Empir Software Eng
DOI 10.1007/s10664-008-9076-6

"Cloning considered harmful" considered harmful: patterns of cloning in software

Cory J. Kapser · Michael W. Godfrey

Editors: Massimiliano Di Penta and Susan Sim

Abstract Literature on the topic of code cloning often asserts that duplicating code within a software system is a bad practice, that it causes harm to the system's design and should be avoided. However, in our studies, we have found significant evidence that cloning is often used in a variety of ways as a principled engineering tool. For example, one way to evaluate possible new features for a system is to clone the affected subsystems and introduce the new features there, in a kind of sandbox testbed. As features mature and become stable within the experimental subsystems, they can be migrated incrementally into the stable code base; in this way, the risk of introducing instabilities in the stable version is minimized. This paper describes several patterns of cloning that we have observed in our case studies and discusses the advantages and disadvantages associated with using them. We also examine through a case study the frequencies of these clones in two medium-sized open source software systems, the Apache web server and the Gnumeric spreadsheet application. In this study, we found that as many as 71% of the clones could be considered to have a positive impact on the maintainability of the software system.

Keywords Clone detection · Clone analysis · Reverse engineering · Case study

1 Introduction

In much of the literature on the topic (Baker 1995; Baxter et al. 1998; Ducasse et al. 1999; Johnson 1994; Kamiya et al. 2002; Kontogiannis et al. 1996; Mayrand et al. 1996), cloning is considered harmful to the quality of the source code. For example,

C. J. Kapser (✉) · M. W. Godfrey
Software Architecture Group (SWAG) David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada
e-mail: cjkapser@uwaterloo.ca

M. W. Godfrey
e-mail: migod@uwaterloo.ca

code clones can cause additional maintenance effort. Changes to one segment of code may need to be propagated to several others, incurring unnecessary maintenance costs (Geiger et al. 2006). Locating and maintaining these clones pose additional problems if they do not evolve synchronously. With this in mind, methods for automatic refactoring have been suggested (Balazinska et al. 1999b; Baxter et al. 1998), and tools specifically to aid developers in the manual refactoring of clones have also been developed (Higo et al. 2004).

There is no doubt that code cloning is sometimes an indication of sloppy design and in such cases should be considered to be a kind of development "bad smell" (Fowler et al. 1999). 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. Thus, a variety of concerns such as stability, code ownership, and design clarity need to be considered before any refactoring is attempted; a manager should try to understand the reason behind the duplication before deciding what action (if any) to take.

This paper introduces eleven cloning patterns that we have uncovered during case studies on large software systems, some of which we reported in (Kapser and Godfrey 2003, 2004, 2006b) and more recently reported in (Kapser and Godfrey 2006a). These patterns present both good and bad motivations for cloning, and we discuss both the advantages and disadvantages of these patterns of cloning in terms of development and maintenance. Our goal is not to categorize clones for purposes of refactoring but to document the types of cloning that occur in software to aid the general understanding of how cloning is used in practice. In some cases, we identify patterns of cloning that we believe are beneficial to the quality of the system. From our observations we have found that refactoring may not be the best solution in all patterns of cloning. Tools need to be developed to aid the synchronous maintenance of clones within a software system, such as Linked Editing (Toomim et al. 2004) and automatic source code generation.

To support our basic argument – that code cloning can be used as a effective and beneficial design practice – we have performed a case study on two widely used open source software systems, the Apache web server and the Gnumeric spreadsheet application. In this study, we report on the observed uses of code cloning by the developers, the apparent rationale behind the uses, and the relative frequency of "good" versus "bad" clones.

This paper introduces the notion of categorizing high level patterns of cloning in a similar fashion to the cataloging of design patterns (Gamma et al. 1995) or anti-patterns (Brown et al. 1998). There are several benefits that can be gained from this characterization of 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 crystallizes a vocabulary that researchers and practitioners can use to communicate about cloning.

As a second contribution, 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 will 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 Section 3, 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 rest of this paper is organized as follows: Section 2 provides a brief background concerning code cloning, Section 3 introduces a template to describe code cloning patterns and then discusses eleven patterns we found in software systems, Section 4 describes a case study investigating the frequency of these clone patterns in two software systems, Section 5 discusses the implications of code cloning patterns on maintenance and tool requirements, Section 6 describes work that has contributed to the understanding of code cloning, and in Section 7 we discuss our conclusions and future work.

2 Code Cloning

Code cloning is considered a serious problem in industrial software (Antoniol et al. 2002; Baker 1995; Baxter et al. 1998; Casazza et al. 2001; Ducasse et al. 1999; Johnson 1994; Kamiya et al. 2002; Kontogiannis et al. 1996; Mayrand et al. 1996). It is suspected that many large systems contain approximately 10%–15% duplicated code (Baker 1995; Ducasse et al. 1999; Kapser and Godfrey 2004, 2006b), and it has been documented to exist at rates of over 50% of the effective lines of code – lines of code that contain more than just white space and comments – in a particular COBOL system (Ducasse et al. 1999). This section describes many motivations that lead to the creation of code clones as well as the positive and negative effects code clones have on several of the quality attributes of source code in a software system.

2.1 Motivations to Clone Code

The literature on the topic has described many situations that can lead to the duplication of code within a software system (Baker 1995; Baxter et al. 1998; Johnson 1994; Kamiya et al. 2002; Kontogiannis et al. 1996; Mayrand et al. 1996). Many of these can be considered ill intentioned cloning. For example, developers may duplicate code because the short term cost of forming the proper abstractions may outweigh the cost of duplicating code. Developers may also duplicate code when they do not fully understand the problem, or the solution, but they are aware of code that can provide some or all of the required functionality. These examples attribute cloning to programmer laziness. Clones can also be introduced as a side effect of programmers' memories; programmers may repeat a common solution, unknowingly introducing clones into the software system (Baxter et al. 1998).

Duplicates can also be introduced with good intentions. In particular, there are four categories of motivations that can be readily identified: code understandability, code evolvability, technology limitations, and external business forces. Code clones created for code understandability are intended to improve readability, conceptual cohesion/coupling, and traceability. Duplicating code can, in some situations, keep software architectures clean and understandable. Duplicates can also be used to keep unreadable, complicated abstractions from entering the system.

Code evolvability refers to the difficulty of introducing changes to existing code in order 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 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 Section 3. 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 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 (Walenstein et al. 2003), or even source code generation. This kind of technique is common in COBOL development, for example, and can lead to *templating* clones, described in Section 3. In these cases, the use of 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 cloning. Cordy notes that financial institutions consider code quality the most important concern when maintaining software (Cordy 2003) because the cost of errors in software can dwarf software maintenance costs. Fixing or modifying an abstraction can introduce risks of breaking existing code and requires that any dependent code be extensively tested, a process that is both costly and time consuming (Cordy 2003). 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 (Cordy 2003). Another external business force is time-to-market and opportunity cost. In some cases the long term cost of maintaining source code may be grossly out weighed by the short term opportunity cost of 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 cloning found in web based applications (Rajapakse et al. 2007).

2.2 Effects of Cloning

Several software maintenance problems have been associated with the use of cloning. In the long term, clones can unintentionally diverge if not carefully managed (Lozano et al. 2007). Code cloning can also lead to an unnecessary increase in code size (Baker 1995; Johnson 1994). Additionally, unused, or "dead", code can remain in the system if clones are left unchecked, resulting in problems with code comprehensibility, readability, and maintainability over the life time of the software system (Johnson 1994). 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 cloning, too. 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 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 (Johnson 1994). Program comprehensibility can be negatively affected by the need to understand the differences between the duplicates.

However, despite these known problems, we have found that developers can and do use cloning as a design tool when they judge that the likely benefits outweigh the risks; that is, these developers believe that the use of cloning can improve the design of the code. For example, aggressive refactoring can sometimes create abstractions that are complex, overly subtle, and unintuitive; in this case, near duplicates may be easier to understand and modify than a solution that employs abstraction, as the study performed by Toomim et al. suggests (Toomim et al. 2004).

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. (2007) found that reducing duplication 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 suggest that avoiding cloning during initial development could contribute to a significant overhead. 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 behavior 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 improving maintainability) may also increase required effort to extend or change code (there by decreasing maintainability). Similar to many development decisions, developers must assess the overall cost of cloning and decide on an individual basis the overall expected gain versus cost. The patterns detailed in Section 3 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.

