"Cloning Considered Harmful" Considered Harmful

Cory Kapser and Michael W. Godfrey
Software Architecture Group (SWAG)
David R. Cheriton School of Computer Science, University of Waterloo
{cjkapser, migod}@uwaterloo.ca

Abstract

Current literature on the topic of duplicated (cloned) code in software systems often considers duplication harmful to the system quality and the reasons commonly cited for duplicating code often have a negative connotation. While these positions are sometimes correct, during our case studies we have found that this is not universally true, and we have found several situations where code duplication seems to be a reasonable or even beneficial design option. For example, a method of introducing experimental changes to core subsystems is to duplicate the subsystem and introduce changes there in a kind of sandbox testbed. As features mature and become stable within the experimental subsystem, they can then be introduced gradually into the stable code base. In this way risk of introducing instabilities in the stable version is minimized. This paper describes several patterns of cloning that we have encountered in our case studies and discusses the advantages and disadvantages associated with using them.

1. Introduction

It is believed that most large software systems contain a non-trivial amount of redundant code. Often referred to as code clones, these segments of code typically involve 10–15% of the source code [24, 25]. Code clones can arise through a number of different activities. For example, intentional clones may be introduced through direct *"copy-and-pasting"* of code. Unintentional clones on the other hand may be the manifestation of programming idioms related to the language or libraries the developers are using.

In much of the literature on the topic [2, 7, 12, 21, 22, 27, 28], cloning is considered harmful to the quality of the source code. 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 [15]. 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 [4, 7], and tools specifically to aid developers in the manual refactoring of clones have also been developed [19].

There is no doubt that code cloning is often an indication of sloppy design and in such cases should be considered to be a kind of development "bad smell". However, we have found that there are many instances where this is simply not the case. For example, cloning may be used to introduce experimental optimizations to core subsystems without negatively effecting 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. [1]

This paper introduces eight cloning patterns that we have uncovered during case studies on large software systems, some of which we reported in [23, 24, 25]. 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. 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 presented by Toomim et al. [29].

This paper introduces the notion of categorizing high level patterns of cloning in a similar fashion to the cataloging of design patterns [14] or anti-patterns [8]. 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

[1] A simple (but trivial) example is the title of this paper. Although there is a kind of duplication in the wording, no "refactoring" of the title would carry the same connotations as the original statement.

software. This documentation crystallizes a vocabulary that researchers and practitioners can possibly 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 severity of the problem in general.

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 eight patterns we found in software systems, Section 4 discusses the implications of code cloning patterns on maintenance and tool requirements, Section 5 describes work that has contributed to the understanding of code cloning, and in Section 6 we discuss our conclusions and future work.

2. Code Cloning

Code cloning is considered a serious problem in industrial software [1, 2, 7, 9, 12, 21, 22, 27, 28]. It is suspected that approximately 10%–15% of many large systems is part of duplicated code [2, 12, 24, 25], and it has been documented to exist at rates of over 50% of the effective lines of code (ELOC) in a particular COBOL system [12]. The literature on the topic has described many situations that can lead to the duplication of code within a software system [2, 7, 21, 22, 27, 28]. 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. 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 [7].

Duplicates can also be introduced with good intentions. Duplicating code can, in some situations, be used to keep software architectures clean and understandable. Duplicates can also be used to keep unreadable, complicated abstractions from entering the system. Lack of expressiveness of a given programming language may lead to the use of "boiler-plated" solutions for particular problems [30], or even source code generation. This kind of technique is common in COBOL development, for example. 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.

There are several problems associated with cloning. Code cloning can lead to unnecessary increase in code size [2, 21]. Cloning code can lead to unused, or "dead", code in the system that left unchecked can cause problems with code comprehensibility, readability, and maintainability over the life time of the software system [21]. Maintenance efforts can be increased when bugs have to be fixed multiple times, and these changes could be prone to errors. Clones of code that is not well understood can introduce bugs. For example, variables may be shared and modified unknowingly [21]. Program comprehensibility can be affected by the increased code size, as well as the need to understand the differences between the duplicates.

Code clones can have beneficial effects on the source code. Code clones can be used to reduce complexity in source code in the cases where abstractions are difficult to form. As a result, duplicates may be easier to understand and modify than a solution that employs abstraction, as the study performed by Toomim et al. [29] suggests. Also, risk to the the stability of the code can be avoided by cloning rather than creating a new abstraction. Cordy notes that financial institutions consider code quality the most important concern when maintaining software [11]. 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. Cloning is a common method of risk minimization that allows code to be maintained and modified separately, containing the risk of introducing errors to a single system or module [11]. Cloning can be 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 unknown to use abstractions.

3. Patterns of Cloning

During our investigations of cloning in large software systems [23, 24, 25], we found several recurring patterns of cloning, or rather 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 are 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. 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 and have not yet been formally validated.

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. 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 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 1.2.

We have divided the eight patterns into three related groups: *Forking*, *Templating*, and *Customization*. 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. In these types of cloning, 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 but appropriate abstraction mechanisms 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.

