JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2006; **18**:267–292
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/smr.333

Practice

Unifying clones with a generative programming technique: a case study

Stan Jarzabek[1,*,†] and Shubiao Li[2]

[1]*Department of Computer Science, School of Computing, National University of Singapore, Lower Kent Ridge Road, Singapore 117543*
[2]*Department of Banking Information Engineering, School of Economics and Finance, Xi'an Jiaotong University, Xi'an 710061, People's Republic of China*

SUMMARY

Software clones—similar program structures repeated in variant forms—increase the risk of update anomalies, blow up the program size and complexity, possibly contributing to high maintenance costs. Yet, programs are often polluted by clones. In this paper, we present a case study of cloning in the Java Buffer library, JDK 1.5. We found that at least 68% of the code in the Buffer library was contained in cloned classes or class methods. Close analysis of program situations that led to cloning revealed difficulties in eliminating clones with conventional program design techniques. As a possible solution, we applied a generative technique of XVCL (XML-based Variant Configuration Language) to represent similar classes and methods in generic, adaptable form. Concrete buffer classes could be automatically produced from the generic structures. We argue, on analytical and empirical grounds, that unifying clones reduced conceptual complexity and enhanced the changeability of the Buffer library at rates proportional to code size reduction (68%). We evaluated our solution in qualitative and quantitative ways, and conducted a controlled experiment to support this claim. The approach presented in the paper can be used to enhance genericity and changeability of any program, independently of an application domain or programming language. As the solution is not without pitfalls, we discuss trade-offs involved in its project application. Copyright © 2006 John Wiley & Sons, Ltd.

Received 5 November 2005; Revised 24 April 2006; Accepted 1 May 2006

KEY WORDS: class libraries; object-oriented methods; maintainability; reusability; generative programming

*Correspondence to: Stan Jarzabek, Department of Computer Science, School of Computing, National University of Singapore, Lower Kent Ridge Road, Singapore 117543.
†E-mail: stan@comp.nus.edu.sg

Contract/grant sponsor: National University of Singapore Research Grant; contract/grant number: R-252-000-239-112

1. INTRODUCTION

As early as in 1993, Batory *et al.* [1] described the 'feature combinatorics' problem hampering the scalability of class/component libraries, reuse and programmers' productivity in general. Batory *et al.* studied C++ data structure class libraries, where typical features relate to data structure, memory allocation scheme, access mode or concurrency. In general, features may relate to any characteristic of a program such as functionality, design solution or platform. The problem manifests itself as follows: features may appear in classes in many different combinations. As we need a unique class for each legal combination of features, we must develop and maintain a large number of similar classes. Batory *et al.* concluded that 'today's method of constructing libraries is inherently unscalable ... Libraries should not enumerate complex components with numerous features ... to be scalable, libraries must offer much more primitive building blocks and be accompanied by generators that compose blocks to yield the data structures needed by application programmers'. In the same paper, Batory *et al.* applied a generation technique of GenVoca [2] to produce classes with the required combination of features from a much simpler set of primitive building blocks than the original classes. In 1994, Biggerstaff further analyzed the library scaling problem and the limits of concrete component reuse caused by 'feature combinatorics' [3].

The problem still exists in today's libraries. As the number of similar classes grows, classes also become polluted with similar, cloned code. In the first part of the paper, we analyze similarity patterns in the Java Buffer library, JDK 1.5, and explain the reasons why conventional techniques were not effective in building a generic, clone-free representation of buffer classes. These reasons could often be traced to the lack of sufficiently strong generic design mechanisms, capable of handling repetitions in a generic way. Current solutions are mostly *ad hoc*, relying on macros, scripts and makefiles. In the Java Buffer library JDK 1.5, the designers escaped to such *ad hoc* solutions, rather than using Java language features and object-oriented design mechanisms, to realize the engineering benefits of much similarity among buffer classes.

