclidean distance.

Clone detection using sub–tree matching of a program's AST was first introduced by Baxter et al. [15]. This method uses well proven compiler technology to build the AST and several simple algorithms for clone detection. Their tool produces macros that can be used to remove the clones without affecting the operation of the system.

The first step to the process finds sub-trees in the AST that are similar. Baxter describes similarity as a function of the nodes that are shared in the sub-trees and those that are

different, as shown in Equation 2.1. S is the number of nodes common between the two sub-trees, L and R are the number of nodes that differ between sub-trees.

$$Similarity = 2 * S/(2 * S + L + R) \tag{2.1}$$

Because comparing every sub-tree to every other sub-tree is infeasible, Baxter reduces the number of comparisons by first hashing each sub-tree into a bucket and then comparing only members of the buckets. Baxter suggests that many clones come from a copy, paste, and edit process where changes tend to be small, therefore only affecting small sub-trees of the full AST. A hash function that does not consider small sub-trees of the sub-tree being hashed is therefore required to avoid hashing near miss clones into different buckets; in this case, the leaves of the tree were ignored [15].

Special attention must be given to certain code sequences, such as declarations or "straight line code" (i.e., a sequence of statements at the same logical nesting level). Straight-line code tends to produce left or right leaning sub-trees with a sequencing operator as the root [15]. Sequences are taken care of by searching the AST for sequence nodes and storing them in a list with the hash values of their sub-trees. The sequences are put into buckets according to their hash and then compared using the similarity function described in Equation 2.1. This process is different from the general sub–tree comparison in that the general sub–tree comparison may find each element of a sequence as a separate clone while the sequence algorithm will find them as a single clone.

The final phase in detecting clones is to look for more general near miss clones. This is done by comparing the parent trees in each pair of cloned sub-trees using the similarity function. This helps detect clones where more changes have been made. During this phase, clones in sub-trees of the larger code clones are removed from the result set, removing non-maximally matching clones.

Another approach to directly comparing the sub-trees of an AST using suffix trees was recently introduced by Koschke et al. [62]. In this approach the AST is serialized using the AST nodes and an attribute specifying the number of child sub-trees for each node. Sub-trees are serialized in the order they appear in the actual source code, preventing spurious false positives or false negatives. Ukkonen's suffix tree algorithm [90] is then used to build

a suffix tree of the serialized representation of the AST and using the suffix tree maximal matches are found. These matches are then post-processed to contain only syntactic units: clones that are not split across code blocks or statements. This approach is faster than Baxter's method. Matches using suffix trees can be found efficiently in $O(n + z)$ where n is the number of nodes in the input and z is the number of matches. This compares very favourably to Baxter's approach which has complexity of $O(n^2)$ for each bucket where n is the number of sub-trees in the bucket, although in practice they have found the cost is much closer to $O(n)$ with optimization. The suffix tree approach also has the advantage of being able to compare all possible sub-tree-pairings as it does not depend on a hashing function to group sub-trees.

Discussion

Using abstract syntax trees as the core data structure has many advantages. The foremost advantage is that the effect of programming style is largely reduced because of the high level of abstraction. However, there are several key disadvantages. First, a properly constructed AST may not include all of the program code, particularly when working with languages that use preprocessor directives, a problem noted by Casazza et al. [4]. Secondly, ASTs are time consuming to create. Koschke et al. assume that the AST is already generated when comparing the running time of their approach to non-AST-based clone detection methods [62]. Finally, this approach requires a language-dependent parser. This may pose serious challenges when applying this technique to large legacy systems for which parsers may not be easily available, or for systems composed of multiple languages [26].

The approach to clone detection described by Jiang et al. is perhaps more flexible than full AST methods while still using only nodes that would appear in an AST [42]. Jiang et al. found their approach handled files with syntax errors better than the tool created by Baxter et al. [42].

2.4.4 Program Dependence Graphs

A program dependence graph (PDG) is an attributed, directed graph representation of source code where vertices are individual statements and predicates, and edges represent

control and data dependencies amongst the vertices [39]. A statement X is *control dependent* on statement Y if Y causes X to be executed. A statement X is *data dependent* on Y if Y defines or modifies data that X will use. Unlike the edges of an AST, the sequence of statements found in the source code is not encoded in the graph, making this data structure more resilient to simple modifications of code clones such as statement reordering. In fact, clones that are found using this approach may not be contiguous sets of statements. Insertions and deletions of statements have a reduced impact on the clone detection process as they may not affect both the control or data dependencies of other statements.

Methods

Clone detection methods based on a *Program Dependence Graph* (PDG) have been studied by Komondoor [59] and Krinke [64]. Krinke [64] uses a modified dependence graph called a *Fine-Grained Program Dependence Graph* that is similar to both the AST and the traditional PDG. The similarity to ASTs is found in the correspondences of the vertices that appear in both. These graphs retain the key advantage of PDGs by only using control and data dependencies as edges in the graphs.

In general, an algorithm for locating clones in software using PDGs attempts to find isomorphic subgraphs of a pair of PDGs. Two graphs are isomorphic if they are structurally similar irrespective of the vertex and edge labels. PDGs are structurally similar when their edges map bijectively onto each other, and the attributes of the matching edges and vertices are the same. This can be done by starting at every pair of matching nodes (nodes with the same attributes) and growing the subgraph in a breadth-first manner. Rather than starting a match with all nodes with matching attributes, a problem that is quadratic in complexity, Krinke use a subset of these nodes [64]. This subset should be based on the feature of the data and is therefore programming language specific. Krinke used predicate vertices for his implementation to locate code fragments [64].

Subgraphs can be built in various ways. Krinke builds subgraphs inductively from two matching nodes in a pair of PDGs summarized by the following algorithm:

1. Given a pair of vertices v and v' in graphs G and G'.
2. Add to the subgraphs g and g' the vertices v and v'.

3. Let the edges leaving v and v' be e and e'. Put e and e' into equivalence classes e_i and e'_i, where edges are equivalent when their attributes match and the attributes of their end vertices match.

4. For each equivalence class e_i and the corresponding class e'_i add each edge to g and g'.

5. For each set of end vertices v_i and v'_i reached by the edge equivalence classes continue this process recursively.

Komondoor et al. used a different algorithm for building subgraphs. Rather than moving forward through the edges, they used backward slices. First, all vertices in the PDGs are put into equivalence classes based on the syntactic structure the represent. Then, for each pair of vertices in an equivalence class, a pair of isomorphic subgraphs is produced. This is done by backward slicing in lock step from the vertex pair. For each step, if a node that has been reached has a matching node in the other graph, the node and the edge are added to the slice. If a *loop* or *if-then-else* predicate is reached, a single forward slice is made to add the matching control-dependent successors.

Discussion

With either algorithm, the clone detection is not sensitive to the re-ordering of statements within a code clone, unlike AST-based or text based algorithms. However, because they are performing subgraph comparisons, their complexity is quadratic making them slower than the non-graph based approaches. Additionally, these techniques are highly language dependent and suffer from the same parsing challenges that AST-based techniques face.

2.4.5 Comparisons of Detection Tools

There are several aspects to be considered when evaluating and comparing clone detection methodologies, including applicability, accuracy and scalability. Applicability can be measured in two ways: the ease of applying a method to various software systems, and the utility of the results it provides. Language dependence plays a major role in determining the applicability. Legacy software systems or web applications can be comprised of several

languages, making detection of clones using language-dependent approaches more costly. Clone detection tools that return matches that are often trivial or uninteresting are also less useful in industrial settings. Accuracy is often measured as the number of false positives the methodology is likely to report; other important metrics include precision and recall, defined earlier in this chapter. Scalability measures how well the detection technique performs as the size of the input increases. Software systems can range in sizes from tens of thousands of lines of code to several million. Depending on the size of the system, accuracy and scale may become a tradeoff.

Rysselberghe et al. evaluated a representative technique for three of the four groups of clone detection techniques described above [81]. Using metrics matching, parametric line matching, parametric token matching, and simple line matching they evaluated portability, accuracy, relevance and scalability of the methods. Portability and relevance contribute to the evaluation of a method's applicability. As expected, metrics matching was the least portable, followed by parameterized token matching, parameterized line matching, and then simple line matching.

In terms of accuracy, they found that both simple line matching and parametric token matching return few false positives. However, this result contradicts our own findings [51, 52, 54] and those of Bellon et al. [16] where parameterized token matching was found to return many false positives. Rysselberghe et al. report parametric line matching returns few false positives. Metric matching returns slightly more false positives when using functions as the level of granularity, and returns many false positives when using blocks of code. This is consistent with our own results [50], where we found the main cause of the false positives to be short blocks or procedures.

Rysselberghe et al. found that simple string matching returned many useless results, that is, results that were considered to be not worth refactoring or maintaining. Because extract string matching returns code clones that are identical or very similar, this is a surprising result. It is likely due to the fact they did not use minimum length restrictions for this technique: many of the useless results were single lines that were the same. Metrics matching found many useless results when using a granularity of programming blocks. Using method bodies produced fewer but still many useless results. On the other hand, metrics did return the most recognizable matches. This was closely followed by parameter-

ized token and line matching. Simple line matching returned the least recognizable results. The metric of "recognizable" was evaluated based on how easy it was to decide whether or not the code clone would likely be of interest to a software maintainer.

In their study Rysselberghe et al. also evaluated scalability [81]. They found simple line matching to be the least scalable because of its quadratic space and time requirements. Parameterized line matching was considered more scalable than parameterized token matching because of the large memory consumption of suffix trees. Metrics matching was also considered more scalable than parameterized token matching for similar reasons.

Burd and Bailey evaluated five clone detection techniques for use in maintenance [19]. MOSS, JPlag, CloneDr, Covet, and CCFinder were tested on a small open source software system, GraphTool. MOSS detects similarity in normalized lines of code; JPlag uses token streams while CCFinder uses parameterized token streams; CloneDr uses AST sub-tree comparison as described by Baxter et al. and Covet uses metrics matching to locate function clones. In the study, the clone detection tools identified a total of 1463 clones, 563 were judged to be false positives. The union of these clones was deemed to represent all of, or the majority of, the clones in the subject system. Using this set of manually evaluated clones, the precision and recall was measured for each tool. CCFinder exhibited the highest recall at 72% with the next highest being Covet at 12%. CloneDr was reported to have perfect precision but the lowest recall at only 9%. CCFinder reported results with only 72% precision, MOSS had 73%, Covet had 63%, and JPlag had 82%. From these results it would appear that CCFinder has a very favourable trade off between precision and recall but this may not be the case. The construction of the data set assumed the combination of the results of all tools would result in a complete set of clones but this may not have been true. However, it is highly likely that there were clones in the data set that were not found by any of the evaluated tools. CCFinder reported approximately 1000 clones that each of the other tools did not report, strongly biasing the recall metric in CCFinder's favour. This bias may be less favourable if a complete set of clones was identified. Also, recall and precision are both difficult to evaluate as the reliable, reproducible identification of false positives is difficult as judges do not agree on the definition/concept of a clone [48, 93]. While general trends are clear, it is difficult to determine the accuracy of the end results and the impact of misclassification.

Bellon [16] and Koschke et al. [62] also measured the precision and recall of clone detection tools. Bellon created a benchmark set of clones by randomly sampling and evaluating a random subset of the union of clones detected by all detectors in the study. This resulted in an oracled set of clones known to be true positives. Each reference clone was classified into one of three types: exact clones (Type 1); parameterized clones (Type 2); and clones with additional changes (Type 3). Six clone detection tools were used in the study: Dup (token-based), CCFinder (token-based), CloneDr (AST sub–tree), Duplix (PDG), CLAN (AST metrics), and Duploc (normalized lines of code). Recall was measured by calculating the number of clones in the reference set that were found by each tool. Token-based and line-based tools performed much better for overall recall than graph-based approaches. CCFinder and Dup had a total recall of 67% and 33% respectively. CCFinder had a recall of 84% for Type 1 and Type 2 clones, but only a recall of 19% for Type 3 clones. This is not surprising as CCFinder searches for contiguous series of parameterized matches. Type 3 clones may have additional lines added, breaking the match for CCFinder. Duplix was the only tool that reported a higher recall for Type 3 clones (36%) but its overall recall was quite low for other clone types.[1] AST-based methods displayed a low recall. CloneDr exhibited a recall of just 9% and CLAN had just 5%. The precision results were nearly the inverse of the recall. AST-based approaches had the highest precision, distantly trailed by token-, line-, and PDG-based approaches. This result is more difficult to evaluate, however. Precision is calculated as the ratio of references from the oracle set found by the tool divided by the total number of candidates found by the tool. This is a somewhat misleading metric as the oracled references may not contain all clones. Many of the clones found by the tool may in fact be un-oracled clones rather than false positives, skewing the results of the precision metric. The token- and line-based approaches report many more candidate code clones than other detection techniques, making this metric biased against them.

Koschke et al. found that the recall of AST-based methods can be dramatically improved depending on the implementation [62]. In their study, they continued the use of the Bellon benchmark to compare the accuracy of their AST suffix tree approach, cpdetector,

[1] A private discussion with the Krinke revealed a bug was found in the tool and the recall is much higher now.

described above as well as a variation of the Baxter et al. approach, ccdiml. An additional 1% of the clones in the benchmark were oracled. ccdiml exhibited a recall of 53% compared to a recall of 61% for CCFinder and the average of 30% for all AST methods (including CloneDr, the tool of Baxter et al.). CloneDr reported a recall of only 10%. The precision of ccdiml was comparable to that of the token-based approaches in the benchmark. The tradeoff for higher recall and precision comes at the cost of scalability. On PostgreSQL, a medium-sized open source relational database system, ccdiml required almost five hours to detect clones compared to 62 seconds for cpdetector.

Both the Bellon and Koschke et al. studies have the same weakness as the Burd and Bailey study. The categorized clones are subjectively evaluated by human reviewers, a process that has been shown to be error prone. Walenstein studied inter-rater agreement when rating function clones for the purpose of refactoring [93]. Using candidate clones from the Bellon study, groups of clones were randomly sampled from groups of clones that constituted similar code of more than 50%, 75% and 95% of the code between functions from three software systems. Raters were asked to decide if a clone was relevant for refactoring tasks. Over four trials the parameters of the study were refined. During the first two trials, the software system WELTAB was used as the study subject. Inter-rater reliability was typically above 80%. The study subject Cook was added in the second phase. Inter-rater agreement was low, only 48.9%. This lack of agreement stemmed from the unconventional style of the C programming used in Cook and disagreement about refactoring functions that acted as constructors and destructors. SNNS was used in the third phase, this time with an additional judge. Overall agreement was again under 80%. In the final phase, raters were asked to judge the clones based only on the information in the source code. Using SNNS again, agreement reached 93.2%. Because of the low agreement in the second and third phases of the study, it may be difficult to judge the reliability of single-rater studies concerning code clones in software.

2.5 Result Post-Processing

Clone detection techniques, in particular parametric string matching, can return very large numbers of potential code clones. Tens of thousands of code clones may be detected for even

a medium-sized software system of several hundred thousand lines of code. Clone analysis on such large data sets is hardly tractable unless measures are taken to identify code clones that are relevant to the clone analysis task at hand. In some cases, language specific filtering might be used to reduce the size of the overall result set. Automatic classification of code clones based on a variety of attributes can also aid in identifying code clones that are relevant to a specific clone analysis task. This section describes filtering approaches we have used to improve the accuracy of *parameterized string matching* techniques and classification approaches that have been used for various clone analysis tasks, including code clone refactoring.

2.5.1 Filtering Results

Clone detection methods return only pairs (or groups) of suspected code clones. These results can suffer from many false positives [50, 51, 52, 54, 62]. We have found that many false positives can be removed from data sets returned by *parameterized string matching* techniques by noting that certain regions of code contribute a large number of false matches. By strictly filtering code falling in these regions we can improve the quality of the candidate code clone set. In particular the filters applied to the clones returned by our clone detection tool are (described in more detail in Chapter 3):

1. **Non-function filter**. This filter operates on structs, union, type definitions, variables, and prototypes. If a clone falls into a region of code comprised of the previously mentioned types, it must have a minimum of 60% of its lines match exactly. These lines can be matched in any order. This filter is not used with our own clone detector, part of the CLICS toolkit, but was used in our prior studies with CCFinder.

2. **Simple call filter**. Clones occurring on statements that are "simple function calls" can often contribute to many false positives when using parametric string matching algorithms. Regions of code that are "simple function calls" are sequences of code of the form

```
function_name(token [, token]*).
```

3. **Logical-structures filter**. We found that clones within simple logical structures, such as switch statements are often false positives. By enforcing a stricter similarity metric, we are able to remove many false positives.

4. **Overlap filter**. Clones whose two segments of code overlap by more than 30% of their length are also removed as such candidates represent only the structural similarity of simple, repeating code (much like the code clones found in `switch` statements or declarations of variables).

2.5.2 Categorizations of Code Clones

This section describes the various categorizations of code clones that have been proposed in the literature. One problem with code clone analysis is that clone detection tools may return a large set of suspected clones, but provide little or no additional information about them to aid the user in their interpretation. This makes clone analysis cumbersome at best and intractable in general. Viewing and classifying thousands of clones manually is time consuming and impractical, but it is necessary if one hopes to manage clones successfully; for example, in our Apache httpd case study [52, 54], a naive use of the clone detection tool CCFinder resulted in 13,062 clone pairs. As a result, little work has been done concerning the in-depth investigation of cloning as it occurs in software systems.

One of the first categorizations of clones in software was proposed by Mayrand et al. [74]. This categorization, summarized in Table 2.1 classifies candidate code clones according to the types and degrees of differences between code segments. The types of differences considered are function names, layout, expressions, and control flow. Using these attributes, clones are categorized in to eight types, described earlier in this chapter and shown in Table 2.1.

Balazinska et al. [8] created a schema for classifying various cloned methods based on the differences between the two functions that are cloned, summarized in Table 2.2. The differences accounted for are in five groups. Changes that do not affect the output or behaviour of a function, called *Identical* and *Superficial Changes* are in Group 1. Differences affecting only one lexical token at a time, types three through nine, are in Group 2. Clones that differ in several of the categories in Group 2 are grouped as types ten through twelve.

1	1	Identical
	2	Superficial Changes
2	3	Called methods
	4	Global variables
	5	Return type
	6	Parameters types
	7	Local variables
	8	Constants
	9	Type usage
3	10	Interface changes
	11	Implementation changes
	12	Interface and Implementation changes
4	13	One long difference
	14	Two long differences
	15	Several long differences
5	16	One long difference, interface and implementation
	17	Two long differences, interface and implementation
	18	Several long differences, interface and implementation

Table 2.2: Balazinska's categorization

Interface changes includes changes to called methods, global variables, parameter types, and/or return types. Implementation changes includes changes to local variable types, constants, and/or type usage. Methods where one or more entities are completely different, types thirteen through fifteen, are in Group 4. Here a difference would be considered a changed statement or expression. Methods differing in several aspects such as interface and a long difference, types sixteen through eighteen, are Group 5. These categories are used by Balazinska et al. to produce software aided re–engineering systems for code clone elimination [9, 10].

A major limitation of the categorizations proposed by Mayrand et al. and Balazinska et al. are their limitation to function clones. Our previous case studies show that only

accounting for function clones may miss a significant portion of other clones in the software system. We have found that function clones may account for only 30% to 50% [50] of the cloning activity in software, indicating that categorization of other types of clones is required to maintain significant coverage of cloning in a software system.

To facilitate our analysis of cloning in software systems, we have proposed our own taxonomy of cloning in software, further described in Chapter 3 and Appendix A. This work differs from Mayrand et al. and Balazinska et al. in that our classification scheme is based on different attributes of the clone including: similarity; locality within the software system; scope of the cloned code; degree of similarity of regions the clones exist in; and the type of region the clone occurs in. We feel that it is important to consider the location of clones in a system as it provides a useful metric for analysis of how cloning is carried out in a system. Additionally, the locality of clones is also an indication of other attributes, such as similarity or rationale for cloning, that can be used to evaluate the severity of cloning in a software system. The purpose of our taxonomy is to be a general categorization of clones that occur in industrial software systems. The goal of Balazinska et al.'s work was to produce a categorization for the purpose of refactoring of software systems.

Koschke et al. define another clone classification using only the types of differences between clones [62]. In this classification only three types of clones are considered, as described above for its use in the Bellon benchmark.

Basit et al. [12] have produced a clone classification scheme that groups clones according to higher level structures such as files or classes. Using the data mining technique "market basket analysis" they search for frequently co-occurring clones. In their model, each file is a basket and each clone class in that file is an item. Their goal is to identify clone classes that occur together frequently and in this way they identify patterns of cloning. Using this technique they classify clones as cloned classes and files, which they call "clone patterns". These patterns can then be further clustered according to the clone classes contained within them and the degree of similarity between the files.

2.6 Code Clone Analysis

Clone analysis is the task of investigating and understanding code clones within a software system after a candidate set of code clones has been detected. This section describes the current work (excluding much of the work of the author which is described in Chapter 3.7) on analyzing code clones in the context of software systems.

Several tools have been developed to aid users in the analysis of clones. Visualization of clones is commonly done using scatter-plots to present matched lines of code [7, 26, 80, 89]. These scatter-plots, shown in Figure 2.2, provide the ability to select and view clones, as well as zoom in on regions of the plot. In practice, we have found scatter-plots do not scale well with medium to large software systems, as the points become so small that it is difficult to pick out all but the most extensive regions of similar code. Additionally, scatter-plots do not easily lend themselves to providing the context of cloning from an architectural perspective. However, scatter-plots are useful when providing views of subsystems or highlighting particularly large regions of suspected code cloning.

Johnson used Hasse diagrams to visualize cloning relationships and investigate changes in software between two versions of GCC, shown in Figure 2.3 [45], and later proposed the use of hyper-linked documents to navigate cloning relationships [46]. In this system users can view groups of similar clones, jump to segments of code in a file, and view clusters of files with matching segments of code. This work provides a structure for navigation of code clones, but does not include architectural views of the software system, nor does it provide metrics to guide users to points of interest. Additionally, there is no way to modify the data set to remove false positives as they are found.

Reiger et al. describes five polymetric views with the goal of showing what parts of the system are connected via code clones and what parts are cloned the most [80]. These views have been designed to aid the user in learning about the code clones in a software system at different levels of abstraction, providing progressively more information about the code clones in the software. Reiger et al. describe the details of each view, its intention, symptoms it highlights, scalability, and over-plotting concerns. The first view, the *Duplication Web*, arranges files in a circle and draws edges connecting files with clones between them. The width of the node shows the degree of cloning within a file, and the width of an edge indicates the amount of cloning between files. This view is intended to

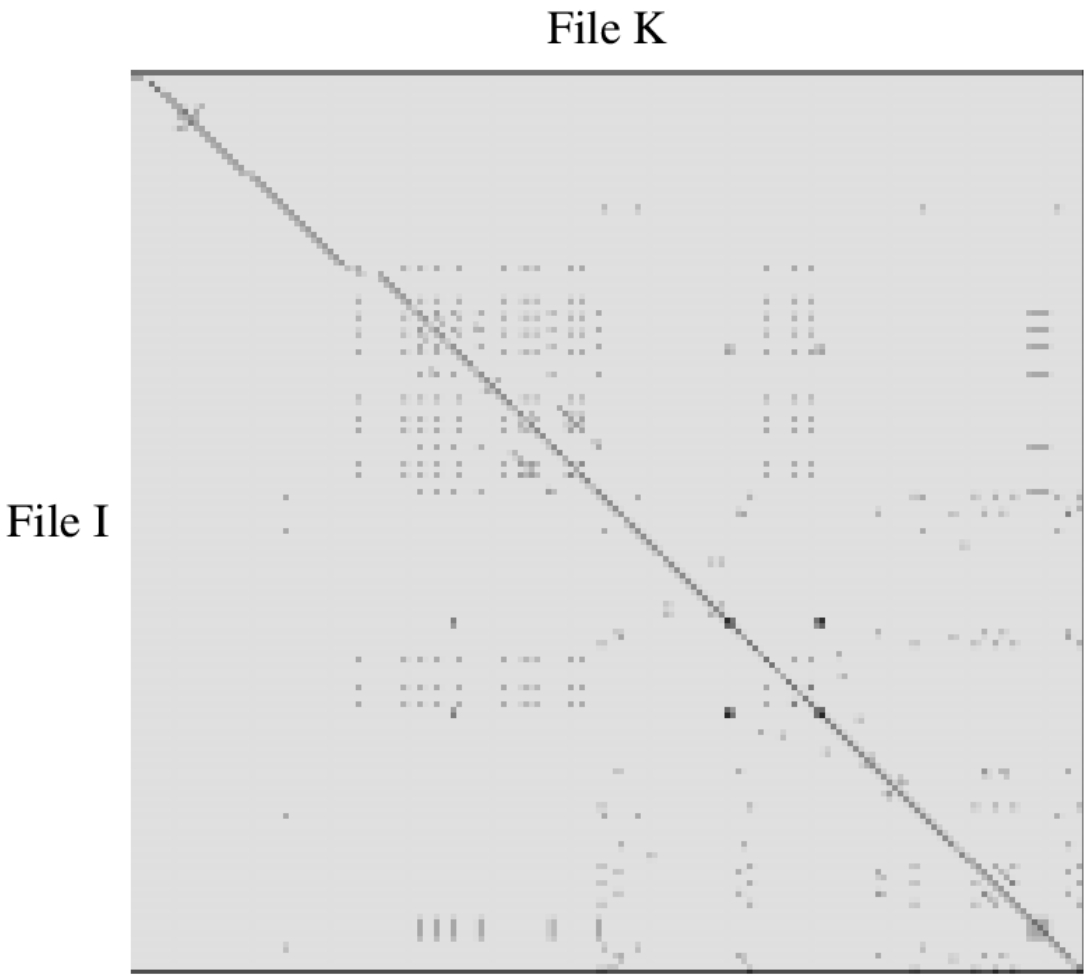

Figure 2.2: Scatter-plot of cloning between two files (taken from [26])

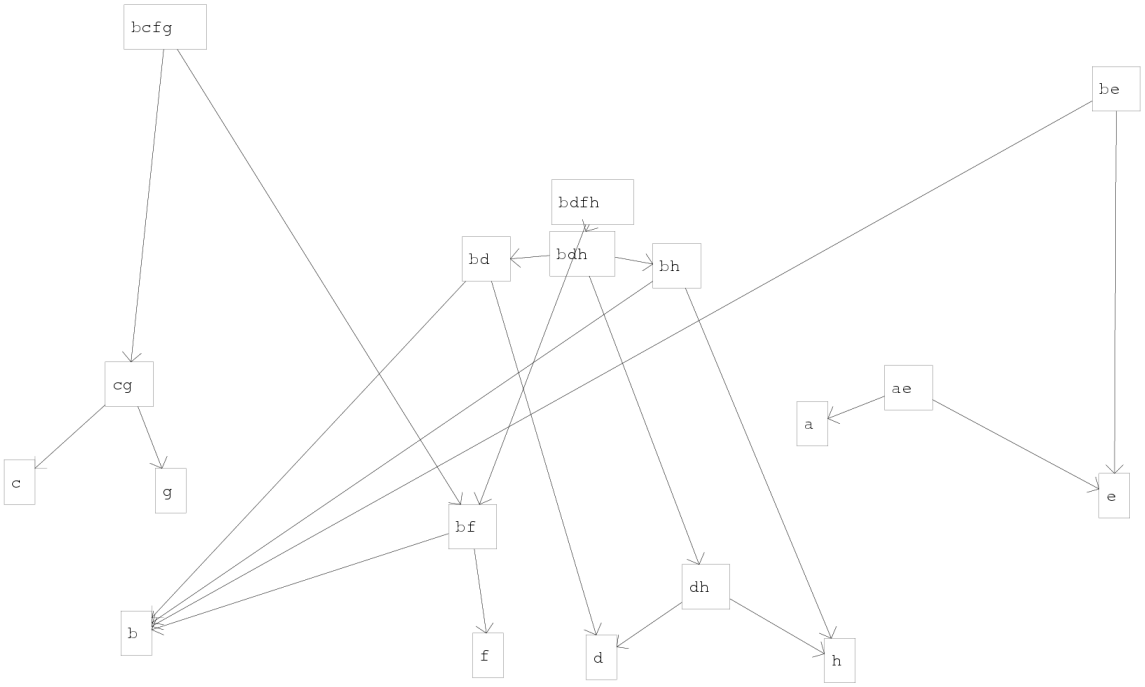

Figure 2.3: Hasse diagram of clone relationships (taken from [45])

indicate the number of files in the system and relate this to the amount of cloning between them. The second view, the *Clone Scatter-plot*, places the same nodes of the *Duplication Web* onto a scatter-plot ordered using LOC and lines of similar code. Edges connect nodes with cloning between them. The key advantage of this view is that it illustrates the ratio of cloning in the file to its size. Clones close to the diagonal contain a high percentage of code that is part of a code clone. The third view, the *Duplication Aggregation Tree Map*, is intended to show the ratio of internal versus external code cloning. The tree is generated according to directory structure and node width illustrates external cloning while height indicates internal cloning. The fourth view, the *System Model View*, visualizes the system as a tree based on directory or inheritance structure. Nodes represent either files or directories and are sized according to the degree of external and internal cloning. Node height indicates external cloning and node width indicates internal cloning. The fifth view, the *Clone Class Family Enumeration*, models the relationship between files and clone classes. Using a nodes-and-edges diagram that is split in the middle, clone classes at the top are connected to files at the bottom. Classes are arranged according to the number of files they include and the number of lines that have been cloned, and files are arranged according to the LOC they contain and the number of clone classes they participate in. *CLICS* has an analogous view to each of these visualizations with the exception of the *Clone Class Family Enumeration*. Unlike *CLICS* these tools do not provide filtering, metrics reporting, or querying facilities.

Gemini [89] and Aries [38] are two tools that use CCFinder as their core clone detection mechanism. In addition to the scatter-plot described above, Gemini also provides visualization through metrics graphs and file similarity tables. It allows users to browse clones either pair-wise, or using clone classes. Clone classes can be browsed and filtered using a graph displaying four metrics: the length of the path to the lowest common ancestor in the directory path (RAD(C)), the number of files containing segments of the clone class (POP(C)), the length of the cloned segment in tokens (LEN(C)) and the number of tokens the program would be reduced by if the clone were eliminated (DFL(C)). This system provides guiding metrics and a system overview but does not tie code clones to a concrete architecture. Also, it does not provide querying support or the ability to modify the data set.

Aries is a refactoring support environment for code clones [38]. It groups clones based on dependencies between them. Yoshida et al. define relationships between methods based on call dependencies of shared variables. They call these methods *chained methods*. Clones are then grouped based on the chained methods they cover. For example, if *Clone A* occurs in *Method A1* and *Method A2*, and *Clone B* occurs in *Method B1* and *Method B2* then *Clone A* and *Clone B* are chained clones. The intuition behind this grouping is that if one clone is refactored, both should likely be refactored. Aries supports this form of refactoring by grouping clones in this way and computing metrics to indicate how each chained clone is distributed throughout the system. Based on the inheritance hierarchy, Aries can recommend a refactoring method to use based on the metrics of the code clones. While Aries provides the capability to refine the displayed clones using queries, it does not support data set refinement or views mapping clones to concrete architecture.

