Chapter 9 Software similarities: symptoms and causes

Summary of this Chapter:

It is the very spirit of reuse-based evolution to identify and leverage from all kinds of similarity patterns arising during evolution. Spotting patterns of evolutionary changes affecting software over time, is essential to avoid complications during evolution. By *similarity patterns* we mean, for example, groups of similar components within a single released system, or across released systems. In the first part of this Chapter, we take a closer look at the software similarity phenomenon and repetitions (software clones) it spawns. We draw observations from empirical studies. One such study, the Java Buffer library, JDK 1.5., we discuss in details. We briefly describe other studies, and comment on the main results.

In the second part of this Chapter, we generalize findings from empirical studies. We trace the roots of the similarity problem and argue about the general nature of the problem's symptoms and causes.

In the next Chapter, we show how mixed-strategy approach deals with unifying similarity patterns.

9.1 The problem of software similarities and cloning

Software similarity pattern is a concept denoting any recurring problem in analysis, design or implementation spaces. *Software clones* are similar program structures that spawn from the fact that similarity exists.

We use terms *similarity patterns*, *similar program structures* and *clones* interchangeably, depending on the context.

Similarity patterns create an opportunity for program simplification via generic design. The aim of *generic design* is to unify differences among similar program structures and represent a group of such structures in unique, generic form. Most of the programming languages support type-parameterization, called generics in Ada [1], Java JDK 1.5, C# [36], and templates in C++ [61], to define generic classes or methods. Design patterns [25] and more recent techniques such as Aspect-Oriented Programming [37] also increase genericity of program solutions. Therefore, not all the similarity patterns must necessarily lead to clones. Having observed similarities, skillful design may help us avoid some of the repetitions.

We define *clones* informally as program structures of considerable size that exhibit significant similarity. The actual size and similarity (which can be measured, for example, in terms of replicated lines of code) varies depending on the context. Clones may represent similar program structures of any kind and granularity.

Clones may or may not represent program structures that perform well-defined functions. We distinguish two types of clones, namely:

- *simple clones:* the same or similar segments of contiguous code, such as class methods or fragments of method implementation, and

- *structural clones*: patterns of inter-related components (e.g., collaborating classes) representing design solutions repeatedly applied by programmers to solve similar problems, so-called "mental templates" [10]), similar program modules or subsystems (so-called architecture-level patterns) comprising many components.

Most of the interesting clones, particularly structural clones, are similar but not identical. Changes among clones result from differences in intended behavior, and from dependencies on the specific program context in which clones are embedded (such as different names of referenced variables, methods called, or platform dependencies).

It is difficult to define the concept of similarity in precise, descriptive terms: similarity is a multi-faceted phenomenon that escapes precise definition. The notion of similarity necessarily involves human judgment and, therefore, is subjective. Whether or not we consider two code structures as clones depends on what we want to do with them and for what purpose. For example, we might want to merely document instances of similar program structures for future maintenance, or unify them with a single generic structure for simplification purpose. The reason why we are interested in clones determines the type and size of similar program structures that we wish to consider as clones.

We can characterize clones that are likely to meet our goals by measures such as the minimum size of clone candidates of our interest, the percentage of common code among code structures [32], types of allowable parametric differences or the Levenstein editing distance [44] among code structures. (The Levenstein editing distance is measured in terms of the number of editing operations required to convert one text fragment to another.) Finally, characterizations of clones may depend on the abstraction level of similarity patterns (design or code), and whether we use automatic or manual process to identify clones in a program. Metrics are essential for automated detection of clones. On the other hand, manual analysis of design-level similarities is better done in a top-down manner, relying on concepts of the underlying application domain and a programming language.

Software clones have received much attention in research. Cloning has been studied in the context of re-engineering [10], refactoring [24] and clone detection [10][32].

Poor design and ad hoc maintenance may induce clones. At time, cloning is done for good cause. Programmers clone code for quick productivity gains, to speed up development and maintenance. Ad hoc cloning is a simple form of reuse, but it does not lead to systematic reuse (e.g., via software product lines [17]). Mayrand observes that cloning is a sign of lack of good re-use processes in place [47]. Sometimes, cloning is done to increase the robustness of life-critical systems, for better performance, or to minimize dependencies among developers in large projects. Recurring problems of a similar structure in analysis and design spaces lead to structural clones. Analysis patterns [23] and design patterns [25] exemplify these situations. Programmers repeatedly apply so-called *mental macros* to solve various programming problems [10]. Mental macros reflect common programming wisdom or programmer's specific experiences. With each application of a mental macro, a redundancy occurs, in a more or less explicit form. Architecture-centric and pattern-driven development encouraged by modern platforms (.NET™ and J2EE™) standardize program solutions, which leads to clones at all levels [70]. Similarities induced by standard way of organizing software are beneficial, as it becomes easier to understand the code written by somebody else - similar problems are being solved in similar ways. Still, pattern-driven development may also pose

certain complication for future maintenance and reuse due to fragmentation and scattering of code in application domain-specific areas. The issue is multi-faceted, most important and interesting, and we revisit it later.

Finally, clones may be generated by tools. Rapid application development features of IDEs (integrated development environments) save programmer's development time by generating code that can be inferred from simple programmers' inputs (e.g., selections of menu items or clicking on a button in IDE's user interface).

Clones arise when developers face difficulties of avoiding them using conventional techniques [30]. Kim estimates that 34% of clones cannot be refactored [39]. In the example of Buffer library discussed in this Chapter, most of the 68% of code contained in simple and structural clones cannot be eliminated using Object-Oriented and other conventional methods, without compromising other important design goals. In yet other studies, also summarized in this Chapter, we found similar rates of cloned code, difficult to refactor with conventional methods.

We close the discussion by classifying clones into the following categories:

1. *Desirable*. Such clones are useful at runtime (e.g., for performance or reliability) and cannot be eliminated from programs. Intentional clones induced by an implementation technique (e.g., by J2EE or .NET architecture and patterns) also belong to this category. We comment on these important type of clones in more detail later.
2. *Avoidable*. Such clones are caused by the programmer's carelessness. For example, similar code fragments introduced by poor design or ad hoc "copy-paste-modify" practice during maintenance often fall into this category.
3. *Problematic*. These are all the clones that are not *desirable* but are difficult to avoid using given design techniques, without compromising important design goals. As the name suggests, nothing definite can be said about problematic clones. They are relative to design techniques and design goals. Despite their hypothetical, imprecise and floating nature, we find the concept useful in discussing cloning problems. Most of the clones discussed in the Buffer library case study belong to this category.

While there are good reasons for creating certain clones, most of them, independently of the reasons why they occur, complicate programs. Cloning has received much attention because of its harmful impact on maintenance. Clones increase the risk of update anomalies, and hinder program understanding during the maintenance in at least, two ways: (1) a programmer must maintain more code than he/she would have to maintain should the clones be removed, and (2) when one logical source of change affects many instances of a replicated program structure scattered throughout a program, to implement a change, a programmer must find and update all the instances of the replicated structure. The situation is further complicated if a replicated structure must be changed in slightly different ways, depending on the context.

There are even deeper reasons to avoid repetitions: By multiplying similar program structures in variant forms we hinder conceptual integrity of the design, which Brooks calls "the most important consideration in system design" [13].

We may merely document clones for future maintenance, or unify them by a macro or function, as it is often done during re-engineering [10], refactor a program to avoid clones [24], or apply generative techniques to eliminate clones at the extra plane level (such as

mixed-strategy representation with XVCL), while leaving clones at the program code level [28][29][52][70]. At times, clone elimination at the program code level may be hindered by risks involved in changing programs [18], or by other design goals that conflict with refactorings [28][29].

To see how significant the problem of cloning is, let us look at the cloning rates reported in various surveys: 50-85% [46], 13-20% [2], 12.7% [10], and 6.4-7.5% [43]. Software examined in re-engineering projects contained around 40% of cloned code [56]. It is important to note that those surveys did not consider design-level structural clones. Instead, they focused on simple clones (exact and similar code fragments), and in many cases addressed only clones found by automatic clone detection tools.

In our studies, we looked at the bigger picture, focusing on both simple clones and large-granularity, design-level structural clones. Then, cloning rates dramatically increase: In lab studies and two industrial projects, we found 50%-90% cloning rates, in well-designed programs, not affected by maintenance. Our studies covered a range of application domains (business systems, Web Portals, command and control systems, and class libraries), programming languages (Java, C++, C#, ASP, JSP, PHP) and platforms (J2EE, .NET, Unix, Windows). For example, the extent of cloning in Java Buffer library was 68% [28] [29](described later in this Chapter), in parts of STL (C++) - over 50% [5], in J2EE Web Portal – 61% [69], and in certain ASP Web portal modules – up to 90% [52]. The last two results are from pilot projects by our industry partner ST Electronics. In many cases, repetitions occurred as there was no simple way of avoiding them with conventional methods, without compromising other important design goals [30]. In case of the J2EE Web Portal, the architecture and patterns induced cloning. However, we could effectively treat the similarities by unifying them with mixed-strategy generic structures. We measured the percentage of cloning by comparing the subject program against a non-redundant mixed-strategy representation of the same program in its original form.

Not always does size reduction lead to program simplification. However, in all our studies, we focused only on repetitions of *significant engineering importance*, meaning that they created reuse opportunities, induced extra conceptual complexity into a program, and/or were counter-productive for maintenance. A survey of 17 Web Applications in which we found 17-60% of code contained in clones [54] was the only exception, as in this study, we were not able to conduct a qualitative analysis of the replicated code.

9.2 Software similarities and reuse

Avoiding repetitions, during development and in software products, is the goal of software reuse. The main themes of software reuse are identifying similarities and differences (variabilities) in requirements (domain analysis [53]), architecture/design and components [17]. With this understanding, we can attempt building generic, adaptable, therefore reusable, program solutions. Current state-of-the-art in reuse is based on architecture-centric, component-based approaches and Product Line concepts [17]. Having identified and scoped variant features (usually modeled as feature diagrams [33]), a Product Line architecture (PLA) is designed to accommodate variant features. With component-based design, genericity of a PLA is achieved by stabilizing component interfaces and localizing the impact of variant features to possibly small number of components. However, for variant features that have crosscutting effect on PLA components, this goal cannot be easily achieved. A component

affected by variant features explodes into numerous versions. Similarities among component versions cannot be easily spotted, and reuse opportunities offered by such similarities are often missed. Given thousands of variant features and complex features inter-dependencies arising in industrial Product Lines, the cost of finding "best matching" component configurations for reuse, and then the follow up component customization, integration and validation, may become prohibitive for effective reuse [21].