3 Patterns of Cloning

Prior to this work, we undertook several investigations to understand code cloning in large software systems (Kapser and Godfrey 2003, 2004, 2006b). We wished to answer the general question of how code clones are used and what types of code are cloned. The study subjects of these initial investigations were the Linux operating system kernel, the Postgresql relational database management system, and the Apache httpd web server (Apache was also used as a study subject in the validation described in this paper). While our initial investigations were not explicitly intended to examine patterns of cloning, during our analysis we uncovered several

recurring ways in which developers duplicated behavior. These patterns are defined by what is duplicated and why, and to some extent how the duplication is done. More specifically, the patterns we consider 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 and how it has been duplicated. In some cases, documentation explicitly states the reasons for code cloning. While this is a subjective assessment, both authors are experienced developers and have a strong understanding of software design and maintainability. When we discuss how the duplication is performed, we describe what the new artifacts will be rather than the tools that are used to perform the duplication. The information described in these patterns is drawn from the case studies we have performed.

To describe our patterns, we use the following template:

- **Name** Describes the pattern in a few words.
- **Motivation** Why developers might use this cloning pattern rather than an appropriate abstraction.
- **Advantages** Description of the benefits of this pattern of cloning compared to other methods of reusing behavior.
- **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 in real systems. In this paper, the examples are drawn from the GNU spreadsheet application Gnumeric version 1.2.12, the relational database management system Postgresql 8.0.1, the web server Apache httpd 2.0.49, and the Java mail client Columba version 1.2.

We have divided the eleven patterns into four related groups: *Forking*, *Templating*, *Customization* and *Exact match*. This partitioning is done based on the high level motivation for the cloning pattern. *Forking* is cloning used to bootstrap 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, by allowing for experimentation to occur away from the core 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 behavior of existing code when appropriate abstraction mechanisms, such as inheritance or generics, are unavailable. *Templating* is used when there is a common set of requirements shared by the clones, such as behavior 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* duplication is typically used to replicate simple solutions or repetitive concerns within the source code.

3.1 Forking

Forking patterns often involve larger portions of code with the intention that the resulting duplicates will need to evolve independently. The duplication can be used as a "springboard" from which to start development and works well in situations where the commonalities and difference 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 duplicates. This section describes three *forking* patterns that we have seen in our case studies.

3.1.1 Hardware Variation

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/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, in particular 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, and the duplicate is then modified to match the new device.

Examples The Linux SCSI driver subsystem has several examples of this pattern of cloning (Godfrey et al. 2000). 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 duplicated and modified in `NCR53C9x.c`. What is interesting in the Linux SCSI drivers is that the authors duplicating the new file explicitly reference the file they have duplicated, making the chain of replications easily verified.

3.1.2 Platform Variation

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 such as a virtualization layer, 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. In the case of source code within virtualization layers themselves, avoiding this complexity is often a reason to clone code. This pattern differs from *hardware variation* in that the drivers are often comprised of lower level source code, not uncommonly comprised of large portions of assembly. The differences in the type of source code in these artifacts raises different types of maintenance concerns.

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 in order to minimize the amount of cloning necessary. When creating the code clones, the variations should be well documented in order to facilitate bug fix propagation.

Long term issues As groups of platform specific code clones grow, the interface that they support will become more brittle and difficult to change because of the number of places where changes will need to be made. In order to guarantee consistent behavior on supported platforms it will be vital to ensure that visible behavior from each of the clones remains consistent.

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 duplicates 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 application program interface (API) calls. With these changes, overall structure remains the same, and in several cases cloned documentation exists providing further information about the cloning.

3.1.3 Experimental Variation

Motivation Developers may wish to optimize or extend pre-existing code but do not want to risk system stability. By forking the existing code, users can have the choice to run the experimental optimized code or the trusted stable code.

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 choose 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 the experimental version closely with the stable version. Changes to the external behavior of the existing stable module will need to be monitored and introduced in the duplicated experimental code in order to maintain a consistent interface.

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

Structural manifestations The cloning pattern will appear as a cloned file, subsystem or class. It may even be labeled as an experimental development effort, as in the case of several Apache modules (Kapser and Godfrey 2006b).

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` (Kapser and Godfrey 2006b). The cloned subsystems are experimental variations on `worker` that are designed to provide better performance. Because they are separated from `worker`, the web server remains stable while optimizations are being developed.

3.2 Templating

Templating occurs when the desired behavior is already known and an existing solution closely satisfies this need. Often *templating* is a matter of parameterization,

as opposed to the complex control flow that might be required for abstraction when *forking* patterns are used instead. For example, one might use this pattern of cloning to achieve the same behavior 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, especially in the case of *boiler-plating*. In the subsections that follow we describe four *templating* patterns.

3.2.1 Boiler-plating Due to Language Inexpressiveness

Motivation Due to language constraints, reusing trusted and tested code may be difficult to achieve. This can occur for example when polymorphism cannot be used. This form of cloning is common in software systems that are developed in the COBOL language.

Advantages *Technology limitations* that hinder reuse are overcome. This pattern can make reuse of trusted code possible. It allows for consistent behavior 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 will very likely require n times the effort for n clones.

Management Documentation or other forms of an explicit link to all duplicates 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 (Duala-Ekoko and Robillard 2007). Tools and methodologies such as Linked Editing (Toomim et al. 2004) should be used to ensure consistent changes are made to all duplicates. Another approach to managing these clones is to create the code at build time using a source code generator (Jarzabek and Shubiao 2003); in this case, the duplicates do not come into existence until the system is being built.

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

Structural manifestations Typically these duplicates 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 10 demonstrates an example of this pattern of cloning.

3.2.2 API/Library Protocols

Motivation Often the use of particular APIs require ordered series of procedure calls to achieve desired behaviors. 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 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 of communication and then parameterize them appropriately to be used 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 duplicating and modifying the code. The duplicated code can flexibly be changed, and often the size of the duplication may not warrant further abstractions.

Disadvantages *Evolvability* can be negatively affected as the severity of changes to the library or API is increased with every clone. We have also experienced several cases where developers have duplicated buggy or incomplete code with the assumption it is correct, degrading the quality of their own code.

Management Locate prevalent cloning of this type and extend the API or library in use with appropriate abstractions. For code clones of this type, rigourous review of the duplicates can ensure that the duplicated code is of high quality.

Long term issues Changes to the API will result in required changes in 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 These duplicates are typically scattered throughout the source code, and are small in size.

Examples In the mail client Columba, this pattern is readily found in the GUI code where buttons are added. This sequence of three operations that create a button, set its action listener, and set its action command is present throughout the system where GUI code is present. An example from the Gnumeric case study presented in Section 4 is shown in Fig. 1.

3.2.3 General Language or Algorithmic Idioms

Motivation Programming idioms are clear and concise implementations of particular solutions. These idioms tend to be self documenting for language experts as they provide information as to how and why the implementation is done in this way. Idioms are commonly discussed, books have been written on this specific topic (Coplien 1992), and there is no shortage of web discussions either. They can be conventional wisdom in the programming community, such as checking the return 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)

Fig. 1 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 duplicated, degrading the quality of the code.

Management Anti-idioms, 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 Fig. 9. At least 15 occurrences of this idiom can be found in the 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. This is an interesting variation as we would expect all copies to behave in this way.

3.2.4 Parameterized Code

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 stdout, stderr, or stdin. In this case, developers may implement a parameterized function that takes an argument as an indicator of what 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 unnecessarily. If the code is copied out of context of the new parameters, such as a different function call, it can violate assumptions and pre/post-conditions of the copied code.

Management The behavior 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 Fig. 2. This example comes from `plugins/fn-eng/functions.c` in Gnumeric.