2.7 Management of Code Clones

Management of clones has been a relatively undeveloped research area in clone detection. This is partially due to the implicit assumption that code clones should be refactored out of the software system. Most clone detection researchers prescribe clone refactoring or unification in one form or another [10, 15, 13, 27]. Basit et al. propose a solution to managing clones using meta-programming [13]. Using the meta-programming language XVCL, Basit et al. show how clones that cannot be refactored using standard programming language constructs, such as templates and generics, can be refactored using code generation based on meta-programming. In their solution, generic meta-components are created with variation points marked as variables. Then, a specific instantiation of that meta-component can be generated using a specification file. This specification file allows for the insertion of statements, parameters, etc. making it much more robust than conventional parametric generics. However, this form of meta-component development and instantiation require the developer to have a very strong sense of the abstractions required to create a set of components and the foresight to know the differences between them. Understanding and anticipating non-parametric variations of components is likely to be a daunting task. Also, this method of refactoring adds an additional layer of abstraction separating the developer

from the source code which could be prohibitive during debugging of the generated source code or even the generation of the source code.

Baxter et al. propose the use of macros to eliminate clones [15]. These can be automatically generated by their tool CloneDr. This approach is rather limited as many languages do not support a macro language. Also macros can also be difficult to debug as they also result in generated code.

Balazinska et al. used Strategy [9] and Template method [10] design patterns to address cloning in Java programs. Using context analysis, the differences between clones are analyzed and in some cases can be automatically refactored using the design patterns. Balazinska et al. also propose another approach that allows computer-aided refactoring rather than automatic refactoring, allowing the user to choose the end method of removing the code clone. While this approach does effectively reduce the size of the code for certain types of clones, the complex design introduced by the refactoring may in fact negatively impact the overall design. In particular, the refactoring introduces an orthogonal set of classes to the original code specifically for handling the differences in the clones, separating the differences in each code clone from the original classes. While this in some ways alleviates the problems associated with discerning the differences between clones during maintenance, it introduces the decoupling of the problem specific details from the problem solution, possibly breaking conceptual models of the developers.

Toomim et al. proposed a method of managing clones through simultaneous editing, called *Linked Editing* [87]. In this approach, copied code can be persistently "linked". This allows users to easily view common segments of code and consistently edit code clones. The linked editor aids the user in propagating changes to clones across all linked code clones. Propagating changes through semi-automated mechanisms rather than those that are fully-automated allows developers to have the flexibility of modifying code as it is applied and provides context in which the developer can make appropriate decisions about the effects of the changes on other clones. Important features include real-time explicit display of differences in clones, selective simultaneous editing and change propagation, and clone removal/introductions support. A major drawback to this work is it requires a central repository of linked code for distributed development environments or additional mechanisms to update the linking database during code check-ins and updates. Corporate

settings would add additional challenges as code ownership may affect how change propagation can take place. Through user studies, Toomim et al. found that linked editing took only 2% of the time needed to form an abstraction to reduce code clones. All users in the study group ranked linked editing above abstraction in all areas of maintenance: maintainability, understandability, changeability, editing speed, and editing effort [87].

Duala-Ekoko et al. also use linked editing to maintain clones within a software system [24]. Similar to Toomim et al. they link segments of code as clones, but in their approach they propose the use of clone region descriptions (CRDs) rather than line numbers to locate clones in subsequent versions of the system. In this way, the limitations of line number links in a distributed development environment are overcome. Using the file and class name, method signature, and a block signature they can search the AST of subsequent versions of the system for the clone segments. If the file or class name changes they search the class space for a similar class. Once the appropriate method is found, the code block containing the clone is located using the block signature which is composed of an identifier string, block type, nesting level, and a corroboration metric. In the case where several methods or blocks may be possible matches the corroboration metric is used as further evidence for selecting the appropriate match. In their implementation, they chose to use cyclomatic complexity as a corroboration metric. On a very small test set they have shown this approach can work for locating the clone segments across several versions of a software system. This approach and its solution is very similar to the origin analysis problem described by Godfrey and Zou [34].

Software product-line engineering is a related field of study [20]. The purpose of software product-line engineering is to enable rapid, efficient instantiation of distinct products based on a core set of components implementing common functionality. One goal of software product-line engineering is maximizing reuse and minimizing duplication of code. Creating a new product is a problem of configuration and customization. Variability is managed through variation points whose role is to provide specific points of extension to the base component, with direct analogies to variation points in XVCL. One problem with managing code clones is managing variations over time. Software product-lines deal with several dimensions of variability over time [67], but these solutions are intended for the end result of a product-line (i.e. a set of distinct products). Code clones may need to vary independently

within the same versioning system and branches, and changes to the shared code base may make this difficult as this results in forced change propagation. Software product-lines engineering has also discussed the tradeoffs of generalizing components early and anticipating many requirements before they are needed (proactive software product-lines) versus refactoring only once a clear understanding of shared core requirements in a product-line or component is attained (reactive software product-lines) [18]. Buhrdorf et al. present a case study of a software company implementing the reactive model of software product-line development. In their study they noted that forming generalized solutions early has a higher overhead than waiting for clear requirements to evolve into the system. By waiting for a fuller understanding of the commonalities and requirements of similar components, effort is not wasted on code that will never be needed or code that has not captured the full set of requirements completely. This has direct analogies to avoiding code clones in early stages of development which can lead to evolving a complex abstraction without understanding what the end requirements can be. Toomim et al. have shown that managing abstractions may lead to significant development overheads [87], and managing them too early will increase the cost of software extension.

2.8 Code Cloning Case Studies

2.8.1 Cloning in Software

Kamiya et al. [47] performed tests on the Java Development Kit (JDK) version 1.3.0 to search for clones within the system, and also studied the cloning behaviour between Linux 2.4.0, FreeBSD 4.0, and NetBSD 1.5. While they observed that clones in the JDK seem to occur in nearby directories or files based on a visual inspection of the scatter plot their tool presents, no quantitative data analysis is discussed concerning this point.

Balazinska et al. measured the refactoring opportunities of cloning. Balazinska et al. measured the types of differences between clone classes in five Java applications using the classification scheme shown in Table 2.2 [8]. From their study they found that the largest group of clones is typically *identical* clones followed by *one long difference, interface and implementation*, *called method* and *global variable*. This contribution is important as

it indicates that while there are many clones that are exact copies, a large portion of them have had subsequent non-trivial changes. This has implications on refactoring and detection results. Balazinska et al. further measured the refactoring opportunities of clones by measuring their coupling to their local context [10]. In this study they analyzed JDK version 1.1.5. They found that while most code clones do not contain context dependent operations or dependencies, there is a large portion of clones that do. This analysis is important from a refactoring perspective: when refactoring clones, those clones without context dependent operations will present fewer challenges than those that do. Quantitative results from this study cannot be generalized as only one system was examined, but it does introduce the importance of understanding difference in clones when measuring or estimating the cost of refactoring software systems. Another key issue with this study is that there is no reporting of the accuracy of the clone methodology used. While the authors report they did not consider methods smaller than six lines because of the high rate of false positives, it is difficult to evaluate the applicability of the results without further analysis.

Balazinska et al. produced an automatic refactoring system based on their analysis of differences in clones [9]. A major shortcoming of these studies on cloning is that there is little or no qualitative investigation into the clones themselves. The main assumption is that cloning is harmful and should be removed. Without considering the rationale behind the cloning this assumption can be harmful to the overall structure of the software system.

Basit et al. have studied the deficiencies of using generics to eliminate clones [13, 14]. Studying the cloning in the Java buffer libraries and the C++ Standard Template Libraries, Basit et al. found many situations where the generics supported by the language were not flexible enough to remove many forms of cloning. For example, non-parametric clones, such as in the cases where methods are added are removed from a class, cannot be supported by Java generics. Non-type parametric changes are also not typically supported; for example, constants or keyword modifiers. Additionally, Basit et al. noticed that Java has type restrictions on their generics. In Java, primitive types cannot be used in parameterization.

Rajapakse and Jarzabek have studied the use of cloning in web applications [78, 79]. Modern web applications are developed in a broad range of programming languages. In fact it is commonplace for a single web application to contain code in several programming languages. In their studies, they found that web applications contain a high level of code

cloning [78]. Only one of 17 web applications analyzed contained less than 20% of the LOC in a code clone, and the average rate of code cloning found was 41%. These results are somewhat higher than those reported for traditional applications [78]. Rajapakse and Jarzabek measured the effective cloning in web-specific languages compared to the general purpose languages that were found in the web applications. In general the web-specific languages contained more cloning than the general purpose languages but the general purpose languages still exhibited high cloning levels. They concluded the amount of cloning in web applications is higher than that of traditional applications. By measuring the degree of cloning over subsequent releases of several web applications Rajapakse and Jarzabek found that the percentage of lines cloned increases over time.

Continuing their work, Rajapakse and Jarzabek performed a case study on the tradeoffs of using server pages to reduce clones in web applications [79]. In their study, they initially built a web application based on the requirements of an industrial partner and then proceeded through two stages of clone elimination. The goal of the initial implementation was to produce a working web application that is representative of a project developed under industrial time pressure. Maintainability concerns, including reducing the amount of cloned code, had a low priority. They then proceeded to unify clones within modules using design patterns. In the final stage they unified the code clones amongst the modules, reducing six modules to one. The overall reduction in size from the initial implementation was 75%. This large reduction in code size is largely attributed to how the system was initially built and extended. New modules were added by cloning and extending functionality from existing modules. This enabled rapid development of the initial working system. Several important tradeoffs that concern software development in general were noted. First, performance was greatly affected by the reduction of code clones: the final reduced code base ran up to three times slower than the initial implementation. Platform and framework conformance also became an issue when removing clones; Rajapakse and Jarzabek provided two examples where removing clones either could not be done or the framework itself had to be modified to remove cloning. They also note that the effort spent on avoiding using code clones will slow the development process, a costly sacrifice in the competitive market of web services. They also note that the evolvability of a web application is adversely affected by clone unification. In their case study, six modules were

unified into one. If an extension or change in functionality is needed for one of the core functionalities, the entire web application needs to be re-tested. Source code packaging can also be negatively affected by removing clones using generalized functions that contain control branches for specific instances of a problem. When repackaging the source code to distribute portions of the overall system, unnecessary code is inadvertently distributed. Later, when updates to the code base are rolled out, systems may be temporarily brought down in order to update code that is in fact unused.

2.8.2 Copy-and-paste

Kim et al. studied how developers used copy-and-paste features of the Eclipse IDE [57]. In this study, they noted that developers often use copy-and-paste to structure and guide the task of extending a software system. For example, they noted that developers will sometimes copy a parent or sibling class to use as a template for writing a new sibling class. They also observed that developers used copy-and-paste to duplicate control structures patterns.

Balint et al. have also studied how developers copy-and-paste code [11]. In their study they focused on analyzing who creates clones and what their relationship is to the original code. They describe five patterns of activity demonstrating how clones are created and maintained. The first pattern is the case where an author creates a family of clones from his/her own code and consistently maintains this code over time. This was considered the least harmful case for cloning. A similar case is where multiple developers clone the same segment of code without modifying it. This case was considered to be harmful because of the risk of future maintenance issues. The third case is when a clone or set of clones is maintained by several authors. In this case individual authors propagate their own changes across all clones in a clone class. This scenario usually involves only a few developers. The final two cases of cloning they observed were inconsistent fixes by a single or multiple authors. In these cases, changes to a segment of code in a clone class are not propagated immediately. In some cases the author introducing the change eventually propagates the change to all of the clones, in other cases these changes are propagated by other authors. This study reveals interesting insights into how clones are managed. However, it only observes how clones are managed and does not consider how the clones are used. Balint

et al. argue that these patterns of cloning are all harmful essentially with the argument that updates may be made inconsistently, leading to bugs or future maintenance problems relating to diverging code clones. One could also argue that with appropriate tool support, such as linked editing, these risks could largely be mitigated.

2.8.3 Evolution

Clone detection case studies have been performed on the Linux kernel [3, 4, 32]. A preliminary investigation of cloning among Linux SCSI drivers was performed [33]. Casazza et al. use metrics-based clone detection to detect cloned functions within the Linux kernel [4]. They performed analysis across the major subsystems, and then on the architecture-dependent code of the memory management subsystem and the kernel core. To evaluate the degree to which cloning occurs, they define a common ratio between two files, which is the percentage of functions in one file that are cloned in another with respect to the number of functions in the first. As noted by Antoniol et al., this common ratio must be used with great care and absolute values should be also be acknowledged, as large disparities in file sizes can make the ratio misleading [3]. The conclusions of this study were that in general the addition of similar subsystems was done through code reuse rather than code cloning, and more recently introduced subsystems tended to have more cloning activity. Antoniol et al. performed a similar study [3], evaluating the evolution of code cloning in the Linux kernel. They too used function metrics clone detection as their detection method and their conclusions were similar, adding that the structure of the Linux kernel did not appear to be degrading due to code cloning activity. The above studies analyzed only clones meeting the *DistinctName* criteria shown in Table 2.1. In our own studies, we found these matching criteria were insufficient for finding function clones as its restrictions are too rigid to detect clones with non-trivial modifications [50]. To detect function clones, we used the following set of metrics:

1. Line counts: total number of lines, blank lines, lines of code, declaration lines, lines of executable code, commented lines.
2. Number of parameters, number of global variables used.
3. Number of parameters or global variables modified.

4. Cyclomatic complexity.

5. Maximum level of nesting.

The metrics we chose are slightly different from those used in other studies [61, 74] but as stated by Antoniol et al., in large systems the choice of metrics does not significantly affect the results [3]. We have used a subset of metrics used in previous work [61, 74], and we expected that our returned pairs would have low precision but high recall as the number of distinguishing data points was lower. Manual inspection of several hundred of the clone pairs revealed that false matches were rare in medium-sized clones, confirming that the choice of metrics did not seem to affect the results and that metrics-based clone detection can be a precise method for detecting clones. However, in the study we demonstrated that clone detection using the *DistinctName* criteria, while being precise for a small range of clones, had disappointing recall for large clones compared to function clones found by parametric string matching. We concluded that the cloning described by Antoniol et al. [3] and Casazza et al. [4] likely includes only copy-and-paste cloning with little or no modifications, and therefore is at best a lower bound of the actual cloning in the Linux kernel.

Recent research has suggested that refactoring clones is not a major concern in the maintenance process. While studying the use of refactorings in the evolution of Tomcat, Rysselberghe and Demeyer compared the use of move method refactorings for removing code clones and encapsulating similar functionality [82]. Their findings show that developers are much more concerned with grouping functionality than removing cloned code.

Kim et al. studied the evolution of code clones over time [58]. Using CCFinder, described above, they detected clones across several versions of two small software systems ($\leq$ 20KLOC). They grouped clones into clone classes and measured how often they were changed together over a series of CVS checkins. They found that in many cases, clones remained in the source code for only a few days, and that many long-lived clones could not easily be refactored for a variety of reasons: standard refactoring techniques could not be used, changes to the design would be required, or because of programming language limitations. They concluded that aggressive refactoring of clones was not always the correct management decision for cloning and that many clones cannot be refactored. They also suggest other maintenance techniques should be considered for this class of clones.

While this work does provide insight into the evolution of clones, there are several serious drawbacks to their approach. Clones that are split because of inconsistent additions or changes in the middle of the cloned code can cause the clone to no longer be detectable, effectively disappearing from view. If changes in subsequent CVS checkins cause the clones to re-emerge as a single clone class, their analysis would view these clones as new entities added to the clone class. Also, the method used is prone to a high number of false positives. Our work differs in that we consider a single version of a public release of a software system. Therefore we do not consider short-lived clones, likely only to be part of a developer's intermediate product, in this work or our classification. We only analyze clones that have become integrated into the system. Also, we evaluate clones based on their harmfulness or helpfulness in maintaining software systems. This rating of the harmfulness is not limited to the feasibility of refactoring the clones, although it is one part of our criteria. Rather, we investigate how clones are actually used from an architectural perspective.

2.8.4 Bugs

Li et al. have proposed an approach to detect copy-and-paste related bugs in source code using the clones uncovered by CP-Miner [70]. They do this by searching for clones with inconsistent renaming of variables. Inconsistent renaming could be an indication that the cloned code has not been modified or updated correctly. In their study of Linux, FreeBSD, Apache httpd, and PostgreSQL, Li et al. detected 421, 443, 17, and 74 possible copy-and-paste related bugs. Of those possible bugs, few were verifiable bugs, shown in Table 2.3. For Linux and FreeBSD 21 and 8 potential bugs respectively were deemed careless programming. The rest of the potential bugs were false alarms. Their results show 99.9% (or more) of code cloning is likely to have been reused correctly by their definition. While Li et al. interpret these results as evidence that copy-and-paste activity is risky, we see it as no more risky than adding a single line of code. If we accept that the industry average bugs per line is approximately 15-20 per KLOC [75], adding a single line of code has a 1.5% chance of introducing a bug. By comparison, the results found by Li et al. indicate 0.1% of modified code clones contain bugs.

System	Detected Clones	Possible bugs	Verifiable Bugs
Linux	122,282	421	28
FreeBSD	101,699	443	23
Apache	4,155	17	5
PostgreSQL	12,105	74	2

Table 2.3: Potential copy and paste bugs found by CP-Miner

2.8.5 Sharing Knowledge

Several studies have examined the reuse of code across software systems [2, 47, 71]. Kamiya et al. analyzed cloning across three popular open source operating systems: Linux, NetBSD, and FreeBSD. In their study, they found that less than 1% of the code was common between Linux, FreeBSD, and NetBSD [47]. In contrast, FreeBSD and NetBSD share 20% of their code. This is not a surprising result as FreeBSD and NetBSD originate from the same code base and have subsequently diverged.

Al-Ekram et al. (including the author) performed an in-depth study of cloning across software systems in two application domains to investigate knowledge sharing in open source software [2]. We found that most cloning was "Cloning by Accident"; these clones are fragments of code that appear to be very similar not because they are intentional code clones but rather because of the inherent form of the solution to a problem. For text editors, most clones across software systems involved GUI code, specifically the setup of the interface. Accidental clones across X window managers involved initialization of structures to interact with the X environment. There were few examples of actual code sharing between systems. Several of the text editors share a regular expression library. Two window managers shared a file for gradients, and another pair shared a specific file for animating the resizing of windows. This study demonstrated that sharing of knowledge in open source software is often not done through sharing of source code but rather by writing code that uses the same library APIs. The findings support the results of Kamiya et al.. In the study, only 0.5% of the total code was shared amongst text editors and 0.2% of the code was shared amongst window managers.

Livieri et al. studied cloning across all software systems comprising the FreeBSD dis-

tribution [71]. Their results further support the results of the previous two studies. Most clones were the result of multiple versions of the same software system. An example of this is Apache httpd 1.3 and 2.0 being included as part of the distribution. FreeBSD supports both in order accommodate users of each branch of the web server. Other clones across software projects occur because of the inclusion of entire source trees of library source, such as php4 and php5. When comparing FreeBSD to a single internally developed software project, SPARS-J, Livieri et al. found many clones related to a file for parsing command line options, *getopt.c*. This instance of code cloning is similar to the regular expression libraries we detected [2]. They also found that the project reused code from two projects in FreeBSD for handling CGI scripts. This is evidence that some code sharing does occur through open source software.

2.9 Summary

This chapter has described the state of the art in software cloning research. We have described the many different approaches for detecting clones. We have also discussed proposed approaches to analyzing and managing clones in software. Finally, we summarized previous studies investigating cloning in software.

While there has been recent quantitative study of code cloning in software, little work has qualitatively investigated how developers use clones as a form of reuse. Section 2.2 proposes several plausible answers to the first question of this thesis — What are the common motivations for developers to use code clones? — but all of these are anecdotal, leaving all three questions of this thesis open to further research. In particular, the largest and most pressing problem in current code clone literature is the contention that code clones in software pose serious maintenance problems. While it seems clear that code clones can sometimes pose maintenance problems, research has not established that cloning is generally a bad practice. Indeed, our experience suggests that the opposite is true: code cloning can sometimes be used advantageously within a disciplined software engineering development process.

In an effort to understand how code clones are used in software, Question 2 of this thesis, the copy-and-paste mechanisms of development environments have been monitored.

The results of that work provide insight into only a single mechanism for creating code clones, ignoring other mechanisms such as file copying or visually copying code through character-by-character input. The study of the evolution of code clones also lacks a well rounded view of how developers actually use code clones, providing no insight into the motivations of the developers.

Finally, no prior work has addressed the issue of investigating or rating the appropriateness of existing code clones in software, Question 3 of this thesis. Correlating code clones to bugs has shown only that a small percentage of code clones contain bugs due to incorrect or incomplete modifications of copied code. These results suggest that we need better tools to create and manage code clones, but does not indicate their actual use is inappropriate.

Empirically investigating these questions will help to evaluate the contention that all cloning is harmful, and will provide insights into the tools needed to find and manage code clones. Understanding how clones are used as part of software will also provide deeper insight into how developers create source code, valuable information that can be used to direct software engineering research in a direction to better enable software developers in processes that are natural and efficient.

Chapter 3

Questions and Methodology

3.1 Introduction

In the previous chapter we discussed the open questions regarding code clones within software systems. In particular, how and why code clones are used in software systems is not well understood. Consider the following scenario: You are the manager of a large software project and one of your quality assurance personnel informs you that 25% of the lines of code in a core subsystem are cloned code. What should your reaction be? Is this bad? Is the subsystem in question on the road to skyrocketing maintenance costs? How can you evaluate this report?

To understand the value of the many metrics reported by the clone detection and analysis literature we must first understand what code clones really are and how they are used in software systems. Currently, we have little understanding about how and why code clones are created, used, and maintained in industrial software. The lack of this base knowledge severely limits our capability to analyze, evaluate, and discuss the reported results. Ultimately, this limits our understanding of maintenance challenges and benefits created by code clones in software.

This chapter describes the research questions that we have investigated for this thesis and the analysis tools that we have developed to answer these questions. This chapter is organized as follows: Section 3.2 discusses the research questions addressed in this thesis, Section 3.3 describes the research methodology of this thesis, Section 3.4 describes the main

study subjects we used to build our understanding of code clones, Section 3.5 describes the code clone detection method used to collect candidate code clones, Section 3.6 describes the post-processing performed on the candidate code clones, Section 3.7 describes the code clone analysis tools used for the investigations presented in this thesis and a set of requirements for clone analysis tools, and Section 3.8 summarizes our overall research methodology.

3.2 Research Questions

As discussed in Chapter 2 there have been several studies that attempt to measure qualitative properties of code clones and code clone evolution. However, for the most part these studies do not provide a complete picture as to how code clones are used within the various software systems. Section 2.2 lists many reasons cited as to why developers clone code, but no study has attempted to evaluate the relevance or accuracy of these intuitive but unconfirmed anecdotes. Work concerning the evolution of clones in software has generally measured the number of clones or how they change [3, 4, 58]. For example, while Kim et al. have investigated how clones change over time, their study focused on how clones were updated and did not address the specific details of how cloning was being used [58]. In their studies of cloning in the Linux kernel Antoniol et al. and Casazza et al. reported the total number of clones in the code base, but did not investigate the types of code clones that were actually found [3, 4].

Investigations into code cloning in software systems have generally been limited in scope. Balazinska et al. focused mostly on measuring the similarity of clones found. Basit et al. noted that generics in programming languages are typically insufficient for abstracting clones, but this is only one type of situation where cloning is necessary. Rajapakse and Jarzabek studied cloning in a particular application domain, web services, and showed that the rate of cloning is typically higher than in traditional applications. This observation provides little insight into why the developers create code clones, although there is some indication that the code cloning is related to the choice of implementation language.

Kim et al. studied how developers copy and paste, but aside from semantic templates, there is little insight into the rationale of why developers introduce code clones into source

code. Balint et al. describe how clones are maintained, but this again does not tell us how clones are being used. Understanding how developers maintain clones is an important topic of research but is only one part of a larger picture. The evaluation of the maintenance challenges is an aspect of code cloning that is important for the creation of tools and techniques to efficiently develop and maintain high quality code. However, this understanding describes only the costs of cloning, it does not provide information about the benefits. This somewhat one-sided view of the problem biases research and leads to statements such as:

> *Many such duplications can be attributed to poor programming practice since programmers often copy-paste code to quickly duplicate functionality.* [42]

Such a statement makes negative implications about code cloning activity, not to mention developer work ethic. Is such a statement justified? Are code clones part of *"poor programming practice"*? With uncertainty about the conventional wisdom that cloning is harmful as well as the ever present questions about what cloning really looks like in software systems (motivated in part by scepticism expressed by industry contacts), our initial research question was formed:

> *How and why is code cloning used in industrial software systems?*

In other words, the current literature does not contain a deep analysis of how clones are used in software. Our initial investigations and case studies of code cloning were motivated by the above question. As we analyzed and compared code clones in several software systems, three research questions arose:

Question 1 *What are the common motivations for developers to use code clones?*

If code clones are the result of lazy programming then we would expect that the resulting code clones are likely to have negative impacts on the system. However, if developers intentionally create code clones for engineering reasons such as mitigating risk, supporting flexible evolution, or avoiding fragmented code within the system, then code cloning may have a positive impact on the system quality.

Whether developers are intentionally using cloning as a conscious design decision or to avoid short term cognitive costs we must ask: in what scenarios are they using code

cloning? Are there regular or repeated patterns of use of code clones, both negative and positive, that can be discovered? This raises our second research question:

Question 2 *Are there common patterns of code cloning that occur in the development of software systems?*

Discovering common patterns of cloning can aid in several aspects of clone detection research. Understanding the types of cloning that occur in software systems provides a better understanding towards the various types of maintenance challenges they present. For example, clones that are likely to be subject to different evolutionary forces, such as platform dependencies, are likely to require careful consideration when propagating changes. More concretely, bug fixes must be carefully considered. Bugs related to communicating with a specific platform are unlikely to appear throughout a set of code clones. However, bugs related to the interaction with the internal software system may be present in all or some of the code clones. In this case, the type of software bug and the type of code clone will dictate the maintenance tasks required.

If patterns can be discovered, can we measure the relative frequency with which they occur? As with any design practice, context will determine the appropriate actions. What can we learn about the harmfulness of cloning in software in relation to code cloning patterns?

Question 3 *How are code cloning patterns used in practice, and to what extent is their use appropriate?*

Measuring the frequency or extent to which code cloning patterns appear in software will provide an indication of the relevance of the clone patterns to the reality of software development and maintenance. Investigating this question also leads to an understanding of how difficult it is to use code cloning in a positive way. If we discover that code clones are repeatedly used with good intent but lead to poor design, over time this would be an indication that the correct use of code cloning is difficult and code cloning should avoided.

3.3 Research Overview

Answering the initial research question requires a thorough, qualitative understanding of code cloning as it occurs within a realistic development environment. Specifically, we must investigate code cloning in source code that is of a reasonable size and has been developed with industry-level quality in mind. For such an investigation, we require:

1. a mixed qualitative/quantitative research methodology,
2. appropriate study subjects,
3. the ability to detect clones, and
4. the ability to analyze and comprehend clones from a design and architectural perspective.

The following sections describe how each of these criteria were met. In particular, we outline the availability of case study subjects, detections tools, and analysis tools. To start, however, we begin with a description of the qualitative research methodology we used to answer the questions in this thesis.

3.3.1 Study Overview and Methodology

The method used to study code clones in this thesis was a series of case studies involving manual analysis of candidate code clones detected in open source software systems. This does not follow the more common research approach of statistical hypothesis testing. We chose this approach because the goal of this research is to generate a deeper understanding of how code clones are used in software, something we do not feel is well addressed by hypothesis testing. In the hypothesis testing approach, one might form the null hypothesis that code cloning is generally used in a principled way. Then we could empirically test this by evaluating a large set of clones, and if we find that within a sufficiently large sample of clones there is a statistically significant number of clones used in an unprincipled way, we could reject the hypothesis that clones are used in a principled way. This finding would provide evidence that not all code clones are produced for justifiable reasons, but would

not tell us anything about how the code clones are actually used, or if there are some clones that are used in a principled way. In the unlikely event that we could not reject this hypothesis, it would not tell us that the hypothesis is correct (that we could accept it), only that our sample could not falsify it so therefore the hypothesis could not be rejected. With either result, little information about actual uses of cloning in software is uncovered.

There are other shortcomings with using hypothesis testing to answer the initial question raised in this thesis. An acceptable experiment to test the hypothesis would require a large, accurate sample to analyze. This would require a precise clone detection tool with high recall, something that does not currently exist. One could use a tool with high recall and manually filter the code clones but any rejection of a hypothesis using this subjective filtering would be highly questionable, as there is no generally accepted definition of what comprises a code clone. Additionally, no tools prior to this work would have provided sufficient mechanisms to evaluate clones within the software context that they exist, further hampering the credibility of any null hypothesis testing.

Another problem with hypothesis testing is that with little understanding of the phenomenon to be studied, it is difficult to form a reasonable null hypothesis to test. This is a problem because if the hypothesis has no basis on the underlying phenomenon, rejecting it will only help you understand the phenomenon in small increments. For these reasons, we have chosen to study the phenomenon of code cloning in detail with the goal of generating accurate descriptions of code cloning that are based on actual instances within software (rather than intuition and general experience with code cloning that are often used to motivate code cloning as a serious problem in software). In the future, these descriptions may provide some of the required information for constructing useful experiments.

Case study research, as a methodology oriented toward developing an accurate view of a phenomenon, can use a combination of quantitative and qualitative data, which is a requirement of our research goals. Question 1 aims to produce qualitative evidence about how cloning is used, and Questions 2 and 3 require both qualitative and quantitative evidence to answer them. In practice, case studies can be used to achieve five different goals [95]:

1. explain presumed causal links of real life phenomena,

2. describe a real life phenomena,

3. illustrate specific aspects of a phenomena,

4. explore a phenomenon that is not well understood, and

5. meta-evaluation (study of an evaluation).

Of the five possible goals of case studies, (2) description and (4) exploration of a phenomena are directly related to the questions posed in this thesis. Prior literature does not present enough evidence to aid in understanding how code clones are used, making our initial case studies exploratory in nature. With the goal of documenting how code clones are used in real software systems, the case studies leading to the results of this thesis were always of a descriptive nature. The overall research methodology used for this thesis is a multi-case holistic case study composed of exploratory single-case and multi-case, holistic case studies and a multi-case descriptive case study (described in Chapter 5).

To answer Question 1 and understand code cloning during initial investigations, the exploratory case studies leading to the results presented in this thesis had a dual purpose: *(1)* to build an understanding of code clones within software and *(2)* to understand and address the problem of analyzing code clones in software. Our initial case study was a holistic single subject case study [49, 50]. The unit of study was a subsystem of a larger software system. The experience of this case study lead to refined data collection and analysis tools (described below), subsequently used in a holistic multiple subject case study [51]. The unit of study in the next work was a subsystem of a software system and a whole software system. Both studies were motivated by Question 1 with no propositions regarding the nature of code cloning. The main results of these three initial case studies are the identification of specific instances of cloning and an understanding of the relationship of these instances of cloning to the software as a whole. Cloning rationales were identified and explained according to the context of code cloning within the software system. Key distinguishing properties of code clones, such as the location of the code segments in the software and the particular code involved, were identified. Later on, this information contributed to the formulation of the patterns detailed in Chapter 4.