In the second part of the paper, we describe a systematic method of clone treatment with a so-called *mixed-strategy approach* in which we build and maintain buffer classes with a generative programming technique of XVCL (XML-based Variant Configuration Language) [4] applied on top of Java. A Java/XVCL mixed-strategy solution represents each group of similar program structures (such as classes or methods) in a unique, generic, but adaptable form. The core functionality of buffer classes is still programmed in Java, but the issues of generic design to unify similarities are delegated to XVCL. The Java/XVCL solution is 68% smaller than the original code, counted as lines of code without blanks or comments. In the Java/XVCL, repeated structures are replaced by generic, adaptable structures, along with information about how to obtain instances of generic structures in the required variant forms. Therefore, code reduction goes hand-in-hand with a reduction in the conceptual complexity of the solution space. The Java/XVCL solution also emphasizes relationships among program elements that matter to programmers who have to understand and maintain classes. In this paper, we evaluate our approach in qualitative and quantitative ways, and conduct a controlled experiment to support the above claim. As the approach is not without pitfalls, we discuss its strengths and weaknesses.

While we can apply XVCL to strengthen generic design and changeability in any software, independently of an application domain or programming language, the results presented in this paper are specific to the Buffer library. Other libraries may suffer from the described problems to a lesser

Table I. Features in the Buffer library.

Level in class hierarchy	Feature dimension	Features
Level 1	buffer data element type	byte, char, int, double, float, long, short
Level 2	memory allocation scheme	Direct, Non-direct
	byte ordering	Little_endian, Big_endian, Native, Non-native
Level 3	access mode	Writable, Read-only

degree or not at all, depending on the domain and the design technique used. Clones are also found in application programs, but for different reasons (e.g., [5]), and the feature combination problem does not affect application programs to the extent that we see it in the Buffer library. Still, we believe some lessons learned from this experiment may also apply in other software.

This paper extends an initial study of the Buffer library (JDK 1.4.1) described in [6] as follows: (1) we analyzed the Java Buffer library JDK 1.5 with generics (type-parameterized classes) and found that generics were not used to unify similar classes; (2) we provided full details of clone analysis; (3) we described a Java/XVCL mixed-strategy solution in more detail than we did in the earlier paper; and (4) we conducted a controlled experiment in which we compare the changeability of the Java/XVCL solution with the original buffer classes.

This paper is organized as follows. In Sections 2–4, we provide the statistics of cloning in the Buffer library and discuss the reasons why cloning was difficult to avoid. In Section 5, we describe a generic Java/XVCL representation of buffer classes. In Section 6, we evaluate the two solutions and discuss the trade-offs involved in applying XVCL. Related work and conclusions end the paper.

2. AN OVERVIEW OF THE BUFFER LIBRARY

A buffer contains data in a linear sequence for reading and writing. The Buffer class library in our case study is a part of the java.nio.* packages JDK 1.5. Buffer classes differ along the dimensions of features such as the buffer element type, memory allocation scheme, byte ordering and access mode (Table I).

Figure 1 shows part of the Buffer library. Class subhierarchies for classes DoubleBuffer, FloatBuffer, LongBuffer and ShortBuffer are analogous to subhierarchies for classes CharBuffer and IntBuffer, so for the sake of brevity we do not depict them in Figure 1. Below, we briefly describe the features addressed in the Buffer library and explain how these features are reflected in classes.

At Level 1, we see seven classes that differ in buffer element data types. A programmer can directly use only Level 1 classes. Therefore, these classes contain many methods providing access to functionalities implemented in the classes below them.

Classes at Level 2 address two memory allocation schemes and two byte orderings. 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.

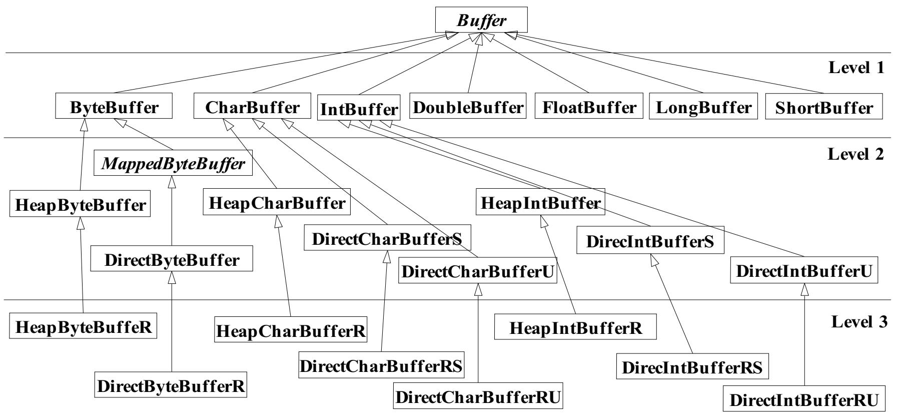


Figure 1. A fragment of the Buffer library.