3.3 Customization

Customization often arises when existing code solves a very similar problem to the current development problem, but additional or differing requirements create the need for extension or modification of the behavior. In some cases, such as concerns about system stability or code ownership, existing code cannot be modified "in place" to encompass the additional behavior. In these cases, code may be cloned

```
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)

Fig. 2 An example of *parameterized code* in Gnumeric

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 are inserted or removed from the clone. While other forms of cloning, such as *templating* and *forking* typically have the goal of maintaining the original behavior to a high degree, *customization* is a reuse of behavior often without requirements that force the behavior 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 behavior may be harder to understand, and the code clones may be harder to detect. In this section we describe two *customization* patterns.

3.3.1 Bug Workarounds

Motivation Due to code ownership issues or unacceptable exposure to risk, it may be difficult to fix a bug at the source, so workarounds may be necessary. Copying the code and fixing the bug in order 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 behavior of the buggy code may cause confusion in the maintenance process if this pattern of duplication is not made explicit.

Management Once the original bug is fixed, remove any duplicates. Planning for this will minimize issues for clone removal.

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 will be duplicated at points where the buggy source code is used.

Examples One of the authors (Godfrey) wrote a Java fact extractor that was built around the internals of Sun's javac compiler. On finding 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, we see an example of duplication 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 duplicated a three line solution three times in three different files: `backend/commands/tablespace.c`, `port/copydir.c`, and `backend/access/transam/xlog.c`.

3.3.2 Replicate and Specialize

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 (Cordy 2003).

Advantages *Maintainability* is improved when used to avoid complex abstractions that may have a high cognitive cost during development and maintenance (Toomim et al. 2004). *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. Long term costs of finding and maintaining these duplicates could outweigh the short term gains.

Management If an appropriate abstraction can be made, deprecating the original code and transitioning to the abstraction may defer testing costs and protect system stability. If the appropriate abstractions cannot be made, explicitly linking the code clones through documentation or tool support will ensure consistent maintenance.

Long term issues Duplicated code can, over time, become more entrenched, with more of the software system dependent upon it. Over time, the cost of refactoring the code may rise. In the case of this pattern, differences in the code may make locating duplicates difficult, making maintenance of clones more costly.

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 duplication to reuse complex logic, an observation also noted by Kim et al. (2004).

Examples This pattern is the most common type 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 Fig. 3. This example is taken from the httpd study described in this paper, 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, but would also unnecessarily create dependencies on a higher level library call that would only be used for these two subsystems, possibly cluttering the higher level system design.

3.4 Exact Matches

Exact matches often arises when a particular problem is repeated within in the software system but is either too small to make the creation of an abstraction worthwhile or is incomplete when taken out of the context of its neighboring source code. In this section we describe two *exact match* clone patterns.

3.4.1 Cross-cutting Concerns

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 (Kiczales et al. 1997). 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 features to abstract this code.

Advantages There is little advantage to cloning cross-cutting concerns: they are typically considered unavoidable. However, like idioms, cross-cutting concerns can clearly present 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.

```
    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)

Fig. 3 An example of *replicate and specialize* in Apache httpd

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 that 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 one place, separate from the rest of the code; when the system is later compiled, the language processing tools weave each aspect into appropriate places 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 duplicated, the more brittle the concern will become making improvements and design changes increasingly difficult.

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

Examples A very common example in Apache is the checking of the command context before executing security sensitive functionality. This clone appears exactly copied throughout the software system. An example of a cross cutting concern that is used throughout the *server* subsystem of Apache httpd is shown in Fig. 4. This

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

Fig. 4 An example of a *cross cutting concern* in Apache httpd

concern ensures the correct security requisites are met before continuing to execute a function.

3.4.2 Verbatim Snippets

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 When it becomes apparent that certain segments of code are copied often, they should be factored out as helper functions.

Long term issues Over time this duplicated code can build up making it difficult to remove. Design decisions that the duplicated code is dependent on will become more brittle and difficult to change.

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

Examples These clones are readily found in any software system. 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 Fig. 5, 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 duplicated.

4 Empirical Evaluation

The purpose of this study was to demonstrate the degree to which these patterns exist in software systems and to assess the relative harmfulness of clones in software systems. In order to measure the prevalence of the cloning patterns in source code

```
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)

Fig. 5 An example of *verbatim* in httpd

we performed a case study of cloning occurring in two open source software systems: *Apache httpd* and *Gnumeric*. The study performed was a multiple case, descriptive case study. Two propositions were formulated:

1. Not all duplication of code is harmful; cloning may be used in sound design decisions.
2. The cloning patterns we described in Section 3 appear with non-trivial frequencies industrial software systems.

Beginning with the patterns of cloning originally described in the previous report on this work (Kapser and Godfrey 2006a) and also described in Section 3 we categorized a random sample of clones from each of the study subjects. As a result of this study we discovered several new patterns of cloning, all of which have been described above. In particular, the patterns for *parameterized code* clones, *cross-cutting concerns*, and *verbatim snippets* were added during this study.

This section describes the study setup and the tools used to detect and visualize the cloned code. It briefly describes the two study subjects and provides several statistics summarizing the overall cloning found in the software systems. The results of the case studies are then reported and discussed.

4.1 Study Setup

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.

4.1.1 Detecting the Clones

The candidate clones were detected using the *CLone Interpretation and Navigation System (CLICS)* clone detection tool. This tool locates common sub-strings within the code using a parameterized string matching method based on suffix trees very similar to the clone detection tool *CCFinder* (Kamiya et al. 2002) or *clones* (Koschke et al. 2006). There are many other clone detection methods available, differing largely on the underlying data structure used to search for similarity. These representations include abstract syntax trees (Baxter et al. 1998; Jiang et al. 2007; Kontogiannis et al. 1996; Mayrand et al. 1996), program dependence graphs (Komondoor and Horwitz 2001; Krinke 2001), normalized lines of code (Ducasse et al. 1999; Johnson 1994), and parameterized token streams (Baker 1995; Kamiya et al. 2002). We chose to use parameterized substring matching based on parameterized token streams because it has been shown to have the highest recall (Bellon

2002; Koschke et al. 2006). We chose *recall* as the most important property because we wanted to ensure that we were able to sample a large variety of clones. This section summarizes the process of parameterized substring matching. For a complete description of the algorithm, please refer to (Kamiya et al. 2002).

There are three steps to this approach: source code parameterization, suffix tree generation, and candidate selection and filtering. In the first step, an abstract representation of the source code is generated. 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, 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 (Kapser and Godfrey 2004, 2005, 2006b) the authors found that parameterizing tokens outside of functions leads to a high occurrence of false positives that require 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.

Using the *p-strings* as input, a generalized suffix tree is constructed and maximal repeats are then extracted from the tree. Maximal repeats are repeated 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 clone. Suffix trees are an attractive data structure for this task because they can be built in $O(n)$ time where n is the length of the string and maximal pairs 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 (Gusfield 1997; Ukkonen 1995).

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);*. 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 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.

One disadvantage with this approach is that it produces a high number of false positives, even after enforcing a mapping. In particular, sections of code with relativity little structural complexity do not have enough distinguishing features to differentiate them. Examples of these types of code are initialization lists, sequences of simple assignments, and *switch* statements. In order to improve the accuracy of our

clones we filter the 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 (Kapser and Godfrey 2006b):

1. **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]*).
   ```

   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 always less then 50% of the total length of the function name. During our studies, we found that typical clones 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 dataset.
2. **Logical-structures filter**. We found that 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 are identical and in the same order. Clones in very 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 dataset.
3. **Overlap filter**. 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.

During this filtering step 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 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 clones is similar to that which is done by

Fig. 6 Two procedures with six "cloned" lines grouped as one RGC

```
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)

Koschke et al. (2006) and Kamiya et al. (2002) 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 them. For each pair of regions with a cloning relationship we group together all the clones that form this relationship; we call 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 Fig. 6 might be considered a code clone, constituting eight code clones forming a cloning relationship between the two procedures. These eight code clones (braces included) are grouped as a single RGC in our analysis. The concept of RGC is useful for both visualizing and filtering clones and is the granularity of the sample set we analyze in the case study.

4.1.2 Sample Selection

In this study, we chose to use RGCs as single elements 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 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 an aspect directly related to how clones are detected, but rather how clones

```
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)

Fig. 7 Two clones occurring in the same region

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 consider clones that occur in the same region as they are most often segments that would not be considered for refactoring or would be considered false positives. Figure 7 shows two segments of code taken from the same procedure in the Apache httpd source code. In this example, the code fragments are detected as 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 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.

4.1.3 Clone Presentation

The randomly sampled RGCs were presented in the CLICS graphical user interface. 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 described in Kapser and Godfrey (2005, 2006b). Combined, these features provide contextual information to aid the evaluation of cloning in a software system.