Following these initial case studies we produced a tool to facilitate more effective in-depth analysis of a software system. This tool was applied in a holistic single subject case study with the goal of characterizing how code cloning was used in the context of

the whole software system [52, 54]. No propositions about the nature of cloning in the study subject were made beforehand. We identified and described code cloning at various levels of abstraction, including the general use of cloning across subsystems and specific instances of cloning across subsystems, within subsystems, and within files. The rationales of the code clones were inferred and documented based on the consideration of source code and design documentation, clone locality (the location of the segments in the source code), code similarity, and number of related clones. The results of this study were a deeper, more refined analysis of code cloning in software. In both this case study and prior case studies we had identified instances of "harmful" and "good" code cloning.

With a deeper understanding of code cloning, we discovered recurring patterns of code cloning in our study subjects. Through iterative, comparative analysis of code clones in the previous study subjects we produced a list of patterns, described in Chapter 4. This list of patterns was then verified through a descriptive case study, described in Chapter 5. The study was a multi-subject, holistic case study. The propositions of the final case study were that the code cloning patterns exist generally in software and that code clones are not always harmful. The results of this study, described in detail in Chapter 5, confirmed both propositions.

3.4 Study subjects

To study how code clones are used in software systems we must study cloning in software that is representative of much of the source code produced for industrial use. To maximize the impact and applicability of the results of our investigations, it is important that the study subjects are *typical* software development projects. There are several aspects one should evaluate when selecting a study subject that represents typical software. In particular, we must consider:

1. if the software was developed in a commonly used development language,
2. if the software was developed with a quality level typical of industry standards,
3. if it is a widely adopted technology or if is satisfies a relatively small niche,

4. if the software is of non-trivial size, and

5. if the source code is publicly available.

We can determine what programming languages are commonly used by referring to publicly available measures of language popularity: development languages commonly used in industrial settings include C, C++, and Java according to the TIOBE Programming Community Index [85]. Studying software systems that use popular programming languages provides us with a large selection of candidates to choose from and ensures our results reflect the current state of software development. Determining a common development methodology is more difficult, especially with the many variations used in development today (eXtreme Programming, agile software development, waterfall, big design up front, etc.). We have chosen to ignore this aspect of software development not only because of the difficulty of determining what methodologies are common but also because we feel that programming language and team dynamics will play a much larger role in how code is reused.

Selecting study subjects that meet industry-level standards is a subjective decision, largely because what is considered to be high quality software is not clearly defined and because rating the quality of open source software is also likely to be subjective. Instead we will consider software systems that have been widely adopted to be high quality under the premise that in most cases only high quality software systems will be embraced by the community. Other metrics for quality could be used to measure the source code quality of the software system; for example, bug reports, number of bugs per line, average complexity, and the ratio of comments to code can all be used to measure various aspects of code quality. This fails to measure whether a given software system meets industry quality standards, however, because data concerning generally accepted values for these metrics is mostly unavailable. According to obvious market mechanisms we would expect high quality software to move rapidly into widespread use. The advantage of using this criterion is that it indirectly measures quality in terms of what is important to the user.

Source code size is important to consider as small software projects are not likely to have the same maintenance constraints and pressures as larger ones. For this reason, we wish to consider software projects of at least 100,000 lines of code. We have found that projects smaller than this often have little or no organized software architecture, which

would inhibit the assessment of cloning on the high-level design of the system. Larger projects, on the other hand, tend to have a more deliberate source code organization, often with subsystems corresponding to directories in the file-system.

Reproducibility of the study must also be considered. Key factors in this are the accessibility of software systems and the availability of results, and so for this thesis, only open source software systems will be studied. In recent studies we have investigated cloning in four open source systems: the file-system subsystem of the Linux operating system kernel [50], the Apache httpd web server [52, 54], the Gnumeric spread sheet application, and the PostgreSQL database management system [51]. In more detail:

- The Linux kernel file-system subsystem is the portion of the Linux kernel that supports file I/O. It implements support for 42 file system formats that are managed through a Virtual Filesystem Switch. It consists of 280,177 LOC with 537 source files. The version of the Linux kernel studied was 2.4.19. The Linux kernel is one of the largest distributed software development projects in the world, with thousands of developers contributing changes and source code [66].

- Apache httpd is an open source web-server designed to run on a wide variety of platforms: BeOS, *BSD, Linux, Netware, OS/2, Unix, and MS-Windows. The core development team of Apache consists of approximately 25 developers who contribute a large majority of new features (88% of new code in 2000 [76]). As of version 2.2.4, Apache consists of 312,460 LOC across 783 files. Our initial studies analyze code clones in version 2.0.49 [52, 54] and the study described in Chapter 5 analyzes version 2.2.4.

- Gnumeric is an open source spreadsheet application, part of the GNOME Desktop environment. It supports a variety of platforms but this platform support is implemented in the libraries it links to (i.e., the various ports of GTK), rather than directly in the main body of the source code of Gnumeric. Using `svn blame` to measure who last modified all of the lines of code in Gnumeric in the last five years reveals that more than 88% of the LOC have been modified by only three developers. This finding is confirmed by documentation distributed with the source code. Gnumeric contains 326,895 LOC across 530 files. The version studied is 1.6.3.

- PostgreSQL is a multi-platform database management system written in C/C++. It has been actively extended and maintained since 1986. Currently the development team includes seven core developers, 24 major contributors, and 32 minor contributors [1]. The version used in the studies is 7.4.2. It consists of 543,387 LOC and 1097 source files.

3.5 Clone Detection

This section describes, in greater detail than Chapter 2, the clone detection tools we used during our case studies and the post-processing steps we took to refine the detection results. The clone detection method we used is parameterized token matching using suffix trees. The choice to use parameterized string matching is motivated by the consistent findings of its high recall [16, 19, 62, 81]. Recall is the most important consideration for this study as we are concerned with observing a large variety of clones rather than observing only true positives. Observing a large variety of clones provides more chances to observe otherwise unseen cloning scenarios. The method of analysis will be largely manual, making precision of automatic detection tools less of a concern. With manual inspection, we can identify and remove false positives from the data set as we analyze the software system.

While precision is of less concern to us than recall, there are several filters that can greatly improve the overall precision of the clone detection results without decreasing the recall, described in Chapter 2.5. In our initial case study we used the clone detection tool CCFinder and manually filtered results. To improve the tractability of the analysis, we introduced post-processing filters to refine the results of clone detection in areas of source code prone to false positives, including *structs*, and series of simple *if/else* statements. These filters were tuned to remove only clones that were false positives with a high probability. The tuning process was an iterative process of reducing the sensitivity of the filters until no true positives were removed from a sample set of clones. We found that these filters reduced the number of candidate code clones by up to 60% [51, 52, 54].

Parameterized string matching begins by creating an abstract representation of the source code. For each file of the source code, a tokenized string is produced and all identifiers that occur within the boundaries of a function are replaced with a generic placeholder,

```
int main () {
  printf ("Hello world!");
}
```

(a) snippet0.c

```
int mainClone () {
  printf("Hello Parallel World!");
}
```

(b) snippet1.c

Figure 3.1: An example source code snippets

```
P P ( ) { P ( P ) ; }
```

(a) transformed snippet0.c

```
P P ( ) { P ( P ) ; }
```

(b) transformed snippet1.c

Figure 3.2: Result of preprocessing snippets shown in Figure 3.1

such as *$P*. The resulting string, containing keywords, operators and separators, and place holders, is called a *p-string*. It is a representation of the underlying structure of the source code. In previous studies [51, 52, 54] we found that parameterizing tokens outside of functions leads to a high number of false positives that requires strict filtering on the resulting clone set. By not parameterizing these tokens in the pre-processing phase, most of the false positives of this type are avoided. This was the approach used in the case study detailed in Chapter 5. Figure 3.2 shows the result of preprocessing the two snippets of C code shown in Figure 3.1 (the reader should note that the representation is identical).

Using the *p-strings* as input, a generalized suffix tree is constructed and maximal repeats are then extracted from the tree. Suffix trees — trees that are used to store and access every suffix of an input string — are an efficient data structure used for searching strings

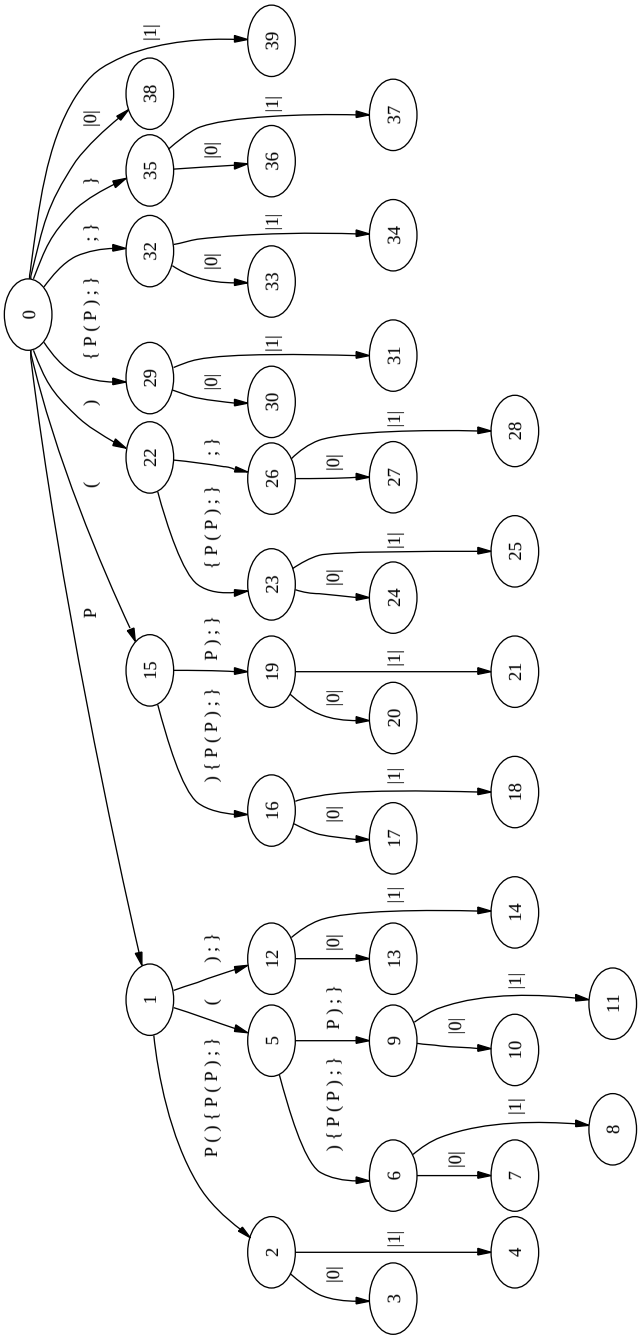

Figure 3.3: Hello world suffix tree

for sub-strings or locating recurring strings. They can be built in $O(n)$ time where n is the length of the string and they can be used to efficiently find sub-strings in the input text. From this data structure, sub-strings that exist in the input string can be extracted in $O(l)$ where l is the length of the desired text. The resulting suffix tree representing the *p-strings* found in Figure 3.2 is shown in Figure 3.3. Figure 3.3 depicts a suffix tree for multiple input strings, also known as a generalized suffix tree. When using generalized suffix trees, each input string is appended with a unique character that does not occur in the input text. In Figure 3.3 these unique characters are represented by |0| and |1|. The edges of the suffix tree represent a sub-string that occurs at least once in the input string. For example, the edge connecting nodes 1 and 2 represents the sub-string "$P(\{P(P);\}$". Edges along a path from the root to node i represent a prefix of two or more suffixes, and the node i represents the point where the remaining portion of the suffixes is different. For example, the path $0-1-5$ represents the string "$P($" which is the prefix of "$P(\{P(P);\}|0|$", "$P(\{P(P);\}|1|$", "$P(P);\}|0|$", and "$P(P);\}|1|$". From this example we immediately see the power of the suffix tree: by walking the edges from the root, we can extract all repeated sub-strings in the input strings. In this example we can see that "$P($" occurs four times in the text. To build suffix trees efficiently in memory, edges store the start index and length of a sub-string rather than the sub-string itself.

In the application of code clone detection, we wish to find maximal repeats: sub-strings with the property that if the sub-strings were extended one character to the right or left they will no longer be an exact match. Only those maximal repeats that are at least as long as a user-specified size are extracted from the tree. In this case, the minimum length is specified as the shortest sequence of tokens that can be considered a code clone. Maximal repeats can be found in $O(n+z)$ where z is the number of maximal repeats. For details on suffix trees and their uses refer to [36, 90].

The maximal repeats are then filtered by checking for an ordered one-to-one mapping between the identifiers of the two repeated strings. For example, *x = 1; f(x, y);* has an ordered one-to-one mapping with the string *a = 1; g(a, b);* but does not have an ordered one-to-one mapping with *b = 1; g(a, b);* even though the *p-string* representations of these three examples are identical. This enforces a stronger structural similarity between the two strings, eliminating many false positives from the results. Stepping through the string

starting from the first token of each string, the mapping is constructed. If a token that breaks the mapping is found then the maximal repeat is split, and the process is restarted at the point of the mismatch. Any sub-matches that are longer than the predefined minimum length are reported as clones found in the source code, and sub-matches that do not meet the minimum length are discarded.

The advantage of using this mapping rather than searching for exact matches is that copied code whose identifiers have been changed can still be detected. In practice, we have found that strictly enforcing this one-to-one mapping may cause the detection process to miss code clones that have not been systematically changed. To account for this, the CLICS clone detection tool allows for up to 7% (1 in 15) mismatches in a given sequence of code.

Finally, the detected matches are mapped from tokens to files, lines, and columns in the original source code. This information is output into a structured text file.

3.6 Post-processing

One disadvantage of using parameterized string matching for clone detection is that it produces a high number of false positives, even after enforcing the identifier mapping described above. In particular, sections of code with relatively little structural complexity do not contain enough distinguishing features to differentiate between them. Examples of these types of code are initialization lists, sequences of simple assignments, and *switch* statements. To improve the accuracy of our candidate code clones we filter the candidate code clone set by enforcing stricter requirements for a match in this type of code. In particular the following additional filters are applied to the clones [54]:

1. **Simple call filter**. Candidate code clones occurring on statements that are "simple function calls" can often contribute to many false positives when using parametric string matching algorithms. Regions of code that are "simple function calls" are sequences of code of the form

   `function_name(token [, token]*).`

   The criterion for a match is that 70% of the function names in either region must be similar. Two function names are similar when their edit distance, as computed by

the Levenshtein Distance algorithm, is less than half the length of the shortest of the two function names being compared. This threshold was determined by examining the edit difference of function calls in a sample of confirmed code clones. This sample was taken from a set of clones found in Apache, Gnumeric, and PostgreSQL that were primarily composed of function calls. In such cases, we found that the edit distance was generally less then 50% of the total length of the function names that were considered similar by the author. During our studies, we found that typical clones of a series of function calls would use mostly the same or similar functions, but did occasionally contain calls to completely unrelated functions. To accommodate this, we adjusted the percentage of function names that must match until true clones were not removed from the data set.

2. **Logical-structures filter**. We found that candidate code clones within simple logical structures such as *switch* statements are often false positives. To filter clones in these areas, we require that 50% of the tokens in these areas be identical and in the same order. Clones in simple *if-then-else* blocks are also filtered in this way. Initial values for this percentage match were found by analyzing cloning in these regions, and counting the number of tokens that remain unchanged in a true clone. We then tuned the filter by making it less strict until we found no true positives were removed from the data set.

3. **Overlap filter**. Candidate code clones whose two segments of code overlap by more than 30% of their length are also removed. This value was determined through observation of overlapping clones, and counting the maximum overlap of true clones. The value was then adjusted through several trials.

Prior to using our own detection tool, we used the parameterized string matching tool CCFinder. CCFinder parameterizes all source code, not just code contained within procedures, resulting in a large number of false positives in the somewhat structurally simple code of *structs* and *unions*. To filter the results from CCFinder we applied an additional filter to the candidate code clones:

4. **Non-function filter**. This filter operates on structs, union, type definitions, variables, and prototypes. If a candidate code clone occurs between a region of the

previously mentioned types and any other region of code, any clone in this relationship must have a minimum of 60% of its lines match exactly. These lines can be matched in any order.

It is important to note the goal when tuning these filters was to improve precision without impacting the recall of the clone detection tool. The constraints on these filters could be made stricter if the goal was to minimize false positives.

During this filtering step candidate code clones are also grouped by the regions of the source code they occur in. Regions are non-overlapping, contiguous lines of code grouped according to syntax. There are eight types of regions: consecutive type definitions, prototypes, and variables; individual macros, structs, unions, enumerations, and functions. Comments are ignored in the analysis. Regions are extracted using a modified version of the open source tool ctags that reports the start and end of important syntactic elements in the code, particularly: macro definitions, type definitions, prototypes, variables, structs, unions, enumerators, and functions. Each line of code in the system maps to a region (regions contain one or more lines). Code clones are split at region boundaries. For example, two identical files containing three procedures each would have three separate clones. This splitting of code clones is similar to that by Koschke et al. [62] and Kamiya et al. [47] where clones are split at the boundary of methods or procedures.

If two regions have cloning between them, we say they have a *cloning relationship*. For example, a code clone between two procedures forms a cloning relationship between the procedures. To help in clone analysis, for each pair of regions with a cloning relationship we group together all the clones between the pair, calling this a *Regional Group of Clones* (RGC). An RGC represents the cloning relationship strictly between two regions as code clones do not cross region boundaries in our analysis. For example, each identical statement shared in the procedures shown in Figure 3.4 might be considered a code clone, constituting six code clones forming a cloning relationship between the two procedures. These six code clones are grouped as a single RGC in our analysis. It is our experience that the concept of an RGC is useful for both visualizing and filtering clones. We have found that it is much easier to understand the context of a single clone when we can easily see all of the code clones in an RGC. In addition, it is often the case that multiple candidate code clones represent a single code cloning action. RGCs help visualize this relationship.

```
int logging_sums(int a, int b, int c, int d, int e)
{
    int sum = a;
    printlog("Sum = %d\n", sum);
    sum += b;
    printlog("Sum = %d\n", sum);
    sum += c;
    printlog("Sum = %d\n", sum);
    sum += d;
    printlog("Sum = %d\n", sum);
    sum += e;
    printlog("Sum = %d\n", sum);
    return sum;
}
```

(a)

```
int sums(int a, int b, int c, int d, int e)
{
    int sum = a;
    sum += b;
    sum += c;
    sum += d;
    sum += e;
    return sum;
}
```

(b)

Figure 3.4: Two procedures with six "cloned" lines grouped as one RGC

3.7 Clone Analysis

In this section we briefly outline our general criteria for a tool used to navigate and understand cloning in a software system. We then describe in more detail the features needed to meet these criteria. This set of criteria and features is derived from much manual work by the author in attempting to understanding cloning in software systems, starting with our studies of the Linux kernel file-system subsystem and the database server PostgreSQL [50, 51]. Some features were also taken from suggestions by students in a senior level graduate course who used the tool Gemini [89] and CLICS, the tool described in this thesis, to perform an analysis of code clones within the Linux kernel source code.

3.7.1 Criteria

The core challenge to the maintenance and management of cloning in software systems is comprehending the actual types of clones and the dependencies they create within the software system. To complete such a task, the code clones must first be detected, and then evaluated throughout the system at different levels of abstraction. However, clone detection tools can return large result sets and viewing every possible clone is generally infeasible. To address this problem, tools and processes need to be developed to help guide the software maintainer (or code clone researcher) toward the information they require to complete their task. We consider that any tool designed to help navigate and understand code cloning in a software system should provide:

1. facilities to evaluate the overall cloning situation,
2. mechanisms to guide users toward code clones that are most relevant to their task, and
3. methods for filtering and refining the analysis of the code clones.

Each of these criteria is described in more detail below.

Overall System Evaluation

As a first step in understanding code cloning within a software system, regardless of the end goal, maintainers must have a general understanding of the code cloning with respect to where code cloning is used and how often it is used. This understanding will allow the user to evaluate the extent and the severity of the code cloning to estimate the cost and/or necessity of the code clone analysis task.

Several mechanisms can be used to evaluate cloning from a high level. Visualization methods, such as scatter-plots [7, 26, 47, 80, 89], are useful for the discovery of highly related sub-systems and high levels of cloning within a subsystem. They are also useful for detecting unusual types of cloning, such as cloning from system libraries to other parts of the software system. Metric–oriented reports, such as reporting the percent of lines cloned, average length of the clone, *etc.* are useful for directing users to points in the system where the most cloning is occurring, or where cloning activities are unusually high in relation to subsystem size.

Guide and Empower the User

The potentially large sets of candidate code clones returned by the clone detection methods make it infeasible to look at each individual clone. For this reason it is important to provide both static views of candidate code clones as well as interactive query facilities. Metrics can be used to query the data set [38]. Some examples of metrics that might be used are the size of the clone, the types of changes made to the clone, and types of external dependencies a code segment has. Such a method can direct users to promising refactoring opportunities.

Other methods of querying the data set can also be used, such as querying based on the location of the clones in the software and the type of source code entity in which the clone exists. For example, a user might be concerned about cloning of macros originating in a particular file. Querying mechanisms provide flexible analysis, allowing users to leverage their own knowledge about the software and cloning, making the user more effective in their task.

Upon an initial survey of a software system, users may not be fully aware of what information they want or need. Query facilities can suffer from this weakness and strong

static analysis of the data set is also required. Static analysis should provide low-level metrics about cloning activities in the system. Additionally, the tool should provide a method of navigating through clones that leverages general knowledge about cloning. An example of this would be the categorization of clones as is done in [10, 51] and described in Chapter 2. This will provide a method of education for novice users, and guide experts more quickly to clones relevant to their task.

It is important to provide views that describe cloning in terms of the concrete architecture and source code organization. We believe that relating code cloning and architecture can have great benefits to comprehension of cloning. Cloning is a type of implicit architectural dependency, and as such can provide information about the high-level design of the system. This also enables users to use their own knowledge of the architecture of the system when evaluating clones (such as the appropriateness of code clones between two subsystems).

Analysis Refinement

Due to the subjective nature of the analysis of clones, from the perspective of the user there will always be candidate code clones that are not relevant to their role or current undertaking. For this reason, it is important that tools supporting the comprehension of cloning provide mechanisms to remove and filter clones from the analysis.

3.7.2 Meeting the Criteria For a Clone Navigation Tool

The following section summarizes the features that we implemented to meet the criteria we described for a clone comprehension tool. It is a proof–of–concept implementation. For a more detailed description and an example of its uses in a comprehensive study of code cloning in a software system please refer to [52].

Overall System Evaluation

To provide a general system overview, we compute a series of metrics encompassing several aspects of the system: system size, percentage of system cloning, and frequency of clone types. The metrics detailing system size include the number of files and LOC.

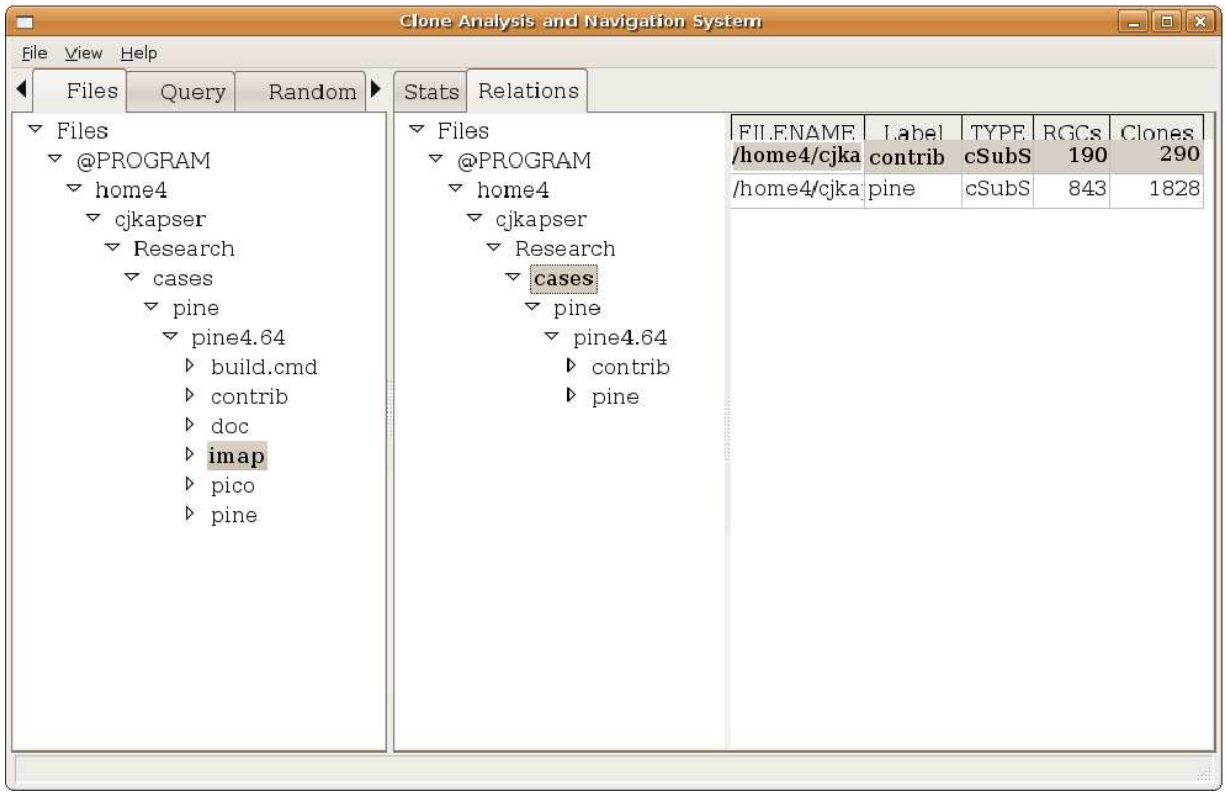

Figure 3.5: CLICS view of the relationship between *imap* and the two subsystems *contrib* and *pine* in the email client *Pine*.

Metrics describing the percentage of system cloning include the percentage of lines that have a clone, the percentage of functions containing a clone, and the percentage of files containing a clone. To describe the frequency of different types of clones in a software system, we list the the number of occurrences of each clone type in the taxonomy.

Guide and Empower the User

CLICS uses several mechanisms to enable the user to perform an in-depth analysis of clones in the system. These mechanisms include visualization of clone relationships between subsystems using a hierarchical containment graph, metrics for entities at all levels of

architectural abstraction, clone navigation through the taxonomy, clone navigation through the subsystem tree, and query facilities.

To visualize cloning as it relates to the system's architecture we use *LSEdit*, which is part of the architecture recovery toolkit, *SWAGKit* [86]. *LSEdit* is a graph visualization tool that is designed for the exploration of software "landscapes", which are graphs that represent software architectures and their relationships. The nodes of the graphs are software artifacts such as subsystems, files, and methods, and the edges of the graph are relationships between two software artifacts, in this case candidate code clones. Graph entities can be hierarchically contained, allowing varying levels of abstraction during analysis.

Complementary to the *LSEdit* visualization, navigation through the system architecture can also be done through the *system navigation tree*, shown in Figure 3.5. The structure of the tree models the subsystem containment hierarchy of the software. In addition to showing the degree of relationship between subsystems, as shown in Figure 3.5, metrics summarizing cloning within the software entity are also provided. For each software artifact within the selected entity, the following metrics are shown in the Stats tab (not shown in Figure 3.5):

- number of code clones involving only sub-entities within the referenced entity,
- number of code clones with one segment within the referenced entity, and one segment somewhere else in the software system,
- percentage of lines of code of the software system contained within the given entity, and
- percentage of total code clones that are involved with the referenced entity.

These metrics give the user information to quickly locate any cloning "hotspots", which are software entities (subsystems, files, methods, etc.) that contain a substantially larger portion of code clones than other entities near it. For example, during our Apache httpd case study we found that the *server* subsystem contains 38.8% of the overall clones in the system, but only has 17% of the lines of code [52, 54].

Using the clone taxonomy described in Appendix A, users can explore the clones in the system by code clone type using the *clone type navigation tree*. Users can view clones in each category, and remove any clones they believe to be false positives. This method of navigation is especially useful when performing the initial analysis of clones in the system as it can provide insight into what types clones are most frequent within the software system. Furthermore, inexperienced users can use this navigation method as a way to become familiar with the clone classifications. This navigation tree is also used to sort the results from the queries described below.

Visualization of a code clone pair is shown in Figure 3.6. On the right we see two segments of source code. These are code clones that occur within two procedures. To make the scope of the regions the code clone is found in more evident, the procedure bodies are highlighted with red text. In this figure, we have selected to show the differences of the two regions rather than just the detected code clone. The common tokens and differing tokens are highlighted with different colours to make the overall similarity clear. Other clones occurring in the file are also highlighted to provide additional context in understanding the candidate code clone presented. The lower left panel is used to annotate the clone according to the code cloning patterns described in Chapter 4.

Currently only limited query support is implemented in *CLICS*. *CLICS* supports querying code clones based on location, clones' relations to code segments, and size. Queries of code clones based on location include clones strictly within a given entity, clones going from one entity to another, and clones that have at least one of its code segments in the entity. Querying for clones related to a region of code includes queries for clones that are directly related to the code, and for clones that are related transitively. Transitive closure queries often uncover code clone relations that are too dissimilar to be detected by the code clone detection tool. Clone relationships that are found in this way typically have several changes in each of the code segments. This happens when changes are introduced to the code segments that make the code segments more dissimilar to some of the related code segments than others. Because of the varying levels of dissimilarity, only a portion of the relationships are detected.