With mixed-strategy approach, we treat similarity patterns induced by variant features as first class citizens, no matter whether they have a crosscutting effect on components or not. Mixed-strategy generic structures unify similarity patterns at any granularity level and of any type – from a subsystem, to pattern of components, to component, to class and to program statement in class implementation. For that reason, mixed-strategy often extends the scope and rates of reuse achievable by means of conventional techniques.

Architecture-centric, patter-driven development which is now supported by major platforms such as .NET™ and J2EE™. Patterns lead to beneficial standardization of program solutions and are basic means to achieve reuse of common service components. IDEs support application of major patterns, or programmers use manual copy-paste-modify to apply yet other patterns. Mixed-strategy can enhance the benefits of modern platforms by automating pattern application, and emphasizing the visibility of patterns in code. Patter-driven design facilitates reuse of middleware service components, but tends to scatter application domain-specific code. With mixed-strategy, we can package and isolate otherwise scattered domain-specific code into reusable generic components. Such extensions improve development and maintenance productivity, and allow reuse to penetrate application business logic and user interface system areas, not only middleware service component layers.

9.3 Software similarities and generic design

Generic design [27][26] aims at achieving non-redundancy by unifying differences among similar program concepts or structures, for simplification purpose. Yet, the evidence abounds that programs are often polluted by redundant code [14][10][22][28][29][32][47][43]. The extent to which similar program structures deliberately spread through programs, hint strongly that potentials of generic design may not be fully exploited. Particularly, as we look at the bigger picture, moving beyond small granularity duplications (e.g., similar code fragments often termed clones in the literature), we find increasing rates of code contained in similar design-level structures (e.g., similar classes, components or patterns of collaborating classes/components). In our studies, in which we concentrated on similarity patterns at all levels, we often find around 60% of code repeated many times, in variant forms. While not all such repetitions are necessarily bad, most of them complicate a program, hinder program understanding, and contribute much to high cost of day-to-day maintenance and long-term evolution.

There are three engineering benefits of generic design (and three reasons to avoid unnecessary repetitions): Firstly, genericity is an important theme of software reuse where the goal is to recognize similarities to avoid repetitions across projects, processes and products. Indeed, many repetitions merely indicate unexploited reuse opportunities. Secondly, repetitions hinder program understanding. Repeated similar program structures cause update anomalies complicating maintenance – vide research on re-engineering, refactoring and clone detection [6]. Thirdly, by revealing design-level similarities, we reduce the number of distinct

conceptual elements a programmer must deal with. Not only do we reduce an overall software complexity, but also enhance conceptual integrity of a program which Brooks calls “the most important consideration in system design” in his famous *The Mythical Man-Month*. Common sense suggests that developers should be able to express their design and code without unwanted repetitions, whenever they wish to do so.

Parameterization is a prime concept in generic design, almost as old as programming. Macros and pre-processing (such as cpp) are simple forms of parameterization that are still used today. Most of the programming languages provide type parameterization called generics (in Ada, Eiffel, Java or C#) or templates (in C++). Despite benefits, type parameterization is not sufficient to unify many similarity patterns that arise in class libraries and application programs.

In the following sections, we show examples and discuss the reasons why conventional programming techniques are not effective in achieving goals of generic design.

9.4 Software similarities, genericity and software complexity

Consider a Project Collaboration Web Portal (PCWP), such as depicted at Figure 1. PCWP supports communication among team members. For that, PCWP maintains information about staff, projects, who works on which projects, tasks assigned to various staff members, and other information that matters in collaboration.

PCWP consists of modules such as Staff, Project or Task. For example, module Staff allows a user to Create, delete and modify data about staff members, assign staff members to projects, etc. CPG-Nuke allows extending its functionality by adding more modules.

PCWP modules are deployed on top of the Portal Foundation that implements common services reused by the portal modules. PCWP also reuses some of the standard modules such as Use, Admin, statistics, History and others.

PCWP may be implemented in of may available Web technologies such as PHP™, ASP™, Ruby on Rails™, ASP.NET™, J2EE™ or even just in Java™ and JavaScript™. The implementation language and platform does not matter for our argumentation.

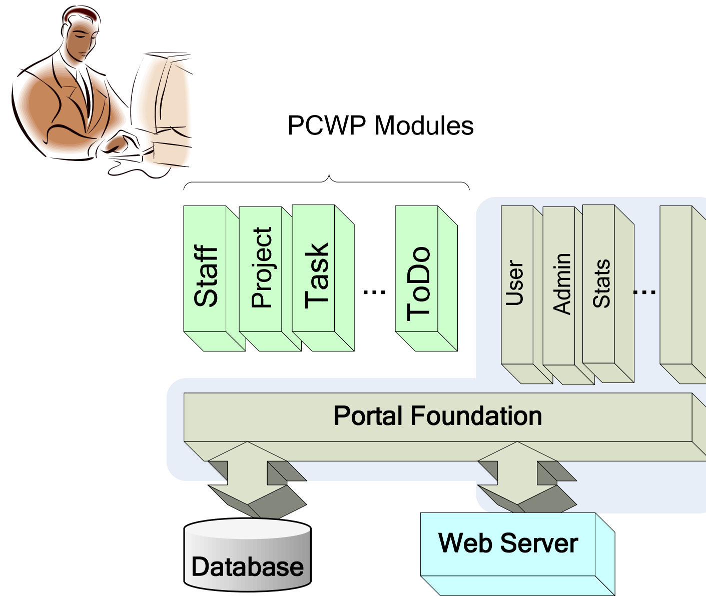

Figure 1. Project Collaboration Web Portal (PCWP)

For each module, there are operations, such as *Create*, *Edit*, *Delete*, *Display*, *Delete*, *Find* or *Copy*. The design of each operation such as *CreateStaff, CreateProject, UStaff or EditProject* involves a pattern of collaborating classes from GUI, service and database layers. Each box in Figure 2 represents a number of classes: GUI classes implement various forms to display or enter data; Business Logic classes implement data validation and actions specific to various operations and/or entities; Entity classes define data access; classes at the bottom contain table definitions.

Classes in corresponding boxes at each level display much similarity. Figure 2 shows two dimensions of DEMS similarity patterns: The first one, results from the same operation applied to different entities, e.g., *CreateStaff*, *CreateProject*, …, *EditStaff*, *EditProject*, …, and others, as shown in Figure 2 (a). The second similarity pattern results from different operations for a given module (e.g., *CreateStaff*, *EditStaff,* …,*CreateProject*, *EditProject*, …, and others, as shown in Figure 2 (b).

However, there are also differences across classes for various operations, implied by different meaning of various entities such as Staff, Project, Task: For example, operation *CreateProject* requires different types of data entry and data validation than *CreateStaff*. A user interface form for *CreateStaff* is displayed with empty data fields, while a form for *EditStaffr* has to fetch data from the Staff table and display it in the proper data fields.

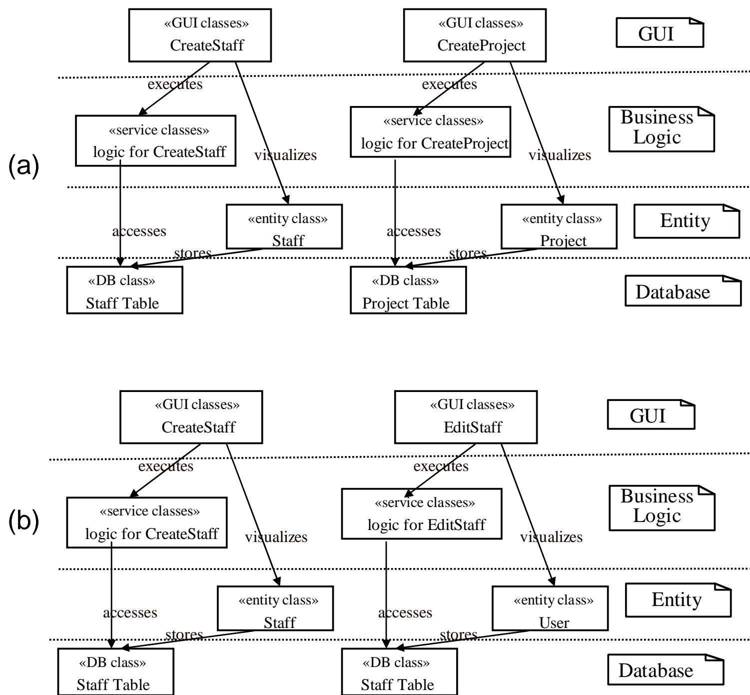

Figure 2. A recurring pattern of components

Suppose we can't find a way to unify differences among operations for different PCWP modules. Then, we have to implement each operation such as *CreateStaff*, *CreateProject*, …, *EditStaff*, *EditProject*, etc. separately, ignoring much similarity that exists among operations.

To visualize the situation, let's assume we have 10 modules and 5 operations. Then, our program solution includes 10*5 = 50 patterns such as shown in Figure 2. If each box consists of only one class, then we have 50*4 = 200 classes and 200 interactions among classes. That's a conservative estimation, as in reality some of the boxes contain more than one class. This situation is shown in Figure 3 (a).

Suppose we can unify differences among the same operation for different modules, so that the 10 modules use only five generic functions Create[M], Edit[M], etc. We reduce the solution space to 10 + 5 = 15 components and 50 interactions, with the slightly added complexity of a generic representation of operations. The interactions have become easier to understand as each form interacts with components that have been merged into five groups rather than 50 distinct components (situation shown in Figure 3 (b).

Suppose we further unify 10 modules with one generic Module. We reduce the solution space to 1 + 5 = 6 components and with five groups of interactions, each group consisting of 10 specific interactions, plus the complexity of a generic representation (situation shown in Figure 3 (c).

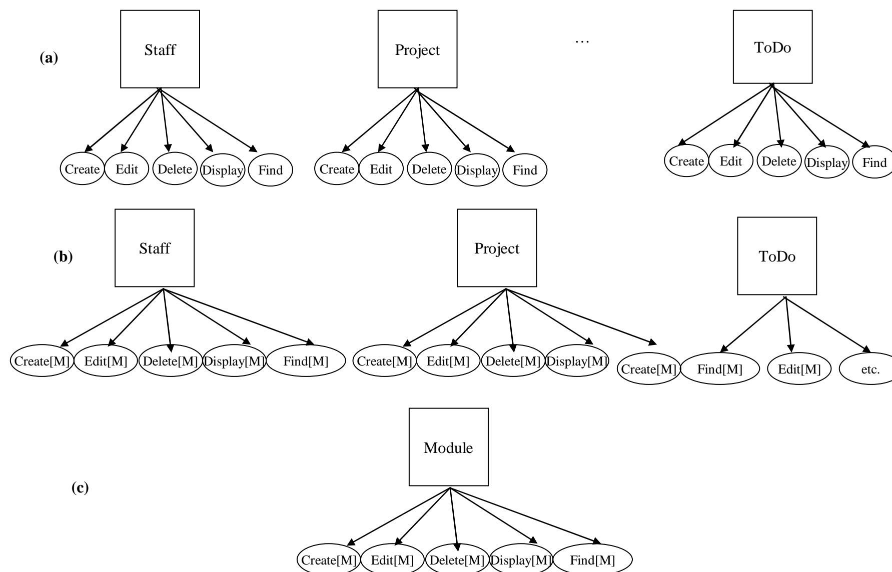

Figure 3. The impact of unifying similarities with generic structures on complexity

Any form of validation at the level of generic representation depicted in Figure 3 (b) or (c) is bound to be much simpler and effective than validation at the level of concrete components (Figure 3 (a)). Similarly, any changes at the level of generic representation are easier to manage than at the level of concrete components due to clearer visibility of change impact, reduced risk of update anomalies, and smaller number of distinct interfaces that have to be analyzed. Finally, generic representation of components and their configurations form natural units of reuse, potential building blocks of Product Line architectures [17].