On the other hand, in the non-direct memory allocation scheme, a buffer is accessed through Java array accessor methods.

Byte ordering matters for buffers whose elements consist of multiple bytes, that is, all of the element types except byte. 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).

When using the direct memory allocation scheme, we must know if buffer elements are stored using native or non-native byte ordering. Twenty new classes at Level 2 in Figure 1 result from combining memory allocation and byte ordering features. (We do not count MappedByteBuffer which is just a helping class.) For each buffer class at Level 1, we have one Heap* class at Level 2 that implements the non-direct memory allocation scheme for that buffer. Classes with suffixes ‘S’ and ‘U’ implement direct memory allocation scheme with native and non-native byte ordering, respectively. We have only one class DirectByteBuffer, as byte ordering does not matter for byte buffers.

The buffers discussed so far are writable. Twenty classes at Level 3 implement read-only variants of buffers.

3. SIMILARITY PATTERNS IN BUFFER CLASSES

We identified the following seven groups of classes with much similarity among classes in each group.

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

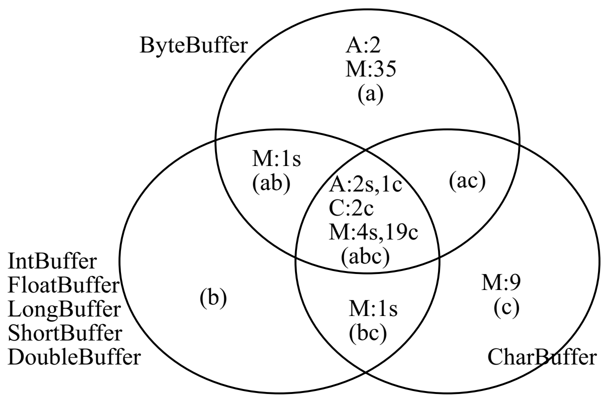


Figure 2. Distribution of clones in classes at Level 1.

4. Direct[**T**]Buffer[**S**|**U**]: 13 classes at Level 2 for combinations of buffer element type, **T**, with byte orderings: **S**, non-native; or **U**, native byte ordering (note that byte ordering is not relevant to buffer element type 'byte').
5. Direct[**T**]BufferR[**S**|**U**]: 13 read-only classes at Level 3 for combinations of parameters **T**, **S** and **U**, as above.
6. ByteBufferAs[**T**]Buffer[**B**|**L**]: 12 classes at Level 2 (see Figure 10 below) 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 Level 3 (see Figure 10 below) for combinations of parameters **T**, **B** and **L**, as above.

In the discussion below, a *code fragment* means a unit of class definitions, such as methods/constructors, segments of method/constructor implementation or attribute declarations sections. A *clone* means a code fragment recurring in the same or similar form.

3.1. Analysis of [T]Buffer classes at Level 1

In Figure 2, circles marked with (a), (b) and (c) represent classes as indicated in the diagram. Note that circle (b) represents five classes. In the overlapping areas, we indicate clones recurring in the respective classes. We refer to the overlapping areas by listing the circles involved such as (ab) or (abc). Non-overlapping areas represent unique code fragments in the respective classes. In Figure 2 (and Figure 4 below), 'A' refers to class attributes, 'C' to constructors and 'M' to methods. Symbol 's' denotes exact clones and 'c' denotes clones recurring with slight changes. For example, (A:2 M:35) in the non-overlapping area of circle (a) means that ByteBuffer class has 37 unique code fragments including two attribute definition sections and 35 method definitions. On the other hand, IntBuffer class has no unique fragments.

Five classes, namely IntBuffer, FloatBuffer, LongBuffer, ShortBuffer and DoubleBuffer differ in type parameter and otherwise are identical. An interesting observation is that this subgroup of classes

```
public abstract class ByteBuffer      public abstract class CharBuffer
   extends Buffer                        extends Buffer
   implements Comparable                 implements Comparable, CharSequence
{ . . .  . . . }                      { . . .  . . . }
```

Figure 3. Declaration of class ByteBuffer and CharBuffer.