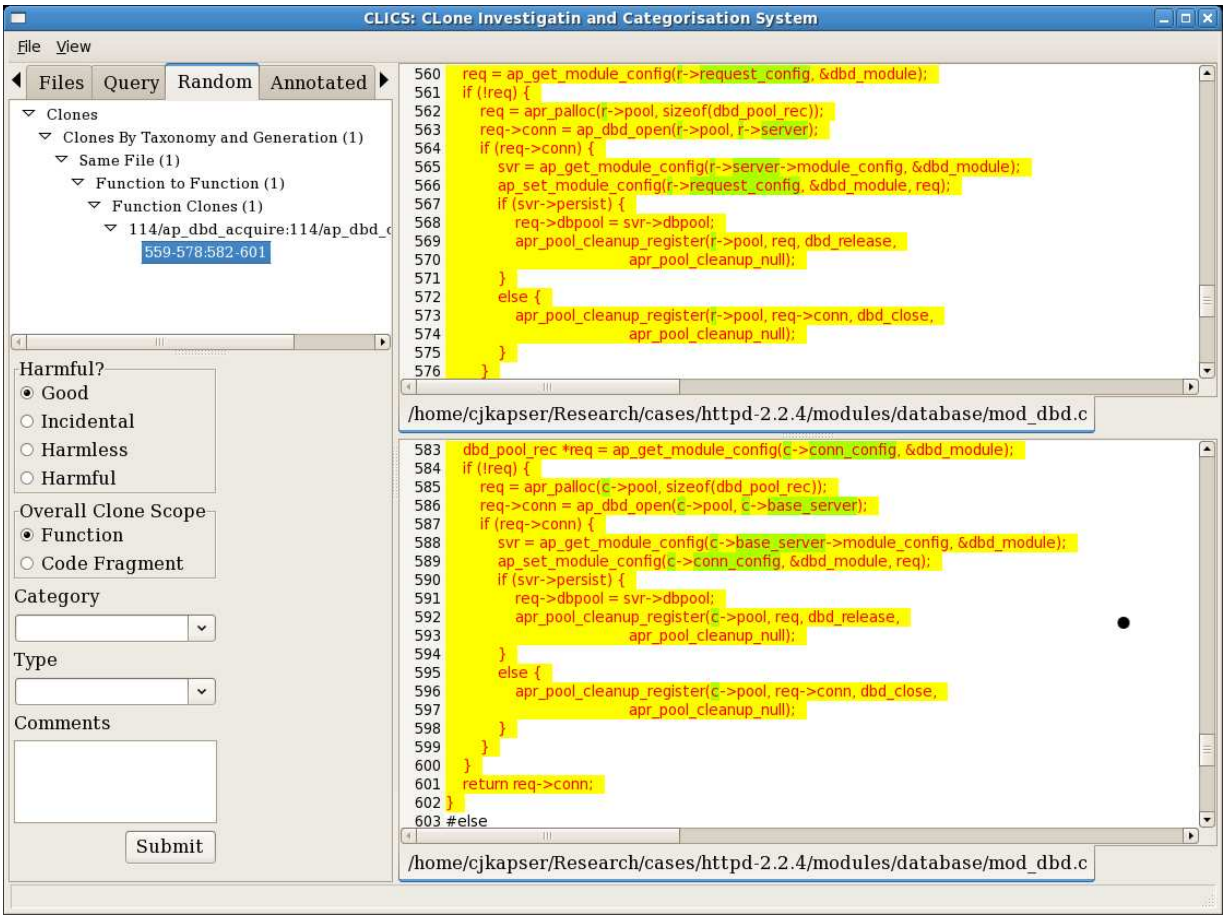

Figure 3.6: Sample visualization of a clone pair

Analysis Refinement

Refinement facilities in CLICS currently allow the manual removal of clones and removal/addition of files from the analysis. Users can remove individual clones, whole RGCs, and clone classes. They can also select files to be excluded from the analysis. More advanced filtering mechanisms are currently being developed.

3.8 Summary

This chapter revisits the questions raised by this thesis. These questions, which can be summarized as why and how do software practitioners create and maintain code clones, have been largely overlooked by the clone detection and analysis research community. Conventional wisdom asserts that code clones are harmful to the quality of software but there has been little or no qualitative or quantitative research to determine the accuracy of this. The development of code clone analysis tools and a series of case studies on code cloning in well-known software systems has led to the discovery of a set of common code cloning scenarios, which will be described in Chapter 4.

There are three contributions of the work presented in this chapter. First, we outline an example of an approach to studying a phenomenon in software that is not well understood and for which analysis tools do not exist. In the approach used for this thesis, case studies were performed to improve our understanding of code clones and that understanding was used to build and enhance a tool for further analysis. More concretely, an initial question of how and why code clones were used in software motivated initial studies that lead to the second and third questions of this thesis. As we improved our understanding of how code clones were used we further refined our analysis tools and our questions, leading to the set of requirements for code clone analysis tools, code cloning patterns described in Chapter 4 and analysis of the use of these patterns described in Chapter 5. The overall process of this approach and the goal of reaching an understanding of a phenomenon through observation of reality is very similar to the qualitative research methodology of grounded theory [30]. Further investigation into a general research methodology for studying ubiquitous phenomena, such as code clones, in source code is needed. While the idea of using several case studies to better understand a phenomena is not novel, the particular combination of

case studies that has taken the author from a general question to a set of common uses and an empirical evaluation of those uses can be used to efficiently answer similar questions.

This chapter also contributes a set of filters for a well-known code clone detection methodology. These filters are specific to parameterized string matching and were created to address particular weaknesses of the approach. The parameters suggested in this work are of a conservative nature: we wished to remove no true positives from the set of candidate code clones. These parameters are specific to the desired characteristics of the candidate code clones: the candidate code clones should contain a large variety of types of code cloning. Adjusting these parameters may improve precision with some impact on recall. The relationship of recall and precision with respect to these filters is the topic of continuing work.

Another contribution is the set of requirements for a code clone analysis tool. The code clone analysis tool presented in this chapter provides key insights regarding analyzing code clones in software. In particular, we have found there is a need for high level metrics relating the phenomena to source code organization as a means of enhancing tractability of the analysis. The work presented in this thesis was the first to present code clones as implicit relationships within a software architecture. Categorization and grouping is also introduced as a means of improving tractability. Code clone analysis tools preceding *CLICS* do not include a means of refining the candidate code clone set either by removing false positives or by hiding instances not currently of interest.

In the following chapters, the results of our case studies into the nature of code cloning are presented. Chapter 4 describes how code clones are used in software in the form of code clone patterns. Chapter 5 evaluates these patterns to determine the prevalence of harmful code cloning and the applicability of the patterns to characterizing code clones in software.

Chapter 4

Patterns of Cloning

4.1 Introduction

As we see in Chapter 2, much anecdotal evidence about the nature of code cloning has been described in the literature. However, most research has focused on clone detection techniques or quantitative analysis of results, with little or no emphasis on the qualitative characterization of code cloning as it exists in software. While these are necessary avenues of research (this thesis would not be possible without them), focused, in-depth, qualitative analysis of code cloning is necessary if we are to evaluate the applicability and relevancy of previous and future code cloning research to software developers and maintainers. Quantitative results citing that 15% of a software system contains cloning provides little useful information without some understanding of the rationale behind those clones and the overall impact on future software development and maintenance. For example, 10% of those clones could be intentionally created to protect system stability while developing new features in a replacement subsystem. In the Linux kernel, the implementation of the Linux filesystem ext3 was in fact cloned from ext2 to add features such as journaling [50].

There is a lack of qualitative research supporting or rejecting the commonly held belief that code cloning is harmful to the maintainability and extendability qualities of software. With this shortcoming in mind, the first question of this thesis, posed in Chapter 3 as:

> **Question 1** *What are the common motivations for developers to use code clones?*

becomes unavoidable if we are to continue this field of study in a productive direction. With this question in mind, the work presented here began with in-depth studies of code cloning in a variety of software systems developed in C/C++. The study subjects of these initial investigations were the Linux operating system kernel, the PostgreSQL RDBMS, and the Apache httpd web server (Apache was also used as a study subject in the validation described in this thesis). While the initial investigations were not explicitly intended to examine patterns of code cloning, during the analysis we discovered several recurring ways in which developers created and used code clones. Further organization of this information resulted in the formulation of the patterns of code cloning that appear to recur in software systems.

This chapter introduces the notion of categorizing high-level patterns of code cloning in a similar fashion to the cataloguing of design patterns [28] or anti-patterns [17]. There are several benefits that can be gained from this characterization of code cloning. First, it provides a flexible framework on top of which we can document our knowledge about how and why cloning occurs in software. This documentation may help in crystallizing a vocabulary that researchers and practitioners can use to communicate about cloning. By using the general definition of code cloning defined in Chapter 1 and refining it to specific types of code cloning observed in software, we hope to provide a set of terms that more broadly defines code clones. It is our hope that this broader set of terms includes many or most of the view points of code clone researchers, and in doing so improves our ability to agree on whether or not two segments are code clones.

This categorization is a first step towards formally defining these patterns to aid in automated detection and classification. These classifications can then be used to define metrics concerning code quality and maintenance efforts. Automatic classifications would also provide us with better measures of code cloning in software systems and the severity of the problem in general. For example, a software system that contains many clones that are intended to evolve separately, such as *experimental variation* clones described in later in this chapter, will require different maintenance strategies and tools compared to a software system containing many clones that need to be maintained synchronously, such as those

clones introduced because of language limitations.

The following sections describe the cloning patterns that have been discovered in the study subjects. These patterns are defined by what artifact is cloned and why, and to some extent the mechanisms of the creation of the code clones. More specifically, the patterns described here concern both cloning of large architectural artifacts, such as files or subsystems, and finer-grained cloning, such as functions or code snippets. The reasons why developers use these patterns range from difficulty in abstracting the code to minimizing the risk of breaking a working software system. These reasons are inferred from the code clones and how they are being used, and the role of the cloned code in the software system. In some cases, documentation, either in the source code or externally, explicitly states the reasons for code cloning. While this is a subjective assessment, the author has experience as a developer and has a strong understanding of software design and maintainability. In regards to how the code clones are created in a pattern, the description includes what the new artifacts will be rather than the tools that are used to perform the code cloning. The information described in these patterns is drawn from the case studies the author has performed.

To describe code cloning patterns the following template will be used:

- **Name.** Describes the pattern in a few words.
- **Intent.** The intention for using the code cloning pattern.
- **Motivation.** Why developers might use this cloning pattern rather than other forms of reuse.
- **Advantages.** Description of the benefits of this pattern of cloning compared to other methods of reusing behaviour.
- **Disadvantages.** Description of the negative impacts of this pattern of cloning.
- **Management.** Advice on how this type of cloning can be managed.
- **Long-term issues.** Issues to be aware of when deciding to use a cloning pattern as a long-term solution.

- **Structural manifestations.** How this type of cloning pattern occurs in the system. This section describes the scope and type of code copied, as well as the types of changes that are expected to be made.

- **Examples.** Examples from real software systems, such as the GNU spreadsheet application Gnumeric version 1.2.12 (observed during the case study described in Chapter 5), the RDBMS PostgreSQL 8.0.1, the web server Apache httpd 2.0.49, and the Java mail client Columba version 1.2.

The code cloning patterns have been divided into four related groups: *Forking*, *Templating*, *Customization* and *Exact match*. This partitioning is based on the high-level motivation for the cloning pattern. *Forking* is cloning used to bootstrap the development of similar solutions, with the expectation that evolution of the code will occur somewhat independently, at least in the short term. A major motivation for forking is to protect system stability, such as allowing for experimentation to occur away from the stable core of the system. In these types of clones, the original code is copied to a new source file and then independently developed. *Templating* is used as a method to directly copy behaviour of existing code when appropriate abstraction mechanisms, such as inheritance or generics, are unavailable or insufficient. *Templating* is used when there is a common set of requirements shared by the clones, such as behaviour requirements or the use of a particular library. When these requirements change, all clones must be maintained together. *Customization* occurs when currently existing code does not adequately meet a new set of requirements. The existing code is cloned and tailored to solve this new problem. *Exact match* code cloning is typically used to replicate, verbatim, simple solutions or repetitive concerns within the source code.

4.2 Forking

Forking patterns often involve large portions of code with the intention that the resulting code clones will need to evolve independently. This pattern of code cloning can be used as a "springboard" from which to start development and works well in situations where the commonalities and differences of the end solutions are not clear. At a later time when

the new code has matured, it may be reasonable to refactor any remaining code clones. This section describes three *forking* patterns that have been observed in software systems: *hardware variation*, *platform variation*, and *experimental variation*.

4.2.1 Hardware variation.

Intent. Rapidly develop support for new hardware devices while maintaining backwards compatibility with related pre-existing hardware devices.

Motivation. When creating a new driver for a hardware family, a similar hardware family may already have an existing driver. However, there are often non-trivial differences in the functionality and features between families of hardware, making it difficult and risky to modify the existing code while preserving compatibility for the original target.

Advantages. Through code cloning, *evolvability* and *testability* of code can be improved over changing the existing driver as testing the driver on older hardware devices can be difficult and time consuming. Cloning the existing driver prevents the need for this type of testing.

Disadvantages. *Maintainability* can be negatively affected by code growth. This can be a particular issue with this pattern of cloning because entire files or subsystems are copied. In addition to the general maintenance issues such as propagating bug fixes, cloned drivers may introduce unexpected feature interactions, particularly in the realm of resource management.

Management. Groups of cloned drivers should be clearly identified to facilitate propagation of bug fixes within the group.

Long-term issues. Dead code can slowly creep into the system unless care is taken to monitor which drivers are still actively supported.

Structural manifestations. Drivers are commonly packaged into a single file. Developers usually copy the entire file, which is then modified to implement the specifications of new device.

Examples. The Linux SCSI driver subsystem has several examples of this pattern of cloning [32]. In one example, the file `NCR5380.c` was copied to the file `atari_NCR5380.c` and adapted for the Atari hardware device. This new file was then cloned as `sun3_NCR5380.c` to be adapted to the Sun 3 platform. Another example of driver cloning is the file `esp.c` which has been copied and modified in `NCR53C9x.c`. What is interesting in the Linux SCSI drivers is that the authors creating the new file explicitly reference the file they have cloned within the source code comments, making the chain of replications easily verified.

4.2.2 Platform variation.

Intent. Reduce code complexity introduced by supporting different APIs and reduce change coupling when supporting the independent evolution of external dependencies.

Motivation. When porting software to new platforms, low-level functionality responsible for interaction with the platform will need to change. Rather than writing portable code using branching or pre-processor directives, it is sometimes easier, faster, and safer to clone the code and make a small number of platform-specific changes. In addition, the complexity of the possibly interleaved platform-specific code may be much higher than several versions of the cloned code, making code cloning a better choice for maintenance. *Hardware variation* may, in some respects, be considered a more specific example of *platform variation*. We have chosen to include the two patterns separately as the body of code that is maintained in the two instances is often quite different in form. Often, drivers are comprised of lower-level source code, intermixed with large portions of assembly or device specific protocols (such as operations on a SCSI or USB attached scanner). In contrast, source code specialized to support a specific software platform, such as an operating system, is comprised of interactions with an API that can be compiled and linked against. The differences in the type of source code in these artifacts raises different types of maintenance concerns. For example, hardware drivers often contain a large portion of constants, often in the form of arrays of hexadecimal values, used to communicate commands and configurations to hardware devices. Managing changes between code clones in the

case of hardware variations includes managing these constants whose values cannot be tested without the physical hardware or a considerably complex testing harness.

Advantages. *Comprehensibility* and *maintainability* of source code may be improved because complex code, inherent to platform-optimized code that is interleaved, is avoided. *Evolvability* is also enhanced because stability for currently supported platforms is maintained. As platforms are likely to evolve independently, maintaining support for one platform will not affect the stability of the code for other platforms.

Disadvantages. *Maintainability* can also can also be negatively affected. The code will evolve along two dimensions: the requirements of the software and the support of the platform. Bug fixes may be difficult to propagate as it may not be clear how or if the bugs are present in each version of the code. Changes to the interface of the platform-specific code become more problematic because these changes will need to be performed across several versions of the library.

Management. The platform-specific interaction should be factored out as much as possible to minimize the amount of cloning necessary. When creating the code clones, the variations should be well documented to facilitate bug fix propagation.

Long-term issues. As groups of platform-specific code clones grow, the interface they expose to the software internals will become more brittle and difficult to change because of the number of places where changes will need to be made. While this is a common problem that can be said for most code clones, it is particularly relevant here as platform variation clones will, in many cases, be part of an abstraction layer that advertises a concise set of behaviours. As a consequence it is vital to ensure that the observable behaviour (from the perspective of the source code dependent on the API) from each of the clones remains consistent. This is not to say that the internal workings of the code clones will not diverge, but their side effects and post-requisites must be consistent with each other.

Structural manifestations. Platform-specific variations often exist in the same subsystem. They often manifest as either cloned files or subsystems.

Examples. *Platform variation* cloning is apparent in several subsystems within Apache's portable library, the Apache Portable Runtime (APR). This subsystem is a portable implementation of functionality that is typically platform dependent, such as file and network access. Two examples of this type of cloning are the `fileio` and `threadproc` subsystems. In these two subsystems, there are four directories: `netware`, `os2`, `unix`, and `win32`. `threadproc` has an additional subsystem `beos`. All of these directories share some cloning that is easily detected by a clone detection tool, but there are also code clones that are sufficiently different that clone detection tools do not detect the similarity. In these cases, changes are typically characterized as insertions of additional error checking or API calls. With these changes, overall structure remains the same, and in several cases cloned documentation exists providing further information about the cloning.

4.2.3 Experimental variation.

Intent. Spring-board development of related or optimized features where the end commonalities are unclear and/or the cost of bug introduction in the core code is high.

Motivation. Developers may wish to optimize or extend pre-existing code but do not want to risk stability of the core code. By forking the existing code, users can choose to run the experimental code or the trusted stable code at deployment.

Advantages. This pattern can contribute to *evolvability*. The stability of the software system is protected while still allowing users access to leading edge development. Further, this eases product distribution by avoiding version control branches that would require multiple releases to be downloaded by users if they wanted to switch between the stable and experimental versions of a feature. Changes made to the experimental fork can be merged with or replace the stable version at a later time. *Risk exposure* and *time-to-market* may also be reduced in some cases.

Disadvantages. Merging code at a later point may be difficult if the corresponding stable version continues to evolve independently, although this may not be a problem if the experimental version is meant to be a replacement rather than a coexisting feature.

Management. Care should be taken to maintain and synchronize the experimental version closely with the stable version. Changes to the external behaviour of the existing stable module will need to be monitored and introduced in the cloned experimental code to maintain a consistent interface.

Long-term issues. As the original and cloned code evolves, consistent maintenance may become more difficult. Documentation of the differences should be maintained to aid program comprehension.

Structural manifestations. The cloning pattern will appear as a cloned file, subsystem or class. It may even be labelled as an experimental development effort, as in the case of several Apache modules [54].

Examples. An example of *experimental variation* can be found in the Apache httpd web server. In the multi-process management subsystem, the subsystem `worker` was cloned multiple times as `threadpool` and `leader` [54]. The cloned subsystems are experimental variations of `worker` that are designed to provide better performance. Because they are separated from `worker`, the web server remains stable while optimizations are being developed.

4.3 Templating

Templating occurs when the desired behaviour is already known and an existing solution closely satisfies this need. Often *templating* is a matter of parametrization, as opposed to the complex insertions and deletions present in *forking* patterns. For example, one might use this pattern of cloning to achieve the same behaviour for `floats` and `shorts` in the C programming language. In this case, the expected changes to the code are only the variable types. When developers use cloning patterns of this type, the evolution of the clones is often expected to be closely related. In the sections that follow four *templating* patterns are discussed: *boiler-plating due to language inexpressiveness*, *API/Library protocols*, *general language or algorithmic idioms*, and *parameterized code*.

4.3.1 Boiler-plating due to language inexpressiveness.

Intent. Overcome language limitations for behaviour reuse when language features are unavailable or unsuitable to permit a maintainable abstraction of the original source code.

Motivation. Due to language constraints, reusing trusted and tested code may be difficult to achieve when the supported data structures of the target code do not match existing implementations. This can occur when polymorphism cannot be used.

Advantages. *Technology limitations* that hinder reuse are overcome. This pattern can make reuse of trusted code possible. It allows for consistent behaviour for related concepts, improving program *comprehensibility*.

Disadvantages. *Maintainability* can be negatively affected due to code growth and change propagation, possibly leading to increased maintenance effort. These code clones will be expected to evolve very closely, and any maintenance efforts increase with each code clone. With appropriate tool support, one would expect changes to one code clone could be automatically applied to all code clones.

Management. Documentation or other forms of an explicit link to all code clones is important to ensure that all clones are modified together. As suggested by Duala et al., these links should be tracked over the evolution of the software system [24]. Tools and methodologies such as Linked Editing [87] should be used to ensure consistent changes are made to all code clones. Another approach to managing these clones is to create the code at build time using a source code generator [41]; in this case, the code clones do not come into existence until the system is being built.

Long-term issues. If maintenance is not performed rigorously, the code clones may become unintentionally different making debugging and testing difficult.

Structural manifestations. Typically these code clones are closely located in the software system, either in the same file or in the same subsystem, with names that are also very similar.

Examples. *Boiler-plating* can be readily found in most software systems. An example of where this pattern was used in PostgreSQL is the `contrib/btree_gist` subsystem where there are a great deal of code clones whose only modification is the data type of the procedure parameters. Figure 4.1 demonstrates an example of this pattern of cloning.

4.3.2 API/Library protocols.

Intent. Use existing source code as a template for a task that is repeated and parameterizable but may be difficult to refactor because it is context dependent.

Motivation. Often the use of particular application program interfaces (APIs) require an ordered series of procedure calls to achieve desired behaviours. For example, when creating a button using the Java Swing API, a common order of activities is to create the button, add it to a container, and assign the action listeners. Similar orderings are common with other libraries as well. The order of activities to successfully set up a network socket in C on Unix systems is well established. Developers will often copy-and-paste these sequences and then parameterize them appropriately for their particular problem.

Advantages. *Development time* can be improved as novice users of the API or library can learn from existing code using cloned code as a form of recipe. Experienced users can reduce coding effort by quickly copying and modifying the code. The copied code can flexibly be changed, and often the size of the code clones may not warrant further abstractions.

Disadvantages. *Evolvability* in terms of maintaining compatibility with newer API versions can be negatively affected as the impact of changes to the library or API is increased with every clone.

Management. Locate prevalent cloning of this type and extend the API or library with appropriate abstractions or create wrapper libraries when modification of the library is not possible. For code clones of this type, rigorous review of the code clones can

```
static PyObject *
py_new_RangeRef_object (const GnmRangeRef *range_ref)
{
        py_RangeRef_object *self;

        self = PyObject_NEW (py_RangeRef_object, &py_RangeRef_object_type);
        if (self == NULL) {
                return NULL;
        }
        self->range_ref = *range_ref;

        return (PyObject *) self;
}
```

(a)

```
static PyObject *
py_new_Range_object (GnmRange const *range)
{
        py_Range_object *self;

        self = PyObject_NEW (py_Range_object, &py_Range_object_type);
        if (self == NULL) {
                return NULL;
        }
        self->range = *range;

        return (PyObject *) self;
}
```

(b)

Figure 4.1: An example of *boiler-plating* in Gnumeric

ensure that the cloned code likely to be used as an exemplar is of high quality and conforms to the API/library use best practices.

Long-term issues. Changes to the API will require changes at multiple sites, and these changes may be problematic in terms of consistency and testing. Using the appropriate abstractions may decrease the maintenance effort by centralizing the required changes.

Structural manifestations. Code clones created using this pattern are typically scattered throughout the source code, and are often small in size.

Examples. In the mail client Columba, this pattern is readily found in the GUI code where buttons are added. A sequence of three operations that create a button, set its action listener, and set its action command is present throughout the Columba system where GUI code is present. An example from the Gnumeric case study presented in Chapter 5 is shown in Figure 4.2.

4.3.3 General language or algorithmic idioms.

Intent. Reuse a generally accepted solution to a ubiquitous problem (e.g. allocating memory and checking the returned value).

Motivation. Programming idioms are clear and concise implementations of particular solutions. These idioms tend to be self documenting for language experts as they implicitly, through a generally accepted interpretation, provide information as to how and why the implementation is done in this way. Idioms are well known in both research and industrial publications. Entire books have been written on them [21], and they remain a popular topic on web-based discussion forums. They can be conventional wisdom in the programming community, such as checking the return value after allocating memory in C programming, or personal dialects of individual developers.

Advantages. Idioms provide structured, standardized solutions to common problems. These solutions become self documenting, improving program *comprehensibility*.

```
w = glade_xml_get_widget (state->gui, "ok");
g_signal_connect_swapped (G_OBJECT (w),
        "clicked",
        G_CALLBACK (cb_do_print_ok), state);
w = glade_xml_get_widget (state->gui, "print");
g_signal_connect_swapped (G_OBJECT (w),
        "clicked",
        G_CALLBACK (cb_do_print), state);
w = glade_xml_get_widget (state->gui, "preview");
g_signal_connect_swapped (G_OBJECT (w),
        "clicked",
        G_CALLBACK (cb_do_print_preview), state);
w = glade_xml_get_widget (state->gui, "cancel");
```

(a)

```
button = glade_xml_get_widget (state->gui, "ok_button");
g_signal_connect (GTK_OBJECT (button),
        "clicked",
        G_CALLBACK (cb_ok_button_clicked), state);
button = glade_xml_get_widget (state->gui, "apply_button");
g_signal_connect (GTK_OBJECT (button),
        "clicked",
        G_CALLBACK (cb_apply_button_clicked), state);
button = glade_xml_get_widget (state->gui, "cancel_button");
g_signal_connect (GTK_OBJECT (button),
        "clicked",
        G_CALLBACK (cb_cancel_button_clicked), state);

/* Make <Ret> in entry fields invoke default */
entry = glade_xml_get_widget (state->gui, "entry1");
```

(b)

Figure 4.2: An example of *API/Library protocols* in Gnumeric

Disadvantages. This code pattern can lead to *bug introduction* if not carefully used. Inconsistencies or faulty implementations of programming idioms may be easily overlooked. Incorrect or inefficient idioms (also known as anti-idioms) can also be copied, degrading the quality of the code.

Management. Anti-idioms — that is, idioms that contribute to poor quality such as inefficiency — should be located and removed. Correct idioms should be located and verified for consistent implementation.

Long-term issues. None.

Structural manifestations. These idioms tend to be distributed throughout the code, as code snippets.

Examples. A common idiom in Apache is how a pointer to a platform-specific data structure is set in the memory pool, shown in Figure 4.3. At least 15 occurrences of this idiom can be found in the Apache APR subsystem. First, the code checks if the data structure containing the pointer exists in the memory pool, and if not space is allocated for it, then the platform-specific pointer is assigned. This idiom exists because the APR library uses similarly defined data structures to point to platform-specific ones, `pthreads` for example. These structures also store platform-specific data that is relevant to the concept, such as the exit status of the thread. A slight variation to this idiom is that in some cases the code checks if the memory pool exists, and returns an error if it does not. The lack of such error checking in other related code clones is likely a bug.

4.3.4 Parameterized code

Intent. Reuse an existing solution that could be parameterized as a function.

Motivation. When implementing a solution to a common problem, it is often the case that this solution can be modified to solve a new problem by changing only a few identifiers or literals in the code. This commonly occurs when implementing basic solutions for very similar problems, such as opening a file descriptor that points to

```
if (pool == NULL) {
     return APR_ENOPOOL;
}
if ((*key) == NULL) {
     (*key) = (apr_threadkey_t *)apr_pcalloc(pool, sizeof(apr_threadkey_t));
     (*key)->pool = pool;
}
```

(a)

```
(*new) = (apr_thread_t *)apr_palloc(pool, sizeof(apr_thread_t));
if ((*new) == NULL) {
     return APR_ENOMEM;
}
(*new)->pool = pool;
```

(b)

Figure 4.3: Two examples of idioms in Apache httpd

`stdout`, `stderr`, or `stdin`. In this case, developers may implement a parameterized function that takes an argument that indicates which descriptor to open. On the other hand, developers may create a new function for each of the three file descriptors.

Advantages. Improves *comprehensibility*. In some cases, this type of cloning can be used to ensure variable names closely match the semantics of the data they represent. This is particularly true with mathematical equations that have commonly accepted variable naming conventions.

Disadvantages. *Maintainability* of the source code may be decreased. This type of code may contribute to unnecessary growth of the software system when used excessively. Unlike *boiler-plating due to language inexpressiveness* whose use may be unavoidable, *parameterized code* can often be avoided using trivial abstractions. Unlike *idioms* whose use is likely associated with small code snippets, it parameterized code may represent larger code fragments with larger impacts on source code size.

Management. The behaviour of these clones is expected to evolve together. Refactoring the code is recommended if such an action does not reduce *comprehensibility* or *traceability*. Otherwise documentation of the clone relationship should be attached to the clones.

Long-term issues. These clones most often contribute to needless code growth, something that can negatively affect the comprehensibility of the source code.

Structural manifestations. These clones most commonly involve entire functions that are within very close proximity of each other.

Examples. An example of this cloning pattern is shown in Figure 4.4. This example comes from `plugins/fn-eng/functions.c` in Gnumeric.

4.4 Customization

Customization often arises when existing code solves a problem that is very similar to the current task at hand, but additional or differing requirements create the need for extension

```
gnumeric_oct2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
        return val_to_base (ei, argv[0], argv[1],
                            8, 2,
                            0, GNM_const(7777777777.0),
                            V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}
```

(a)

```
gnumeric_hex2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
        return val_to_base (ei, argv[0], argv[1],
                            16, 2,
                            0, GNM_const(9999999999.0),
                            V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}
```

(b)

Figure 4.4: An example of *parameterized code* in Gnumeric

or modification of the behaviour. In some cases, such as concerns about system stability or code ownership, existing code cannot be modified in place to encompass the additional behaviour. In these cases, code may be cloned and customized to suit the specific development task. These patterns differ from *templating* in that *customization* requires more than simple parametric changes to the copied code. For example, lines of code may be inserted or removed from the clone. While other forms of cloning, such as *templating* and *forking* typically have the goal of maintaining the original behaviour, *customization* is a reuse of behaviour often without requirements that force the observable behaviour to remain the same or similar. The sometimes unstructured editing that occurs in *customization* clones sets them apart from other clones in important ways: their differences can be harder to spot, the effects of the changes on behaviour may be harder to understand, and the code clones may be harder to detect. In this section we describe two *customization* patterns: *bug workarounds*, and *replicate and specialize*.

4.4.1 Bug workarounds.

Intent. Overload existing code to fix a bug that for some reason cannot be addressed directly.

Motivation. Due to code ownership issues or unacceptable exposure to risk, it may be difficult to fix a bug within the original source, so workarounds may be necessary. Copying the code and fixing the bug to overload the broken code may be the only available solution. In other situations, it may be possible to guard the points where the buggy code is used. This guard is then copied as part of the usage of the procedure.

Advantages. Improves *time-to-market*. Problems can be solved without requiring retesting of other code that may be external to the organization. This solution can allow for progress in development, although it should only be a temporary measure.

Disadvantages. *Maintainability* is reduced and *evolvability* may also be reduced. The source of the bug is not addressed, causing further replication of code or, even worse, new code may not even address the existence of the bug. Also, changes to the

behaviour of the buggy code, such as fixing the bug, may cause confusion if this pattern of code cloning is not made explicit.

Management. Once the original bug is fixed, remove any code clones associated with the bug. Planning for this will minimize issues for clone removal, in particular detection of the code clones.

Long-term issues. The code clone may not be removed when the bug is fixed. This forgotten fix may confuse maintenance efforts later on.

Structural manifestations. These clones can appear as locally overloaded procedures or methods, or as procedures with very similar names to the original source. Cloned guarding statements may appear at points where buggy source code must be used.

Examples. The supervising professor of this dissertation, Michael W. Godfrey, wrote a Java fact extractor that was built around the internals of Sun's `javac` compiler. When he found a small bug in the javac source code, he cloned the offending code into a descendant class and fixed the bug there. Because he didn't have write access to the class that contained the offending method, he could not make bug fix directly in the javac code-base (he created a bug report instead).