For the purposes of the study, the tool presents to the user a single randomly selected RGC at a time. Figure 8 shows an example of how a random RGC is presented to the user. In the upper left a tree indicates the automatic classification of

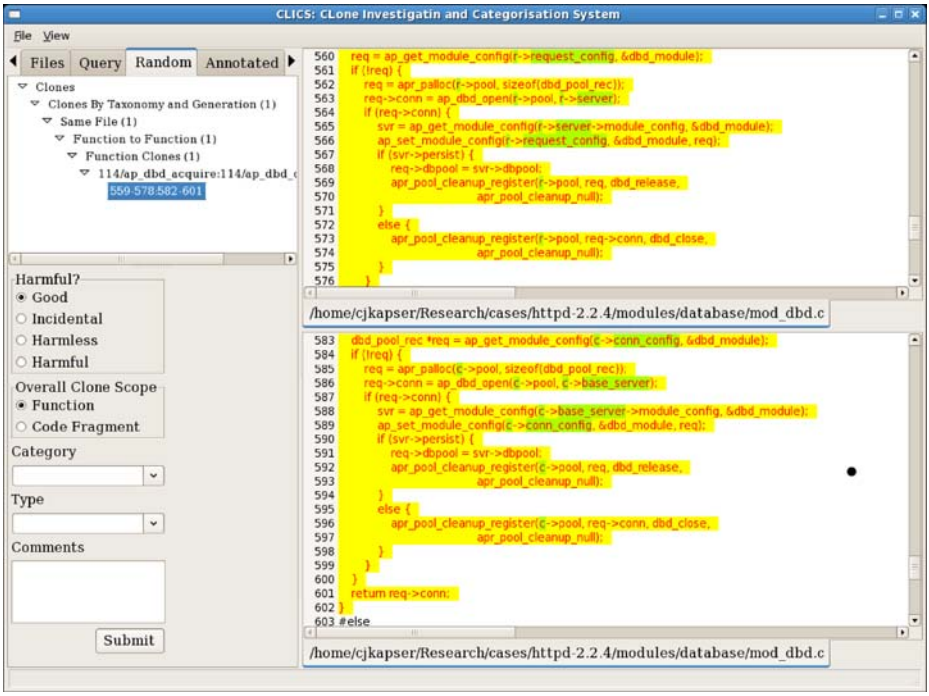

Fig. 8 Sample visualization of randomly selection clone

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 clone shown in the diagram 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 Fig. 8 we see the code encompassed within the two regions of the RGC indicated by highlighted text. Using CLICS we can show the differences of the two regions rather than just the detected clone. The common tokens and differing tokens are highlighted with different colors to make the changes in the duplication 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 our cloning patterns.

4.2 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 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 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 affects the software system is undoubtedly the most controversial aspect of this study, and also the most subjective. The range of the rating was good, incidental, harmless, and harmful. 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 considered good, we felt that certain quality attributes 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 prorogation*. Incidental clones are clones that cannot be abstracted further and therefore are neither good nor harmful. Generally this means the code is at the highest level of abstraction, such as referencing a single function or multiple calls to the same function, and the parameters to that function are changed in a non-systematic way. Harmless clones are those clones that are likely to have no impact on maintenance or development, usually small fragments of code. Harmful clones are those clones we believe will have a negative effect on the maintainability of software system. In our study, a clone was considered harmful when we judged that the overall negative effects on quality attributes of source code out-weighed the positive effects. In effect, this is an assessment of the perceived net gain/loss to source code quality.

In our rating of the harmfulness of the duplication we tried to take into account several considerations including likelihood of the clones requiring co-evolution and how difficult this would be to maintain, complexity of abstracted code, and effects of clones on semantics or understandability of the code. First, we asked how likely is it that changes in one code segment of the clone will 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 behavior. 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 flavor 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 on the 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 duplicate is more likely to

be harmful because changes to the internal concepts will need to be reflected in all of the cloned code.

Our second consideration was evaluating the complexity of forming an abstraction in order to refactor the code. There is evidence that abstractions can be harder to maintain than managing duplicated code (Toomim et al. 2004). 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 duplicated code, it is likely harder to maintain the clone than maintaining the straightforward abstraction.

Our third consideration was evaluating how refactoring the clone would affect the semantics or understandability of the code it occurred in. This is particularly important when evaluating cloned code fragments rather than cloned regions. 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 duplication was considered to be the whole region. In cases where the clone detector only found several fragments of code, the region diff-ing utility was used to visualise the overall similarity. If a large portion of the two regions in question is 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 clones into patterns was done manually, based on the descriptions we have documented in Section 3. 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 duplication. This required an understanding of programmer intent 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 behavior 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 behavior 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. This analysis results in an understanding of the programmers' intent 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 programmer intent 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 difficulty to form a more general abstraction, in order to infer motivation for forming the code clone. With this information in hand, we then compare the information we have compiled with the various patterns listed in Section 3. This process was very time consuming in the beginning of the sample analysis of each system, but as we analyzed more clones the process became easier as we could reuse much of the knowledge about the system we gained over time.

4.3 Study Subjects

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 (Mockus et al. 2000)). 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 (Mockus et al. 2000).

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 implemented in the libraries it depends on rather 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.

4.4 Sample Set

Two sets of clones were detected for each study subject. Several patterns of cloning described in Section 3 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. In order 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

Table 1 Detected clones in Apache and Gnumeric

	Min Clone Size							
	30 Tokens				60 Tokens			
	Num RGCs	Sampled	Clones	Sampled	Num RGCs	Sampled	Clones	Sampled
Apache	10,657	100	61,481	204	1580	100	21,270	2655
Gnumeric	23,129	230	84,028	807	3437	100	11,400	405

not be considered for refactoring, creating a bias in the results. In order 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 code clone, and therefore any clone larger than the minimum will also be detected.

A summary of the total number of clones returned by the clone detector is shown in Table 1. In the table, the columns "RGCs" indicates the number of RGCs we detected, and the column directly to its right indicates the number of RGCs we randomly selected to be categorized. 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 viewed a large number of code clones throughout the systems. Only a small percentage of the code clones available were categorized due to time constraints. 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 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 largely comprised of false positives but also contains many of the smaller clones that contribute to larger clones such as in the *Replicate and Specialize* clone pattern.

The total number of false positives found in the sample sets are shown in Table 2. This table shows the number of RGCs from the clone sample set that were considered to be false positives. As we predicted, the number of false positives was dramatically reduced after increasing the minimum length of a clone to 60 tokens.

4.5 Results

The results of the subjective categorizations are summarized in Tables 3, 4, 5, 6. Each row in the tables indicates a clone pattern and the frequency of good, incidental,

Table 2 False positives in sample sets

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

Table 3 Clones by type - Apache httpd 2.2.4–30 Tokens

Group	Pattern	Good		Incidental		Harmless		Harmful		Total
		#	%	#	%	#	%	#	%	
Forking	Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Forking	Platform variation	9	100	0	0	0	0	0	0	9
Forking	Experimental variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Templating	Boiler-plating	8	100	0	0	0	0	0	0	8
Templating	API	0	0	7	87.5	0	0	1	12.5	8
Templating	Idioms	0	n/a	0	n/a	0	n/a	0	n/a	0
Templating	Parameterized	1	16.7	0	0	0	0	5	83.3	6
Customize	Replicate and specialize	10	76.9	0	0	1	7.7	2	15.4	13
Customize	Bug workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact match	Cross-cutting	12	92.3	0	0	1	7.7	0	0	13
Exact match	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

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.