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 is often 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 examples of *forking* patterns that we have seen in our case studies.

3.1.1. Hardware variations.

Name. Hardware variations

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. The risk of changing the existing driver is especially high in this situation 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. 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. Code growth can be a particular issue with this pattern of cloning because entire files or subsystems are copied.

Management. Groups of cloned drivers should be clearly identified, and potential bug fixes should be investigated 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 for simplicity of use within the system. 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 [16]. 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.

Name. 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.

Advantages. Code complexity that is inherent to platform optimized code that is interleaved is avoided. Additionally, stability for currently supported platforms is maintained. As platforms are likely to evolve independently, maintaining support for one platform will not effect the stability of the code for other platforms.

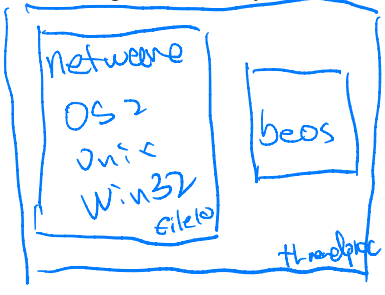
Disadvantages. The code will evolve along two dimensions: the requirements of the software and the support of the platform. Bug fixes may be difficult to propagate as it may not be clear how or if the bugs are present in each version of the code. Changes to the interface of the platform specific code 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 cloning the code 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 rigid 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 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.

Name. 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. System stability is protected while still allowing users access to leading edge development. Changes made to the experimental fork can be merged with or replace the stable version at a later time.

Disadvantages. Merging code at a later point may be difficult if the stable version continues to evolve independently.

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.

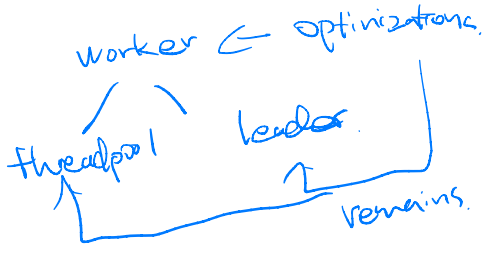
Long term issues. As the original and duplicate code evolves, consistent maintenance may become more difficult. Documentation of the differences should be maintained in order to aid program comprehension.

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 [25].

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` [25]. The cloned subsystems are experimental variations on `worker` 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 parametrization, 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 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 three examples of *templating* patterns.

3.2.1. Boiler-plating due to language in-expressiveness.

Name. Boiler-plating due to language in-expressiveness
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. Can make reuse of trusted code possible. Allows for consistent behavior for related concepts, improving program comprehensibility.
Disadvantages. 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 that makes an explicit link to all duplicates is important. Tools and methodologies such as Linked Editing [29] should be used to ensure consistent changes are made to all duplicates. Another approach to managing these clones is to use generated code at build time [20], making the duplicate exist only when the source code is compiled.
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 is a great deal of duplication whose only modification is the data type of the procedure parameters.

3.2.2. API/Library protocols.

Name. API/Library protocols
Motivation. Often the use of particular application program interfaces (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 parametrize them appropriately to be used for their particular problem.
Advantages. Novice users of the API or library can learn from other code. 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. Developers may be duplicating buggy or fragile code, 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, rigorously review the duplicates to ensure the duplicated code is high quality.
Long term issues. Changes to the API will require 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.

3.2.3. General language or algorithmic idioms.

Name. 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 [10], and there is a 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.
Disadvantages. Inconsistencies or faulty implementations of programming idioms may be easily overlooked. Incorrect or inefficient idioms (or anti-idioms) can also be duplicated, degrading the quality of the code.
Management. Anti-idioms 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. 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.

3.3. Customization

Customization often arises when code solving a very similar problem to the current problem exists, but additional or differing requirements create the need for extension or modification. For a variety of reasons, such as concerns about system stability or code ownership, the existing code can not be modified to encompass this additional problem. In these cases, code may be cloned and customized to suit the specific development task. In this section we describe two examples of *customization* patterns.

3.3.1. Bug workarounds.

Name. 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 work arounds 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. The problem can be solved without requiring retesting of other code. This solution can allow for progress in development, although it should only be a temporary measure.
Disadvantages. 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 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: `src/backend/commands/tablespace.c`, `src/port/copydir.c`, `src/backend/access/transam/xlog.c`.

3.3.2. Replicate and specialize.

Name. 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 [11].
Advantages. Reduces immediate costs in testing and refactoring. Additionally, the high cognitive cost of developing the abstraction is avoided [29].
Disadvantages. Long term costs of finding and maintaining these duplicates could out-weigh 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 can not 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. 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. [26].

Examples This pattern is the most common type of cloning. 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. However, how the items are chosen to be added to the menu differs, causing a minor change and addition of several lines. Another small difference is the way in which the menu title is made near the end of the procedure. In addition to these customizations, the data type containing the list of entities is also different, performed as a parametric change.