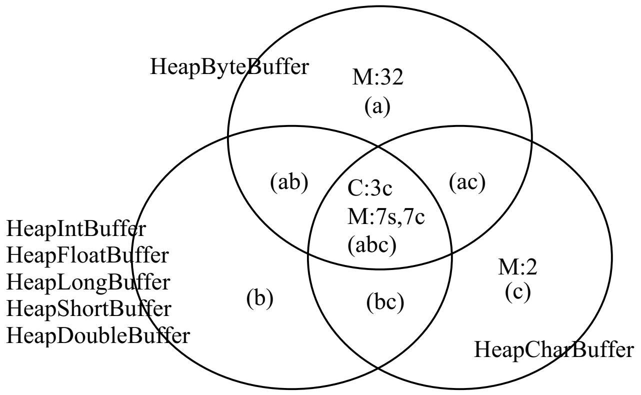


Figure 4. Distribution of code fragments in Heap* classes at Level 2.

has no unique code fragments, that is it is solely built of code fragments that recur in other [T]Buffer classes in the same or similar form.

Declarations of classes ByteBuffer and CharBuffer (Figure 3) differ only in the *implements* clause. We did not include this type of clone into our statistics.

3.2. Analysis of classes at Level 2

Classes at Level 2 address memory allocation schemes and byte ordering, as well as the buffer element type. This new combination of features brings in two new subclasses of ByteBuffer and three new subclasses for each of the remaining classes at Level 1. Not surprisingly, we found more clones in classes at Level 2.

Figure 4 shows code fragments in Heap* classes. Five classes, namely HeapIntBuffer, HeapFloatBuffer, HeapLongBuffer, HeapShortBuffer and HeapDoubleBuffer differ in type parameter and are otherwise identical. As before, this subgroup of classes has no unique code fragments, that is it is solely built of code fragments that recur in other Heap[T]Buffer classes.

Figure 5 shows method **put()** recurring in classes HeapByteBuffer and HeapCharBuffer. Analogical methods **put()** recur in each of the remaining Heap* classes.

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

Figure 5. Method **put()** in HeapByteBuffer and HeapCharBuffer classes.

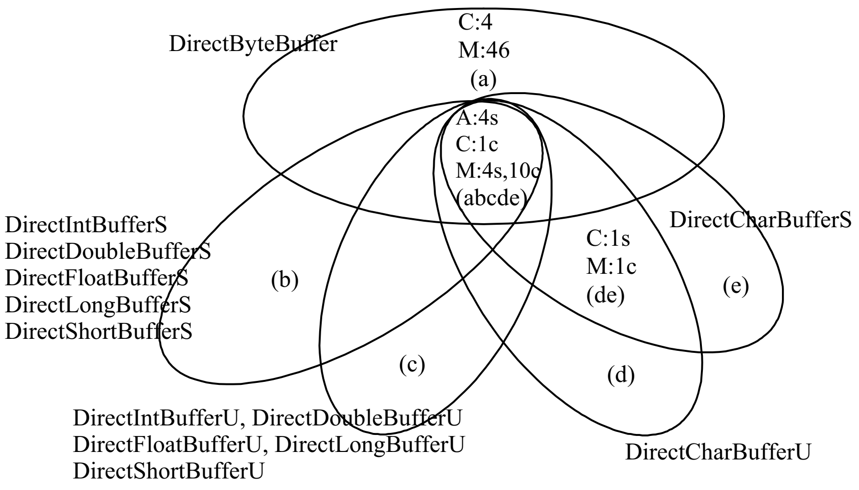


Figure 6. Distribution of code fragments in Direct* classes at Level 2.

Classes implementing the direct memory allocation scheme (named Direct* in Figure 1) also contain many clones, as shown in Figure 6.

For example, method **slice()** recurs 13 times in all of the Direct* classes with various combination of values highlighted in bold. In Figure 7, we see method **slice()** from class DirectByteBuffer.

Implementation of method **order()** appears in 12 Direct* classes, all but DirectByteBuffer, with differences in operators only.

We did not find interesting clones across Direct* and Heap* classes at Level 2 (Figure 1) such as DirectByteBuffer and HeapByteBuffer. Most of the methods in those classes deal with accessing buffer elements and are highly dependent on the specific memory allocation scheme.