4.5.1 Cloning in Apache httpd

When comparing the result of the two clone sets sampled for Apache httpd, shown in Tables 3 and 4, the most striking observation is the very large difference in the number of harmful RGCs. In the Apache case study where the minimum clone length was 30 tokens, 71% of the 59 true positive RGCs were considered good and only 14% were considered harmful. In the sample set where the minimum clone length was 60

Table 4 Clones by type - Apache httpd 2.2.4–60 Tokens

Group	Pattern	Good		Incidental		Harmless		Harmful		Total
		#	%	#	%	#	%	#	%	
Forking	Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Forking	Platform variation	10	100	0	0	0	0	0	0	10
Forking	Experimental variation	4	100	0	0	0	0	0	0	4
Templating	Boiler-plating	5	100	0	0	0	0	0	0	5
Templating	API	0	0	17	100	0	0	0	0	17
Templating	Idioms	0	0	0	0	0	0	12	100	12
Templating	Parameterized	5	27.8	0	0	1	5.6	12	66.7	18
Customize	Replicate and specialize	12	75	0	0	0	0	4	25	16
Customize	Bug workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact match	Cross-cutting	2	100	0	0	0	0	0	0	2
Exact match	Verbatim snippets	1	12.5	0	0	0	0	8	88.9	9
Total		39	41.9	17	18.3	1	1.1	36	38.7	93

Table 5 Clones by type - Gnumeric 1.6.3–30 Tokens

Group	Pattern	Good		Incidental		Harmless		Harmful		Total
		#	%	#	%	#	%	#	%	
Forking	Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Forking	Platform variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Forking	Experimental variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Templating	Boiler-plating	3	100	0	0	0	0	0	0	3
Templating	API	0	0	5	83.3	0	0	1	16.7	6
Templating	Idioms	2	100	0	0	0	0	0	0	2
Templating	Parameterized	5	23.8	0	0	0	0	16	76.2	21
Customize	Replicate and specialize	9	56.3	0	0	0	0	7	43.8	16
Customize	Bug workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact match	Cross-cutting	2	50	0	0	0	0	2	50	4
Exact match	Verbatim snippets	2	11.1	0	0	2	11.1	14	77.8	18
Total		23	32.9	5	7.1	2	5	40	57.1	70

tokens, 42% of the clones were considered good and 39% were considered 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 biases the sample set toward simplistic clones that can be clearly refactored, and are therefore have no value as clones in the system. This observation is supported by the large increase in the number of RGCs classed as *templating* clones.

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*

Table 6 Clones by type - Gnumeric 1.6.3–60 Tokens

Group	Pattern	Good		Incidental		Harmless		Harmful		Total
		#	%	#	%	#	%	#	%	
Forking	Hardware variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Forking	Platform variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Forking	Experimental variation	0	n/a	0	n/a	0	n/a	0	n/a	0
Templating	Boiler-plating	6	85.7	0	0	0	0	1	14.3	7
Templating	API	0	0	8	88.9	0	0	1	11.1	9
Templating	Idioms	1	100	0	0	0	0	0	0	1
Templating	Parameterized	10	29.4	0	0	0	0	24	70.6	34
Customize	Replicate and specialize	15	93.8	0	0	0	0	1	6.25	16
Customize	Bug workarounds	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact match	Cross-cutting	0	n/a	0	n/a	0	n/a	0	n/a	0
Exact match	Verbatim snippets	1	25	0	0	0	0	3	75	4
Total		41	57.7	0	0	0	0	30	42.3	71

```
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)

Fig. 9 Two examples of idioms in Apache httpd

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. The authors noted 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 found only in the sample of larger clones of Apache httpd. This is a surprising result as we expected precisely the opposite. Two examples of the idioms found are shown in Fig. 9. 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 value of the allocation method is checked, and if memory could not 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, Fig. 9(b) should be implemented in 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. Our observation is 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 considered incidental were involved in testing. 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 consider 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. Only 25% of these clones were considered 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 considered harmful as they involved reasonably simple code that was 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 that abstraction would introduce 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 duplicates were considered 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 similar enough to be matched by the duplication detection but was not actually cloned code.

The *verbatim snippets* clones found in the small clone size set were code fragments involving control flow. It was deemed that these code fragments were helpful. Abstracting them would adversely affect the conceptual cohesion of the functions they appeared in. *Verbatim snippets* clones found in the large clone size set were mostly considered harmful. In these cases, the snippets consisted of complete conceptual units of code. For example, 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.

4.5.2 Cloning in Gnumeric

The results from the categorization of the clones in Gnumeric are summarized in Tables 5 and 6. The sample size for the data shown in Table 5 was 230 RGCs and the sample size of the data shown in Table 6 was 100 RGCs. In Table 5 we see that 57% of the RGCs sampled were considered harmful. This does not sharply contrast the results of the second sample set of the larger clones, where 42% of the clones were considered harmful. The Gnumeric source code has a large amount of mathematical computations. The interface between the spreadsheet and the mathematical computations 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 duplication of initialization of variables and then the identifier calling a particular function 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 very 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. While this does not rule out their existence in Gnumeric, it does demonstrate their relative rarity in the code. This is very likely due to the fact that there are very 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 Fig. 10. Abstracting these clones would not improve the source code in any way. The single exception rated harmful had an obvious abstraction that could be made without adding complexity.

```
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)

Fig. 10 An example of *boiler-plating* in Gnumeric

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 considered harmful were clone pairs that appeared frequently in the system and had an obvious abstraction using macros. Frequency can be a concern when considering 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 consider good because they were not at risk of being poorly implemented contrary to the case in Apache.

Parameterized code clones were considered 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. Fifteen of the clones of this type were considered good. In these cases, the identifiers were changed to reflect the standard notation of the variables in the mathematical functions they represented. This was considered 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 considered 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 abstracted.

When considering *replicate and specialize* clones in the sample of small clones, nearly 44% of the RGCs were considered 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 dialog. 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 considered them to be beneficial because of the semantic traceability they create to the mathematical equations they represent. A clone of this type is shown in Fig. 11.