Independently of an implementation technique (ASP, C#, J2EE or PHP), it was not possible to represent operations for different modules in a generic form. This resulted in repetitions that increased the complexity of a program solution, and contributed much to development and maintenance effort. It also hindered reuse of similar program solutions across PCWP modules, and in building other Web Portals.

A case study of a Buffer library illustrates typical problems in unifying similarity patterns with conventional programming techniques, and explains some of the general roots of the problem.

9.5 Similarity patterns in the Buffer library: an example

9.5.1 An overview of the Buffer Library

A buffer contains data in a linear sequence for reading and writing. A Buffer class library in our case study is a part of the java.nio.* packages JDK 1.5. Depending on the purpose, programmers need buffers with specific features, for example, buffers holding elements of different types such as integer, float or character, read-only or writable buffers, and so on. In particular, Java buffer classes are characterized by the following features:

buffer data element types: byte, char, int, float double, long, short

memory allocation scheme: Direct, Non-direct (Heap)

byte ordering: Little_endian, Big_endian, Native, Non-native.

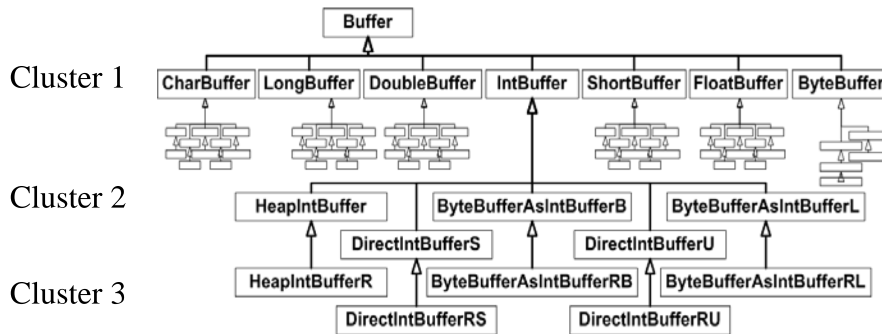

Figure 4. A fragment of the Buffer library

Figure 4 shows some of the buffer classes covered in our study. A class name reflects a specific combination of features a given class implements. For example, class HeapIntBuffer implements a buffer of type integer (Int) with memory allocation scheme on the Heap. A generic form of a class name is a template: [MS][T]Buffer[AM][BO], where MS – memory scheme: Heap or Direct; T – type: Byte, Char, Int, Double, Float, Long, or Short; AM – access mode: W – writable (default) or R - read-only; BO – byte ordering: S – non-native or U – native; B – BigEndian or L – LittleEndian. All the classes whose names do not include 'R', by default are writable ('W'). VB – View Buffer is yet another feature that allows us to interpret byte buffer as Char, Int, Double, Float, Long, or Short (not shown in Figure 4).

Class sub-hierarchies for classes CharBuffer, LongBuffer, DoubleBuffer, ShortBuffer, FloatBuffer, are analogous to sub-hierarchies for classes and IntBuffer, so for the sake of brevity we do not depict them in Figure 4. A class sub-hierarchy under ByteBuffer differs slightly from classes under other buffer element types which we explain later.

Each legal feature combination – that is, a combination of features that a programmer may want - yields a class. As most feature combinations are legal, not surprisingly the number of classes explodes. This phenomenon was observed in other class libraries in early 1990's, and was termed *feature combinatorics* problem [8][11].

All the buffer classes are derived from the top abstract class Buffer. Below class Buffer, we see a Cluster 1 of seven classes [T]Buffer that differ in buffer element data types T, where T stands for Char, Long, Double and other element types. A programmer can directly use only [T]Buffer classes. Therefore, these classes contain many methods providing access to functionalities implemented in the classes below them.

Classes in Cluster 2, in addition to various element types, implement two memory allocation schemes and two byte orderings, in various combinations.

The direct memory allocation scheme allocates a contiguous memory block for a buffer and uses native access methods to read and write buffer elements, using a *native or non-native byte ordering* scheme. In the non-direct memory allocation scheme, a buffer is accessed through Java array accessor methods.

For a variety of historical reasons, different CPU architectures use different native *byte ordering* conventions. For example, Intel microprocessors put the least significant byte into the lowest memory address (which is called Little_Endian ordering), while Sun UltraSPARC processors put the most significant byte first (which is called Big_Endian ordering). Byte ordering matters for buffers whose elements consist of multiple bytes, that is all the element types but byte.

When using the direct memory allocation scheme, we must know if buffer elements are stored using native or non-native byte ordering. When combining memory allocation and byte ordering features, we obtain twenty classes, as shown in Cluster 2 in Figure 4. For each [T]Buffer class, there is a Heap[T]Buffer counterpart class in Cluster 2 that implements the non-direct memory allocation scheme for that buffer. For each [T]Buffer class with the exception ByteBuffer, there are also two classes implementing direct memory allocation scheme with native (suffix 'S') and non-native (suffix 'U') byte ordering. Their names are Direct[T]BufferS and Direct[T]BufferU, respectively. We have only one class DirectByteBuffer, as byte ordering does not matter for byte buffers. There are yet other classes in Cluster 2, but they are not important for our discussion here.

Classes whose names do not include letter 'R', by default represent writable buffers. In particular, all the buffers in Clusters 1 and 2 are writable. Twenty classes in Cluster 3 implement read-only variants of buffers, shown by the letter 'R' in the class name.

9.5.2 Analysis method and experiment summary

The experiment involved 74 classes that contained 6,719 LOC (physical lines of code, without blanks or comments). At the time of the experiment, we did not have an automated clone detector. The small size and transparent design of the library made it possible to analyze classes manually in around four-person-days. In software reuse, analysis of similarities and differences that aims at identifying reuse opportunities is called domain analysis [17][53]. Our analysis combined bottom-up examination of similarities and differences in the existing family of buffer classes, with elements of top down domain analysis.

We gained general understanding of buffer classes first. Based on that we hypothesized which classes were likely to be similar to each other, and therefore could be candidates for structural clones. To validate if our hypothesis was true, we further analyzed simple clones in each group of such classes – similar class methods and attribute declaration sections.

Classes that differ in only one feature we call *peer classes*. For example, classes IntBuffer, CharBuffer and other classes in Cluster 1 form a group of peer classes. Another example is a group of Heap[T]Buffer classes. Our analysis often focused on various groups of peer classes to gain insights a global and detailed impact of a specific feature on buffer classes.

Classes [T]Buffer (Cluster 1) differ only in the type of a buffer element and were the most obvious candidates for structural clones. there were classes in Cluster 1. Yet other groups of classes that we thought could be similar are listed below.

1. [**T**]Buffer: 7 classes at Cluster 1 that differ in buffer element type, **T**: Byte, Char, Int, Double, Float, Long, Short
2. Heap[**T**]Buffer: 7 classes at Cluster 2, that differ in buffer element type, **T**
3. Heap[**T**]BufferR: 7 read-only classes at Cluster 3

4. Direct[**T**]Buffer[**S|U**]: 13 classes at Cluster 2 for combinations of buffer element type, **T,** with byte orderings**:** **S** – non-native or **U** – native byte ordering (notice that byte ordering is not relevant to buffer element type ‘byte’)
5. Direct[**T**]BufferR[**S|U**]: 13 read-only classes at Cluster 3 for combinations of parameters **T**, **S** and **U**, as above
6. ByteBufferAs[**T**]Buffer[**B|L**]: 12 classes at Cluster 2 for combinations of buffer element type, **T,** with byte orderings: **B** – Big_Endian or **L** – Little_Endian
7. ByteBufferAs[**T**]BufferR[**B|L**]: 12 read-only classes at Cluster 3 (not shown in Figure 4) for combinations of parameters **T**, **B** and **L**, as above.

Examination of methods and attribute declarations in classes within each of the above groups confirmed that indeed there were many similarities among classes. But there were differences, too:

1. the same attributes and/or method signatures recurred with different data type parameters,
2. fragments of constructor/method implementation recurred with different types, keywords, operators, or minor editing changes (additions or deletions),
3. some methods with the same signature had different implementation in classes under the common parent class,
4. certain classes in a group of otherwise similar classes, had some extra methods and/or attributes as compared to other classes in that group,
5. finally, certain classes in a group of otherwise similar classes, differed in details of the ‘implements’ clause.

Within each of the seven groups of similar classes, we identified unique, and similar methods and sometimes yet smaller code fragments (e.g., fragments of method implementation or attribute declaration sections). In analysis below, term *code fragment* means any unit of similarity below the class level, such as a class method, constructor, declaration section or a fragment of method/ constructor implementation. A *simple clone* means a code fragment that recurs in the same or similar form in a number of places in class definitions. We paid our attention only to similar code fragments that played a specific role in the Buffer domain or in class construction, and whose noticing could enhance program understanding, and help in eventual modifications.

9.5.3 Cluster 1: Analysis of [T]Buffer classes

In seven [T]Buffer classes (Cluster 1), we found 30 simple clones, 28 of which recur in all the seven classes. For example, simple clones include:

- two attribute definition sections recurring in the same form and one attribute definition section recurring with slight change,
- two constructor definitions recurring with slight change,
- four method definitions recurring in the same form, and
- 19 method definitions recurring with slight changes.

Five classes, namely IntBuffer, FloatBuffer, LongBuffer, ShortBuffer and DoubleBuffer differ in type parameter and otherwise are identical.

Class ByteBuffer has two unique attribute declaration sections and 35 unique methods. Class CharBuffer has 13 unique methods.