3.3. Analysis of read-only classes at Level 3

Level 3 brings in 20 new read-only buffer classes. Clones in read-only buffer classes are shown in Figures 8 and 9.

```
/*Creates a new byte buffer containing a shared
   subsequence of this buffer's content. */
public Buffer 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 7. Method **slice()** recurring with slight changes in 13 Direct* classes.

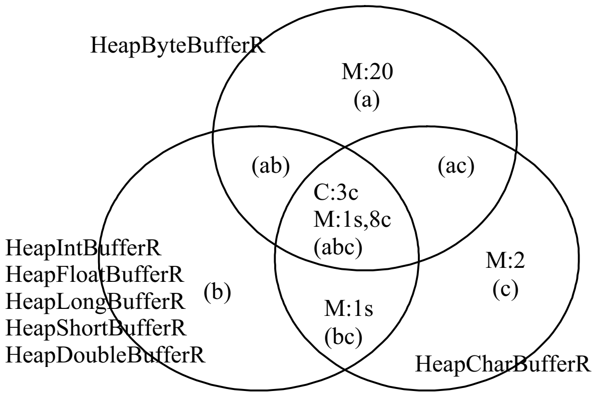


Figure 8. Distribution of code fragments in non-direct read-only classes.

Five classes, namely HeapIntBufferR, HeapFloatBufferR, HeapLongBufferR, HeapShortBufferR and HeapDoubleBufferR differ in type parameter and are otherwise identical. Class HeapByteBufferR has 20 unique methods. Class HeapCharBufferR has two unique methods. Thirteen classes in the Direct[T]BufferR[S|U] group display much similarity. Thirteen code fragments recur in either the same form or with slight changes in more than one class in the group, among which 11 fragments recur in all of the 13 classes.

This time, clones also exist across horizontal classes such as HeapByteBufferR and HeapCharBufferR. A few clones appear across parent classes and their subclasses as well.

3.4. Analysis of the remaining buffer classes

Figure 10 shows 25 new classes derived from Level 1 classes. ByteBufferAsCharBufferB reforms ByteBuffer to CharBuffer, using Big_Endian byte ordering; ByteBufferAsCharBufferL does the same

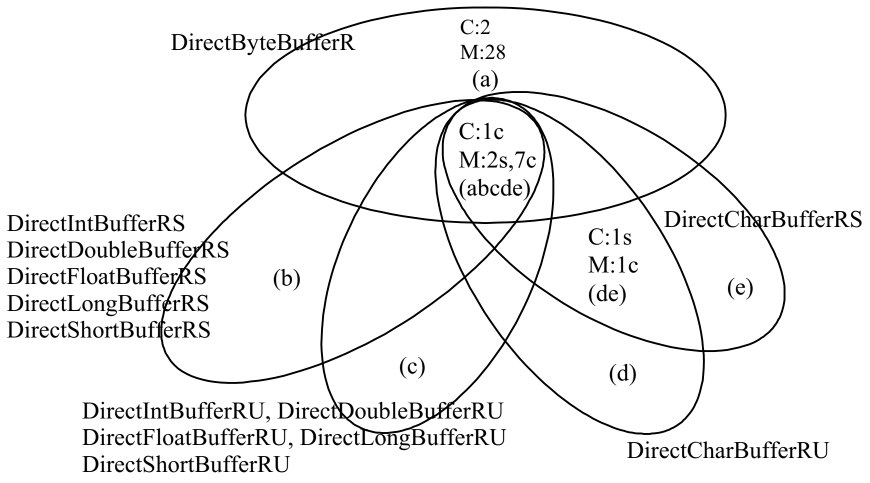


Figure 9. Distribution of code fragments in direct read-only classes.

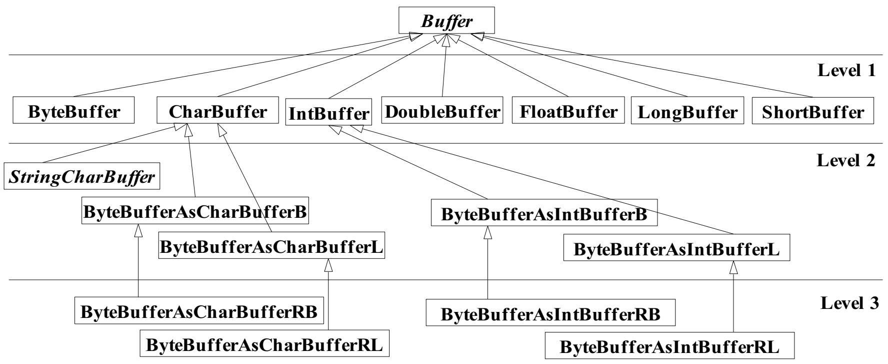


Figure 10. Remaining buffer classes.

using Little_Endian. Buffers for Int, Double, Float, Long and Short types, also have similar subclasses. StringCharBuffer class in italic at Level 2 is mainly for making a non-direct, read-only CharBuffer from a CharSequence. We did not include this class in our analysis and statistics.

All classes at Levels 2 and 3 share many clones as shown in Figures 11 and 12.

Classes in each of the two groups, namely ByteBufferAs[T]Buffer[B|L] and ByteBufferAs[T]BufferR[B|L] share much similarity in respective groups.

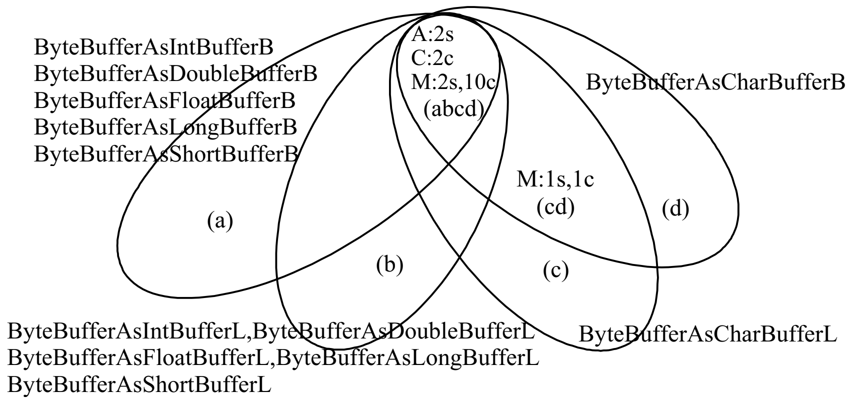


Figure 11. Distribution of clones in classes ByteBufferAs*.

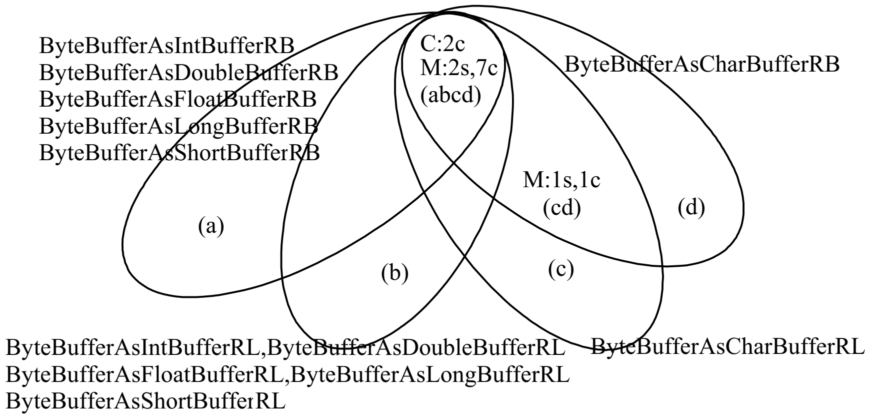

ByteBufferAsIntBufferRB
ByteBufferAsDoubleBufferRB
ByteBufferAsFloatBufferRB
ByteBufferAsLongBufferRB
ByteBufferAsShortBufferRB
C:2c
M:2s,7c
(abcd)
ByteBufferAsCharBufferRB
M:1s,1c
(cd)
(a)
(b)
(c)
(d)
ByteBufferAsIntBufferRL,ByteBufferAsDoubleBufferRL
ByteBufferAsFloatBufferRL,ByteBufferAsLongBufferRL
ByteBufferAsShortBufferRL
ByteBufferAsCharBufferRL

Figure 12. Distribution of clones in read-only classes ByteBufferAs*.

4. REASONS FOR CLONING IN THE BUFFER LIBRARY