There were four *cross-cutting concerns* seen in the sample set from the small clones. The two that were considered 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 considered 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 setting the

```
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)

Fig. 11 An example of *replicate and specialize* in Gnumeric

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 considered harmful, in this case 63%. Only clones that were small fragments were considered good as the code was small and incomplete on its own. The resulting abstraction would have break the understandability of the procedure they were found in.

4.6 Case Study Discussion

There were two goals for this case study. The first goal was to empirically evaluate a generally accepted wisdom: "Cloning considered harmful". We proposed that not all duplication is harmful. 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, 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. While a non-trivial number of clones were considered good it should be noted that, as described in the patterns listed in Section 3, these clones were not considered 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 considered while developing software systems.

Second, we proposed that the cloning patterns described in Section 3 appear in industrial source code. We wanted to measure whether the patterns we found during previous case studies appeared regularly in source code, and if they did, how often. The results clearly show that these patterns appear in 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 we have observed this form of cloning, it may not be as common as we first believed and may not be justified as a pattern. In order to investigate this further, one direction of our future work is to detect and analyze more clones of this type. In the case of *hardware variation* we did not expect to see this pattern as neither software system directly interacts with a hardware device. However, the pattern we describe in this paper is clearly visible in the Linux kernel. Section 3.1.1 lists two families of drivers that demonstrate this pattern. To properly evaluate this pattern, studies focussing on the analysis of driver code must be performed. Candidate study subjects might include Xorg, Linux, and FreeBSD.

Comparing Tables 3 and 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 set 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.

Tables 5 and 6 also show that specific types of clones were classified similarly, with the notable exception of *replicate and specialize* clones. The larger *replicate and specialize* clones found in Gnumeric the 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.

The total number of verified code clones sampled varies largely between the two samples in Apache (Tables 3 and 4) but does not vary between the samples in Gnumeric (Tables 5 and 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 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

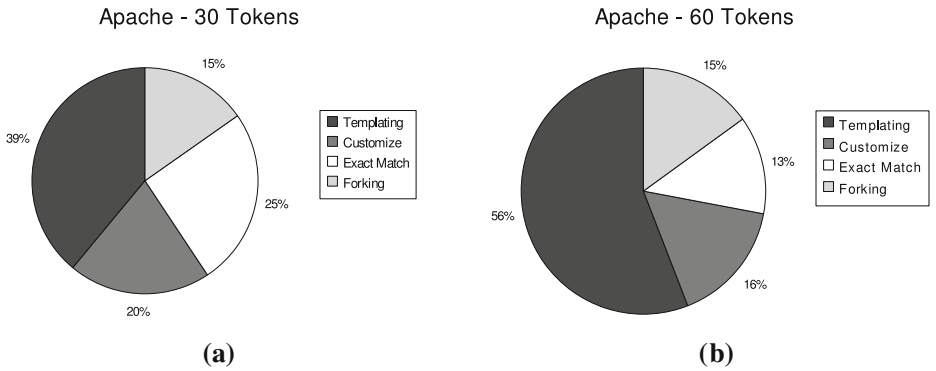

Fig. 12 Percentage of clone types in Apache. **a** Thirty tokens. **b** Sixty tokens

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*, could not have be made unless we had sampled the set of smaller clones. This observation and the results shown in Table 2 emphasize the need for more sophisticated filtering of clone detection results.

Cloning in Apache httpd and Gnumeric appears to be qualitatively different. Figures 12 and 13 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 is largely due to the interface between the spreadsheet and the mathematical functions. The differences in the types of

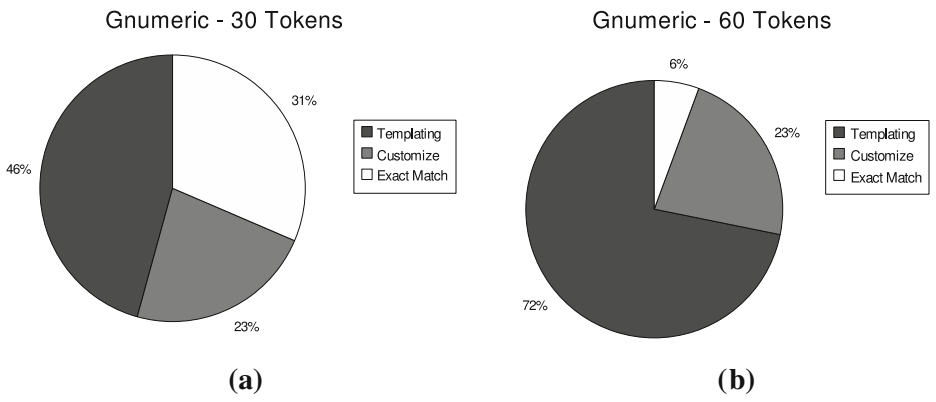

Fig. 13 Percentage of clone types in Gnumeric. **a** Thirty tokens. **b** Sixty tokens

cloning found in the two study subjects reflects 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, and the code using this interface, not surprisingly, will have much in common.

In both studies *templating* clearly represent 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 substring 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 form larger clones, and this grouping is less sensitive to changes in logic and structure.

As can be seen in Figs. 12 and 13, 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 their relative frequency, indicating that the size of the code plays a role in developers' decisions to duplicate or abstract.

Though there are differences in the overall frequency of the clone patterns in the study subjects, all clones sampled were categorized using the catalog 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 7 and 8 summarize the scope of the clones for each cloning pattern found in 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 latter being the most frequent case. These tables clearly show that cloning most 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

Table 7 Scope of clones by type - Apache httpd 2.2.4–30 Tokens and 60 Tokens

Group	Pattern	Scope	30 Tokens		60 Tokens	
			Count	% in sample	Count	% in sample
Forking	Experimental variation	Full	0	N/A	4	100
		Fragment	0	N/A	0	0
Forking	Platform variation	Full	9	100	10	100
		Fragment	0	0	0	0
Templating	Boiler-plating	Full	7	88	5	100
		Fragment	1	22	0	0
Templating	API	Full	6	75	17	100
		Fragment	2	25	0	0
Templating	Idioms	Full	0	N/A	0	0
		Fragment	0	N/A	12	100
Templating	Parameterized	Full	4	67	15	83
		Fragment	2	33	3	17
Customize	Replicate and specialize	Full	10	77	16	100
		Fragment	3	33	0	0
Exact match	Verbatim snippets	Full	0	0	4	44
		Fragment	2	100	5	56
Exact match	Cross-Cutting	Full	0	0	0	0
		Fragment	13	100	2	100

more likely to be reimplemented rather than sought out for duplication and this reimplementation will be more difficult to detect as clones.

4.7 Threats to External Validity

There are several threats to the validity of this analysis. First, the RGCs were judged by a single expert observer who is one of the authors of this paper. Without additional

Table 8 Scope of clones by type - Gnumeric 1.6.3–30 Tokens and 60 Tokens

Group	Pattern	Scope	30 Tokens		60 Tokens	
			Count	% in sample	Count	% in sample
Templating	Boiler-plating	Full	3	100	7	100
		Fragment	0	0	0	0
Templating	API	Full	1	17	9	100
		Fragment	5	83	0	0
Templating	Parameterized	Full	16	84	34	100
		Fragment	5	16	0	0
Templating	Idioms	Full	1	50	0	0
		Fragment	1	50	1	100
Customize	Replicate and specialize	Full	14	88	16	100
		Fragment	2	14	0	0
Exact match	Cross-cutting	Full	0	0	0	N/A
		Fragment	4	100	0	N/A
Exact match	Verbatim snippets	Full	6	32	3	75
		Fragment	13	68	1	25

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.

The choice of study subjects may also be a threat to the validity of this analysis. Both are medium size (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 through-out 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 (Mockus et al. 2000). In the case of Gnumeric, we found that the code is maintained by very 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.

The detector and analysis environment is also a threat to the validity. The CLICS clone detector uses parameterized string matching and does not find clones with reordered statements or statements added/removed. These types of clones would be considered *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. Clone detection methodologies using program dependence graphs overcome this limitation (Krinke 2001) but tools implementing this technology were not available to us for this study. However, parameterized substring matching is one of the top clone detection methods in terms of recall (Bellon 2002; Koschke et al. 2006) and we feel it is well suited for the goals of our study.

5 Discussion

In describing the patterns of code cloning, we see different management strategies that should be considered. 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, preferably through an automated approach such as source code generation. These varying maintenance strategies require a variety of different tools.

In the case of *templating* patterns, as mentioned above, it is clear that there is a need for synchronous editing, as suggest by Toomim et al. (2004), to manage 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, compared to performing the proper abstraction, if Linked Editing is used.

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 Kapser and Godfrey (2003, 2004, 2006b) the authors used cloning relationships visualized as architectural relationships as aids to locate several examples of these *forking* patterns.

In addition to locating *forking* cloning patterns, it is important that development tools also explicitly outline the similarities and differences in the code. 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 very important as it affects the decisions of how and when to propagate changes to duplicated code.

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 duplicated code. Such a tool would iterate over all candidate code clones and selectively patch clones according to human (expert) decisions. In all cases, it is vital that all clones are tracked to ensure no clones are excluded from an update (Duala-Ekoko and Robillard 2007).

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 behavior, such as in sibling classes of an object-oriented program, refactorings should be performed if the language supports this. In situations where the behavior 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.

We do not consider the patterns described in this work are exhaustive. The case study described in Section 4 confirms the patterns we describe occur in software systems, but also demonstrates that the list of patterns we originally reported (Kapser and Godfrey 2006a) was incomplete. During the study, three new patterns were added. We believe that this list of patterns will grow as more case studies are performed, and this is the focus of our future work on this topic.

Finally, the results of the case study identify a set of patterns that are most often harmful, namely *verbatim snippets* and *parameterized code*. While there were several examples of good usage of these clone patterns, the majority were deemed harmful. This may be an indication that developers should avoid this form of cloning. On the other hand several patterns were found to be mostly good: *boiler-plating*, *replicate and specialize*, and *cross-cutting concerns*. While not always good, when used with care (as with any form of design or implementation decision) these patterns are more likely to achieve an overall beneficial effect on the software system.

6 Related Work