Declarations of classes ByteBuffer and CharBuffer (shown in Figure 5 and Figure 6, respectively) differ only in the *implements* clause.

```
public abstract class ByteBuffer
    extends Buffer
    implements Comparable <ByteBuffer>
{
```

Figure 5. Declaration of class ByteBuffer

```
public abstract class CharBuffer
  extends Buffer
  implements Comparable<ByteBuffer>,Appendable,CharSequence,Readable
{
```

Figure 6. Declaration of class CharBuffer

The reader should refer to relevant entries in Table 1 for statistics of similarities in classes at Cluster 1.

9.5.4 Cluster 2: Analysis of classes Heap[T]Buffer and Direct[T]Buffer[S|U]

Classes Heap[T]Buffer and Direct[T]Buffer[S|U] address memory allocation schemes (Heap or Direct), and byte ordering (S or U), in addition to the buffer element type (T). This new combination of features results in two new sub-classes of ByteBuffer, and three new sub-classes for each of the remaining classes in Cluster 1. Though the impact of various features visibly affects class implementation, there is still remarkable similarity among classes in Cluster 2.

We examine similarities in seven Heap[T]Buffer classes first. 18 simple clones recur in more than one class in the group, among which 17 clones recur in all the seven classes.

Five classes HeapIntBuffer, HeapFloatBuffer, HeapLongBuffer, HeapShortBuffer and HeapDoubleBuffer differ in type parameter and otherwise are identical.

Class HeapByteBuffer has 32 unique methods. Class CharBuffer has two unique methods.

For example, method **put()** recurs in seven Heap[T]Buffer classes with small changes. Figure 7 and Figure 8 show method **put()** in classes HeapByteBuffer and HeapCharBuffer, respectively. Analogical methods **put()** recur in each of the remaining classes in this group.

```
//Writes the given byte into this buffer at the current position.
public ByteBuffer put(byte x) {
  hb[ix(nextPutIndex())] = x;
  return this;
}
```

Figure 7. Method **put()** in HeapByteBuffer class

```
//Writes the given character into this buffer at the current position.
public CharBuffer put(char x) {
    hb[ix(nextPutIndex())] = x;
    return this;
}
```

Figure 8. Method **put()** in HeapCharBuffer class

12 classes in the Direct[T]Buffer[S|U] group implement the direct memory allocation scheme. Classes in this group display much similarity. 23 simple clones recur in either the same form or with slight changes in more than one class in the group, among which 21 fragments recur in all the 12 classes.

Five classes DirectIntBufferS, DirectFloatBufferS, DirectLongBufferS, DirectShortBufferS and DirectDoubleBuffersS differ in type parameter and otherwise are identical. Five classes DirectIntBufferU, DirectFloatBuffeU, DirectLongBuffeU, DirectShortBuffeU and DirectDoubleBuffeU – likewise. As before, these two sub-groups of classes have no unique code fragments, that is they are solely built of simple clones.

Class DirectByteBuffer has 44 unique methods, 3 unique constructors and one unique attribute declaration section. Class DirectCharBufferS shares with DirectCharBufferU two cloned methods that do not appear in other classes in the group.

For example, method **slice()** is cloned in all the Direct[T]Buffer[S|U] classes with various combination of values highlighted in bold. In Figure 9, we see method **slice()** from class DirectByteBuffer**.**

```
/*Creates a new byte buffer containing a shared
subsequence of this buffer's content. */
 public ByteBuffer slice() {
    int pos = this.position();
    int lim = this.limit();
    assert (pos <= lim);
    int rem = (pos <= lim ? lim - pos : 0);
    int off = (pos << 0);
    return new DirectByteBuffer(this, -1, 0, rem, rem, off);
 }
```

Figure 9. Method **slice()** recurring with small changes in Direct[T]Buffer[S|U] classes

Method **order()** is cloned in 12 Direct[T]Buffer[S|U] classes, all but DirectByteBuffer. Figure 10 shows implementation of method **order()** in six classes Direct[T]BufferU and Figure 11 shows implementation of method **order()** in six classes Direct[T]BufferS. Both implementations differ very little.

```
public ByteOrder order() {
 return ((ByteOrder.nativeOrder() != ByteOrder.BIG_ENDIAN)
      ? ByteOrder.LITTLE_ENDIAN : ByteOrder.BIG_ENDIAN);
 }
```

Figure 10. Method order() in six Direct[T]BufferU classes

```
public ByteOrder order() {
 return ((ByteOrder.nativeOrder() == ByteOrder.BIG_ENDIAN)
      ? ByteOrder.LITTLE_ENDIAN : ByteOrder.BIG_ENDIAN);
 }
```

Figure 11. Method order() in six Direct[T]BufferS classes

We did not find interesting similarities among groups Heap[T]Buffer and Direct[T]Buffer[S|U]. This is not surprising as most of the methods in those classes deal with accessing buffer elements and highly depend on the specific memory allocation scheme.

We would like to turn the reader's attention to a striking symmetry in similarity situations in all the above class groups.

The reader should refer to relevant entries in Table 1 for statistic of similarities in classes in Cluster 2.

9.5.5 Cluster 3: Analysis of read-only classes

In some applications, we need immutable, read-only buffers. This new feature brings in 20 new classes, namely seven Heap[T]BufferR classes and 13 Direct[T]BufferR[S|U] shown in Cluster 3 in Figure 4.

We examine the similarities in seven Heap[T]BufferR classes first. The results will not surprise the reader: We see 13 simple clones, of which 12 recur in all the seven classes.

Five classes Heap[T]BufferR, where T is Int, Float, Long, Short or Double, differ in type parameter and otherwise are identical.

Class HeapByteBufferR has 20 unique methods. Class CharBufferR has two unique methods.

13 classes in the Direct[T]BufferR[S|U] group display much similarity. There are 13 simple clones, of which 11 recur in all the 13 classes.

Five classes DirectIntBufferRS, DirectFloatBufferRS, DirectLongBufferRS, DirectShortBufferRS and DirectDoubleBufferRS differ in type parameter and otherwise are identical. Five classes DirectIntBufferRU, DirectFloatBuffeRU, DirectLongBuffeRU, DirectShortBuffeRU and DirectDoubleBuffeRU – likewise.

Class DirectByteBufferR has 26 unique methods, and 2 unique constructors. Class DirectCharBufferR shares with DirectCharBufferRU two cloned methods that do not appear in other classes in the group. These two classes have no unique fragments.

We note that this time, simple clones also exist across horizontal classes such as HeapByteBufferR and HeapCharBufferR. There are many simple clones among the parent classes and subclasses, as well. Table 1 shows some examples.

Table 1. Similar methods across parent classes and their subclasses

Parent class	subclass	No. of similar methods	
		S	C
HeapByteBuffer	**HeapByteBufferR**	**1**	**8**
HeapCharBuffer	**HeapCharBufferR**	**2**	**3**
HeapIntBuffer	**HeapIntBufferR**	**1**	**2**
……	**…**	**…**	**…**
DirectByteBuffer	**DirectByteBufferR**	**3**	**8**
DirectCharBufferU	**DirectCharBufferRU**	**2**	**3**
DirectCharBufferS	**DirectCharBufferRS**	**2**	**3**
DirectIntBufferU	**DirectIntBufferRU**	**1**	**2**
DirectIntBufferS	**DirectIntBufferRS**	**1**	**2**
……	……	……	……
……	……	……	……

S: same content recurring in parent class and subclass
C: small change when recurring in parent class and subclass

9.5.6 Analysis of the remaining buffer classes

Here, we summarize results of similarity analysis in the remaining classes in the Buffer library. Two groups of classes, namely ByteBufferAs[T]Buffer[B|L] and ByteBufferAs[T]BufferR[B|L] are derived from Cluster 1 classes. There are six classes in each group for types Char, Int, Float, Long, Short or Double For example, subclasses ByteBufferAsCharBufferB and ByteBufferAsCharBufferL allow us to view classes ByteBuffer as CharBuffer, using Big_Endian and Little_Endian ordering, respectively. Each of the Int, Double, Float, Long and Short buffer classes has analogical two subclasses.

ByteBufferAs[T]Buffer[B|L] and ByteBufferAs[T]BufferR[B|L]

Finally, a read-only class is derived from each of the ByteBufferAs[T]Buffer[B|L] and ByteBufferAs[T]BufferR[B|L] classes, yielding 12 more classes.

Each of the ByteBufferAs[T]Buffer[B|L] and ByteBufferAs[T]BufferR[B|L] class groups displays remarkable similarities. There are 19 simple clones recurring in all six ByteBufferAs[T]Buffer[B|L] classes. In addition, classes ByteBufferAsCharBufferB and ByteBufferAsCharBufferL have two unique methods.

There are 11 simple clones recurring in all six ByteBufferAs[T]BufferR[B|L] classes. In addition, classes ByteBufferAsCharBufferRB and ByteBufferAsCharBufferRL have two unique methods.

Method **ix**(int i) is cloned in ByteBufferAs[T]Buffer[B|L] classes with different values of the integer value shown in bold (Figure 12).

```
protected int ix(int i) {
  return (i << 1) + offset;
}
```

Figure 12. Method **ix**(int i)

Method **order()** shown in Figure 13 is cloned in classes ByteBufferAs[T]BufferB, while method **order()** shown in Figure 14 is cloned in classes ByteBufferAs[T]BufferL.

```
public ByteOrder order() {
  return ByteOrder.BIG_ENDIAN;
}
```

Figure 13. Method **order**() in classes ByteBufferAs*BufferB

```
public ByteOrder order() {
  return ByteOrder.LITTLE_ENDIAN;
}
```

Figure 14. Method **order**() in classes ByteBufferAs*BufferL

The reader should refer to relevant entries in Table 2 or statistic of similarities in classes ByteBufferAs[T]Buffer[B|L] and ByteBufferAs[T]BufferR[B|L].

9.5.7 Why Do Similar Classes Arise?

Usability is an important design goal for class libraries. To make buffer classes easy to use, designers decided to reveal to programmers only the top eight classes (Figure 4). Functionalities related to lower-level concrete classes can be accessed via special methods provided in the top eight classes. Conceptual clarity and high performance are yet other important design goals for the Buffer library. For conceptual clarity, designers did not multiply classes beyond what was absolutely needed. We see almost one-to-one mapping between legal feature combinations and buffer classes. While in many situations one could introduce a new abstract class or a suitable design pattern to avoid repetitions, such a solution would compromise the above design goals.

Most of the similarity patterns in buffer classes are the result of feature combinations. As features cannot be implemented independently of each other in separate implementation units (e.g., class methods), program structures affected by feature combinations must appear in many variant forms, depending on the context. Whenever such program structure cannot be parameterized to unify the variant forms, and placed in some upper-level class for reuse via inheritance, similar code structures must be replicated.

To observe the impact of feature combinations on buffer classes, we compared classes in various peer class groups (that is, classes differing in one feature only). For example, we compared classes that differed in element type (e.g., DirectCharBufferS and DirectIntBufferS), in byte ordering (e.g., DirectIntBufferS and DirectIntBufferU) and in access mode (e.g., DirectIntBufferS and DirectIntBufferRS).