In PostgreSQL there exists an example of code cloning of a guard for the event of an error due to bugs. In this case, the source code is dependent on MinGW, an external set of libraries required for platform compatibility. This library has a bug in it that has not been fixed for the current release. Because of this, the PostgreSQL developers cloned a three line solution three times in three different files: `backend/commands/tablespace.c`, `port/copydir.c`, and `backend/access/transam/xlog.c`.

4.4.2 Replicate and specialize.

Intent. Adapt an existing solution to solve a new problem that requires similar but distinct behaviours.

Motivation. As developers implement solutions, they may find code in the software system that solves a similar problem to the one they are solving. However, this code may not be the exact solution, and modifications may be required. While the developer could generalize the original code, this may have a high cost in testing and refactoring in the short term. Code cloning may appear to be a more attractive alternative, and is commonly used in practice to minimize costs associated with risk [22].

Advantages. *Maintainability* is improved when this pattern is used to avoid complex abstractions that may have a high cognitive cost during development and maintenance [87]. *Time-to-market* and *risk exposure* can also be affected. This pattern reduces immediate costs in testing and refactoring existing code that may be entrenched in the software system.

Disadvantages. *Maintainability* can also be negatively affected, particularly in the cases of change propagation. The long-term costs of finding and maintaining these code clones could outweigh the short term gains.

Management. Creators of the newly cloned code should carefully document the intention of the specializations. If the appropriate abstractions cannot be made, explicitly linking the code clones through documentation or tool support can help to ensure consistent maintenance. Because the behaviour of the original code has been changed, any refactorings to remove the code clone should include stringent regression testing. Alternatively, deprecating the original code and transitioning to the abstraction will defer testing costs and protect system stability.

Long-term issues. In this pattern, differences in the code clones may make locating them difficult, making maintenance of these code clones more costly. The differences between cloned segments of code may mean there is no clean way of unifying them. This can make the refactoring process error prone, so it is advisable that a strong testing mechanism be in place before refactoring is performed. In many cases, the difficulty of abstracting the code clones will require alternative solutions. Long-term maintenance of these code clones can be greatly facilitated by linking all cloned

segments of code, either via documentation or meta-data that can be used to strictly enforce careful maintenance.

Structural manifestations. These code clones are often snippets or procedures located near each other, but can be more widely distributed as well. In some cases these clones can be particularly hard to detect due to the changes that have been made. Often the copied code contains control structures, suggesting that developers use code clones to reuse complex logic, an observation also noted by Kim et al. [57].

Examples. This pattern is a common form of cloning that we have found in our studies. In one example in Gnumeric, we see this pattern in use for developing the procedures that build the locale and character encoding selection menus. The procedures can be found in the files `src/widgets/widget-charmap-selector.c` and `src/widgets/widget-locale-selector.c`. The control flow of both procedures is very similar but distinct. Another example of this pattern is shown in Figure 4.5. This example is taken from the httpd study described in this thesis, located in the files `httpd-2.2.4/srclib/apr/file_io/unix/readwrite.c` and `httpd-2.2.4/srclib/apr/network_io/unix/sendrecv.c`. Here we see the action within the *do/while* loop has been changed. Because of the small size of the clone, the changes made to it, and their near proximity within the source code, these clones are considered good. Their proximity leads us to believe that updates to the clones are unlikely to be overlooked. The abstraction would be not only be non-trivial to implement, but would also unnecessarily create dependencies on a higher level library call that would be used only for these two subsystems, possibly cluttering the higher level system design.

4.5 Exact Matches

Exact matches often arise when a particular problem is repeated within the software system but it is either too small to make the creation of an abstraction worthwhile or is incomplete when taken out of the context of its neighbouring source code. In this section two *exact match* clone patterns will be discussed: *cross-cutting concerns* and *verbatim snippets*.

```
    apr_status_t arv = apr_wait_for_io_or_timeout(thefile, NULL, 1);
    if (arv != APR_SUCCESS) {
        *nbytes = bytes_read;
        return arv;
    }
    else {
        do {
            rv = read(thefile->filedes, buf, *nbytes);
        } while (rv == -1 && errno == EINTR);
    }
```

(a)

```
apr_status_t arv = apr_wait_for_io_or_timeout(NULL, sock, 1);
if (arv != APR_SUCCESS) {
    *len = 0;
    return arv;
} else {
    do {
        rv = recvfrom(sock->socketdes, buf, (*len), flags,
                      (struct sockaddr*)&from->sa, &from->salen);
    } while (rv == -1 && errno == EINTR);
}
```

(b)

Figure 4.5: An example of *replicate and specialize* in Apache httpd

4.5.1 Cross-cutting concerns

Intent. Replicate pervasive semantic properties throughout the system.

Motivation. Cross-cutting concerns are semantic properties of the software systems that cut across otherwise unrelated functionality. Typical examples of cross-cutting concerns are access control, logging, and debugging [56]. Clones involving these concerns are typically unavoidable in programming languages based on traditional programming paradigms, such as C or Java, because they do not have the appropriate language mechanisms to abstract this code.

Advantages. There is little advantage to cloning cross-cutting concerns: they are typically unavoidable. However, like idioms, cross-cutting concerns can clearly document semantics of the code, improving *comprehensibility*. In the case of cross-cutting concerns checking assertions, cross-cutting concerns document the preconditions or post-conditions of the code they are near.

Disadvantages. *Evolvability* can be negatively affected. Clones of cross-cutting concerns can entrench design decisions as they create repeated dependencies on the concern and its current design. Changes to the design of the modules on which the concern is dependent on will have broad reaching impacts to all the clones involving it.

Management. Aspect-oriented programming is a recent solution to this type of cloning. In this case, the clones are completely removed from the main application code, and are maintained in central location, separate from the rest of the code; when the system is later compiled, the language processing tools weave each aspect into appropriate areas in the source. This solution may remove certain types of maintenance problems but also removes the implicit documentation that exact match cross-cutting concerns provide. Transformation languages and linked editing can also be used to maintain the code. These methods have the advantage of leaving the code in place and with it the implicit documentation.

Long-term issues. The more often these concerns are cloned, the more brittle the concern will become making improvements and design changes increasingly difficult.

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
    return err;
}
```

Figure 4.6: An example of a *cross-cutting concern* in Apache httpd

Structural manifestations. These code clones are often snippets scattered throughout the software system. The examples that we have seen are are exact copies and are small fragments of code.

Examples. A common example in Apache is the checking of the command context before executing security sensitive functionality. This clone is copied verbatim throughout the software system. An example of a cross-cutting concern that is used throughout the *server* subsystem of Apache httpd is shown in Figure 4.6. This concern ensures the correct security requisites are met before continuing to execute a function.

4.5.2 Verbatim snippets

Intent. Reuse the exact solution of an existing problem with no modification (neither parameterized nor customized) required.

Motivation. Often small repetitive fragments of logic must be reused throughout the source code (e.g. branching control). These fragments, not having significant semantics on their own, will be copied rather than implemented as reusable functions. These differ from *cross-cutting concerns* in that they do not implement a specific aspect or property of the system, rather they are general purpose fragments.

Advantages. *Comprehensibility* is improved in the form of conceptual cohesion. Conceptual integrity of modules or functions is maintained by keeping code simple and close together. This pattern can also help to reduce interface bloat by avoiding the accumulation of a many small procedures.

Disadvantages. As with all code, assumptions are made about the data that is being manipulated. Because these clones can be difficult to find due to their small size, changes that affect these assumptions may be difficult to propagate.

Management. Often these snippets cannot be refactored. Because they often constitute only incomplete concepts and are small in size, it is unlikely this form of code cloning poses a serious maintenance risk. If it becomes apparent that certain segments of code that do constitute complete concepts are copied often, they should be factored out as helper functions or macros.

Long-term issues. In most cases these code clones are small and probably do not have a large impact on the overall system design. However, over time this cloned code can build up making it difficult to remove, or making data structures difficult to change due to the amount of code that is dependent on them.

Structural manifestations. These clones generally appear as small fragments of code scattered throughout the code base. Typically the number of similar code fragments is low.

Examples. These clones are readily found in many software systems. Common examples include the initial lines of *for* loops and fragments of error or condition checking. An example of such a clone is shown in Figure 4.7, taken from Apache httpd 2.2.4, in the file `httpd-2.2.4/srclib/pcre/pcretest.c`. Verbatim snippets can also occur as cloned data structures where the entire region is copied and pasted.

4.6 Summary

To evaluate and maintain code clones within a software system, it is necessary to understand the decisions behind creating individual code clones, an area of research that has largely been overlooked in the literature. Over-simplified generalizations about the harmfulness of cloning does little to aid developers and maintainers deal with existing clones or understand the trade offs associated with creating new ones. To better understand

```
for (i = 0; i < sizeof(utf8_table1)/sizeof(int); i++)
  if (cvalue <= utf8_table1[i]) break;
```

(a)

```
for (j = 0; j < sizeof(utf8_table1)/sizeof(int); j++)
  if (d <= utf8_table1[j]) break;
```

(b)

Figure 4.7: An example of *verbatim* in httpd

code cloning as it exists in real software systems and to help software practitioners make justifiable decisions, this chapter presents common patterns of code cloning, derived from qualitative analysis of several open source software systems, along with qualitative advantages and disadvantages, management trade offs, and real life examples of cloning. These patterns are intended to provide a more descriptive view of code cloning resulting in a richer vocabulary to discuss the topic as well as evaluate the quality of software systems in practice.

These patterns may be used to provide guidance in software development and maintenance decisions regarding code clones. While code abstraction, polymorphism, and other generally accepted methods of behaviour reuse are taught by most software development educators, their negative qualities (such as high code complexity) and limitations seem to be overlooked. Until now, patterns such as *boiler-plating due to language inexpressiveness* and *experimental variation* have not been discussed as plausible solutions to code reuse in the appropriate settings. This oversight may leave software developers without the necessary tools to most effectively solve particular types of behavioural reuse problems. Not unlike design patterns, documented code cloning patterns provide more information and context to developers and help them avoid reinventing the decision making process.

The patterns presented in this chapter may also be useful as an additional metric for code quality evaluations. Categorizing code clones according to these patterns can provide a means of determining whether or not the code clones are useful or harmful artifacts. *Platform variation* clones of essentially identical, co-evolving platform dependent code

may be considered unreasonable. On the other hand, if the two platforms are quite similar but are not expected to evolve together the decision to produce a code clone is likely well founded. Understanding the rationale behind the clone, in this example trying to implement similar behaviour on two similar platforms, provides clues to how to evaluate the quality of the decision.

This catalogue of code cloning patterns is not presented as a complete, final set of cloning patterns. Further studies may reveal new patterns of cloning. The format and organization of the patterns is intended to permit flexible growth. New patterns can either be added to the existing groups of patterns, or new groups of patterns can be added. As a means of facilitating referencing and understanding code cloning patterns, the catalogue prescribes the structure of the pattern documentation with the intention that patterns are consistently documented. Future work in this area is required to test effectiveness of this structure in both teaching the patterns to software practitioners as well as facilitating the addition of new ones.

Chapter 5

Empirical Evaluation

5.1 Introduction

In this thesis we present a set of code cloning patterns that recur in source code. Prior to this work, there was little research to improve our understanding of code clones in software and their implications on software development and maintenance. The literature gives us little or no concrete evidence about the common motivations behind code clone usage: most motivations cited in literature appear as either anecdotes or plausible scenarios without empirical or concrete support (see e.g., [15, 26, 44]). Further, many of these motivations cast a negative light on code clones, often presenting code cloning as a severe problem in software systems. Beginning with an exploration into the motivations behind code cloning in real software systems, we discovered a recurring set of similar code cloning activities that we presented as code cloning patterns in Chapter 4. These patterns present common forms of cloning, connecting motivations to various manifestations of the phenomena, with concrete examples to support their relevance to real software development activities. These patterns are intended to further our knowledge about how code cloning occurs in software and provide structure for analyzing code cloning in software systems. The patterns provide some indication that not all clones are equally "bad" and provide a means to measure the extent of cloning in a software system and the severity of code cloning (if it is a problem at all).

While these patterns are supported by concrete examples found in real software systems,

as yet we have not shown that these patterns are relevant or recurring outside of the examples listed in the previous chapter, evidence that is necessary to answer the second question of this thesis:

Question 2 *Are there common patterns of code cloning that occur in the development of software systems?*

To answer this question, we must measure the degree to which the code cloning patterns listed in Chapter 4 recur in a software system. As part of this evaluation, we also investigate the third question of this thesis:

Question 3 *How are code cloning patterns used in practice, and to what extent is their use appropriate?*

By evaluating whether code clones in each pattern are harmful to the quality of the source code of a software system, we can gain insights into whether or not certain patterns of cloning are harmful and whether or not code clones are harmful in general.

The study described in this chapter provides empirical evidence to support the proposition that the cloning patterns described in this thesis are relevant to software systems in the real world. We have also found that the widely accepted belief that code clones are harmful is not true for all code clones. The study results suggest certain patterns of cloning are often used in constructive ways while other patterns are often used to create harmful clones.

The rest of this chapter is organized as follows: Section 5.2 reviews the clone detection and analysis tools used to evaluate the patterns described in Chapter 4 as well as the classification criteria used to classify each sample, Section 5.3 describes the two study subjects, Section 5.4 describes the sample selection used, Section 5.5 describes the results of the case study, Section 5.6 relates the study results to the thesis questions, Section 5.7 discusses the threats to the validity of the case study presented in this chapter, and Section 5.8 concludes this chapter.

5.2 Study Setup

The purpose of this study is to demonstrate the degree to which the patterns presented in Chapter 4 exist in software systems and to assess the relative harmfulness of code clones. To measure the prevalence of the cloning patterns in source code we performed a case study of cloning in two open source software systems: *Apache httpd* and *Gnumeric*. The study performed was a multiple case, descriptive case study [95]. Two propositions were formulated:

1. Not all code cloning is harmful; cloning may be used in sound design decisions.
2. The cloning patterns described in Chapter 4 appear with non-trivial frequencies in software systems.

We began the study using the set of cloning patterns that were described in our original report on this work [53], and we set out to categorize a random sample of candidate code clones from each of the study subjects. However, as a result of performing this study we discovered several new cloning patterns: parameterized code clones, cross-cutting concerns, and verbatim snippets. We subsequently added these to our revised pattern set — the one described in Chapter 4 — and revised the results of the study to be consistent with the revised set.

In this section four aspects of the study setup are discussed: the clone detection methodology, the granularity of the sampling and analysis, the clone presentation, and the classification criteria.

5.2.1 Clone Detection

The candidate clones were detected using the *CLone Interpretation and Classification System (CLICS)* clone detection tool described in Section 3.5. This tool locates common sub-strings within the code using a parameterized string matching method based on suffix trees similar to the clone detection tools *CCFinder* [47] and *clones* [62] described in Chapter 2. We chose to use parameterized sub-string matching based on parameterized token streams because this method has been shown to have the highest recall compared to other clone detection methods [16, 62]. We chose *recall* as the most important property when

choosing a clone detection tool because we wanted to ensure that we were able to sample a large variety of clones at the cost of a more time consuming study.

5.2.2 Sample Selection and Clone Presentation

In this study we chose to use Regional Group of Clones (RGCs), described in Section 3.5, as the unit of analysis rather than individual clones. In our previous work, we found that cloned code is often modified in non-trivial ways, causing the clone detection tools to detect several individual clones with breaks between them. These groups of clones in reality are part of a single larger clone. RGCs are better representations of cloning than individual code clones because they present these groups of clones as a single clone, providing more context for each code clone as well as results that more accurately reflect the cloning occurring in a software system. For example, a large number of small segments of code can often be the result of a single code cloning action. The choice of analyzing clones in this way is not directly related to how the candidate code clones are detected, but rather how they are presented, as most clone detection techniques identify segments of similar code, leaving differences as breaks in between the segments.

To sample the RGCs for categorization, we used a uniform random sample of the RGCs that do not occur in the same region. We chose to not sample clones that occur in the same region as experience suggests they are most often segments that would not be likely candidates for refactoring or would be considered false positives. Figure 5.1 shows two segments of code taken from the same procedure in the Apache httpd source code. In this example, CLICS identified these code fragments as candidate clones. However, while they follow similar logic, the fragments test different conditions, and assign different values and return using different variables. While this is not a false positive, it is not a clone that would be likely considered for refactoring. Because of their relatively high frequency in the clone sets, using them in the studies would bias our results toward "good" clones.

The randomly sampled RGCs were presented in the CLICS graphical user interface described in Chapter 3. Key features of the tool that were relevant to this study include the ability to query for clones related to the ones being presented, visualization of the differences in the code comprising the selected clone or RGC, highlighting of other clones occurring between the two files, and automatic classification of clones via a taxonomy

```
if (ret != 0) {
    if (sql->trans) {
        sql->trans->errnum = ret;
    }
    return ret;
}
if (!*results) {
    *results = apr_pcalloc(pool, sizeof(apr_dbd_results_t));
}
(*results)->res = res;
(*results)->ntuples =
```

(a)

```
if (rv == 0) {
    if (sql->trans) {
        sql->trans->errnum = 1;
    }
    return 1;
}
if (!*results) {
    *results = apr_pcalloc(pool, sizeof(apr_dbd_results_t));
}
(*results)->random = seek;
(*results)->handle =
```

(b)

Figure 5.1: Two clones occurring in the same region

described in [52, 54]. Combined, these features provide contextual information to aid the evaluation of cloning in a software system.

For the purposes of this study, the tool presents to the user a single randomly selected RGC at a time. Figure 3.6 in Chapter 3 shows an example of how a random RGC is presented to the user. In the upper left a tree indicates the automatic classification of the cloning between the two regions. This classification indicates the relative location of the clones in the software system, the type of region they occur in, the degree of similarity between the two regions and possibly the type of code the clones occur in. For example, the code clone shown in the figure occurs in the same file and the category *function clone* indicates the detected clone covers more than 60% of the functions it occurs in.

On the right of Figure 3.6 we see the code encompassed within the two regions of the RGC indicated by highlighted text. Using CLICS we can show the differences between the two regions in addition to displaying the detected clone. The common tokens and differing tokens are highlighted with different colours to make the changes in the candidate code clone clear to the user. CLICS also highlights other clones occurring in the file to provide additional context in understanding the clone presented (this feature is not shown here). The lower left panel is used to annotate the clone according to the cloning patterns described in Chapter 4. From this point, we can use the various features of CLICS (described in Chapter 3) to understand the context and motivations of the clone.

5.2.3 Classification Criteria

In the case study, we performed a subjective classification of each of the randomly selected RGCs. For each RGC presented, we rated four attributes of the RGC:

1. The likely effect of the clone on the quality of the software system.
2. The scope of the cloning (Does it cover the majority of the two regions in the RGC or is the candidate code clone only a fragment of code?).
3. The high-level classification of the clone (forking, templating, customization, or exact matches).
4. The low-level classification (the specific pattern).

Rating how a code clone is likely to affect the software system is undoubtedly the most controversial aspect of this study, and also the most subjective. We used a nominal scale of four values: incidental, good, harmless, and harmful. Incidental clones are those that cannot be refactored as they are already at the highest possible level of abstraction and therefore are neither good nor harmful. Examples of these code clones that we have observed reference a single function or multiple calls to the same function, and the parameters to that function are changed in a non-systematic way. In contrast, good, harmless, and harmful code clones can be refactored. Good clones are clones we believed to have an overall positive effect on maintenance and development. In the study below, if a code clone was deemed to be good, we felt that certain quality attributes that were improved by the clone, such as *comprehensibility* or *evolvability*, and the degree to which they were improved had a larger contribution to overall source code quality than the quality attributes that were negatively affected, such as the *bug fix propagation*. Harmful clones are the opposite of good clones. Such clones we believe will have a negative effect on the maintainability of software system. In our study, a clone was deemed to be harmful when we judged that the overall negative effects on quality attributes of the source code outweighed the positive effects. In effect, this is an assessment of the perceived net gain/loss to source code quality. Harmless code clones are those clones that are likely to have no impact on maintenance or development. In other words, there is no intrinsic value in abstracting the code clone, nor does code cloning solve quality attributes such as *evolvability* or *understandability* of the source code. Usually these code clones are small fragments, as small as a variable assignment and single function call. We chose the term "harmless" as these code clones are not strictly necessary (as many *boiler-plating* clones are) yet there is no advantage to removing them.

In our rating of the harmfulness of the candidate code clone we tried to take into account several considerations including the likelihood of the clones requiring co-evolution and how difficult this would be to maintain, the likely complexity of abstract refactored code, and the likely effects of code clones on understandability of the code. First, we asked how probable it was that changes in one code segment of the clone would need to be propagated to the other. This question is often most easily answered by determining what requirements the code is most dependent on, and how many of those requirements are shared between the

cloned code segments. If the cloned code involves concepts internal to the system (such as managing memory, parsing data streams, converting arguments to the appropriate type for a function call, etc.) then it is likely that the clones are strongly dependent on a similar set of requirements and these segments of code will need to evolve together. For example, clones that implement adding and removing items from a request queue will be comprised of standard queue operations, plus error checking and data conversion code. In most cases, these clones will need to maintain similar if not identical behaviour. Conversely, if the cloned code acts primarily as an interface to independent external systems, then the requirements of the cloned code segments are less likely to evolve together because it is unlikely the two external systems are evolving together. An example of an external set of requirements is the interface with database management systems such as PostgreSQL, MySQL, and Oracle. While interaction with these systems is very similar, each system has its own protocol for connection management and implements its own flavour of SQL. Also, each system will add, remove, and change features independently. The developer of a virtualization layer supporting these systems will have to decide how these commonalities will be dealt with. In this case, code cloning is less likely to be harmful and more likely to be beneficial as it enables maintainers to freely evolve the code. On the other hand, in the case of the internal example the code clone is more likely to be harmful because changes to the internal concepts will need to be reflected in all of the cloned code. Further, if the source code in the segments of a code clone support a single external system, they should be treated the same as an internal dependency as they will be expected to evolve together in a similar fashion.

Our second consideration was evaluating the complexity of forming an abstraction to refactor the code. There is evidence that abstractions can be harder to maintain than managing code clones [87]. Forming abstractions for *replicate and specialize* clones can be difficult if the modifications have been interleaved with the cloned code. In cases where the code is already complex, forming abstractions may only exacerbate the complexity. On the other hand, when an obvious abstraction exists, such as when the specialization is restricted to the end of the code clone, it is likely harder to maintain the clone than maintaining the straightforward abstraction. It is important to note that the task of unifying the candidate code clone must be considered in the context of all related code clones (the code clone class)

rather than just the two segments comprising the code clone being inspected. If several code clones exist with unique specializations in varying locations the logical branches to deal with these variations can become very complex, and the end result of unifying these code clones may have high maintenance cost due to high complexity.

Our third consideration was evaluating how refactoring the clone would affect the understandability of the code it occurred in. This is particularly important when evaluating candidate code clones that are code fragments rather than complete syntactic units of code (such as function definitions). In several cases in *Gnumeric* variable names are changed to closely reflect the common mathematical notation the functions represent, such as the parameters for statistical distributions. This type of cloning acts as a form of documentation to ensure that future maintainers will immediately grasp the meaning of the code. Refactoring some of this code may in fact break the conceptual ties the variable names create. Another example where code cloning aids understandability is the cloning of small code fragments. These fragments do not have meaning on their own and refactoring would result in breaking the conceptual cohesiveness of source code.

Evaluating the scope of the code cloning — whether the RGC involves a fragment of code or the majority of the two regions in the region pair — is done by examining all of the common code between the two regions. In cases where the code clones in the clone relationship covered most or all of the two regions they occur between, the scope of the code clone was considered to be the whole region. In cases where the clone detector found several fragments of code, the region differencing utility was used to visualize the overall similarity. If a large portion of the two regions in question was found to be shared, the scope of the clone was considered to be the whole of the regions. If the degree of similarity between regions pairs was restricted to fragments of code, the scope of the cloning is considered to be a fragment.

Classifying the code clones into patterns was done manually, based on the descriptions we have documented in Chapter 4. The high-level classification was one of the four clone pattern groups: *Forking*, *Templating*, *Customization* and *Exact match*. The low-level classification is chosen from one of the patterns in the group of patterns specified by the high-level classification. The primary mechanism for classification was to infer the motivation for the code clone. This required an understanding of the intent of the pro-

grammer for each code fragment individually and also the types of changes made to the cloned code. To determine the purpose of the source code fragments, we analyzed the code fragments in the context of the software system. Relevant documentation (either found within the source code or distributed with the source code), data structures, and data flow were referenced to gain as much information about the source code as possible. Documentation can provide a clearer picture of the external behaviour of a segment of code (pre- and post-conditions for example). Data structures used by both segments of code often enriched this information by making more explicit the low level behaviour requirements, such as the range of valid values for a variable. Data flow, included calling and called functions or procedures, provides more information about how the segments of code manipulate data, enriching the view of how the procedures operate. The goal of this analysis is to arrive at an improved understanding of the intent of the programmers for each segment of code individually. Next we analyzed the differences between the clones. These differences include not only the textual differences of the cloned code fragments, but also the differences in the intent of the programmer uncovered in our analysis of the purpose of the code. For example, we analyzed the differences in the data structures used by the two code fragments. In a few cases, this required referencing external documentation relevant to shared libraries provided by external projects. We also analyzed the purpose of the file and subsystems containing the clones. Combining information about the intent of individual code fragments with an understanding of their differences, we could then assess individual attributes, such as forces that will affect evolution of the source code and the difficulty of forming a more general abstraction, to infer the motivation for forming the code clone. With this information, we then compare the information we compiled with the various patterns listed in Chapter 4. This process was time consuming in the beginning of the sample analysis of each system, but progressively the process became easier as we could reuse much of the knowledge about the system we gained over time.

5.3 Study Subjects

The study subjects were described in Chapter 3 and the description is restated here for clarity. The two study subjects of this experiment were Apache httpd, version 2.2.4, and

Gnumeric, version 1.6.3. Both software systems are of medium size: Apache is 312,460 LOC across 783 files and Gnumeric is 326,895 LOC across 530 files. Apache httpd is an open source web-server designed to run on a wide variety of platforms: BeOS, *BSD, Linux, Netware, OS/2, Unix, and MS-Windows. The core development team of Apache consists of approximately 25 developers who contribute a large majority of new features (88% of new code in 2000 [76]). While there is no explicit policy on code ownership, contributors tend to defer decisions concerning changes to more experienced developers. As a result, small groups, rather than individual developers, modify modules and files [76].

Gnumeric is an open source spread sheet application, part of the GNOME Desktop environment. It supports a variety of platforms but this platform support is not implemented directly in the source code, but rather in the libraries it depends on (i.e., the GTK framework, which has been ported to many different OSs). Using `svn blame` to measure who last modified all of the lines of code in Gnumeric in the last five years reveals that more than 88% of the LOC have been modified by only three developers. This finding is confirmed by documentation distributed with the source code.

5.4 Sample Set

Two sets of clones were detected for each study subject. Several patterns of cloning described in Section 4 are typically only present as small fragments of cloned code and therefore are detectable only when the minimum threshold for a match is set to a low value. To detect these types of clones we ran the clone detector searching for clones with a minimum length of 30 tokens. When this is done, however, many false positives are detected as well as many small code fragments that would normally not be considered for refactoring, creating a bias in the results. To detect and sample both smaller cloning patterns and larger ones, we chose to detect two sets of clones for each subject using two minimum lengths: 30 and 60 tokens. The resulting clones detected with a minimum length of 60 tokens is a direct subset of the results returned when detecting clones with a minimum length of 30 tokens. This is because we are only setting the minimum size of an acceptable code clone, and therefore any clone larger than the minimum will also be returned in the result set.

A summary of the total number of clones returned by the clone detector is shown in

Table 5.1. In the table, the columns "RGCs" indicates the number of RGCs that were detected by CLICS and sampled by us, and the columns "Clones" indicates the number of clones that were detected and sampled as part of the RGC sampling selection. For each data set, we categorized the maximum of 100 RGCs or 1% of the total RGCs. The minimum of 100 RGCs was chosen to ensure we observed a large number of code clones throughout the systems. For the data sets where a minimum clone size of 30 tokens was used, we randomly sampled 0.93% of the total RGCs in Apache and 0.99% of the RGCs in Gnumeric. For the clone sets detected with a minimum size of 60 tokens we randomly selected 6.3% of the RGCs in Apache and 2.9% for Gnumeric. One can see from Table 5.1 that there is a significant difference in the number of clones detected when adjusting the minimum length, especially in the case of the Gnumeric clone sets. As will be presented in the results, this large difference in clones is mainly comprised of false positives but also contains many of the smaller clones that contribute to larger groups of code clones such as in the case of *Replicate and Specialize* code clones, which are overlooked in the sample set of larger clones.

The total number of false positives found in the sample sets are shown in Table 5.2. This table shows the number of RGCs from the clone sample set that were considered to be false positives. As one might expect, the number of false positives was dramatically reduced after increasing the minimum length of a clone to 60 tokens.

5.5 Results

The results of the subjective categorizations are summarized in Tables 5.3 - 5.6. Each row in the tables indicates a clone pattern and the frequency of good, incidental, harmless and harmful RGCs seen in the sample set. The column "Total" indicates the total number of RGCs classified as that pattern of cloning. The bottom row of each table indicates the total number of good, incidental, harmless, and harmful RGCs.

In this section we will discuss the specific results of each study subject and compare the results obtained from the two different samples extracted from each subject.

System	30 Tokens			
	RGCs		Clones	
	# found	# sampled	# found	# sampled
Apache	10,657	100	61,481	204
Gnumeric	23,129	230	84,028	807
	60 Tokens			
	RGCs		Clones	
	# found	# sampled	# found	# sampled
Apache	1580	100	21,270	2655
Gnumeric	3437	100	11,400	405

Table 5.1: Detected clones in Apache and Gnumeric

	Min. Clone Size			
	30		**60**	
System	*RGCs*	*% of sample*	*RGCs*	*% of sample*
Apache	41	41%	7	7%
Gnumeric	159	69%	29	29%

Table 5.2: False positives in sample sets

5.5.1 Cloning in Apache httpd

When comparing the results of the two clone sets sampled for Apache httpd, shown in Table 5.3 and Table 5.4, perhaps the most striking result is the very large difference in the number of harmful RGCs. In the Apache case study where the minimum clone length was 30 tokens, we judged 71% of the 59 true positive RGCs were deemed to be good and only 14% were deemed harmful. In the sample set where the minimum clone length was 60 tokens, 42% of the clones were judged as good and 39% were judged as harmful. We believe this difference is caused by the types of clones found when increasing the minimum clone size. The code clones in the sample of larger clones tend to be more similar and clones with complex changes are overlooked by the matching algorithm. Line insertions and deletions are more likely to cause clones to be overlooked when a higher minimum length is used. This observation is supported by the large increase in the number of RGCs classed as *templating* clones. This biases the sample set toward simplistic clones that can be clearly refactored with simple abstractions, and are unlikely to provide benefit as code clones in the system.

The clone sets for Apache httpd contain a total of 19 RGCs for *platform variation*. Additionally, four RGCs in the set of larger clones were related to *experimental variation*. In each of these cases we felt that the code clones were beneficial to the comprehensibility and evolvability of the source code.

Boiler-plating due to language constraints was present in both sets of clones. These were exclusively due to changes in the data types. Several of these clones could be refactored using a combination of anonymous pointers and passing individual members of composite data types to the functions. However it was felt that the added complexity to both the abstracted function and the source code dependent on the function outweighed the advantage of reduced code size.

Idioms were observed only in the sample of larger clones of Apache httpd. This is a surprising result as we expected the opposite to be true. Two examples of the *idioms* found are shown in Figure 5.2. Both *idioms* allocate memory for a pointer in a memory pool. In the first example, the existence of the pool is asserted, then if the variable *key* is *NULL*, memory is allocated for it and a pointer to the memory pool it has been allocated in is set. In the second example, memory is allocated for the variable *new*. The return

Pattern	**Good**		**Incidental**		**Harmless**		**Harmful**		**Total**
	#	%	#	%	#	%	#	%	
Forking	**9**	**100%**	**0**	**0%**	**0**	**0%**	**0**	**0%**	**9**
Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Platform variation	9	100%	0	0%	0	0%	0	0%	9
Experimental variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Templating	**9**	**40.9%**	7	**31.8%**	**0**	**0%**	**6**	**27.3%**	**22**
Boiler-plating	8	100%	0	0%	0	0%	0	0%	8
API	0	0%	7	87.5%	0	0%	1	12.5%	8
Idioms	0	n/a	0	n/a	0	n/a	0	n/a	0
Parameterized	1	16.7%	0	0%	0	0%	5	83.3%	6
Customize	**10**	**76.9%**	**0**	**0%**	**1**	**7.7%**	**2**	**15.4%**	**13**
Replicate and Specialize	10	76.9%	0	0%	1	7.7%	2	15.4%	13
Bug Workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact Match	**14**	**93.3%**	**0**	**0%**	**1**	**6.7%**	**0**	**0%**	**15**
Cross-cutting	12	92.3%	0	0%	1	7.7%	0	0%	13
Verbatim snippets	2	100%	0	0%	0	0%	0	0%	2
Total	42	71.2%	7	11.9%	2	3.4%	8	13.6%	**59**

Table 5.3: Clones by type - Apache httpd 2.2.4 - 30 Tokens

Pattern	**Good**		**Incidental**		**Harmless**		**Harmful**		**Total**
	#	%	#	%	#	%	#	%	
Forking	**14**	**100%**	**0**	**0%**	**0**	**0%**	**0**	**0%**	**14**
Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Platform variation	10	100%	0	0%	0	0%	0	0%	10
Experimental variation	4	100%	0	0%	0	0%	0	0%	4
Templating	**10**	**19.2%**	**17**	**32.7%**	**1**	**1.9%**	**24**	**46.2%**	**52**
Boiler-plating	5	100%	0	0%	0	0%	0	0%	5
API	0	0%	17	100%	0	0%	0	0%	17
Idioms	0	0%	0	0%	0	0%	12	100%	12
Parameterized	5	27.8%	0	0%	1	5.6%	12	66.7%	18
Customize	**12**	**75.0%**	**0**	**0%**	**0**	**0%**	**4**	**25.0%**	**16**
Replicate and Specialize	12	75%	0	0%	0	0%	4	25%	16
Bug Workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact Match	**3**	**27.3%**	**0**	**0%**	**0**	**0%**	**8**	**72.7%**	**11**
Cross-cutting	2	100%	0	0%	0	0%	0	0%	2
Verbatim snippets	1	11.1%	0	0%	0	0%	8	88.9%	9
Total	39	41.9%	17	18.3%	1	1.1%	36	38.7%	**93**

Table 5.4: Clones by type - Apache httpd 2.2.4 - 60 Tokens