Cataloging of software engineering principles and behaviors is not a new idea. Other works have cataloged common scenarios that arise in software development and maintenance. Godfrey et al. (Godfrey and Zou 2005) describe several scenarios in which maintenance activities lead to new functions in a software system. Fowler et al. documented approximately 70 refactorings (Fowler et al. 1999). Refactorings are patterns of behavior-preserving restructuring of source code used to eliminate bad design or source code entities, including duplicated code. Gamma et al. have described many design patterns to aid in making more flexible and reusable code (Gamma et al. 1995).

Clone classification schemes have been previously suggested, usually based on the degree of similarity of segments of code and also the type of differences (Balazinska et al. 1999a; Mayrand et al. 1996). In the work presented by Mayrand et al. (1996) and Balazinska et al. (1999a) these classifications are limited to function clones only. In previous work (Kapser and Godfrey 2003, 2005, 2006b) the authors present a classification scheme based on locality, size, code type, and similarity. The classification includes clones varying in scope from functions down to code fragments. This classification scheme was used to aid the analysis of cloning in large software systems. Balazinska et al. (2000) used a classification of function clones to produce software aided re-engineering systems for code clone elimination.

The classification of cloning presented in this study differs from the above categorizations both in the type of categorization and the goal of the work. In this paper, cloning is categorized primarily from a motivational perspective, while other categorizations focus on the structural properties of the clones. The goal of this paper is not to categorize clones for purposes of refactoring but to document the types of cloning that occurs in software to aid the general understanding of how cloning is used in practice.

Several case studies on cloning in software systems have contributed to the source of information for compiling these cloning patterns. Clone detection case studies on the Linux kernel have been reported (Antoniol et al. 2002; Casazza et al. 2001; Godfrey et al. 2000). Casazza et al. (2001) use metrics based clone detection to detect cloned functions within the Linux kernel. 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. (2002) did a similar study, evaluating the evolution of code cloning in the Linux, concluding that the structure of the Linux kernel did not appear to be degrading due to code cloning activities. Godfrey et al. present (Godfrey and Tu 2000) a preliminary investigation of cloning among Linux SCSI drivers. The authors recently investigated cloning in several large software systems (Kapser and Godfrey 2003, 2004, 2006b). These studies provide insight into the types of code that are cloned and why; in particular the authors (Kapser and Godfrey 2006b) describe an in-depth investigation into the sources of duplication in the Apache httpd web server.

Cordy reports on the use of code cloning as a method of minimizing and containing risk during maintenance and extensions of financial software (Cordy 2003). Often occurring in the form of customization, developers may use cloning to reuse the design of an existing application. Cloning is also used to separate the dependencies

of custom views on data that several modules or applications may have. Cloning in this way prevents the introduction of bugs into working code, and confines testing to a smaller subset of source code. Cordy also suggests that developers may not want to universally propagate bug fixes across clones as this may break dependent code (Cordy 2003).

Jarzabek et al. (Jarzabek and Shubiao 2003) and Basit et al. (2005) performed case studies for reducing duplication in the Java buffer classes and the STL. In their studies, they used a meta-language XVCL to reconstruct the code at compile time. (Jarzabek and Shubiao 2003) report that many clones existed because of language limitations and were removed using templates. Basit et al. (2005) show that the STL made heavy use of generics to reduce redundant code but redundant code still existed in a form analogous to *customization* and *boiler-plating* patterns where operators were modified. In a recent study, (Rajapakse et al. 2007) studied the effects of reducing cloning in web applications using server pages. During their study they initially build a web application based on the requirements of an industrial partner and then went through two stages of clone removals. While they were able to reduce they size of the source code by 75%, during this process they made several observations supporting our findings that not all cloning should be considered harmful. First, the initial version of the web application was developed rapidly because less time was spent trying to reduce duplicated code. In fact, as part of their development process they cloned modules to accelerate development of new ones. They also found that the overall performance of the final application was three times slower than the initial version. Clone unification also adversely affecting the overall evolvability of the software system. Six modules were unified into one, and if a single feature was changed for one of the original six modules, all modules would require retesting. Rajapakse and Jarzabek also suggested that distribution of the application might be adversely affected because a large amount of unnecessary code (embedded in branches of the unified module) must be distributed with the application. Updates that are distributed may unnecessarily cause downtime if only unused code us updated.

Balazinska et al. (1999a) measured the number of clones with various degrees of similarity, and found that exact duplicates were the most common followed by duplication with larger changes. The third and forth most prominent groups appeared to be clones where the called methods have been changed or a global variable has been changed. These last two types are similar to a *templating* pattern.

Research has also 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 duplicated code and encapsulating similar functionality (Rysselberghe and Demeyer 2003). Their findings show that developers are much more concerned with grouping functionality than removing duplicated code.

Kim et al. studied how developers used copy-and-paste features of the Eclipse IDE (Kim et al. 2004). 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. Kim et al. also observed usage patterns similar to the *templating* pattern noted here. The also observed that developers used copy-and-paste to duplicate control structures, similar to our *replicate and specialize*

pattern. The work presented here differs in that it focuses on how duplicated code that persists in source code is used as part of a design decision.

Kim et al. studied the evolution of code clones over time (Kim et al. 2005). In this study 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 only remained in the source code for several days, and that many long lived clones (clones that remained in the system for extended periods of time) could not be easily refactored for a variety of reasons: standard refactoring techniques could not be used, changes to 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. Our work differs in that we consider a single version of a public release of the software. Therefore we do not consider short lived clones 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 decision.

In order 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 (Aversano et al. 2007). Defining a set of evolutionary patterns based on the work of Kim et al. (2005), they analyzed how maintenance activities affected clone classes. In particular, they investigated how and why some code clone classes change 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 (Aversano et al. 2007)), 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 (Aversano et al. 2007)). They also found a large number of clones evolved independently, indicating developers use cloning as a development practice for starting new code (Aversano et al. 2007). Lozano et al. suggest that cloning should not always be considered harmful as they do not generally pose a risk of inconsistent updating. Lozano et al. (2007) report somewhat different results to those of Aversano et al. on the same study subject, DNSJava. In their findings, Lozano et al. report that most procedures that share a code clone do not co-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. The difference in these results may be due to 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. (1998) was used. This method of clone detection has been shown to have very high precision but low recall (Bellon 2002; Koschke et al. 2006). This may have resulted in overlooking types of clones that have several changes. Lozano et al. chose to use CCFinder (Kamiya et al. 2002), a parameterized suffix tree approach similar to the one presented in this paper. This approach is shown to have very high recall but very low precision (Bellon 2002; Koschke et al. 2006). The low precision of this approach may have falsely

indicated that many procedures have a cloning relationship when in reality they do not, skewing the overall number of procedures with cloning that are not consistently changed.

While interviewing and surveying developers and how they develop software, LaToza et al. (2006) uncovered six patterns of cloning based on the motivation for duplication: repeated work, example, scattering, fork, branch, and language. 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 largely enough that simple similarity matching will not uncover this type of duplication. Based on these patterns, LaToza et at. 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. is another indication that most cloning is unlikely to be created with ill intentions.

7 Conclusions

Code cloning is often presented as a negative design characteristic in software systems, usually attributed to the limitations of the developers. Often referred to as a bad "code smell", many negative effects of code cloning have been cited as reasons to remove code duplicates from source code. During our case studies of large software systems, we found that code cloning can often be used in a positive fashion.

In this paper we list several patterns of cloning that are used in real software systems. In our descriptions of these cloning patterns we discuss the pros and cons of using cloning and suggest methods of managing these code clones. We also discuss long term issues that may arise and provide concrete examples of these cloning patterns in real software systems. These insights provide evidence to support the notion that clones can be a reasonable design decision and that tools should be developed with long term maintenance of duplicates in mind.

In the future we would like to identify more patterns of cloning, and develop methods/tools to automatically identify these patterns in order to aid developers in maintenance and refactoring decisions. We would also like to identify the degree to which these patterns exist in software systems as well as occasions where using the cloning pattern was a successful development method and when it was not.