Usability, conceptual clarity and high performance are important design goals for the Buffer library. To simplify the use of the Buffer library, the designers decided to only reveal the top eight classes at Level 1 to programmers (Figure 1). Functionalities related to lower-level concrete classes can be accessed via special methods provided in the top eight classes. For conceptual clarity, we saw almost one-to-one mapping between legal feature combinations and buffer classes. While in some situations cloning could be avoided by introducing a new abstract class or a suitable design pattern, such a solution would inject other complications.

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

Figure 13. Cloned method **hasArray()**.

The main reason for cloning in the Buffer library was that conventional techniques were not effective in unifying clones without compromising other important design goals.

It is quite common that software is developed under multiple, sometimes competing, design goals. Achieving non-redundancy may be one of the goals, but is not usually the most important. Any solutions to unifying clones must be evaluated in view of this reality, which is normally very different to and more complex than the isolated examples we find in textbooks.

Most of the clones in buffer classes spark from feature combinations. As features cannot be implemented independently of each other in separate implementation units (e.g., class methods), code related to specific features appears in many variant forms in different classes, depending on the context. Clones arise whenever such code cannot be parameterized to unify the variant forms or be placed in some upper-level class for reuse via inheritance. To observe the impact of feature combinations on cloning, we compared a number of classes that differed 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, say class A, need a certain method (or data) and other classes derived from A do not need that method. We could create a new abstract parent class just to make that method available to the classes that need it. Creating many such classes would, however, complicate the class hierarchy and hinder performance. We could also place such a method in the parent class A. However, this solution would either be error-prone or require us to write extra code to disable the method in the classes that do not need it. In yet other situations, a certain method is needed in all the classes derived from the same parent, say class A, but in some of those classes the method requires 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 place a method into each class that needs it, creating clones.

Method **hasArray()** shown in Figure 13 illustrates a simple yet interesting case. This method is cloned in each of the seven classes at Level 1. Although method **hasArray()** recurs in all seven classes, it 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**.