```
if (pool == NULL) {
     return APR_ENOPOOL;
}
if ((*key) == NULL) {
     (*key) = (apr_threadkey_t *)apr_pcalloc(pool, sizeof(apr_threadkey_t));
     (*key)->pool = pool;
}
```

(a)

```
(*new) = (apr_thread_t *)apr_palloc(pool, sizeof(apr_thread_t));
if ((*new) == NULL) {
     return APR_ENOMEM;
}
(*new)->pool = pool;
```

(b)

Figure 5.2: Two examples of *idioms* in Apache httpd

value of the allocation method is checked, and if memory cannot be allocated the function returns an error code. If there is no error, the pointer to the memory pool is set. Both *idioms* occur frequently in the source code. While these *idioms* were detected separately, we noticed that they are related. The second idiom, Figure 5.2(b) should be implemented using the first. That is to say, the first idiom should check the return of the allocation method before attempting to assign a value to a member of the newly allocated variable. Also, the second idiom should check that *pool* is not *NULL* before attempting to allocate memory from it. In both cases, attempting to assign a value to a *NULL* pointer will result in a segmentation fault. We felt that both instances of these *idioms* are incomplete and are therefore harmful. All *idioms* we found were clones of these two examples, and were rated as harmful to the software system.

The *API* clones in both studies that were deemed incidental were part of a test suite. In these clones, the same function is repeatedly called to build a test suite and then the test suite is returned. In all cases, the function call was used in the same way, with the only variation being the function pointer passed as an argument. We deemed this type of cloning incidental because the API usage is already abstracted to the highest level, resulting in the series of repeated function calls to the same function.

Parameterized code clones were found in both clone sets for Apache httpd. We judged only 25% of these clones were to be good. Samples rated as good clones were code fragments that were cloned very few times in the system (typically, once) or would have become complex if abstracted due to the presence of *ifdef*s or *switch* statements. In one case the parameterized code fragment was cloned between two subsystems. We felt this small clone was more beneficial if left where it was. More commonly, these types of clones were judged to be harmful as they involved simple code that could be trivially abstracted. In these cases, the code would be more compact and easier to understand if the clones were removed.

The clones classified as the *replicate and specialize* pattern were rated as good 76% of the time. In all cases this was because the complexity introduced by unifying the code clones would make the code difficult to understand and maintain. In these cases, non-trivial changes were interleaved throughout the cloned regions. Examples of these changes were adding or removing statements, unsystematic changes to identifiers, and changes to data types. Several clones of this pattern were rated as harmful because obvious abstractions

were available and were deemed to make the code more clear if used. In these cases, the specialization could be neatly modularized into a single block of code or could be guarded by control flow statements.

In the sample set of the small clones, 12 *cross-cutting concern* patterns were rated as good. These were aspects related to security, in particular checking that the command was safe to run in the current program context. These code clones were judged to be good as they explicitly stated that the function was security sensitive, something that was not included in the source code documentation. A single RGC of this type was rated as a harmless clone because it comprised of a single procedure call. *Cross-cutting concerns* appeared very infrequently in the sample set of the larger clones. This was not surprising because this type of clone typically appears as small portions of code. The clones of this type were found essentially by chance: the subsequent code to the aspect was structurally similar enough to be matched by the code clone detection but was not actually cloned code.

The *verbatim snippets* clones found in the sample set of smaller candidate code clones were code fragments involving control flow. It was deemed that these code clones were good as abstracting them would adversely affect the conceptual cohesion of the functions they appeared in. *Verbatim snippets* clones found in the set code clones with a minimum size of 60 tokens were mostly deemed to be harmful. In these cases, the snippets consisted of complete conceptual units of code, such as initializing a data structure and then error checking, or dealing with differences in how a new line is represented in various operating systems. In one particular case of verbatim cloning, a whole file (`abts.c`) was cloned.

5.5.2 Cloning in Gnumeric

The results from the categorization of the clones in Gnumeric are summarized in Table 5.5 and Table 5.6. The sample size for the data shown in Table 5.5 was 71 RGCs and the sample size of the data shown in Table 5.6 was 71 RGCs. In Table 5.5 we see that 57% of the RGCs sampled were judged to be harmful. This does not sharply contrast the results of the second sample set of the larger clones, where 42% of the clones were judged to be harmful. The Gnumeric source code has a large amount of mathematical computations. The interface between the spreadsheet and the mathematical computations

Pattern	**Good**		**Incidental**		**Harmless**		**Harmful**		**Total**
	#	%	#	%	#	%	#	%	
Forking	**0**	**n/a**	**0**	**n/a**	**0**	**n/a**	**0**	**n/a**	**0**
Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Platform variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Experimental variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Templating	**10**	**31.3%**	**5**	**15.6%**	**0**	**0%**	**17**	**53.1%**	**32**
Boiler-plating	3	100%	0	0%	0	0%	0	0%	3
API	0	0%	5	83.3%	0	0%	1	16.7%	6
Idioms	2	100%	0	0%	0	0%	0	0%	2
Parameterized	5	23.8%	0	0%	0	0%	16	76.2%	21
Customize	**9**	**56.3%**	**0**	**0%**	**0**	**0%**	**7**	**43.8%**	**16**
Replicate and Specialize	9	56.3%	0	0%	0	0%	7	43.8%	16
Bug Workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact Match	**4**	**18.2%**	**0**	**0%**	**2**	**9.1%**	**16**	**72.7%**	**22**
Cross-cutting	2	50%	0	0%	0	0%	2	50%	4
Verbatim snippets	2	10.5%	0	0%	3	15.8%	14	73.7%	19
Total	23	32.9%	5	7.1%	2	2.9%	40	57.1%	**71**

Table 5.5: Clones by type - Gnumeric 1.6.3 - 30 Tokens

Pattern	**Good**		**Incidental**		**Harmless**		**Harmful**		**Total**
	#	%	#	%	#	%	#	%	
Forking	**0**	**n/a**	**0**	**n/a**	**0**	**n/a**	**0**	**n/a**	**0**
Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Platform variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Experimental variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Templating	**17**	**33.3%**	**8**	**15.7%**	**0**	**0%**	**26**	**51.0%**	**51**
Boiler-plating	6	85.7%	0	0%	0	0%	1	14.3%	7
API	0	0%	8	88.9%	0	0%	1	11.1%	9
Idioms	1	100%	0	0%	0	0%	0	0%	1
Parameterized	10	29.4%	0	0%	0	0%	24	70.6%	34
Customize	**15**	**93.8%**	**0**	**0%**	**0**	**0%**	**1**	**6.25%**	**16**
Replicate and Specialize	15	93.8%	0	0%	0	0%	1	6.25%	16
Bug Workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact Match	**1**	**25%**	**0**	**0%**	**0**	**0%**	**3**	**75%**	**4**
Cross-cutting	0	n/a	0	n/a	0	n/a	0	n/a	0
Verbatim snippets	1	25%	0	0%	0	0%	3	75%	4
Total	33	46.5%	8	11.3%	0	0%	30	42.3%	**71**

Table 5.6: Clones by type - Gnumeric 1.6.3 - 60 Tokens

requires a function that initially converts the values of the cell data to data types that can be used for mathematical computations (such as integers and floating point numbers). A typical scenario we found in both sample sets was cloning that consisted of the unchanged duplication of the initialization of variables and then the identifier naming the procedure to be called was changed. This high percentage of "harmful" cloning is a reflection of this type of scenario. The interface between computation functions and the spreadsheet also contributed to the high percentage of false positives in this study. When the long series of cell conversions are abstracted to a *p-string*, they become identical, and because they usually do not repeat identifiers in the assignment, the one-to-one mapping almost always succeeds. This points out a particular weakness in the method of clone detection used in this case study.

The reader will note that no *forking* patterns were found in this study. This is likely due to the fact that there are few external dependencies on other systems such as databases or operating systems in this source code.

Boiler-plating clones found in the Gnumeric study were generally rated as good clones with one exception. These clones were again caused by changes to data types. An example of this type of cloning is shown in Figure 5.3. Refactoring these segments of code would require not only wrapper functions to provide explicit type information but also a deeper refactoring of the data structures of *py_RangeRef_object* and *py_Range_object* to allow the reuse of the assignment of the member *range_ref* and *range*. Due to considerations such as this, we felt that the clarity of these simple code clones likely outweigh the benefits of unifying the code. The single exception rated harmful had an obvious abstraction that could be made without adding complexity.

Most cloning categorized as *API/Library protocol* clones were rated as incidental for the same reason as they were in the Apache clone samples. In this case, the cloning involves the connection of UI signals to actions. These clones cannot be abstracted further because they are already calling a single function. While there is high degree of similarity between many individual clone pairs of this type, the overall variability of the total set make it very difficult to create an appropriate abstraction that would reduce code size without introducing a very high degree of complexity. The two exceptional cases where this pattern of cloning was judged to be harmful were clone pairs that appeared frequently

```
static PyObject *
py_new_RangeRef_object (const GnmRangeRef *range_ref)
{
        py_RangeRef_object *self;

        self = PyObject_NEW (py_RangeRef_object, &py_RangeRef_object_type);
        if (self == NULL) {
                return NULL;
        }
        self->range_ref = *range_ref;

        return (PyObject *) self;
}
```

(a)

```
static PyObject *
py_new_Range_object (GnmRange const *range)
{
        py_Range_object *self;

        self = PyObject_NEW (py_Range_object, &py_Range_object_type);
        if (self == NULL) {
                return NULL;
        }
        self->range = *range;

        return (PyObject *) self;
}
```

(b)

Figure 5.3: An example of *boiler-plating* in Gnumeric

in the system and had an obvious abstraction using macros. Frequency can be a concern when assessing the overall maintenance time that might be saved by reducing all of the clones to a single, simple abstraction.

Only three *idioms* were found in the sample sets for the study subject Gnumeric. These *idioms* were rated as good because they were relatively simple and provided context to what the code was doing. One idiom involved differing data types that could not be refactored. In the other two cases the *idioms* were deemed to be good because they were not at risk of being poorly implemented contrary to the case in Apache and provided clear context of the code responsibility.

Parameterized code clones were judged to be harmful 76% of the time in the sample of small clones, and 71% of the time in the sample of large clones. In nearly all of these cases, passing a function pointer as an argument to a single function would remove many of these clones without negatively impacting the comprehensibility created by the variable names in the functions. Fifteen of the clones of this type were judged to be good. In these cases, the identifiers were changed to reflect the standard notation of the variables in the mathematical functions they represented. We deemed this to be a good type of clone because it provided traceability from the code to the documentation (the mathematical function). In cases where *parameterized code* clones were judged to be harmful, we found that the parameterized code represented such similar concepts (such as related numerical functions) that there would likely be no loss in conceptual traceability of the code if it were to be abstracted.

When assessing *replicate and specialize* clones in the sample of small clones, nearly 44% of the RGCs were judged to be harmful. Three of these clones were variations on the type of *parameterized code* clones described above, with additional error checking added. In these cases, the abstraction was obvious. Three cases of harmful *replication and specialization* were the creation of a dialogue. The additional code added to one function could be factored out and the clones could be parameterized and merged.

Good cloning of the type *replicate and specialize* had non-trivial changes. While these changes were typically smaller than the ones seen in Apache httpd, we deemed them to be beneficial because of the semantic traceability they created to the mathematical equations they represent. A clone of this type is shown in Figure 5.4.

```
gnumeric_randnegbinom (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
        gnm_float p = value_get_as_float (argv[0]);
        int failures = value_get_as_int (argv[1]);

        if (p < 0 || p > 1 || failures < 0)
                return value_new_error_NUM (ei->pos);

        return value_new_float (random_negbinom (p, failures));
}
```

(a)

```
gnumeric_tinv (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
        gnm_float p = value_get_as_float (argv[0]);
        int dof = value_get_as_int (argv[1]);

        if (p < 0 || p > 1 || dof < 1)
                return value_new_error_NUM (ei->pos);