A typical situation that leads to cloning is when some classes derived from the same parent need a certain method (or data), and other classes derived from A do not need that method. One solution could be to place such a method in a new parent class created for the purpose abstracting a common method. However, creating many such classes would complicate the class hierarchy and hinder performance. Another solution could be to place such a method in the parent class. But then any attempt to invoke a method for a class that does not need the

method would result in a runtime error. To avoid such an error-prone solution, we would have to write an extra code to disable the method in the classes that do not need it.

In yet other situations, a certain method may be needed in all the classes derived from the same parent, but in some of those classes the method may require different parameters, return type or implementation than in other classes. Furthermore, implementations of such a method in different classes may refer to non-local attributes defined in the context of different classes. In the above cases, designers often choose to replicate a method in each class that needs it. Method **hasArray()** shown in Figure 15 illustrates a simple yet interesting case. This method is cloned in each of the seven [T]Buffer classes. Method **hasArray()** cannot be implemented in the parent class Buffer, as variable **hb** must be declared with a different type in each of the seven classes. For example, in class ByteBuffer the type of variable **hb** is **byte** and in class IntBuffer, it is **int**.

```
/* Tells whether or not this buffer is backed by
 an accessible byte array. */
public final boolean hasArray() {
return (hb != null) && !isReadOnly; }
```

Figure 15. Recurring method **hasArray**()

We found some instances of methods defined in a parent class and then repeated in children classes. For example, the same method **_get()** is defined in classes HeapByteBuffer and HeapByteBufferR, and the same method **order()** is defined in classes HeapCharBuffer and HeapCharBufferR, HeapIntBuffer and HeapIntBufferR, etc. Those clones may exist for performance reasons or due to deficiency of the design.

Many repetitions arise due to the inability to specify small variations in otherwise identical code fragments. For example, some attributes, methods or even classes may differ only in data types, constants, keywords or operators. Classes or methods that differ in type parameters are candidates for generics. JDK 1.5 supports generics, based on the earlier proposal JSR-14 [12]. However, generics have not been applied to unify similarity patterns described in our study. As generics are an important language-level feature to address problems of unifying similarities, we have done a detailed analysis of Java generics in the context of the Buffer library. Groups of classes that differ only in data type are obvious candidates for generics. There are three such groups comprising 21 classes, namely [T]Buffer, Heap[T]Buffer and Heap[T]BufferR. In each of these groups, classes corresponding to Byte and Char types differ in non-type parameters and are not generics-friendly. This leaves us with 15 generics-friendly classes whose unification with three generics eliminates 27% of code. There is, however, one problem with this solution. In Java, generic types cannot be primitive types such as int or char. This is a serious limitation, as one has to create corresponding wrapper classes just for the purpose of parameterization. Wrapper classes introduce extra complexity and hamper performance. Application of generics to 15 buffer classes is subject to this limitation.

Many simple clones differ in parameters representing constants, keywords or algorithmic elements rather than data types. For example, method **slice()** is cloned 13 times in all the Direct[T]Buffer[S|U] classes with small changes highlighted in bold in Figure 9. This happens when the impact of various features interacts in code fragments, affecting data type

names, constant values or details of algorithms. Generics are not meant to unify this kind of differences in classes. We found yet other cases of similar but generics-unfriendly classes and we refer the reader to further details of the generics solution (including code) to our case studies at the XVCL Web site [69]. In summary, it is unlikely that generics can help unify similarities patterns in software that exhibits high degree of feature combinations such as the Buffer library.

It is interesting to note that small variations appear in otherwise the same code fragments across classes at the same level of inheritance hierarchy, as well as in classes at different levels of inheritance hierarchy. Programming languages do not have a proper mechanism to handle such variations at an adequate (that is a sufficiently small) granularity level. Therefore, the impact of a small variation on a program may not be proportional to the size of the variation.

In summary, we observed substantial similarities within each of the seven groups classes, listed in Section 9.5.2. Table 2 shows the statistics of class methods/constructors, as well as smaller class building blocks (e.g., attribute declaration sections or method implementation fragments), cloned in many classes.. 'Clusters' in the first column of Table 2 refer to classes shown in Figure 4. As we will see in the next Chapter, by unifying similarities with generic mixed-strategy solutions, we can collapse code size by 68%, reducing program's conceptual complexity by rates proportional to the code size reduction.

Copyright (C) 2007 Stan Jarzabek

Table 2. Buffer library statistics

Classes	Fragments							LOC
	Recurring fragments						Unique	
	times	2	6	7	12	13		
Cluster 1 7 classes [T]Buffer	same form		2	6			50	3720
	small changes			22				
Cluster 2 7 classes Heap[T]Buffer	same form		1	3			34	914
	small changes			14				
Cluster 2 13 classes Direct[T]Buffer[S|U]	same form	1				8	48	2428
	small changes	1			1	13		
Cluster 3 7 classes Heap[T]BufferR	same form		1	1			22	521
	small changes			11				
Cluster 3 13 classes Direct[T]BufferR[S|U]	same form	1				2	28	979
	small changes	1			1	9		
subtotal for 47 classes	same form	2	4	10		10	182	8562
	small changes	2		47	2	22		
other classes in Cluster 2 (12 classes)	same form	1			4		0	1014
	small changes	1			12			
other classes in Cluster 3 (12 classes)	same form	1			2		0	556
	small changes	1			9			
subtotal for other classes	same form	2			6		0	1570
	small changes	2			21			
total	same form	4	4	10	6	10	182	10132
	small changes	4		47	23	22		

9.5.8 Closing comments on the Buffer library example

The Buffer library is a very special kind of a program. In other libraries, classes may serve distinct purposes, displaying less similarity than we observed in buffer classes. Application programs have a more monolithic structure than buffer classes, involve multi-tier architectures with user interface, business logic and database layers, and today often rely on functionally provided by underlying component platforms (such as .NET™ or J2EE™). While this is all true, we believe the Buffer library allows us to observe in a pure and distilled form some common roots of software similarities and reasons why they may be difficult to unify with conventional programming techniques.

In particular, in other experiments, some of which we review in sections below, we observed that feature combination was one of the major forces triggering similarity patterns that caused spreading similar program structures across programs under study. Ad hoc, irregular nature of

variations among similar program structures made it difficult to unify the differences among similar program structures with conventional generic design techniques. This difficulty was further magnified by the fact that programs had to meet other important design goals, and conventional generic design solutions, even if existed, would require designers to compromise these goals, therefore were not used.

9.6 Similarity patterns and clones in STL

The Standard Template Library (STL) [61] is a hallmark of powerful and elegant generic design. Due to light integration of templates with the C++ language core, template parameters are less restrictive than parameters of Java generics. Unlike Java generics, C++ templates also allow constants and primitive types to be passed as parameters. Not only does STL use the most advanced template features and design solutions (e.g., iterators), but it is also widely accepted in the research and industrial communities as a prime example of the generic programming methodology. Therefore, STL provides a good opportunity to strengthen observations made in the Buffer Library case study.

STL consists of containers (such as stack, set or map), algorithms (such as sort or search), iterators, function objects and adaptors. Algorithms and data structures commonly used in computer science are provided in the STL. There are plenty of algorithms that need to work with many different data structures. Without generic containers and algorithms, the STL's size and complexity would be enormous. Such simple-minded solution would unwisely ignore similarity among data structures, and also among algorithms applied to different data structures, which offers endless reuse opportunities. Redundant code sparking from unexploited similarities would contribute much to the STL's size and complexity, hindering its evolution. All the components of STL have been heavily parameterized so that a single implementation of the container template can be used for all types of contained elements. In STL, generic solutions are mainly facilitated by templates and iterators.

Generic containers form the root of the STL. We analyzed associative containers - variable-sized containers that support efficient retrieval of elements based on keys. These are either sequence containers or associative containers. In sequence containers, all members are arranged in some order. In associative containers, the elements are accessed by some key and are not necessarily arranged in any order.

We selected associative containers for further detailed manual analysis because of its high level of cloning. An associative container is a variable-sized container that supports efficient retrieval of its elements based on keys. Containers are characterized by the following features:

ordering: hashed or sorted. The elements of a 'hashed' associative container are not guaranteed to be in any meaningful order. 'Sorted' associative containers use an ordering relation on their keys.

key type: simple or pair: In a 'simple' associative container, elements are their own keys. A 'pair' associative container associates a key with some other object.

uniqueness: in a 'unique' associative container each key in the container is unique, which need not be the case in a 'Multiple' associative container.

Any legal combination of the above features yields a unique class template. For example, the container 'set' represents an associative container where Storage=sorted, Uniqueness=unique,

and Key type=simple. There is much similarity across associative containers independently of the specific features they implement which leads to clones.

We started the experiment by running automated clone detectors CCFinder [32] and Clone Miner [6] on STL. We further analyzed manually STL regions that showed high cloning rates, to identify possible design-level similarities. We found that containers displayed much similarity and code repetition. Four 'sorted' associative containers and four 'hashed' associative containers contained 57% of cloned code. Stack and queue contained 37% of cloned code. Algorithms set union, intersection, difference, and symmetric difference (along with their overloaded versions) 52% of cloned code.

There were many non-type-parametric differences among associative container group of templates. For example, certain otherwise similar methods, differed in operators or algorithmic details. Figure 16 shows a simple example of cloned templates.

```
template <class _Key, class _Compare, class _Alloc>
inline bool operator== (
    const set<_Key,_Compare,_Alloc>& __x,
     const set<_Key,_Compare,_Alloc>& __y) {
  return __x._M_t ==__y._M_t;
}
```

```
template <class _Key, class _Compare, class _Alloc>
inline bool operator< (
    const set<_Key,_Compare,_Alloc>& __x,
     const set<_Key,_Compare,_Alloc>& __y) {
  return __x._M_t <__y._M_t;
}
```

Figure 16. Cloned templates differing in operators

Other replicated templates differed in non-parametric ways (e.g., swapped lines of code or inserted lines of code).While it is possible to treat many types of non-parametric differences using sophisticated forms of C++ template meta-programming, often the resulting code becomes "cluttered and messy" [19]. We did not spot such solutions in STL, and believe their practical value needs to be further investigated.

The reader may find full details of the STL case study in [5].

9.7 Similarity patterns in application programs

9.7.1 Domain Entity Management Subsystem

We now discuss a similarity situation in a Domain Entity Management Subsystem (DEMS) written in C#. DEMS was a part of a command and control system developed by the author's industry collaborator ST Electronics Pte Ltd (STEE). The system involved domain entities such as *Task*, *User* or *Resource*. For each entity, the system implemented up to 10 operations, such as *Create*, *Update*, *View*, *Delete*, *Find* or *Copy*.