4. 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, such as that suggested in [29], 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. [29] provides some evidence to suggest that there is less cognitive load to manage the duplicated code, rather than 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 [23, 24, 25] 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 difference 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 the differences in the code clones is very important as it effects the decisions of how and when to propagate changes to duplicated code.

In the cases of the *customization* patterns, the tool requirements are a combination of the *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.

While we believe that not all clones require refactoring of abstractions, 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.

5. 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. Zou et al. [18] describe several scenarios in which maintenance activities lead to new functions in a software system. Fowler et al. documented approximately 70 refactorings [13]. 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 [14].

Clone classification schemes have been previously suggested, usually based on the degree of similarity of segments of code and also the type of differences [3, 28]. In the work presented by Mayrand et al. [28] and Balazinksa et al. [3] these classifications are limited to function clones only. In previous work [23, 24, 25] 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 blocks. This classification scheme was used to aid the analysis of cloning

in large software systems. Balazinska et al. [5] used a classification of function clones to produce software aided re-engineering systems for code clone elimination.

The classification of cloning presented here differs from the above categorizations in both the type of categorization and the goal of the work. In this paper, cloning is categorized primarily from 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 occur in software to aid the general understanding of how cloning is done 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 in [1, 9, 16]. In [9], Casazza et al. 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. [1] 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. In [17] a preliminary investigation of cloning among Linux SCSI drivers was performed. The authors recently investigated cloning in several large software systems [23, 24, 25]. These studies provide insight into the types of code that are cloned and why, in particular [25] describes 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 [11]. 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 [11].

Jarzabek et al. [20] and Basit et al. [6] performed case studies for reducing duplication in on the Java buffer classes and the STL. In their studies, they used a meta-language XVCL to reconstruct the code at compile time. In [20] many clones existed because of language limitations and were removed using templates. In [6] 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. Balazinska et al. [3] 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.

Kim et al. studied how developers used copy-and-paste features of the Eclipse IDE [26]. In this study, Kim et al. 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. Kim et al. 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 the source code is used as part of a design decision.

6. 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 way.

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 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

[1] G. Antoniol, U. Villano, E. Merlo, , and M. D. Penta. Analyzing cloning evolution in the linux kernel. In *Information and Software Technology 44(13)*, 2002.

[2] B. S. Baker. On finding duplication and near-duplication in large software systems. In *WCRE '95: Proceedings of the Second Working Conference on Reverse Engineering*, page 86, Washington, DC, USA, 1995. IEEE Computer Society.

[3] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 292–303, 1999.

[4] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial redesign of java software systems based on clone analysis. In *The Proceedings of the 6th. Working Conference on Reverse Engineering*, pages 326–336, 1999.

[5] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced clone analysis to support object-oriented system refactoring. In *Proceedings of the 7th. Working Conference on Reverse Engineering*, pages 98–107, 2000.

[6] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. 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*, pages 451–459, New York, NY, USA, 2005. ACM Press.

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 368, Washington, DC, USA, 1998. IEEE Computer Society.

[8] W. J. Brown, R. C. Malveau, H. W. M. (III), and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1st edition, 1998.

[9] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. D. Penta. Identifying clones in the linux kernel. In *First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 92–100. IEEE Computer Society Press, 2001.

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

[11] J. R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. In *IWPC*, pages 196–206. IEEE Computer Society, 2003.

[12] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings ICSM'99: International Conference on Software Maintenance*, pages 109–118. IEEE, 1999.

[13] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1st edition, 1999.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1st edition, 1995.

[15] R. Geiger, B. Fluri, H. Gall, and M. Pinzger. Relation of code clones and change couplings. In *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006*, volume 3922 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2006.

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

[17] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the 2000 International Conference on Software Maintenance*, 2000.

[18] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2), 2005.

[19] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring support environment based on code clone analysis. In *The 8th IASTED International Conference on Software Engineering and Applications(SEA 2004)*, pages 222–229, 2004.

[20] S. Jarzabek and L. Shubiao. 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*, pages 237–246, New York, NY, USA, 2003. ACM Press.

[21] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the International Conference on Software Maintanence*, pages 120–126, 1994.

[22] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*, pages 654–670. IEEE Computer Society Press, 2002.

[23] C. Kapser and M. W. Godfrey. Toward a taxonomy of clones in source code: A case study. In *Evolution of Large Scale Industrial Software Architectures*, 2003.

[24] C. Kapser and M. W. Godfrey. Aiding comprehension of cloning through categorization. In *Proc. of 2004 International Workshop on Principles of Software Evolution (IWPSE-04)*, pages 85–94, 2004.

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

[26] M. Kim, L. Bergman, T. Lau, and D. Notkin. 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)*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

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

[28] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International*

Conference on Software Maintenance, pages 244–253. IEEE Computer Society Press, 1996.

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

[30] A. Walenstein, N. Jyoti, J. Li, Y. Yang, and A. Lakhotia. Problems creating task-relevant clone detection reference data. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE-03)*, pages 285–294. IEEE Computer Society Press, 2003.