        return value_new_float (qt (p / 2, dof, FALSE, FALSE));
}
```

(b)

Figure 5.4: An example of *replicate and specialize* in Gnumeric

There were four *cross-cutting concerns* seen in the sample set from the small clones. The two that were judged to be good performed assertion checks of the parameters of the function and, implemented as macros, caused the function to return if they failed. The RGCs of this type that were deemed to be harmful were fragments of code consisting of several steps responsible for removing references to dynamically allocated variables. In these cases, removing references involved a call to a function that decrements the number of references to the data and then sets the pointer to *NULL*. We felt this could be better encapsulated as a macro or procedure call. This would avoid the risk of missing the step of setting the pointer to *NULL*.

As with the Apache study subject, most of the *verbatim snippets* clones were judged to be harmful, in this case 74%. Only clones that were small fragments were deemed to be good as the code was small and incomplete on its own. We felt the resulting abstraction would have degraded the understandability of the procedure they were found in.

5.6 Case Study Discussion

In the case study presented in this chapter, we try to answer Questions two and three of this thesis:

Question 2 *Are there common patterns of code cloning that occur in the development of software systems?*

Question 3 *How are code cloning patterns used in practice, and to what extent is their use appropriate?*

While Chapter 4 partially answers Question two, it is necessary to measure how often these patterns appear in source code. The results clearly show that these patterns appear with non-trivial frequencies in the two study subjects we investigated, with the exception of *bug-workarounds* and *hardware variations*. In the case of *bug-workarounds* this exception is possibly an indication of the rarity of such a pattern and that examples of its use may be special cases. While this form of cloning has been observed during prior case studies, it may not be as common as first believed and may not be justified as a pattern. To investigate this further, one direction of future work is to detect and analyze more clones

of this type. In the case of *hardware variation* it was not expected to be sampled as neither software system directly interacts with a hardware device. However, from previous work we know that this pattern is easily discernible in other systems, such as the Linux kernel [32, 53]. Chapter 4.2.1 lists two families of drivers that demonstrate this pattern. To properly evaluate this pattern, studies focusing on the analysis of driver code must be performed. Candidate study subjects might include Xorg, Linux, FreeBSD, and sane.

Comparing Table 5.3 and Table 5.4 we can see that, in most cases, specific types of clones were classified similarly in both sample sets. Two notable exceptions to this observation are the *idiom* clones and the *verbatim snippets* clones. In the case of *idiom clones*, as mentioned previously, this is a surprising result as we expected to find *idioms* in the sample set of clones detected using a minimum token length of 30. In the case of the two *verbatim snippets* seen in the sample with a minimum clone size of 30 tokens, they were considered good because these were very small fragments contributing to control flow and not complete fragments of code on their own. In the sample set of clones with a minimum size of 60 tokens, eight of the clones were large, identical clones representing complete conceptual units that had obvious abstractions that would improve the code. Only one clone of this type was classified as good in the sample set. This is not surprising as one would expect that in many cases large, identical blocks of code would be more easily maintained when refactored into a single unit. Because the set of clones with a minimum size of 60 tokens is a subset of the set of clones with a minimum size of 30 tokens, increasing the number RGCs sampled from latter set would have likely revealed these clones.

The total number of verified code clones sampled varies largely between the two samples in Apache (Table 5.3 and Table 5.4) but does not vary between the samples in Gnumeric (Table 5.5 and Table 5.6). This is due to the larger number of RGCs sampled out of the clones detected in Gnumeric with a minimum size of 30 tokens. The lack of precision of the sample sets with a smaller minimum threshold, shown in Table 5.2 and highlighted in the comparison of the number of actual clones sampled, demonstrates a particular weakness of parameterized string matching. Smaller sequences of tokens become increasingly similar as less information is provided to differentiate them. Tokens become "anonymous" when they are unique within a sequence because there are no back references to them, removing any information about order. Smaller sequences of code are more likely to contain a high

percentage of unique tokens, making these sequences appear to be clones. The imprecision of the clone sample sets, particularly when using a minimum token size of 30, had a large impact on the tractability of our study. While precision is greatly increased when the minimum threshold is increased, certain observations, such as the difference between the classification of *replicate and specialize* in Gnumeric, could not have be made unless we had sampled the set of smaller clones. This observation and the results shown in Table 5.2 emphasize the need for more sophisticated filtering of clone detection results.

Cloning in Apache httpd and Gnumeric appears to be qualitatively different. Figure 5.5 and Figure 5.6 summarize the percentage of RGCs categorized into each of the four groups of cloning patterns. *Forking* did not appear in Gnumeric, and the *templating* patterns appear to be a much larger contributing factor to the overall cloning when looking at the larger clones. This high number of *templating* clones appears to be largely due to the interface between the spreadsheet and the mathematical functions. The differences in the types of cloning found in the two study subjects can be seen as a reflection of the differences in design and purpose of the software systems. The focus of Apache httpd is to be a highly stable, portable web server. The focus on stability implies changes to stable core code will be done cautiously, or even avoided, when adding new functionality. This can lead developers to clone and change rather than modify working code. The focus on portability requires the implementation of a large set of common functionalities for many different platforms, with each of these common functionalities differing in implementation in non-trivial ways, again leading to cloning to avoid inter-dependencies amongst largely unrelated platform-specific code. The focus of Gnumeric is to be a broadly featured spreadsheet application. This functionality is accessed through a common interface, the cells of the spreadsheet, and the code using this interface, not surprisingly, will have much in common.

In both studies *templating* clearly represents the majority of patterns sampled when considering the sample set of larger clones. We feel this phenomenon is largely due to the type of clone detector we are using. Specifically, the type of clone that parameterized string matching finds is the *templating* pattern because of the definition of a match: a sub-string whose identifiers have a bijective mapping. Structural or logical changes will not match. We feel that the results from the sample sets using smaller clones are a better representation of their relative frequencies because smaller clones have been grouped to

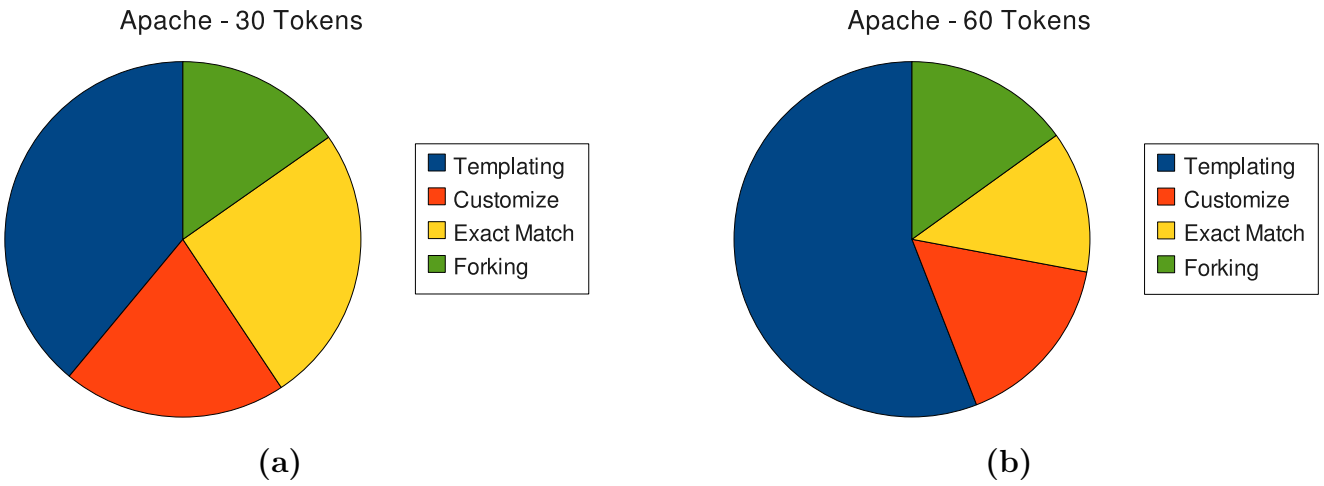

Figure 5.5: Percentage of clone types in Apache.

form larger clones, and this grouping is less sensitive to changes in logic and structure.

As can be seen in Figure 5.5 and Figure 5.6, the ratio of *exact matches* decreases dramatically when the minimum clone size is increased to 60 tokens. This indicates that developers are less likely to directly copy-and-paste large segments of code if they do not require enhancement or modification. In contrast, *customization* patterns did not decrease in relative frequency, indicating that the size of the code plays a role in developers' decisions to create a code clone or abstract.

Although there are differences in the overall frequency of the clone patterns in the study subjects, all clones sampled were categorized using the catalogue of clone patterns. Also, most patterns were found in the samples demonstrating their relevance to real software systems. The absence of hardware variation clones and the qualitative difference in the types of clones found in the two study subjects suggests that factors such as the design and purpose of the software can affect the types of code clones one can expect to find. Future work in characterizing software systems and the relative occurrences of these patterns will benefit software practitioners as it can provide a point of comparison to assess the quality of their own code.

Tables 5.7 - 5.8 summarize the scope of the clones for each cloning pattern found in

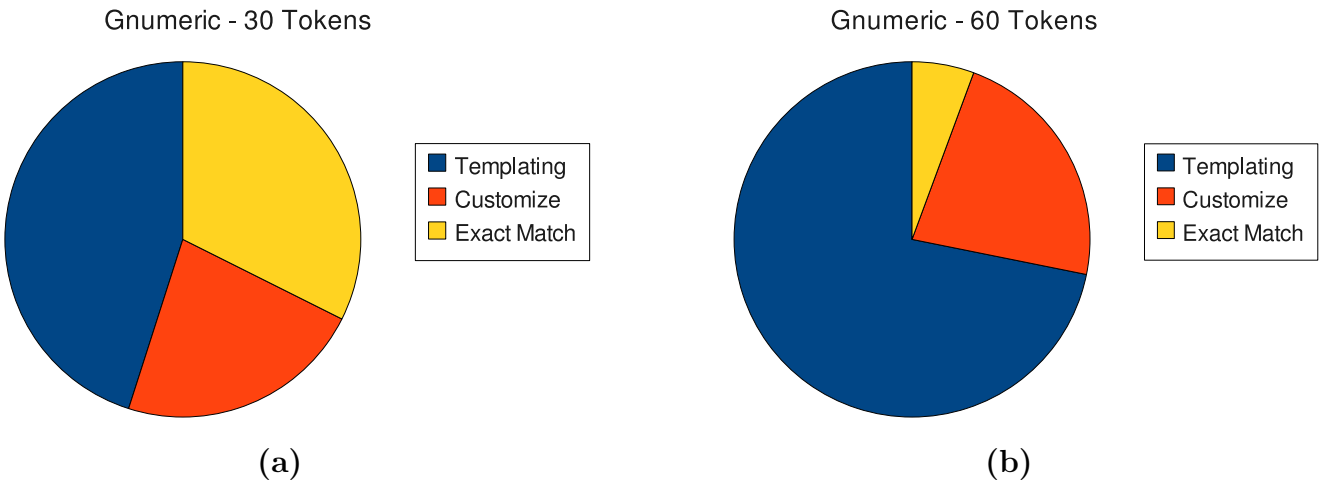

Figure 5.6: Percentage of clone types in Gnumeric.

the sample sets. In these tables, the column "Scope" refers to the scope cloning was considered to have over the region. "Full" indicates that the cloning covered the majority of the regions and "Fragment" indicates the cloning covered a fragment of the regions. As can been seen in the tables, in general the intuition in the pattern descriptions is supported by the evidence collected in the case study, with a few exceptions. In the cases where *verbatim snippets* is "Full" scope, the cloning either occurs in very small functions, data structures, or preprocessor directives, the last being the most frequent case. These tables indicate that the cloning we observed often involves entire regions, usually functions. This is likely due to the ease of finding and duplicating entire functions as opposed to code fragments. Code fragments are more likely to be re-implemented rather than sought out for cloning and this re-implementation will be more difficult to detect as it will likely only have semantic and not representational similarity.

By rating the harmfulness of code clones, we address Question three of this thesis. Our results show that a non-trivial number of RGCs can be categorized, according to our criteria and our judgement, as good forms of cloning: in Apache 71% were classified as good in the sample set of clones with a minimum length threshold of 30 tokens and 42% were classified as good in the sample set of clones with a minimum length of 60 tokens,

		30 Tokens		60 Tokens	
Pattern	**Scope**	**Count**	**% in sample**	**Count**	**% in sample**
Forking	**Full**	**9**	**100%**	**14**	**100%**
	Fragment	**0**	**0%**	**0**	**0%**
Experimental variation	Full	0	N/A	4	100%
	Fragment	0	N/A	0	0%
Platform variation	Full	9	100%	10	100%
	Fragment	0	0%	0	0%
Templating	**Full**	**17**	**77.3%**	**37**	**71.2%**
	Fragment	**5**	**22.7%**	**15**	**28.8%**
Boiler-plating	Full	7	87.5%	5	100%
	Fragment	1	12.2%	0	0%
API	Full	6	75%	17	100%
	Fragment	2	25%	0	0%
Idioms	Full	0	N/A	0	0%
	Fragment	0	N/A	12	100%
Parameterized	Full	4	66.7%	15	83.3%
	Fragment	2	33.3%	3	16.7%
Customize	**Full**	**10**	**76.9%**	**16**	**100%**
	Fragment	**3**	**23.1%**	**0**	**0%**
Replicate and Specialize	Full	10	76.9%	16	100%
	Fragment	3	23.1%	0	0%
Exact Match	**Full**	**0**	**0%**	**4**	**36.4%**
	Fragment	**15**	**100%**	**7**	**63.6%**
Verbatim snippets	Full	0	0%	4	44.4%
	Fragment	2	100%	5	55.6%
Cross-Cutting	Full	0	0%	0	0%
	Fragment	13	100%	2	100%

Table 5.7: Scope of clones by type - Apache httpd 2.2.4 - 30 Tokens and 60 Tokens

		30 Tokens		60 Tokens	
Pattern	**Scope**	**Count**	**% in sample**	**Count**	**% in sample**
Templating	**Full**	**21**	**65.6%**	**50**	**98%**
	Fragment	**11**	**34.4%**	**1**	**2%**
Boiler-plating	Full	3	100%	7	100%
	Fragment	0	0%	0	0%
API	Full	1	16.7%	9	100%
	Fragment	5	83.3%	0	0%
Parameterized	Full	16	76.2%	34	100%
	Fragment	5	23.8%	0	0%
Idioms	Full	1	50%	0	0%
	Fragment	1	50%	1	100%
Customize	**Full**	**14**	**87.5%**	**16**	**100%**
	Fragment	**2**	**12.5%**	**0**	**0%**
Replicate and Specialize	Full	14	87.5%	16	100%
	Fragment	2	12.5%	0	0%
Exact Match	**Full**	**6**	**26.1%**	**3**	**75%**
	Fragment	**17**	**73.9%**	**1**	**25%**
Cross-cutting	Full	0	0%	0	N/A
	Fragment	4	100%	0	N/A
Verbatim snippets	Full	6	31.6%	3	75%
	Fragment	13	68.4%	1	25%

Table 5.8: Scope of clones by type - Gnumeric 1.6.3 - 30 Tokens and 60 Tokens

and in Gnumeric 33% were classified as good in the sample set of clones with a minimum length of 30 tokens and 57% were classified as good in the sample set of clones with a minimum length of 60 tokens. In other words, we have empirically evaluated the generally accepted wisdom "Cloning considered harmful" and found it to not be generally true. While a non-trivial number of clones were judged to be good it should be noted that, as described in the patterns listed in Chapter 4, these clones were not judged to be free of maintenance risk either. As with any principle of design or development, there are trade-offs and maintenance concerns that should always be evaluated while developing software systems.

Table 5.3 through Table 5.6 also show that specific types of clones' harmfulness were classified fairly consistently, with the notable exception of *replicate and specialize* clones. The larger *replicate and specialize* clones found in the Gnumeric set of clones with a minimum size of 60 tokens had more non-trivial changes compared to many of the clones sampled in the sample set with a lower minimum size. From these results we can see that in cases where larger clones are non systematically modified the complexity of the abstraction makes cloning a better alternative to refactoring.

5.7 Threats to Validity

There are several threats to the internal validity of this analysis. First, the RGCs were judged by a single expert observer who is the author of this thesis. Without additional judges in this study there is no way to measure bias. However, experts in code cloning research are uncommon making a larger study much more difficult and costly to conduct. Further, disagreement on the definition of code clone within the research community remains an obstacle to overcome before a group study of this nature can be performed.

The choice of detector and analysis environment is another threat to internal validity as it may bias the results of the studies. The CLICS clone detector uses parameterized string matching and does not find clones with reordered statements or statements added and removed. These types of clones would be *customization* clones. The number of missed clones is difficult to estimate; however, by examining the clones using a small minimum size for a clone match we have provided a reasonable estimate of the lower bound on the percentage

of RGCs in a software system that would be regarded as *customization* clones. Grouping smaller clones, as in the case of the sample sets of candidate code clones of a minimum size of 30 tokens, largely overcomes statement insertions and deletions when several lines remain structurally similar on either side of the changes. Clone detection methodologies using program dependence graphs overcome this limitation [64] but tools implementing this technology were not available to us for this study. However, parameterized sub-string matching is one of the top clone detection methods in terms of recall [16, 62] and we feel it is well suited for the goals of our study.

The choice of study subjects is a threat to the external validity of this analysis. Both are medium-sized (just over 300,000 LOC) software systems, and the results may change when analyzing larger systems. Also, these study subjects are open-source software projects with many developers distributed throughout the world. Management and organizational partitions of responsibility that are present in closed-source software projects may not be mirrored in open-source software projects. The typical development model for these projects does not restrict developers from modifying code throughout the system. However, in the case of Apache we see that most changes to the source code are made by relatively few developers and there appears to be an implicit agreement of ownership within the development community [76]. In the case of Gnumeric, we found that the code is maintained by few developers. In both cases, these scenarios are unlikely to differ by a large degree from closed-source development environments. Without further study into the qualitative similarities and differences of open-source and closed-source software systems, generalizing results is difficult. More broadly speaking, a better understanding and characterization of attributes of software is required before these results, or the result of many studies of software, can be applied or generalized to groups of software systems. For example, what is the effect of the problem domain on the overall design? This question has direct implications regarding the types of clones we can expect to find in a software system.

5.8 Summary

To better understand code cloning as it occurs in practice, we carried out qualitative investigations of code cloning in widely-adopted, medium-sized to large software systems,

which led to the discovery and documentation of the code clone patterns presented in Chapter 4. This chapter presented an empirical study of the prevalence of these patterns in software systems, showing that most patterns occur frequently in both study subjects and answering the second question of this thesis: yes, there are patterns of code cloning present in software systems.

This study also revealed that in many cases code clones are used appropriately. This suggests that the conventional wisdom the code clones are "bad" is not generally true. Further, popular maintenance advice to refactor code clones may be counterproductive. Benefits provided by code clones, when correctly used, will be lost by refactoring. Further, refactoring may result in introducing artifacts that that the code clone was used to avoid, such as complex code or dependencies between unrelated subsystems.

While many code clones in the study were judged to be good, the majority were still deemed to have a negative impact on the quality of the software system. This does suggest that code cloning is still a problem that must be managed. However, this problem may in part be the result of a lack of decision making tools regarding code cloning or even a lack of understanding of the tradeoffs for different types of clones: both situations may, in part, be the result of overly simplified views of cloning that have persisted in the software development community and education system.

Chapter 6

Discussion

6.1 Introduction

Throughout this thesis it has been mentioned that there have been relatively few investigations into how and why developers use code cloning. Most research prior to this work has focused on detecting, refactoring, and, more recently, tracking clones in software. While these are important avenues of research, we need a deeper understanding of how and why code clones are used in real software systems before we can evaluate the effectiveness of code clone detectors and proposed code clone maintenance techniques. To provide more insight into how code clones are used in software, this thesis presents a set of code cloning patterns found in real software systems. The descriptions of these patterns include motivations for creating code clones, anticipated maintenance concerns, and suggested maintenance activities regarding each clone pattern.

With a documented set of patterns of usage we not only hope to improve our understanding of how code clones are used in software, but we also hope to aid software practitioners in making better decisions about using code cloning. The belief that code clones are always bad in software is clearly false, in the general sense, to experienced software developers and maintainers. However, it may not be as clear to these same developers when the use of code clones is appropriate, as is indicated by the large number of harmful code clones sampled in Chapter 5. As code cloning patterns are further investigated and developed, they may help to enable software practitioners to make more informed decisions

and possibly make the decision making process faster. Furthermore, these code cloning patterns could be used to document existing code cloning activities. This documentation would facilitate maintenance decisions to be made in the future.

This chapter revisits the three motivating questions of this thesis and discusses the answers found through our case studies of code cloning. This chapter will also discuss some of the possible maintenance implications specific to individual cloning patterns and addresses these issues with suggestions for tool support and development processes. Finally, we introduce a novel design for a tool, integrated with source code control systems, to track and document code clones and provide maintenance advice according to the code cloning patterns described in this thesis.

6.2 Research Questions

Chapter 1 raised three questions about code cloning in software. This section summarizes the findings regarding those questions, as observed in the case studies performed by the author.

6.2.1 What are the Common Motivations for Cloning?

Question 1 *What are the common motivations for developers to use code clones?*

From our prior case studies it can be concluded that not all motivations for code cloning are negative reflections on the developers [49, 50, 51, 52, 54, 53]. While programmer laziness, lack of understanding, or quick fixes may contribute to some code cloning activities, we have found repeated examples where this is not the case. Common motivations that have been observed in the case studies discussed in this thesis indicate that many developers have principled engineering intentions in mind when duplicating code. A few of these motivations include:

- risk avoidance, a contributing factor to code cloning when new code to be introduced may affect the stability of the core or stable functionality of a working system;
- change decoupling, found when code clones are not expected to evolve synchronously such as in the case of *platform variation*;

- language constraints, occurring when a programming language does not provide the required abstraction facilities to create a unified solution;
- common code usages, such as idioms and API usage patterns, are often self-explanatory or generally understood and may not have useful abstractions;
- avoiding complex and hard-to-maintain code, as code clones may in some cases be easier to understand and change than a complex unification of the various similar problems;
- copying a temporary solution for work-arounds, which may not warrant refactoring of code when the work-around is inserted on a temporary basis;
- avoiding code fragmentation, a particular problem when code clones are comprised of small code snippets that have little or no meaning outside of their current context; and
- implementations of latent semantic properties of the software system (i.e., cross cutting concerns).

We have drawn this list of motivations for creating code clones from concrete sources such as in-line source code documentation and external design documentation. We have also made inferences about the motivations for creating clones based on the context of code clones within the overall software. For example, when examining the code cloning between the file-system implementations of the *ext2* and *ext3* file-systems, it is likely that cloning *ext2* and modifying it to add the new features for *ext3* was safer than modifying *ext2* to support both file-systems [50]. In this case one would consider this motivation to be risk avoidance. Instances of cloning between subsystems in Apache httpd that support operating-system-specific data types and behavioural nuances are likely motivated by change decoupling and avoidance of complex abstractions. Changes to Linux threading could create a multitude of testing and debugging problems if the code to support Linux and *BSD platforms were unified.

While we claim that many code clones are created with sound engineering goals in mind, we do not suggest that all such code clones are so easily justifiable. Code clones

created with good intentions may have negative maintenance consequences, such as challenges with change propagation, or run-time side effects, such as increased memory usage. As with any design decision code cloning must balance the negative and positive effects of the action in the context of the overall software system and its environment. For example, in software projects where the cost of the risk associated with a change is high, developers may disregard maintainability concerns completely when considering the use of code cloning. With this in mind, the evaluation of whether or not it is appropriate to use code clones in software must be made within the context of the particular software system, its development environment, and any other relevant engineering and business concerns.

6.2.2 Are there Common Patterns of Code Cloning?

Question 2 *Are there common patterns of code cloning that occur in the development of software systems?*

During the progression of the case studies conducted to answer the first question of this thesis, we noticed recurring patterns in how software practitioners used code cloning. Repeated patterns of cloning would suggest there are specific problems developers are trying to solve with this development "tool". The case studies we conducted revealed a set of code cloning patterns, described in Chapter 4. The evidence provided in Chapter 5 suggests patterns of code cloning do occur in software, though their relative frequencies vary between software systems.

The patterns of code cloning described in this thesis demonstrate a relationship between the problem the code cloning is solving, the intent of creating a code clone, and the manifestation of the code clone. For example, *experimental variation* will often encompass entire files or subsystems while *boiler-plating* generally encompasses only procedures. *Replicate and specialize* patterns of cloning exhibit different types of changes as compared to *general language idioms*. Each of these patterns has its own set of advantages and disadvantages, with some overlap, and in many cases, different maintenance strategies are required. This suggests investigations and evaluations of code cloning within a software system cannot be restricted to the measurement of code cloning as a single homogeneous entity but should be finer grained, possibly considering the various aspects of the patterns discussed in this

thesis.

We do not assert the patterns described in this work are exhaustive. The case study described in Chapter 5 confirms the patterns we describe occur in software systems, but also demonstrates that the list of patterns we originally reported [53] was incomplete. During the study described in Chapter 5, three new patterns were added. This list of patterns may grow as more case studies are performed, and this is one focus of future work on this topic.

6.2.3 How are Code Cloning Patterns Used?

Question 3 *How are code cloning patterns used in practice, and to what extent is their use appropriate?*

We have found that the use of code cloning patterns in software varies from system to system. It appears that factors affecting their use include the software domain, the programming language used, the use of external dependencies such as libraries, and external forces such as market pressures. For example, a software system that depends largely on external software devices (such as an operating system or video decoder) is expected to contain *hardware variation* more than office processing software. Languages such as COBOL are not known to have the same abstraction features as C or C++, perhaps leading to more *boiler-plating*. Most modern applications must use external sources of code and behaviour to meet project goals efficiently. For example, libraries such as Qt and GTK+ provide platform abstraction to aid in writing more portable code. If these libraries do not provide a high level of abstraction in their interface, or their default behaviours do not fit the target application, one might expect to find more *API/Library protocol* clones than one might when using a library with a higher level of abstraction or one whose default behaviour better matches the target application's requirements.

In our studies we found that a significant portion of the code clones in software systems are using cloning patterns appropriately. We have identified motivations for code cloning and common code cloning patterns found in software. From these patterns and motivations we see that developers often use code cloning as an alternative to other forms of reuse when the overall benefits of code cloning are perceived to outweigh the negative. While many

code clones appear to be created with good intentions, it is important to know what proportion of code clones are not actually harmful to the quality of the software systems. Our analysis of sample sets of code clones, taken from two software systems, shows that a significant portion of code clones may have a beneficial impact on source code quality. The results of the case study described in Chapter 5 show a range of 13.6% to 57.1% of the code clones in the sample sets were judged to be harmful. This indicates that a significant number of code clones are either good, incidental, or harmless. These findings provide strong evidence that code clones should not be deemed to be generally harmful to software systems.

6.2.4 Use and Maintenance of Code Clones in Software

The discovery of code cloning patterns in source code, and the evidence supporting that they are often used appropriately, has implications on the interpretation of past code clone research as well as the direction of future research. While it has been suggested that code cloning is a serious problem in industrial software [3, 6, 15, 4, 26, 44, 43, 47, 60, 61, 74], this dissertation provides evidence that this is not always the case. Furthermore, the large number of code clones we deemed to be good or harmless suggests further research is needed regarding when and how factoring techniques, such as those proposed by Baxter [15], Balazinska et al. [8], Basit et al. [13], or Higo et al. [38], should be applied. For example, if code clones have been introduced to avoid complex code or design, the particular use of design patterns suggested by Balazinska et al. may contradict this rationale [8]. In other cases, such as the parameterized code clones we deemed harmful, automated refactoring tools could greatly reduce the effort required to consolidate the code clones.

The cloning patterns described in this dissertation also suggest the need for strong tool support for maintaining code clones, especially in the case where code clones should persist in the source code. The code cloning patterns also suggest that there are different management strategies required for each of the patterns. For example, *experimental variation* requires developers to monitor changes to the external interface of a cloned subsystem to make decisions on whether or not to propagate changes to the duplicated code. On the other hand, *boiler-plating* requires close synchronization of the maintenance effort, possibly through an automated approach such as source code generation. These varying

maintenance strategies require a variety of different tools.

Templating patterns strongly suggest a need for synchronous editing, as suggested by Toomim et al. [87], to manage code clones where evolution between the duplicates should be tightly coupled but abstraction is not possible. Even in the cases where abstractions are possible, such as in the case of *API and Library Protocols*, Toomim et al. provides evidence to suggest that there is less cognitive load required to manage the duplicated code with Linked Editing than performing the traditionally accepted abstraction.

In cases of duplication where the evolution of the duplicates may not be so tightly coupled, as in the cases of *forking* patterns, architectural and historical dependencies of cloning can guide developers to related points in the software system that should be taken into consideration during a maintenance operation. In [50, 51, 54] the author used cloning relationships visualized as architectural relationships as aids to locate several examples of these *forking* patterns. To support continued maintenance of these clones, these relationships should be part of the source code documentation, or even development environment, throughout the lifetime of the forked code.

In addition to locating *forking* cloning patterns, it is important that development tools also explicitly outline the similarities and differences in code clones. During our case studies, we noted that while it was easy to see similarities in code, it was far more difficult to find and understand the differences in the code. Identifying and understanding differences in the code clones is important as it affects decisions of how and when to propagate changes across code clones.

In the cases of *customization* patterns, the tool requirements are a combination of *forking* and *templating* patterns. In extreme cases of customization, automated tool support may not be possible for editing, and may not be desirable. Semi-automated approaches for patching code clones may be necessary, especially in cases of large groups of code clones. Such a tool would iterate over all candidate code clones and selectively patch clones according to human (expert) decisions.

While we believe that not all clones require refactoring, we also believe there are situations that warrant the effort. In cases where code is directly copied to duplicate behaviour, such as in sibling classes of an object-oriented program, refactorings should be performed if the language supports this. In situations where the behaviour of the clones is similar but

not the same, the effect of the costs of refactoring, such as effects on program comprehension and exposure to risk, should be measured against the expected gain in maintainability or extendability of the system.

6.2.5 Harmfulness of Code Cloning in Recent Work

To empirically measure the common belief that cloning is harmful due to the possibility of inconsistent updates, Aversano et al. closely analyzed how clones are modified over time [5]. Defining a set of evolutionary patterns based on the work of Kim et al. [58], they analyzed how maintenance activities affected clone classes. In particular, they investigated how and why some code clone classes changed together and others did not. Their findings show that in the majority of cases, clone classes are changed together (classified as a *consistent change* pattern [5]), particularly in the case of bug fixes and other forms of maintenance where it would be risky to not propagate changes to all clones. Aversano et al. note that non-risky changes (such as modifying visibility of class members) may not be propagated immediately but may be delayed (classified as a *late propagation* pattern [5]). They also found a large number of clones evolved independently, indicating developers use cloning as a development practice for starting new code [5].

Lozano et al. measured the consistency of changes between procedures that share code clones in DNSJava, a software system also analyzed in the two studies mentioned above [72]. Lozano et al. reported that procedures with clones between them change inconsistently, suggesting code clones may be harmful as they may pose a risk of inconsistent updating. They report that most procedures that share a code clone do not change together, and those procedures that do contain clones tend to change more often, presumably because the developers are not aware of clone relationships between procedures. This result would suggest that developers *do not* update clones together. This result sharply contrasts the findings of Aversano et al., who found that in the case of DNSJava 74% of clones change together. The differences in these results may be caused by the differences in the clone detection tools used. In the case of Aversano et al. an AST-based clone detection tool similar to the method proposed by Baxter et al. [15] was used. This method of clone detection has been shown to have high precision but low recall [16, 62]. This may have resulted in overlooking clones that have several changes, which one might speculate are

more likely to have inconsistent changes over time. Lozano et al. chose to use CCFinder as a detection tool[47], a parameterized suffix tree approach similar to the one presented in this thesis. This approach is shown to have high recall but low precision [16, 62]. It is the experience of the author, supported by the findings in Table 5.2 in Chapter 5, that the low precision of this approach will falsely indicate that many procedures have a cloning relationship. These false positives will skew the overall number of procedures with a cloning relationship that are not consistently changed, and could perhaps have contributed to their conclusions.

Krinke suggests that when code clones are not changed consistently bugs may be introduced into the source code [65]. In studying changes made to code clones in five open-source systems over the period of 200 weeks Krinke found that code clones were changed consistently (changes were propagated to all code clones in a clone class) only half of the time. While it was suggested this may indicate that code clones are difficult to maintain, the code cloning patterns presented in this thesis may offer another explanation. Certain patterns of code cloning, such as *Forking* or *Customization* patterns, would not be expected to be changed consistently. Krinke also found that changes were not propagated later in time, suggesting that code clones remain inconsistent once an inconsistent change is made. This again is not unexpected for code cloning patterns where code is added or removed. However, if this is an indication that developers have difficulty changing code clones because they are unaware of their existence then tools should be provided to aid in the continued maintenance of code clones.

While interviewing and surveying developers and how they develop software, LaToza et al. uncovered six patterns of cloning based on motivation: repeated work, example, scattering, fork, branch, and language [68]. Repeated work occurs when two or more developers unknowingly duplicate effort to solve a similar problem. Example cloning is similar to our *idioms* and *API patterns*. Scattering directly maps to our *cross-cutting concerns*. Fork clones are similar to our *replication and customization* and *forking* patterns. Branch is not strictly a clone, but represents the repeated work required to propagate changes across branches of the entire source tree. Language involves implementing the same code in multiple languages. Two patterns, repeated work and language, are not covered by our cloning patterns. This is likely due to the fact that clone detection algorithms are unlikely

to find these types of clones. The implementation details, either due to developer design decisions or syntactic differences, vary enough that representational similarity matching will not uncover this type of duplication. Based on these patterns, LaToza et al. found that developers rarely clone code in the basic copy-and-paste fashion cited in much of the literature. For each pattern, LaToza et al. found that less than half of the developers interviewed thought the pattern was a problem. The findings of LaToza et al. are another indication that not all cloning is created due to poor development practice. The overall similarity of their patterns to our own provides additional support for the accuracy and relevance of the cloning patterns described in this thesis.

6.3 An Application of Cloning Patterns in Software Development

As a direct consequence of code clones requiring varied maintenance tools and strategies, code clones in a software system must be identified and categorized accordingly. To aid in maintenance decisions, suggestions should be given by the development environment about how to proceed with future maintenance. We believe that the code cloning patterns described in this thesis can help to address this problem. In this section we propose a high-level approach to integrating the identification and maintenance of code clones using a maintenance advisor based on categorization according to the patterns described in this thesis.

In this application of code cloning patterns, the maintenance advisor provides suggestions to the user about maintaining code clones based on the maintenance suggestions given in Chapter 4. Based on the type of code cloning pattern identified, this maintenance advice might be the automatic propagation of changes, as would be the case in *boiler-plating* clones, or suggestions regarding refactoring when the differences and similarities between code clones stabilize, as would be the case in *experimental variation*. The tool would also provide advice when the original intent of a code clone becomes less relevant or in retrospect is determined to be incorrect. The benefit to developers would be the automated support for maintaining code clones, an informed decision-making process based on clone-pattern-specific information, semantics attached to implicit software dependencies,

a trail of documentation when changes are not propagated, and an iterative process for identifying and classifying code clones that is much more tractable than the current process of analyzing a single snapshot of the source code.

To begin our discussion of an application of code clone patterns, consider the following scenario:

1. A developer has worked on a portion of source code, and now wants to commit it to a versioning system.

2. The developer runs the command to commit the code, and a clone detector runs on all the changed files (comparing changed files to themselves and to the set of files that have not changed). The detected clones are compared to a database (repository) of known clones.

   (a) Clones that have disappeared are further analyzed to determine if the disappearance was a result of refactoring or that the extent of changes have made the code clone impossible to detect.

   (b) New clones would be compared against existing clones to determine if a clone class has been expanded.

   (c) Old clones that still exist would be analyzed to see if changes have been made to any of the segments of code contained within them.

3. Based on the above the analysis, several actions will be required from the user:

   (a) New clones will need to be annotated and classified by the user.

   (b) Clones that disappeared should be confirmed by the user.

   (c) Clone classes that are expanding should be analyzed and a maintenance effort metric computed. If the maintenance appears to be growing more expensive, refactoring should be considered.

   (d) Code clones with changed segments of code, depending on their type, should be analyzed for potential change propagation. In cases where changes have not been propagated to all cloned segments of code, the user should be prompted to do so or document why the change is not to be propagated.

(e) Metrics concerning code volatility should be computed for code clones that have been changed. Code clones changing often and changing synchronously can either be refactored or remain in the code, depending on the type of clone. (*Platform variation* may never be refactored, where as experimental variation may be refactored once changes involving common code are being propagated most of the time).

4. After the user completes the analysis process, all results are stored in a central repository.

In the above example, the advisor uses historical data and the cloning patterns to suggest maintenance advice. This proposed solution requires a code clone detection tool, tools to track changes in files and clones, a user interface for classifying detected clones, and a data storage mechanism for storing data about the categorized code clones that can be accessed with the source code repository.

To maintain code clones in software, the clone detection tool requires high recall. It is better to have the user mark false positives than to miss clones that should be tracked. While parameterized string matching is one of the best detection methods with respect to recall, it is still not perfect. An improvement to the detection algorithm might be to use copy-and-paste activities as hints for closer scrutiny, along with the standard algorithm. Such an approach would log all copy-and-pastes and use this information to perform fine-grained analysis, such as common sub-sequence extraction, of the source code in and around copy-and-paste activities. Because code clones may not always arise through copy-and-paste within the development environment traditional clone detection should still be used.

Detecting new clones and analyzing the history of existing clones requires tools for retrieving previous versions of the source files. Tracking changes in source code can be done using source code versioning systems. These systems provide mechanisms for efficiently storing submitted, or committed, versions of files over time. Mechanisms to retrieve copies of specific versions are always included as part of these systems. Versioning systems can either be centralized (e.g., cvs and subversion) or distributed (e.g., bazaar, git, and mercurial). Centralized versioning systems store the source code versions in a central location and have the advantage that those with access can see all commits as soon as they are performed. Distributed versioning systems store all of the versioning information locally

on an developer's/maintainer's computer and committed versions are only made public if they are transferred and imported into another developer's source tree. These types of systems have the advantage of allowing disconnected development where commits are only made public when the developer is ready. Distributed version control systems are rising in popularity, and thus the maintenance tool proposed here should support this functionality.

One problem with tracking changes that is not solved by source code control is determining if changes are the result of creating new code or if the code has been moved from elsewhere, a problem known as *origin analysis* [34]. When code is moved, it appears as if one segment of code has been removed while another has been added. When analyzing and tracking code clones found in the original segment of code it would appear as if clones have been removed and new ones have been added to the software. While source code versioning can provide access to previous versions of the source code, they do not provide mechanisms for interpreting the differences between two versions. Early work on origin analysis identifies the origin of procedures, taking advantage of call graphs, procedure signatures, and text similarity [34]. The approach described by Zou and Godfrey is only semi-automated and would require improvements to automate the process.

The interface presented to the user does not require many of the aspects of clone analysis discussed in Chapter 3.7 as they should already be familiar with the code they are maintaining and adding to the system. The interface should present a list of new code clones to the user and for each code clone allow them to select the cloning pattern used, the motivation behind creating the clone, and the intended duration of the code clone's existence (which may be inferred by the motivation for creating the code clone). All of these attributes should be predefined as they are used to determine how future maintenance of the code clone should be carried out. Pre-existing clones that have been modified should also be displayed to the user. For each code clone, the maintenance history should be accessible, including prior maintenance decisions regarding the code clones, and the system should provide maintenance advice whenever possible. There should be facilities to document any decisions made.

Code clones and their annotations must have the same degree of accessibility as the versioning system. In the case of centralized versioning systems, information about code clones may be stored in a central database. For large systems with many code clones it

may be necessary to take advantage of the optimizations modern database management systems provide for data processing. Distributed version control systems would be limited by a central database. For such systems, the data storage would have to be part of the versioning meta-data so that it can be imported and exported to other copies of the source tree. In both cases, the code clones and their annotations must support all operations that the chosen version control system provides for manipulating the source code, including renames, commits, rollbacks, branch merging, etc.

Duala-Ekoko and Robillard have also described a tool that will track clones across versions and notify the user when clones are not modified consistently [24]. The tool proposed here improves on their idea by suggesting that code clones can be documented when they are initially added to the software system. Also, maintenance advice can be given by generating advice specific to each code clone, based on historical metrics and identified code cloning patterns.

6.4 Summary

To improve our understanding of code clones, we have investigated three open questions concerning the use of code cloning in software systems. In this chapter, we discuss our first question: what are common motivations for code cloning? Our investigations into code clones leading up to and including this thesis have revealed a number of motivating factors when using code clones. In this chapter we discussed these motivations and discovered that many reasons to create code clones relate to maintaining an extendable, understandable, and testable code base.

While answering our second question, are there recurring ways in which code cloning is used in software, we have described a catalogue of cloning patterns. This catalogue, derived from case studies into code cloning in real software systems, connects specific motivations for creating code clones with typical manifestations of the pattern in source code and maintenance implications that should be evaluated. We have shown these patterns are repeated in two software systems, with varying degrees of frequency. We expect that this catalogue of patterns will grow in the future as applications in different domains and states of maturity are investigated.

To understand how code cloning patterns are used, and the appropriateness of their use, we subjectively evaluated a sample set of code clones and their effects on software quality. We concluded that a large number of code clones are used appropriately and their use improves source code quality. However, there remains a similar number of code clones that are harmful to overall source code quality, leading us to believe that software practitioners may require more guidance on the use of code cloning as a principled engineering practice.

Based on the evaluation of code cloning patterns in software systems, we found that specific maintenance strategies are advisable for each cloning pattern. In general, we believe that code cloning can be used as a principled engineering practice if the appropriate maintenance tools are utilized. In the preceding text we have suggested possible tools for maintaining specific patterns of code cloning and outlined some of the desirable features these tools should have.

Finally, we suggested a novel process to complement source code control in managing and documenting code clones as they are created or committed into the source code. We outlined the behaviour of a source code advisor that would track and advise on maintenance actions for code clones created using each type of code cloning pattern. In doing so, we suggested requirements for such a tool as well as a possible architecture for integration with modern source code control systems.

Chapter 7

Conclusions

Prior to the work presented here, code clone research had largely ignored the question of how the practice of code cloning is employed in developing software systems. In particular, there was little evidence supporting the conventional wisdom about the harmfulness of code clones, nor was there much investigation into the types of code clones present in software. The work presented in this thesis tries to address these questions through investigations of code clones in real software systems. We found evidence that software practitioners often use code cloning in a principled way. By identifying code cloning patterns, we identified common code cloning motivations and manifestations. We also evaluated the appropriateness of the use of the code cloning patterns in two open-source software systems. Our evaluation provides strong evidence that a large portion of code clones in software systems are the results of sound engineering decisions.

The code cloning patterns presented in this thesis are a step toward building a language for discussion that can be further used to evaluate and interpret research in this area. This language, in the form of code cloning patterns, endeavours to provide a flexible framework on top of which we can document our knowledge about how and why cloning occurs in software and how resulting code clones should be maintained over the life of a software system. It is our hope that this documentation will crystallize a vocabulary that researchers and practitioners can use to communicate about cloning.

These patterns may also help to address the problem of forming a generally accepted definition of a code clone. While the high-level concept of what constitutes a clone is

generally accepted — a code clone is two or more similar segments of code — there is little agreement on a more concrete definition. Often factors such as code cloning rationale, future maintenance strategies, and a specific definition of similarity play a role in deciding what a code clone is. The patterns of code cloning presented in this thesis encompass many of these varying interpretations of the general definition as specific instances of code cloning. This has the advantage over the generic term "code cloning" as each pattern can be defined more precisely without the risk of excluding an individual's definition of code cloning; a definition of code cloning not covered by the patterns presented here is likely to be yet another code cloning pattern.

The list of patterns presented in this thesis is not asserted to be complete. Further studies may reveal code cloning patterns that we have not yet observered. For instance, we can forsee code cloning patterns in scenarios where multiple languages are used. Another source of additional code cloning patterns is in the domain of functional programming, something this work has not addressed.

A major contribution of this thesis is the insights it provides into how and why developers clone code rather than forming higher level abstractions. These insights are based on evidence taken from real software systems and has applications in the clone detection and analysis research community. This understanding of the types of clones that exist in source code may help focus work on improving clone detection methodologies. Understanding common ways that developers use code clones may also aid in developing tools and methods to better manage code clones as they are created or discovered. One such tool is suggested in Chapter 6.

We also described a set of observations that can provide guidance into the design of future code clone analysis tools. Throughout the investigation of cloning in software we gained insights into the problem of analyzing clones in large systems [31, 50, 49, 51, 52, 54, 53]. These insights led to a high-level description of the requirements of an effective clone analysis environment and the development of a prototype [52, 54] currently being used in on-going research.

Furthermore, in this work, an additional categorization of cloning is introduced and discussed in Appendix A. This categorization is a taxonomy of clones in software systems that takes into account the locality of the clones within the software architecture, the type

of code the clones reside in, and the similarity of the regions the clones reside in. This has not been done in previous work and was an important step toward understanding the problems involved with the management of code clones in software systems. Our clone navigation tool, the *Clone Interpretation and Classification System (CLICS)*, incorporates the taxonomy as a way of displaying clones, as well as a method of filtering clones.

7.1 Future Work

We see two main streams of future work. Recent experience in industry by the author has made it evident that code cloning in large, multi-project software systems is a common method of reusing behaviour. While investigations into code cloning amongst projects in the open-source community have been undertaken, these studies analyzed projects that were generally not tightly coupled. In the context of large inter-communicating projects, code cloning can be employed as a method to reuse input and output behaviour, separate runtime or build time dependencies, and enable backwards compatibility with deprecated products or services. While these code cloning patterns have been observed, and used, by the author, their prevalence and usefulness over other forms of reuse must be evaluated. How does code cloning across tightly coupled software projects compare qualitatively to code cloning within a single large software system?

In the case studies leading up to and including this thesis, the study subjects have been primarily written in the programming language C or C++. Further investigations are required to compare code cloning in other languages, including fourth generation languages such as SQL that have not been discussed in prior literature. In addition, the identification of language features that assist in avoiding potentially harmful code cloning may lead to better design of new languages, and enhancement of existing languages.

Appendix A

A Taxonomy of Clones

A.1 Introduction

The following appendix describes one of the major tools in our analysis of code cloning in software, a taxonomy of clones. This taxonomy has been created through manual inspection of thousands of code clones from several large software systems. The resulting artifact has been validated and used in several case studies [49, 51, 50, 52, 53, 54].

Such an artifact is useful for several purposes. In this thesis, we use the taxonomy to gain more insight into the clones we are investigating. The taxonomy is a method of displaying clones for exploration, as shown in Chapter 3. The taxonomy provides additional information about the clones such as location and scope that can be used when identifying maintenance opportunities. A design-centric taxonomy, such as the one presented here, was essential in growing insights into what, where, and how code clones are used in software. By partitioning code clones into identifiable groups, such as function clones and clone blocks, we are able to ask more directed questions thereby facilitating the growth of our understanding of code clones. For example, by further analyzing clone blocks — code clones that comprise only small portions of a function — we identified the use of code cloning for initialization and verification of preconditions for functions.

The taxonomy is also used for filtering code clones, as discussed in Section 2.5. Each clone type has its own characteristics of layout and complexity, making each type of code clone susceptible to different types of false positives. Sorting clones into a taxonomy

provides opportunities to use specialized filters that are designed based on the structure of the code in question [51].

This appendix describes the structure of our code clone taxonomy, including a description of each code clone category, the subgroups contained in each category where applicable, and observations we have made throughout our case studies.

A.2 The Code Clone Taxonomy

The code clone taxonomy is a hierarchical classification of code clones, shown in Figure A.1. Dark arrows indicate subgroups and dotted lines connecting dark arrows indicate that the connected groups are also included as subgroups. For example *Same File Clones* contains *Same Region Clones*, *Function to Function Clones*, *Structure Clones*, *Macro Clones*, *Heterogeneous Clones*, and *Misc. Clones*.

The categorization of clone types at higher levels of the taxonomy are grouped according to the pairs of regions they reside within. This group is called a *Regional Group of Clones (RGC)*. A *RGC* is a group of clones that share the same two regions. For example, all clones between Function A and Function B will be grouped as a single *RGC*. In some cases, these *RGC*s can be used form a single large clone when the two regions are sufficiently similar. For example, two functions that have 60% of their tokens in common with each other are considered *Function Clones*.

The rest of this chapter is organized as follows. Each subsection describes a specific level in the taxonomy and the clone types that it are contained within it. Each subsection includes a description of how clones are categorized at this level, an informal description of each group in that level, and several observations we have made about each group. When it is appropriate, we provide examples of the types of clones found.

A.2.1 Partition by Location

Description

At the first level of the taxonomy, clones are partitioned based on the location of the two code segments within the software architecture. We consider software to be structured as

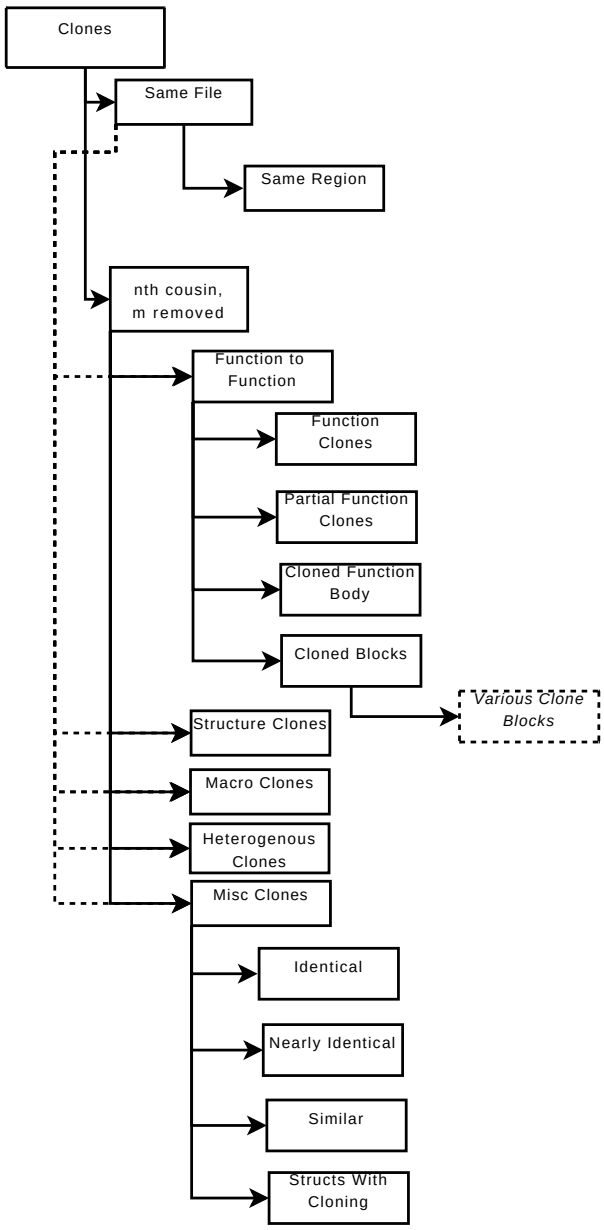

Figure A.1: Taxonomy of clones

a set of subsystems hierarchically contained. This partitioning of clones is done according to the location of the code segments in terms of files and subsystems. For example, one code segment of a clone could appear in Directory X, File Y, and the other code segment in Directory A, File B. In this case, the closest common ancestor is computed, determining the family tree relationship of the files. If A == X, then the clones are sibling files, or 0th cousin clones, also referred to as *Same Directory Clones*. If one file is located deeper in system hierarchy then the other, we show this difference by computing the difference of the depth of the files from the root of the architecture. The difference in the depth of two files is referred to as the number of generations removed. For example, consider a clone with segments of code in the following two locations:

1. *httpd/server/core.c* and
2. *httpd/modules/experimental/mod_example.c*.

In this case, the clone would be considered a second cousin clone, once removed. It is a second cousin because the closest ancestor (httpd) is two levels away from *core.c*. The clone is once removed because the depth of *core.c* is one level less than *mod_example.c*.

Groups

Clones that occur in the same region of the same file are called *Same Region Clones*. Clones that have both code segments within the same file are called *Same File Clones*, while clones that span different files but in the same directory are called in *Same Directory Clones*. Clones where the code segments are in different directories are considered *nth cousins, m removed* as described above.

Observations

This partitioning is significant as we have found that the dominant traits of each of these groups are different. For example, clones in the group *Same Directory Clones* tend to be function clones, clones occurring in different subsystems tend not to be function clones, and the *Function Clones* that are in different subsystems appear to have larger differences than those function clones that occur in the same directory or same file. In several of our

case studies we have found that a large percentage of clone pairs (70 - 80%) are either *Same File Clones* or *Same Directory Clones*, indicating that either clones occurring across subsystems are less frequent or they are harder to detect due to the larger changes.

A.2.2 Partition by Region

Description

The next level of the taxonomy partitions each of the above groups, with exception of *Same Region Clones*, by the types of regions the code clones occur in. For example, a list of declarations and a function are two different types of regions. A code segment is said to belong to the region type that makes up the majority of the lines of code in the clone. For example, if we have a code segment containing 50 tokens of function prototypes and 7 tokens in a struct, the segment is considered to be a prototype.

Groups

There are five groups: clones between two functions are called *Function to Function Clones*; clones between two programming structure regions such as unions, enumerators and structs are called *Structure Clones*; clones between macros are called *Macro Clones*; clones between two regions of different types are classified as *Heterogeneous Clones*; clones between external variable definitions, prototypes, and type defines are considered *Misc. Clones*. A clone with one segment in a function and the other segment as part of a macro is classified as a *Heterogeneous Clone*.

Observations

Detected clones that are not *Function to Function Clones* are often false positives in parameterized string matching algorithms. Because of this phenomenon, strict filters such as exact match line matching must be used to remove these false positives. In the case of parameterized string matching methods, we were able to remove up to 65% of the reported clone pairs using stricter filters, without removing many, if any, true clones.

After filtering, *Function to Function Clones* make up the bulk of the detected clones. Less than 1% of the clone pairs reported were not *Function to Function Clones*.

A.2.3 Function to Function Clones

Description

Function to Function Clones are those clones that occur between two functions. At this level of the taxonomy we partition the RGCs, as described above, by the degree to which the code is considered cloned between two functions.

Groups

Function to Function Clones are divided into four subtypes: *Function Clones*, *Partial Function Clones*, *Cloned Function Body* and *Cloned Blocks*. *Function Clones* are functions that share a minimum of 60% of each function's tokens. *Partial Function Clones* are near miss function clones. These function pairs must have one function with at least 40% and less than 60% of its code shared with the other, and the other function must have a minimum of 60% shared code. *Cloned Function Body Clones* are functions where one smaller function has been copied into a considerably larger one. This class of clones requires one function to share a minimum of 60% of its code with a function where less than 40% of the function code is cloned. *Cloned blocks* are blocks of code that are not large enough or numerous enough to be in one of the above clone classifications. At this point, clones are individually classified within their RGC. *Cloned blocks* is further subdivided as shown below.

Observations

Function Clones appear to be the most promising points of refactoring of all the clone types. They are larger and tend to have a significant amount of code in common. *Function Clones* and *Partial Function Clones* can account for up to 80% of the clone pairs found by CCFinder in a software system [49, 51, 50]. *Partial Function Clones* tend to be functions where one function has been cloned and extended slightly.

A.2.4 Cloned Blocks

Description

Cloned Blocks can exist in various parts in a function and can have many different roles. This group has been subdivided based on the functional properties of the code in the clones. In this group we try to capture the function or structure of the code that is cloned.

Groups

Cloned Blocks are subdivided into 14 groups of clones. Here we describe their rational and the constraints we use to classify them.

Clones that occur at the beginning of functions are called *Initialization Clones*. *Initialization Clones* start in the first 5 lines of a function and finish before the middle of the function.

Clones occurring at the end of functions are called *Finalization Clones*. *Finalization Clones* start after the middle of the function and end within the last 5 lines. For this group we have found that restricting only one clone to fit this criteria is still accurate but allows for functions where labelled sections of code have been appended at the end.

There are several clone types associated with loops in our taxonomy. *Loop Clones* are code clones that cover at least 60% of the length of the loop and the loop must cover at least 60% of the clone. This clone type allows multiple loops to be part of the clone, as long as 60% of the clone is covered by loops and 60% of the total loops are covered. *Starting Loop Clones* are typically clones of initialization at the beginning of a loop. These clones start before the first three lines of a loop, and cover less than half of the loop. *Partial Loop Clones* are loops where one loop is cloned and then extended. The criteria for a clone in this categorization is that at least one of the loops must contain at least 60% cloned code.

Clones occurring in *switch* statements are called *Clones in Switch*. The only criteria for this type of clones is that 60% of the tokens of both segments of the clone pair must be within a switch statement.

Conditional Clones are code clones that occur within *if/else* statements. These clones must cover at least 60% of the tokens of the block forming the *if/else* and the block must cover at least 60% of the code clone. In this clone type, *if* and *else* blocks are considered

either as separate entities or as a single unit, which ever decision favours placing the clone in this group. *Multi-Conditional Clones* are similar to the previous clone type but allows multiple control-flow statements to be involved. This is useful when regions of many single *if* statements are strung together and cloned. *Partial Match Conditionals* are similar to *Cloned Function Body Clones*. They are clones where the body of an *if* statement has been copied and used in a section of code that is not a conditional or when one smaller *if* statement is copied into a larger one. Clones of this type must have one clone covering 60% of a conditional and the conditional must cover 60% of the code clone. The thresholds reported here were tuned based on several study subjects.

The next group of clones we discuss involve calling functions: *Simple Call Clones* and *Less Simple Call Clones*. These clones are sequences of function calls, often cloned as a template. *Simple Call Clones* are clones where at least 70% of the tokens are part of very simple function calls. A simple function call is defined as:

```
functionCall := [assignment] function_name (parameter [, parameter]*);
function_name := token && !"if"
parameter := token
assignment := token =
token := alphanumeric[alpanumeric]*
alphanumeric := a-z|A-Z|0-9|_
```

This group of clones is particularly susceptible to false positives so filtering must be done. Currently we have constraints that require the function calls to be similar in name.

Less Simple Call Clones is a group of clones where the parameters of a function call can be statements in addition to tokens. Similar to *Simple Call Clones*, *Less Simple Call Clones* require that 70% of the tokens in a clone be part of a function call. In this case the function call is defined as follows:

```
functionCall := [assignment] function_name (params);
function_name := token && ! "if"
params := NOT ";"
assignment := token =
token := alphanumeric[alphanumeric]*
```

```
alphanumeric := a-z|A-Z|0-9|_
```

The final *Cloned Blocks* clone we will discuss is *Clone Islands*. While the other clones in this group have attributes related to the source code and their location in it, these clones do not. They are clones that are the only cloned between two functions. Clones of this type are often non-trivial fragments that occur as a result of a generic task, such as calling a function and checking its return value. For clones which cannot be easily placed we put them in a group called *Unclassified*.

Observations

Initialization Clones and *Finalization Clones* are an interesting set of code clones. Many of these clones embody the logic for ensuring preconditions and post-conditions for entering and exiting a function are met. Such clones would be difficult to remove, as they often appear to be modified, but are important to be aware of as they often handle issues of memory management and exit conditions.

Loop Clones appear much less often than first anticipated. We in fact found very few such clones in our case studies.

Clones in Switch is an interesting group of clones. They appear to difficult to refactor as switches with clones very often have insertions and deletions of cases and refactoring would make the code quite unreadable. We feel this group of clones is a good candidate for documentation rather than refactoring.

A.2.5 Structure Clones, Macro Clones, Misc. Clones, and Heterogeneous Clones

Description

These groups of clones are partitioned based on their degree of similarity.

Groups

These groups of clones are sub-grouped as *Identical*, *Nearly Identical* , *Similar* and *Structs with Cloning*. *Identical Clones* are structural entities that are 100% duplicates of each other.

Nearly Identical are 80% duplicates, and *Similar* are at least 60% duplicated. *Structs with Cloning* share less than 60% of their code.

Observation

Code segments that fall into these types of regions in source code are often of simple structure and are very prone to false positives, especially in the case of parameterized string matching, For example, the following two structs would produce a clone match:

```
struct {                        struct {
  int a;                          char ch1;
  int b;                          char ch;
  float f                         int  clock;
  char c;                         float elapsed;
  char* charstar;                 float* intervals;
} struct1;                      } struct2;
```

Because of this, filters should be applied to clones of these types.

A.3 Extensions

There are several points of extension that can be made to this framework which will be addressed in future work. *Function Clones* can be further divided to reflect the closeness of the match and the types of differences in the code clones. Balazinska has described such a categorization that could be used to further extend our description of function clones [8, 74]. This is an important extension that should be addressed as it will aid users in differentiating between functions that should be refactored to remove redundancy and functions that are dissimilar enough to require documentation. A similar extension should be made for *Partial Function Matches* and *Cloned Function Body* clones.

Bibliography

[1] Postgresql: Contributor profiles. "http://www.postgresql.org/community/contributors/", 2008.

[2] Raihan Al-Ekram, Cory Kapser, Richard C. Holt, and Michael W. Godfrey. Cloning by accident: an empirical study of source code cloning across software systems. In *IEEE International Symposium on Empirical Software Engineering (ISESE), 17–18 November, 2005, Noosa Heads, Australia*, pages 376–385. IEEE Computer Society, 2005.

[3] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the linux kernel. *Journal of Information & Software Technology*, 44(13):755–765, 2002.

[4] Giuliano Antoniol, Umberto Villano, Massimiliano Di Penta, Gerardo Casazza, and Ettore Merlo. Identifying clones in the linux kernel. In *First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM-01), 10 November, 2001, Florence, Italy*, pages 92–100. IEEE Computer Society Press, 2001.

[5] Lerina Aversano, Luigi Cerulo, and Massimiliano Di Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR-07) 21–23 March, 2007, Amsterdam, Netherlands*, pages 81–90. IEEE Computer Society, 2007.

[6] Brenda S. Baker. A program for identifying duplicated code. In *Computing Science and Statistics*, volume 24, pages 49–57, 1992.

[7] Brenda S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of the Second Working Conference on Reverse Engineering (WCRE-95), 14–16 July, 1995, Toronto, Canada*, pages 86–95. IEEE Computer Society, 1995.

[8] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the Sixth International Software Metrics Symposium (METRICS-99) 4–6 November, 1999, Boca Raton, Florida, USA*, pages 292–303. IEEE Computer Society, 1999.

[9] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Partial redesign of java software systems based on clone analysis. In *Proceedings of the Sixth Working Conference on Reverse Engineering (WCRE-99), 6–8 October, 1999, Atlanta, Georgia, USA*, pages 326–336. IEEE Computer Society, 1999.

[10] Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Lague, and Kostas Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE-00), 23–25 November, 2000, Brisbane, Australia*, pages 98–107. IEEE Computer Society, 2000.

[11] Mihai Balint, Radu Marinescu, and Tudor Girba. How developers copy. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), 14–16 June, 2006, Athens, Greece*, pages 56–68. IEEE Computer Society, 2006.

[12] Hamid Abdul Basit and Stan Jarzabek. Detecting higher-level similarity patterns in programs. 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), 5–9, September, 2005, Lisbon, Portugal*, pages 156–165. ACM Press, 2005.

[13] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: a study of clones in the STL and some general implications. In *Proceedings of the*

27th IEEE International Conference on Software Engineering (ICSE-05), 15–21, May, 2005, St. Louis, MO, USA, pages 451–459. ACM Press, 2005.

[14] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. An empirical study on limits of clone unification using generics. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE-05), July 14–16, 2005, Taipei, Taiwan, Republic of China*, pages 109–114. Knowledge Systems Institute, 2005.

[15] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th IEEEInternational Conference on Software Maintenance (ICSM-98), 16–20 November, 1998, Bethesda, MD, USA*, pages 368–377. IEEE Computer Society, 1998.

[16] Stefan Bellon. Detection of software clones — tool comparison experiment. `http://www.bauhaus-stuttgart.de/clones`, 2009.

[17] William J. Brown, Raphael C. Malveau, Hays W. McCormick (III), and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, Hoboken, NJ, 1st edition, 1998.

[18] Ross Buhrdorf, Dale Churchett, and Charles W. Krueger. Salion's experience with a reactive software product line approach. In *5th International Workshop Software Product-Family Engineering (PFE-2003), Revised Papers, November 4–6, 2003, Siena, Italy*, pages 317–322. Springer, 2003.

[19] Elizabeth Burd and John Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'02), 1 October, 2002, Montreal, Canada*, pages 36–43. IEEE Computer Society, 2002.

[20] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[21] James O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison Wesley Professional, 1st edition, 1992.

[22] James R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension (IWPC 2003), May 10–11, 2003, Portland, Oregon, USA*, pages 196–206. IEEE Computer Society, 2003.

[23] James R. Cordy, Thomas R. Dean, and Nikita Synytskyy. Practical language-independent detection of near-miss clones. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research (CASCON-04), 5–7 October, 2004, Markham, Ontario, Canada*, pages 1–12. IBM Press, 2004.

[24] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07), 20–26 May, 2007, Minneapolis, MN, USA*, pages 158–167. IEEE Computer Society, 2007.

[25] Stephane Ducasse, Oscar Nierstrasz, and Matthias Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution*, 18(1):37–58, 2006.

[26] Stephane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the 15th IEEE International Conference on Software Maintenance (ICSM-99), 30 August - 3 September, 1999, Oxford, England, UK*, pages 109–118. IEEE Computer Society, 1999.

[27] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition, 1999.

[28] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1995.

[29] Reto Geiger, Beat Fluri, Harald Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Proceedings of the Ninth International Conference Fundamental*

Approaches to Software Engineering (FASE-06), 27–28 March, 2006, Vienna, Austria, volume 3922 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2006.

[30] Barney G. Glaser. *Emergence vs. Forcing: Basics of Grounded Theory Analysis*. Sociology Press, P.O. Box 400 (31 Oxford Ave.), Mill Valley, CA, 1992.

[31] Michael Godfrey, Xinyi Dong, Cory Kapser, and Lijie Zou. Four interesting ways in which history can teach us about software. In *2004 International Workshop on Mining Software Repositories (MSR-04)*, pages 58–62, 2004.

[32] Michael W. Godfrey, Davor Svetinovic, and Qiang Tu. Evolution, growth, and cloning in Linux: A case study. A presentation at the 2000 CASCON workshop on 'Detecting duplicated and near duplicated structures in largs software systems: Methods and applications', on November 16, 2000, chaired by Ettore Merlo; available at `http://plg.uwaterloo.ca/~migod/ papers /cascon00-linuxcloning.pdf`, 2000.

[33] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of the 2000 International Conference on Software Maintenance (ICSM-00), 11–14 October, 2000, San Jose, California, USA*, pages 131–142. IEEE Computer Society, 2000.

[34] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[35] Sam Grier. A tool that detects plagiarism in pascal programs. In *Proceedings of the 12th SIGCSE Technical Symposium on Computer Science Education, 26–27 February, 1981, St. Louis, Missouri, United States*, pages 15–20. ACM Press, 1981.

[36] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, New York, NY, USA, 1997.

[37] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, NY, USA, 1977.

[38] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Aries: Refactoring support environment based on code clone analysis. In *Proceedings of the Eighth*

IASTED International Conference on Software Engineering and Applications (SEA-04), 9–11 November, 2004, Cambridge, USA, pages 222–229. ACTA Press, 2004.

[39] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th IEEE International Conference on Software Engineering (ICSE-92), 11–15 May, 1992, Melbourne, Australia*, pages 392–411. ACM Press, 1992.

[40] H. T. Jankowitz. Detecting plagiarism in student pascal programs. *The Computer Journal*, 31(1):1–8, 1988.

[41] Stan Jarzabek and Li Shubiao. Eliminating redundancies with a "composition with adaptation" meta-programming technique. In *Proceedings of the Ninth European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-11), 1–5 September, 2003, Helsinki, Finland*, pages 237–246. ACM Press, 2003.

[42] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th IEEE International Conference on Software Engineering (ICSE-07), 20–26 May, 2007, Minneapolis, MN, USA*, pages 96–105. IEEE Computer Society, 2007.

[43] J. Howard Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON-93), 24–28 October, 1993, Toronto, Ontario, Canada*, pages 171–183. IBM Press, 1993.

[44] J. Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the 1994 IEEE International Conference on Software Maintenance (ICSM 1994), September, 1994, Victoria, BC, Canada*, pages 120–126. IEEE Computer Society, 1994.

[45] J. Howard Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON-94)*, page 32. IBM Press, 1994.

[46] 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)*, page 16. IBM Press, 1996.

[47] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *Transactions on Software Engineering*, 8(7):654–670, 2002.

[48] Cory Kapser, Paul Anderson, Michael Godfrey, Rainer Koschke, Matthias Rieger, Filip van Rysselberghe, and Peter Weißgerber. Subjectivity in clone judgment: Can we ever agree? In *Duplication, Redundancy, and Similarity in Software*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[49] Cory Kapser and Michael W. Godfrey. A taxonomy of clones in source code: The reengineers most wanted list. In *2nd International Workshop on Detection of Software Clones (IWDSC-03), 13 November, 2003, Victoria, British Columbia, Canada*, 2003.

[50] Cory Kapser and Michael W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Architectures (ELISA-03), 23 September, 2003, Amsterdam, The Netherlands*, pages 67–78, 2003.

[51] Cory Kapser and Michael W. Godfrey. Aiding comprehension of cloning through categorization. In *Proceedings of the 7th IEEE International Workshop on Principles of Software Evolution (IWPSE-04), 6-7 Sept. 2004, Kyoto, Japan*, pages 85–94. IEEE Computer Society, 2004.

[52] Cory Kapser and Michael W. Godfrey. Improved tool support for the investigation of duplication in software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), 25–30 September, 2005, Budapest, Hungary*, pages 305–314. IEEE Computer Society, 2005.

[53] Cory Kapser and Michael W. Godfrey. 'Cloning considered harmful' considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE*

2006), 23–27 October, 2006, Benevento, Italy, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[54] Cory Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, 2006.

[55] Cory Kapser and Michael W. Godfrey. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 13(6):645–692, 2008.

[56] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming (ECOOP-97), 9–13 June, 1997, Jyvaskyla, Finland*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[57] 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 IEEE International Symposium on Empirical Software Engineering (ISESE-04), 19–20 August, 2004, Redondo Beach, CA, USA*, pages 83–92. IEEE Computer Society, 2004.

[58] 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), 5–9, September, 2005, Lisbon, Portugal*, pages 187–196. ACM Press, 2005.

[59] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the Eighth International Symposium on Static Analysis (SAS-01), 16–18 July, 2001, Paris, France*, pages 40–56. Springer-Verlag, 2001.

[60] Kostas Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proceedings of the 4th Working Conference on Reverse*

Engineering (WCRE-97), 6–8, October, 1997, Amsterdam, The Netherlands, pages 44–55. IEEE Computer Society Press, 1997.

[61] Kostas Kontogiannis, Renato de Mori, Ettore Merlo, M. Galler, and Morris Bernstein. Pattern matching for clone and concept detection. 3:77–108, 1996.

[62] Rainer Koschke, Raimar Falke, and Pierre Frenzel. Clone detection using abstract syntax suffix trees. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), 23–27, October 2006, Benevento, Italy*, pages 253–262. IEEE Computer Society, 2006.

[63] Rainer Koschke, Andrew Walenstein, and Ettore Merlo. 06301 abstracts collection – duplication, redundancy, and similarity in software. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[64] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE-01), 2–5 October, 2001, Suttgart, Germany*, pages 301–309. IEEE Computer Society, 2001.

[65] Jens Krinke. A study of consistent and inconsistent changes to code clones. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE-07), 28–31 October, 2007, Vancouver, BC, Canada*, pages 170–178. IEEE Computer Society, 2007.

[66] Greg Kroah-Hartman, Jonathan Corbet, and Amanda McPherson. Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it. `"http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php"`, 2008.

[67] Charles W. Krueger. Variation management for software production lines. In *Proceedings of the Second International Conference on Software Product Lines (SPLC-2),*

19–22 August, 2002, San Diego, California, pages 37–48, London, UK, 2002. Springer-Verlag.

[68] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th IEEE International Conference on Software Engineering (ICSE-06), 20–28 May, 2006, Shanghai, China*, pages 492–501. ACM, 2006.

[69] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[70] 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 (OSDI-04), 6–8 December, 2004, San Francisco, California, USA*, pages 20–20. USENIX Association, 2004.

[71] Simone Livieri, Yoshiki Higo, Makoto Matushita, and Katsuro Inoue. Very-large scale code clone analysis and visualization of open source programs using distributed CCFinder: D-CCFinder. In *Proceedings of the IEEE 29th International Conference on Software Engineering (ICSE-07), 20–26 May, 2007 Minneapolis, MN, USA*, pages 106–115. IEEE Computer Society, 2007.

[72] Angela Lozano, Michel Wermelinger, and Bashar Nuseibeh. Evaluating the harmfulness of cloning: A change based experiment. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR-07), 19–20 May, 2007, Minneapolis, MN, USA*, page 18. IEEE Computer Society, 2007.

[73] Nazim H Madhavji. Compare: a collusion detector for pascal. *Technique et Science Informatiques*, 4(6):489–497, 1985.

[74] 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 1996 International Conference on Software Maintenance (ICSM-96), 4–8 November, 1996, Monterey, CA, USA*, pages 244–253. IEEE Computer Society Press, 1996.

[75] Steve Mcconnell. *Code Complete*. Microsoft Press, 2nd edition, 2004.

[76] Audris Mockus, Roy T. Fielding, and James Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE-00), 4–11 June, 2000, Limerick Ireland*, pages 263–272. IEEE Computer Society, 2000.

[77] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE-01), 2–5, October 2001, Suttgart, Germany*, pages 13–22. IEEE Computer Society Press, 2001.

[78] Damith C. Rajapakse and Stan Jarzabek. An investigation of cloning in web applications. In *Special interest tracks and posters of the 14th international conference on World Wide Web (WWW-05), 10–14 May, 2005, Chiba, Japan*, pages 924–925. ACM Press, 2005.

[79] Damith C. Rajapakse and Stan Jarzabek. Using server pages to unify clones in web applications: A trade-off analysis. In *Proceedings of the 29th International Conference on Software Engineering, (ICSE-07), 20–26 May, 2007, Minneapolis, MN, USA*, pages 116–126. IEEE Computer Society, 2007.

[80] Matthias Rieger, Stphane Ducasse, and Michele Lanza. Insights into system-wide code duplication. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE-04), November 8–12, 2004, Delft, The Netherlands*, pages 100–109. IEEE Computer Society, 2004.

[81] Filip Van Rysselbergh and Serge Demeye. Evaluating clone detection techniques. In *Proceedings of the International Workshop on Evolution of Large Scale Industrial Software Architectures (ELISA-03), 23 September, 2003, Amsterdam, The Netherlands*, pages 25–36, 2003.

[82] Filip Van Rysselberghe and Serge Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE-03), 1–2 September, 2003, Helsinki, Finland*, page 126. IEEE Computer Society, 2003.

[83] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD-03), 9–12 June, 2003, San Diego, California*, pages 76–85. ACM Press, 2003.

[84] Robert Sedgewick. *Algorithms, 2nd Edition.* Addison-Wesley, 1988.

[85] TIOBE Software. Tiobe programming community index. `"http://www.tiobe.com/tpci.htm"`, 2007.

[86] Nikita Synytskyy, Richard C. Holt, and Ian Davis. Browsing software architectures with lsedit. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC-05), 15–16 May, 2005, St. Louis, MO, USA*, pages 176–178. IEEE Computer Society, 2005.

[87] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC-04), 29 September, 2004, Rome, Italy*, pages 173–180. IEEE Computer Society, 2004.

[88] Qiang Tu and Michael Godfrey. An integrated approach for studying software architectural evolution. In *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC-02), 27–29 June, 2002, Paris, France*, pages 127–136. IEEE Computer Society, 2002.

[89] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Gemini: Maintenance support environment based on code clone analysis. In *Proceedings of the Eighth IEEE Symposium on Software Metrics (METRICS-02), 4-7 June, 2002, Ottawa, Canada*, pages 67–76. IEEE Computer Society Press, 2002.

[90] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[91] Andrew Walenstein. Code clones: Reconsidering terminology. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[92] Andrew Walenstein, Mohammad El-Ramly, James R. Cordy, William S. Evans, Kiarash Mahdavi, Markus Pizka, Ganesan Ramalingam, and Jürgen Wolff von Gudenberg. Similarity in programs. In *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[93] Andrew Walenstein, Nitin Jyoti, Junwei Li, Yun Yang, and Arun Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE-03), 13–16 November, 2003, Victoria, Canada*, pages 285–294. IEEE Computer Society Press, 2003.

[94] Richard Wettel and Radu Marinescu. Archeology of code duplication: Recovering duplication chains from small duplication fragments. In *Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC-05), 25–29 September, 2005, Timisoara, Romania*, page 63. IEEE Computer Society, 2005.

[95] Robert K. Yin. *Case Study Research : Design and Methods (Applied Social Research Methods, Third Edition)*. SAGE Publications, December 2003.