DEMS was to serve a range of command-and-control applications, so it was meant to be easily adaptable and extensible. The intention was to make DEMS an integral part of the product line architecture for those applications. DEMS comprised 18,823 LOC of C# code, with 117 classes covering GUI, service and database layers.

The project started by porting an existing DEMS written in Visual Basic into C#. At the same time, new variant features were considered to enhance reusability of the DEMS in a wider range of applications. In the process of re-writing the system, STEE analyzed similarity patterns to identify potential reuse opportunities. C# and .NET mechanisms were applied whenever possible to unify observed similarity patterns.. The option of applying generics was taken into account based on the generics proposed for C# [36]. Similarities that could not be unified using conventional mechanisms, and that were leading either to repetitions or to inefficiencies during customization of the DEMS for reuse in target applications, were noted and documented. One of such similarity pattern is shown in Figure 17.

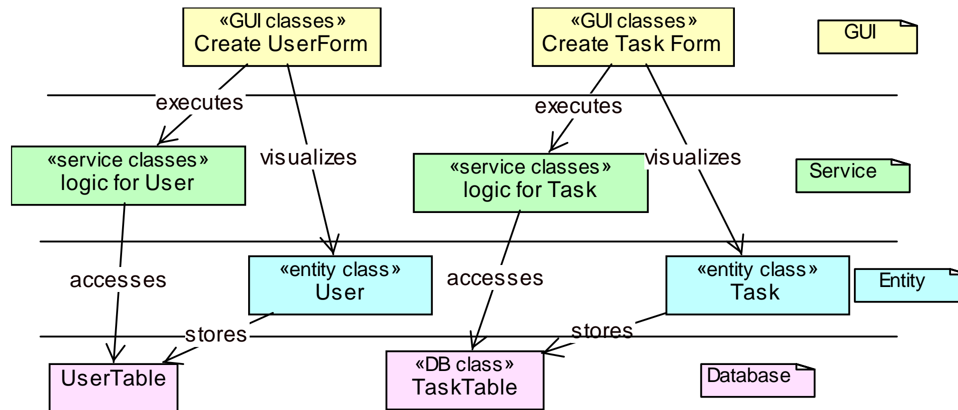

Figure 17. A recurring pattern of components

The design of each operation, say *CreateUser*, involved a pattern of collaborating classes from GUI, service and database layers (Figure 17). The design of operation *Create* for other entities, e.g., *CreateTask,* was quite similar. Each box in Figure 17 represents a number of classes, with much similarity across classes implementing similar concepts for all the domain entities, in their respective operations.

However, there were also differences across classes for various operations, caused by different meaning of domain entities: For example, operation *CreateTask* required different types of data entry and data validation than *CreateUser*.

Despite striking similarities, it was not possible to design a generic solution for groups of similar operations. To implement generic operations, one would have to first unify groups of classes in GUI, Service, Entity and DB layers such as Create[entity-type]Form, Update[entity-type]Form, etc. However, the nature of variations across operations for different entity types was such that neither inheritance nor type parameters (such as in generics proposed for C#) could be used to implement operations in a generic way. Therefore, implementation of operations for different entities was replicated many times, with required variations. Structural clones, two of which we depicted in Figure 17, that sparked from this similarity pattern contained an estimated 68% of DEMS code.

9.7.2 Similarity patterns in Web Portals on J2EE platform

The goal of this study was to evaluate J2EE™ (Java™ 2 Platform, Enterprise Edition) as a platform for Web Portal (WP) product line development. An earlier project by STEE revealed 60% similarity rates in Active Server Pages (ASP) WPs, and in certain areas as high

as 90% [52]. It was interesting to know the effectiveness of J2EE mechanisms in unifying various kinds of similarity patterns.

J2EE provides standardized architecture and supports reuse of common services via portlet technology. Unlike ASP, J2EE supports inheritance, generics and other OO features via Java 1.5. We worked with a portal developed by STEE, called CAP-WP. CAP-WP supported collaborative work and included 14 modules such as Staff, Project or Task (Figure 18).

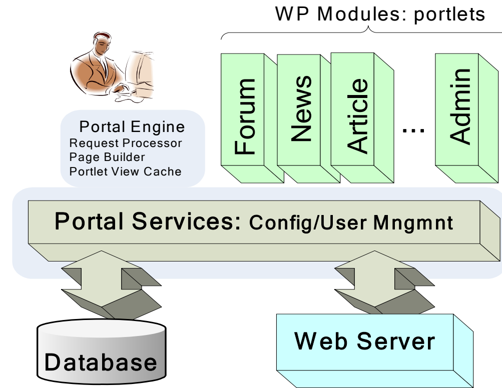

Figure 18. CAP-WP modules (portlets) and architecture

Figure 19 shows patterns of classes in CAP-WP that formed structural clones. CAP-WP is composed of a set of modules where each module is a portlet. There are 25 modules in CAP-WP. Each module contains all the code required for implementing a particular core feature in CAP-WP (e.g., Forum, News, etc.). Each module follows standard design implied by the 5-tier J2EE architecture model.

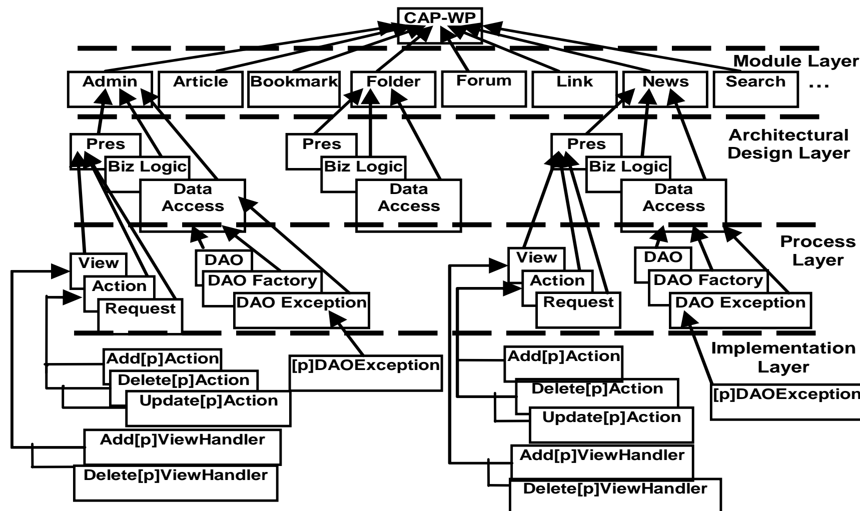

Figure 19. Similarities in CAP-WP

We studied similarity patterns in the presentation layer (Pres), the business logic layer (Biz Logic) and the data access layer. Each layer of the architecture is subdivided into the various processes or functions (i.e., viewing of data) within that layer. Finally, each process is

implemented by classes (*Implementation Layer*) of Figure 19. Parameter [p] in class names represents the module name (e.g., Article, Forum, or News).

We studied both intra- and intra-module similarities in CAP-WP design. The nature and degree of intra- and inter-module similarities varied. Within modules, we found 75% of code contained in exact clones, and 20% of code contained in similar clones (leaving only 5% of code unique). Analysis across modules, revealed design-level similarities, with 40% of code contained in structural clones. Both intra- and inter-module similarities were important for clarity of the design, however they could not be unified with generic design solutions expressed by J2EE mechanisms.

Inter-Portlet Similarities:

In particular, there was much similarity among various Implementation Layer classes implementing the same concepts in different portal modules (i.e., for different values of parameter [p]). For example, [p]ViewHandler classes for all 25 modules are very similar to each other. They all contain code shown in Figure 20, with variations indicated in underlined font, and slight changes in the contents of the method render. Such similar classes were common across the different CAP-WP modules.

```
class AddContentTypeViewHandler implements ViewHandler {
    private PortletConfig _config;
    public void init (PortletConfig config) {
        _config = config;
    }
    public void render (PortletRequest req, PortletResponse resp,
        AppResult [] result) throws ViewException {
                    ......
    }
    public void renderException (PortletRequest req, PortletResponse resp,
        PortletException ex) throws ViewException {

        PortletContext context = _config.getPortletContext ();
        PortletRequestDispatcher disp = context.getNamedDispatcher ("ADMIN_ERROR");
        req.setAttribute (RequestKey.EXCEPTION, ex);
        req.setAttribute (RequestKey.ERROR_PAGE_TITLE, "Add Content Type Fail");
        try {
                        disp.include (req, resp);
        }
        catch (Exception e) {
                    Log.error ("AddContentTypeViewHandler.renderException - Unable to
            render error page: " + e.getMessage ());
                    throw new ViewException
            (ErrorCode.ERROR_CREATE_CONTENT_TYPE, e);
                }
            }
```

Figure 20. View Handlers

Intra-Portlet Similarities:

Some of the CAP-WP modules have more than one view handler. For example, the Admin module has the following view handler classes: *AddContentTypeViewHandler*, *UpdateContentTypeViewHandler*, *DeleteContentTypeViewHandler* and *ViewContentType-ViewHandler*. The structure of all these classes is similar to the one shown in Figure 20.

Also classes implementing Add and Update operations for a given module are very similar one to each other (for example, *AddQueryRequestHandler* and *UpdateQueryRequestHandler* for the Query Management module). Here the degree of similarity is even higher than in request handlers within the same module, for example *DeleteQueryRequestHandler*.

The reader may find full details of this project in [70].

9.7.3 Similarities in the Entity Bean data access layer

Our final example shows explosion of look-alike Entity Beans triggered by feature combinations. We addressed variant features affecting Entity Beans depicted in Figure 21. Domain Entity feature group represents database tables for different portal modules. Entity Beans receive query request from the presentation or business logic layers, connect to the specified database (DBMS feature group in Figure 21), by the specified connectivity mechanism (Connection), receive query results and pass them back to the caller in the required format (Input/Output Parameter). As any combination of variant features shown in Figure 21 could be implemented in Entity Beans in some application, the number of required Entity Beans was huge. While Entity Beans were similar across modules, there were differences among them, too. J2EE/EJB provided little support to abstract similarities among Entity Beans and to deal with the impact of variations in a generic way. Sometimes, we could implement "typical variant combinations" into components, allowing a developer to select variants needed in a WP using EJB mechanisms. However, using this method we covered only a small number of simple cases.

An alternative solution was to have a separate component version for each legal combination of features. But then we would end up with huge number of look-alike components. Both solutions were prohibitive to systematic reuse. Consequently, a new application was basically built with components (and component configurations) that already appeared in some modules. These components could be reused in building new applications. As such, they required much manual work during customization. These experiences are in line with problems reported in industrial product lines, where many variant features and feature dependencies lead to thousands of component versions, hindering reuse .