References

Antoniol G, Villano U, Merlo E, Penta MD (2002) Analyzing cloning evolution in the linux kernel. Inf Softw Technol 44(13):755–765

Aversano L, Cerulo L, Di Penta M (2007) How clones are maintained: an empirical study. In: CSMR '07: proceedings of the 11th european conference on software maintenance and reengineering. IEEE Computer Society, Los Alamitos, pp 81–90

Baker BS (1995) On finding duplication and near-duplication in large software systems. In: WCRE '95: proceedings of the second working conference on reverse engineering. IEEE Computer Society, Washington, DC, pp 86–95

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (1999a) Measuring clone based reengineering opportunities. In: Proceedings of the sixth international software metrics symposium. IEEE Computer Society, Los Alamitos, pp 292–303

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (1999b) Partial redesign of java software systems based on clone analysis. In: The proceedings of the 6th. working conference on reverse engineering. IEEE Computer Society, Los Alamitos, pp 326–336

Balazinska M, Merlo E, Dagenais M, Lague B, Kontogiannis K (2000) Advanced clone analysis to support object-oriented system refactoring. In: Proceedings of the 7th. working conference on reverse engineering. IEEE Computer Society, Los Alamitos, pp 98–107

Basit HA, Rajapakse DC, Jarzabek S (2005) Beyond templates: a study of clones in the STL and some general implications. In: ICSE '05: proceedings of the 27th international conference on software engineering. ACM, New York, pp 451–459

Baxter ID, Yahin A, Moura L, Sant'Anna M, Bier L (1998) Clone detection using abstract syntax trees. In: ICSM '98: proceedings of the international conference on software maintenance. IEEE Computer Society, Washington, DC, p 368

Bellon S (2002) Detection of software clones—tool comparison experiment. In: International workshop on source code analysis and manipulation. Montreal, October 2002

Brown WJ, Malveau RC, McCormick HW III, Mowbray TJ (1998) AntiPatterns: refactoring software, architectures, and projects in crisis, 1st edn. Wiley, New York

Casazza G, Antoniol G, Villano U, Merlo E, Penta MD (2001) Identifying clones in the linux kernel. In: First IEEE international workshop on source code analysis and manipulation. IEEE Computer Society Press, Los Alamitos, pp 92–100

Coplien JO (1992) Advanced C++ programming styles and idioms, 1st edn. Addison Wesley, Reading

Cordy JR (2003) Comprehending reality—practical barriers to industrial adoption of software maintenance automation. In: Proceedings of the 11th IEEE international workshop on program comprehension. IEEE Computer Society, Los Alamitos, pp 196–206

Duala-Ekoko E, Robillard M (2007) Tracking code clones in evolving software. In: 29th international conference on software engineering (ICSE 2007). IEEE Computer Society, Los Alamitos, pp 158–167

Ducasse S, Rieger M, Demeyer S (1999) A language independent approach for detecting duplicated code. In: Proceedings ICSM'99: international conference on software maintenance. IEEE Computer Society Press, Los Alamitos, pp 109–118

Fowler M, Beck K, Brant J, Opdyke W, Roberts D (1999) Refactoring: improving the design of existing code, 1st edn. Addison-Wesley Professional

Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns: elements of reusable object-oriented software, 1st edn. Addison-Wesley, Reading

Geiger R, Fluri B, Gall H, Pinzger M (2006) Relation of code clones and change couplings. In: Fundamental approaches to software engineering, 9th international conference, FASE 2006, Lecture notes in computer science, vol 3922. Springer, Heidelberg, pp 411–425

Godfrey MW, Tu Q (2000) Evolution in open source software: a case study. In: Proceedings of the 2000 international conference on software maintenance. IEEE, Piscataway, pp 131–142

Godfrey MW, Zou L (2005) Using origin analysis to detect merging and splitting of source code entities. IEEE Trans Softw Eng 31(2):166–181

Godfrey MW, Svetinovic D, Tu Q (2000) 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. http://plg.uwaterloo.ca/~migod/papers/2000/cascon00-linuxcloning.pdf

Gusfield D (1997) Algorithms on strings, trees, and sequences: computer science and computational biology. Cambridge University Press, New York

Higo Y, Kamiya T, Kusumoto S, Inoue K (2004) Aries: refactoring support environment based on code clone analysis. In: The 8th IASTED international conference on software engineering and applications (SEA 2004). MIT, Cambridge, pp 222–229

Jarzabek S, Shubiao L (2003) Eliminating redundancies with a "composition with adaptation" meta-programming technique. In: ESEC/FSE-11: proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering. ACM, New York, pp 237–246

Jiang L, Misherghi G, Su Z, Glondu S (2007) DECKARD: scalable and accurate tree-based detection of code clones. In: ICSE '07: proceedings of the 29th international conference on software engineering. IEEE Computer Society, Los Alamitos, pp 96–105

Johnson JH (1994) Substring matching for clone detection and change tracking. In: Proceedings of the international conference on software maintanence. IEEE, Piscataway, pp 120–126

Kamiya T, Kusumoto S, Inoue K (2002) CCfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Trans Softw Eng 8(7):654–670

Kapser C, Godfrey MW (2003) Toward a taxonomy of clones in source code: a case study. In: Evolution of large scale industrial software architectures. Amsterdam, 23 September 2003

Kapser C, Godfrey MW (2004) Aiding comprehension of cloning through categorization. In: Proc. of 2004 international workshop on principles of software evolution (IWPSE-04). IEEE Computer Society, Los Alamitos, pp 85–94

Kapser C, Godfrey MW (2005) Improved tool support for the investigation of duplication in software. In: ICSM '05: proceedings of the 21st IEEE international conference on software maintenance (ICSM'05). IEEE Computer Society, Washington, DC, pp 305–314

Kapser C, Godfrey MW (2006a) 'Cloning considered harmful' considered harmful. In: WCRE '06: proceedings of the 13th working conference on reverse engineering (WCRE 2006). IEEE Computer Society, Washington, DC, pp 19–28

Kapser CJ, Godfrey MW (2006b) Supporting the analysis of clones in software systems. J Softw Maint Evol Res Pract 18(2):61–82

Kiczales G, Lamping J, Menhdhekar A, Maeda C, Lopes C, Loingtier J-M, Irwin J (1997) Aspect-oriented programming. In: Akit M, Matsuoka S (eds.) Proceedings European conference on object-oriented programming, vol. 1241. Springer, Berlin Heidelberg New York, pp 220–242

Kim M, Bergman L, Lau T, Notkin D (2004) An ethnographic study of copy and paste programming practices in oopl. In: ISESE '04: proceedings of the 2004 international symposium on empirical software engineering (ISESE'04). IEEE Computer Society, Washington, DC, pp 83–92

Kim M, Sazawal V, Notkin D, Murphy G (2005) An empirical study of code clone genealogies. In: ESEC/FSE-13: proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on foundations of software engineering. ACM, New York, pp 187–196

Komondoor R, Horwitz S (2001) Using slicing to identify duplication in source code. In: SAS '01: proceedings of the 8th international symposium on static analysis. Springer, Heidelberg, pp 40–56

Kontogiannis K, DeMori R, Merlo E, Galler M, Bernstein M (1996) Pattern matching for clone and concept detection. Autom Softw Eng 3(1/2):77–108

Koschke R, Falke R, Frenzel P (2006) Clone detection using abstract syntax suffix trees. In: WCRE '06: proceedings of the 13th working conference on reverse engineering (WCRE 2006). IEEE Computer Society, Washington, DC, pp 253–262

Krinke J (2001) Identifying similar code with program dependence graphs. In: WCRE '01: proceedings of the eigth working conference on reverse engineering (WCRE 2001). ACM, New York, pp 301–309

LaToza T, Venolia G, DeLine R (2006) Maintaining mental models: a study of developer work habits. In: ICSE '06: proceedings of the 28th international conference on software engineering. IEEE Computer Society, Los Alamitos, pp 492–501

Lozano A, Wermelinger M, Nuseibeh B (2007) Evaluating the harmfulness of cloning: a change based experiment. In: MSR 2007: proceedings of the 4th int'l workshop on mining software repositories. IEEE Computer Society, Los Alamitos, pp 18–22

Mayrand J, Leblanc C, Merlo E (1996) Experiment on the automatic detection of function clones in a software system using metrics. In: Proceedings of the international conference on software maintenance. IEEE Computer Society Press, Los Alamitos, pp 244–253

Mockus A, Fielding R, Herbsleb J (2000) A case study of open source software development: the Apache Server. In: Proceedings of the 22nd international conference on software engineering (ICSE 2000). ACM, New York, pp 263–272

Rajapakse D, Stan Jarzabek S (2007) Using server pages to unify clones in web applications: a trade-off analysis. In: Proceedings ICSE '07: 29th international conference on software engineering. IEEE Computer Society, Los Alamitos, pp 116–126

Rysselberghe FV, Demeyer S (2003) Reconstruction of successful software evolution using clone detection. In: IWPSE '03: proceedings of the 6th international workshop on principles of software evolution. IEEE Computer Society, Washington, DC, p 126

Toomim M, Begel A, Graham SL (2004) Managing duplicated code with linked editing. In: VLHCC '04: proceedings of the 2004 IEEE symposium on visual languages - human centric computing (VLHCC'04). IEEE Computer Society, Washington, DC, 173–180

Ukkonen E (1995) On-line construction of suffix trees. Algorithmica 14(3):249–260

Walenstein A, Jyoti N, Li J, Yang Y, Lakhotia A (2003) Problems creating task-relevant clone detection reference data. In: Proceedings of the 10th working conference on reverse engineering (WCRE-03). IEEE Computer Society, Los Alamitos, pp 285–294

Cory J. Kapser graduated from the University of Alberta with a B.Sc. in Computer Science in 2002. He is currently pursuing a Ph.D. at the David R. Cheriton Computer Science, University of Waterloo under the supervision of Dr. Michael Godfrey. Currently he is interested in analysis and comprehension of large software systems.

Michael W. Godfrey is Associate Professor at the David R. Cheriton School of Computer Science in the University of Waterloo, where he is also a member of SWAG, the Software Architecture Group. He holds a PhD in Computer Science from the University of Toronto (1997), and between 2001 and 2006 he held an Associate Industrial Research Chair in telecommunications software engineering sponsored by Nortel Networks and NSERC. His main research area is software evolution: understanding how and why software changes over time. His research interests include empirical studies, software tool design, reverse engineering, and program comprehension.