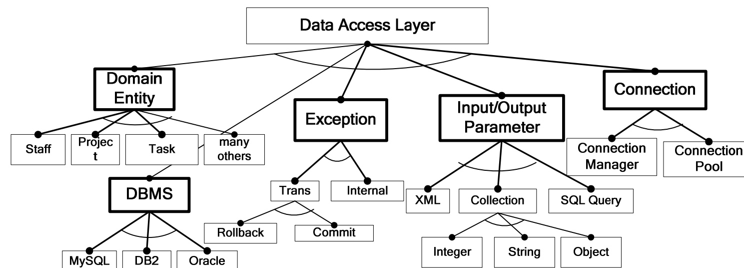

Figure 21. Variant features in data access layer

Copyright (C) 2007 Stan Jarzabek

Even though we did not create many look-alike components, Entity Beans still displayed much similarity that could not be unified by generic solutions. For example, many Entity Beans had similar interfaces and business logic related to handling tables. However, they differed in attributes and some other details. Unfortunately, we could not abstract commonalities among Entity Beans and parameterize them for reuse.

9.8 General implications

Just like we can't avoid the crosscutting impact of some of the computational aspects on modules [37][62], we can't avoid repetitions stemming from certain similarity patterns.

What are similarity patterns and what do they represent?

Some similarity patterns can be observed at the level of software requirements. Others are the result of recurring design solutions of similar structure, or can be induced by the underlying platform (e.g., by standardized architectures and patterns of J2EE or .NET). Design-level similarities, showing as patterns of classes or components, often represent domain-specific abstractions (e.g., operations on domain entities in DEMS, Figure 17). Generic structures unifying such similarities are particularly important as they describe how domain concepts are realized in the solution space. Such similarity patterns create opportunities for reuse within a given system, or even across similar systems. In the evolution context, instances of a generic structure in system releases show how the design and implementation of the same domain concept have been changing during system evolution.

Some similarity patterns do not have a corresponding functional abstraction suitable for its representation. Ad hoc, irregular nature of variations among similar program structures is another reason why it is difficult to unify the differences among similar program structures with conventional generic design techniques. Finally, software design is often constrained by multiple, sometimes competing, design goals, such as development cost, usability, simplicity, performance, or reliability. A generic solution to unify similarity patterns will only be applied if it is natural and synergistic with all the other design goals that matter.

Feature combinatorics problem is the main reason why similar program structures in many variant forms arise and spread through the buffer classes. We observe feature combination phenomenon within a single system (vide DEMS or a Web Portal examples) or class library (vide the Buffer library or STL). We observe feature combinations across a family of similar systems forming a product line [17]. Finally, we observe feature combination across system releases arising from software evolution.

In all the above situations, a cost-effective solution to containing the impact of feature combination should rely on some kind of a structure representing the underlying similarity pattern in a generic form.

Differences among similar program structures, an effect of combining features, tend to be chaotic, and rarely confined to type parameters. Programs are designed under the constraints of the design goals (such as usability or performance), and the symptoms of the *feature combinatorics* problem must be considered in the context of those constraints. Therefore, it becomes difficult to define effective generic solutions unifying similarity patterns without compromising other design goals, using conventional techniques.

While not all such recurring similar program structures are necessarily bad, most of them hinder program understanding and contribute much to high maintenance cost [10][22][32]. There are a number of reasons why cloned structures increase program complexity, as perceived by programmers, hindering program understanding: Not only do clones blow up the size of a program, but also increase the number of conceptual elements programmers must deal with and hinder important relationships among program elements that must be understood whenever program is to be changed. Whenever change happens to a cloned structure, a programmer must locate all its instances, analyze them in the context to see which clone instances are affected by change and how. In that sense, cloned structures – untapped program similarity patterns in general - increase the risk of update anomalies. Finally, design-level similarities recurring in multiple forms across a program, hinder the conceptual integrity of the design, which Brooks calls "the most important consideration in system design" [13].

More detailed argumentation is best done with a reference to a non-redundant mixed-strategy program representation. We continue discussion in the next Chapter.

9.9 Identifying similarity patterns and automated detection of clones

The objective of domain analysis is to identify similarity patterns based on top-down analysis of an application domain. We can identify similarities within a program, and across different programs in a domain. We further characterize differences among instances of similarity patterns. Feature diagrams are often used for that purpose [33]. Many similarity patterns become apparent only during architecture design, detailed design and implementation.

To find clones in large legacy systems, some level of automated analysis is essential. Different techniques have been proposed to automate finding clones in existing programs. Most of the techniques focus on simple clones. We can broadly categorize clone detection techniques based on the program representation and the matching technique. For program representation, the different options are plain text [22], tokens [2][32], abstract syntax trees [10], program dependence graphs [41][42], and software metrics [40]. The different matching techniques include suffix tree based token matching [32], text fingerprints matching [31], metrics value comparisons [40], abstract syntax trees comparisons [10], dynamic pattern matching [40] and neural networks [20].

CCFinder [32] is easily configurable to read input in different programming languages like C, C+, Java and COBOL. A suffix-tree matching algorithm is used for the discovery of clones. Some optimizations are applied to reduce the complexity of token matching algorithm. These optimizations include the alignment of token sequences to begin at tokens that mark the beginning of a statement like #, {, class, if, else etc. Considering only these tokens as leading tokens reduces the resulting suffix tree size to one-third. Another optimization is the removal of short repeated code segments from the input source like case statements in switch-case constructs and constant declarations. Very large files are truncated and "divide-and-conquer" strategy is applied to find duplicates in them.

Dup is another token based clone detection tool [22]. It finds identical clones as well as strictly parameterized clones, i.e., clones that differ only in the systematic substitution of one set of parameter values for another. These parameters can be identifiers, constants, field names of structures, and macro names but not keywords like *while* or *if*. The tool is text- and token-based and the granularity of clones is line-based where comments and white spaces are

not considered. The algorithm used in this tool is based on a data structure called parameterized suffix tree [3], which is a generalization of a suffix tree. Dup is developed in C and Lex and only C code can be processed through Dup.

For finding maximal parameterized matching, Dup's lexical analyzer generates a string of one non-parameter symbol and zero or more parameter symbols for each line. The parameter and their positions are recorded in this parameterized string. This string is encoded in such a way that the first occurrence of each parameter is replaced by zero and every later occurrence is replaced by the distance in the string since the previous occurrence. Non-parameter symbols are left unchanged. This tokenization scheme ensures the strict parameterization requirement of Dup. The clones detected by Dup cannot cross file boundaries but can cross function boundaries.

With the experience gained from the clone analysis of different software, it was observed that some short repeated segments of code turn up as multiple clone classes which are not very useful [32]. In Dup, these fragments of code had to be removed by hand. In CCFinder, the first two repetitions are reported and the rest are ignored. Clone Miner [6] does not report these short repetitions as clone classes, but rather reports them separately as repetitions.

Clone detection tools typically produce much information that needs to be further analyzed, with programmer involvement, to filter useful clones. By useful clones, we mean clones that represent some significant program elements or concepts that play a role in program understanding and maintenance. Gemini [65] is a visualization tool that produces graphical views of the cloning situation (such as clone scatter plot) to help a programmer find useful clones based on the CCFinder output. Kapser and Godfrey propose analysis through the sorting of clones into categories based on clone taxonomy and then displaying the clones in the form of an architectural landscape view [35]. This view is a graph where the edges are clone relations and the nodes are software artifacts such as files and methods. Reiger et al. use a duplication web to show the overview of cloning in a system. The nodes of the web represent files while the edges represent a clone relation between the files [55]. They also use a clone scatter plot for an overview of cloning. This plot is a variation of the standard scatter plot in which they attempt to resolve the scaling problem of the original by applying logarithmic scales to the lines of code metric in files.

The above mentioned techniques are based on simple clone detection only. They do not address design-level similarities in the form of structural clones Apparently, there is a gap between simple clone detection and useful design information, and visualization techniques may help us narrow down this gap only to some extent.

Clone Miner [6] applies a mixture of techniques to recover design-level similarities, so-called structural clones discussed in this Chapter. Clone Miner uses token-based techniques to find simple clones first. Then, Clone Miner applies data mining techniques to find configurations of simple clones that are likely to form higher-level similarity patterns. As other clone detection tools, Clone Miner produced much information not all of which represents useful design concepts. The follow up visualization, filtering and abstraction techniques, implemented into a Clone Analyzer, assist programmers in filtering useful structural clones from the Clone Miner output.

9.10 Conclusions of this Chapter

We discussed the problem of software similarities and the related problem of software clones. We traced the roots of the problem and argued about the general nature of the general nature of the problem's symptoms and causes. We discussed feature combination as a common trigger of software similarities. Software is affected by multiple features. Similarities arise from feature combinations that have to be implemented into software modules such as classes or components. However, given the overall design goals, in many cases, similarity patterns are difficult to avoid - refactored or otherwise unified, with conventional programming techniques. In some cases, such unification was conflicting with other design goals. In yet other situations, such unification was not possible at all given the programming technology used. We supported our argument with examples from class library, application programs and J2EE domains. While not all the clones are necessarily bad, most of them hinder program understanding and contribute much to high maintenance cost.

References

[1] ANSI and AJPO 1983. Military Standard: Ada Programming Language, ANSI/MIL-STD-1815A-1983, February 17, 1983

[2] Baker, B. S. "On finding duplication and near-duplication in large software systems," *Proc. 2nd Working Conference on Reverse Engineering*. 1995, pages 86-95.

[3] Baker, B. S. "Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance," *SIAM Journal of Computing*, October 1997.

[4] Bassett, P. *Framing software reuse - lessons from real world*, Yourdon Press, Prentice Hall, 1997

[5] Basit, H.A., Rajapakse, D.C., and Jarzabek, S. "Beyond Templates: a Study of Clones in the STL and Some General Implications," *Int. Conf. Software Engineering, ICSE'05,* St. Louis, USA, May 2005, pp. 451-459

[6] Basit, A.H. and Jarzabek, S. "Detecting Higher-level Similarity Patterns in Programs," *ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2005, Lisbon, pp. 156-165

[7] Batory, D. and O'Malley, S. 1992. "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Trans. on Software Engineering and Methodology*, Vol. 1, No. 4, Octobe 1992, pp. 355-398

[8] Batory, D., Singhai, V., Sirkin, M. and Thomas, J. "Scalable software libraries,", *ACM SIGSOFT'93: Symp. on the Foundations of Software Engineering*, Los Angeles, California, Dec. 1993, pp. 191-199

[9] Batory, D., Sarvela, J.N. and Rauschmayer, A. 2003 "Scaling Step-Wise Refinement," *Proc. Int. Conf. on Software Engineering, ICSE'03*, May 2003, Portland, Oregon, pp. 187-197

[10] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. "Clone detection using abstract syntax trees," *Proc. Int. Conf. on Software Maintenance* 1998, pp. 368–377

[11] Biggerstaff, T. "The library scaling problem and the limits of concrete component reuse," *3rd Int. Conf. on Software Reuse, ICSR'94*, 1994, pp. 102-109

[12] Braha, G., Cohen, N., Kemper, C. Marx, S., et al. JSR 14: Add Generic Types to the Java Programming Language, http://www.jcp.org/en/jsr/

[13] Brooks, P.B *The Mythical Man-Month*, Addison Wesley, 1995

[14] Burd, E., Munro, M., "Investigating the Maintenance Implications of the Replication of Code," *Proc. IEEE Intl. Conference on Software Maintenance (ICSM '97*), pp. 322-329.

[15] Burd, E., and Bailey, J., "Evaluating Clone Detection Tools for Use during Preventative Maintenance," *Second IEEE Intl. Workshop on Source Code Analysis and Manipulation (SCAM'02*), pp. 36-43

[16] Clarke, S., Harrison, W., Ossher, H. and Tarr, P. 1999 "Subject-oriented design: toward improved alignment of requirements, design, and code," Proc. OOPSLA'99, November 1999, Denver, pp. 325-339

[17] Clements, P. and Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, 2002

[18] Cordy, J. R., "Comprehending Reality: Practical Challenges to Software Maintenance Automation," *Proc. 11th IEEE Intl. Workshop on Program Comprehension, (IWPC 2003)*, pp. 196-206

[19] Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications,* Addison-Wesley, 2000
[20] Davey, N., Barson, P., Field, S., Frank, R., and Tansley, D. The development of a software clone detector. *International Journal of Applied Software Technology*, 1(3-4): 219-236, 1995.
[21] Deelstra, S., Sinnema, M. and Bosch, J. "Experiences in Software Product Families: Problems and Issues during Product Derivation," *Proc. Software Product Lines Conference, SPLC3,* Boston, Aug. 2004, LNCS 3154, Springer-Verlag, pp. 165-182
[22] Ducasse, S., Rieger, M. and Demeyer, S. 1999. "A language independent approach for detecting duplicated code," Int. Conference on Software Maintenance, ICSM'99, September 1999, Oxford, UK pp. 109-118
[23] Fowler, M. *Analysis Patterns: Reusable Object Models*, 1997, Addison-Wesley
[24] Fowler M. *Refactoring - improving the design of existing code*, 1999, Addison-Wesley
[25] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns – Elements of Reusable Object-Oriented Software, 1995, Addison-Wesley
[26] Garcia, R. et al., "A Comparative Study of Language Support for Generic Programming," Proc. 18th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, 2003, pp. 115-134.
[27] Goguen, J.A. 1983. "Parameterized Programming," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, Sept. 1984, pp. 528-543
[28] Jarzabek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003, Helsinki, pp. 237-246
[29] Jarzabek, S. and Li, S. "Unifying clones with a generative programming technique: a case study," *Journal of Software Maintenance and Evolution: Research and Practice* John Wiley & Sons, Volume 18, Issue 4, July/August 2006, pp. 267-292
[30] Jarzabek, S. "Genericity - a "Missing in Action" Key to Software Simplification and Reuse," to appear in *13th Asia-Pacific Software Engineering Conference, APSEC'06*, IEEE Comp. Soc., 6-8 December 2006, Bangalore, India
[31] Johnson, J. H. Substring Matching for Clone Detection and Change Tracking. In Proc. Intl. Conference on Software Maintenance (ICSM '94), pages 120–126.
[32] Kamiya, T., Kusumoto, S., and Inoue, K. "CCFinder: A multi-linguistic token-based code clone detection system for large scale source code", IEEE Trans. Software Engineering, 2002, 28(7): pp. 654-670
[33] Kang, K et al. "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Pittsburgh, Nov. 1990
[34] Karhinen, A., Ran, A. and Tallgren, T. 1997. "Configuring designs for reuse," International Conference on Software Engineering, ICSE'97, Boston, MA., 1997, pp. 701-710.
[35] Kapser C., and Godfrey, M. "Improved tool support for the investigation of duplication in software," Intl. Conference on Software Maintenance, ICSM'05, Budapest, Sept. 2005, pp.
[36] Kennedy, A. and Syme, D., "Design and implementation of generics for the .Net Common Language Runtime," Proc. ACM Conf. on Programming Languages Design and Implementation (PLDI -01), New York, June 2001, pp 1-12
[37] Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. "Aspect-Oriented Programming," Europ. Conf. on Object-Oriented Programming, Finland, Springer-Verlag LNCS 1241, 1997, pp. 220-242
[38] Kiczales, G., Hilsdale, E., Hugunin, J. et al. 2001. "An Overview of AspectJ," Proc. of 15th European Conference on Object-Oriented Programming, Budapest, Hungary, June 18-22, 2001, Lecture Notes in Computer Science, 2072, pp. 327-353
[39] Kim, M., Bergman, L., Lau, T. and Notkin, D. "An Ethnographic Study of Copy and Paste Programming Practices in OOPL," Proc. Int. Symposium on Empirical Software Engineering , ISESE'04, August 2004, Redondo Beach, California, pp. 83-92
[40] Kontogiannis, K.A., De Mori, R., Merlo, E., Galler, M., and Bernstein, M. Pattern Matching for Clone and Concept Detection. J. Automated Software Eng., vol. 3, pp. 770-108, 1996.
[41] Komondoor, R., and Horwitz, S. Using slicing to identify duplication in source code. In Proc. 8th International Symposium on Static Analysis, 2001, pages 40-56.
[42] Krinke, J. Identifying Similar Code with Program Dependence Graphs. In proceedings of the Eight Working Conference on Reverse Engineering, Stuttgart, Germany, October 2001, pp. 301-309.

[43] Lague, B., Proulx, D., Mayrand, J., Merlo,E.M., Hudepohl, J. "Assessing the Benefits of Incorporating Function Clone Detection in a Development Process," Proc. Int. Conf. on Software Maintenance (ICSM '97) October, 1997 Bari, Italy 314-321
[44] Levenshtein, V.I. 1966. "Binary codes capable of correcting deletions, insertions, and reversals," Cybernetics & Control Theory 10-8, 1966, pp. 707-710.
[45] Loughran, N., Rashid, A., Zhang, W. and Jarzabek, S. "Supporting Product Line Evolution with Framed Aspects," 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, ACP4IS'04, March 22-26, 2004, Lancaster UK
[46] Maginnis, N. 1986 "Specialist: Reusable code helps increase productivity," in Computerworld, Nov. 1986
[47] Mayrand, J., Leblanc, C., and 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,1996, pp.244-253
[48] Meyer, B. 1998. Object-Oriented Software Construction, Prentice-Hall, London, 1988
[49] Musser, D. and Saini, A., 1996. STL Tutorial and Reference Guide: C++ Programming with Standard Template Library, Addison-Wesley, Reading (MA), USA.
[50] Neighbours, J. 1984. The Draco Approach to Constructing Software from Reusable Components. IEEE Trans. on Software Eng., SE-10(5), September 1984, pp. 564-574
[51] Parnas, D., 1972. On the Criteria To Be Used in Decomposing Software into Modules, Communications of the ACM, Vol. 15, No. 12, December, 1972, pp.1053-1058
[52] Pettersson, U., and Jarzabek, S. "Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach," *ESEC-FSE'05, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press, September 2005, Lisbon, pp. 326-335
[53] Prieto-Diaz, R. "Domain analysis for reusability," Proc. COMPSAC'87, October 1987, Tokyo, Japan, pp. 23-29
[54] Rajapakse, D.C and Jarzabek, S. "An Investigation of Cloning in Web Portals," *Int. Conf. on Web Engineering, ICWE'05,* July 2005, Sydney, pp. 252-262
[55] Rieger, M., Ducasse, S. and Lanza, M. "Insights into System-Wide Code Duplication," Proceedings of 11th Working Conference on Re-verse Engineering (WCRE'04), 2004, pp. 100–109.
[56] Sneed, H. private conversation
[57] Shaw, M. and Garlan, D. Software Architecture: Perspectives on Emerging Discipline, Prentice Hall, 1996
[58] Smaragdakis, Y. and Batory, D.2000. "Application generators," in Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering, J. Webster (ed.), John Wiley and Sons, 2000
[59] Smyth, W.F. 2003. Computing patterns in strings, Pearson-Addison-Wesley
[60] Soe, M.S., Zhang, H. and Jarzabek, S. 2002. "XVCL: A Tutorial," Proc. 14th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE'02, ACM Press, July 2002, Italy, pp. 341-349
[61] STL, http://www.sgi.com/tech/stl/
[62] Tarr, P., Ossher, H., Harrison, W. and Sutton, S. "N Degrees of Separation: Multi-Dimensional Separation of Concerns", Proc. International Conference on Software Engineering, ICSE'99, Los Angeles, 1999, pp. 107-119
[63] Tarr, P., Ossher, H. 2000. Hyper/J User and Installation Manual, http://www.research.ibm.com/hyperspace/, IBM
[64] Thompson, S., "Higher Order + Polymorphic = Reusable", unpublished manuscript available from the Computing Laboratory, University of Kent. http://www.cs.ukc.ac.uk/pubs/1997
[65] Ueda, Y., Kamiya, T., Kusumoto, S., and Inoue, K., "Gemini: Maintenance Support Environment Based on Code Clone Analysis," *Proc. Eighth IEEE Symposium on Software Metrics*, pp. 67-76, 2002.
[66] Wong, T.W., Jarzabek, S., Myat Swe, S., Shen, R. and Zhang, H.Y. "XML Implementation of Frame Processor," *Proc. ACM Symposium on Software Reusability, SSR'01,* Toronto, Canada, May 2001, pp. 164-172
[67] Zhang, H.Y., Jarzabek, S. and Soe, M. S. "XVCL Approach to Separating Concerns in Product Family Assets", *Proc. Generative and Component-based Software Engineering (GCSE 2001)*, Erfurt, Germany, September 2001, pp. 36-47
[68] Zhang, H. and Jarzabek, S., "An XVCL-based Approach to Software Product Line Development", *Proc. 15th Int. Conf. on Software Engineering and Knowledge Engineering* (SEKE'03), San Francisco, USA, 1 - 3 July, 2003

Copyright (C) 2007 Stan Jarzabek

[69] XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, http://fxvcl.sourceforge.net

[70] Yang, J. and Jarzabek, S. "Applying a Generative Technique for Enhanced Reuse on J2EE Platform," 4th Int. Conf. on Generative Programming and Component Engineering, *GPCE'05*, Sep 29 - Oct 1, 2005, Tallinn, Estonia, pp. 237-255