Much cloning arises 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. For example, method **slice()** recurs 13 times in all of the Direct* classes with slight changes highlighted in bold in Figure 7. Classes or methods that differ in type parameters are candidates for generics. The newest release of JDK 1.5 supports generics. However, generics have not been applied to unify clones described in our study. 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, where T stands for Byte, Char, Int, Double, Float, Long or Short. 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 (represented as circles (b) in Figures 2, 4 and 8) whose unification with three generics saves 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. Application of generics may also be hampered by couplings among classes across the library. We describe further cases of similar but generics-unfriendly situations in case studies at the XVCL Web site: http://fxvcl.sourceforge.net/.

In summary, it is difficult to increase the genericity of the Buffer library with Java generics. C++ templates [7] are free of some of the above-mentioned limitations (e.g., generics parameters may be primitive types, keywords or constants), but still cannot deal with more complicated forms of parameterization in a natural and practical way [8]. Our study of STL [9] points to some specific problems in that area.

It is interesting to note that small variations appear in otherwise the same clones 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.

5. CONSTRUCTION OF BUFFER CLASSES WITH XVCL

In Java/XVCL mixed strategy, each similarity pattern that matters in class construction is represented by a generic, adaptable meta-component. We organize an overall solution into a hierarchy of meta-components shown on the left-hand-side of Figure 14, 'normalized' to eliminate clones. Lower-level meta-components are building blocks for upper-level meta-components. Each of the groups of similar classes (identified in Section 3) is represented by a meta-class from which we can generate all of the classes in a given group. Meta-methods and meta-fragments represent groups of similar class methods and smaller class building blocks, respectively.

The top-most meta-component is called a specification meta-component (SPC for short). It specifies how to construct all of the buffer classes.

Meta-components contain Java code inter-mixed with XVCL commands that mark variation points. XVCL commands indicate how a meta-component can adapt meta-components below it, and how a meta-component can be adapted by meta-components above it. The XVCL processor traverses meta-components as indicated by <adapt> commands in depth-first order, starting with the SPC. For each visited meta-component, the processor interprets XVCL commands embedded in that meta-component and generates source code accordingly.

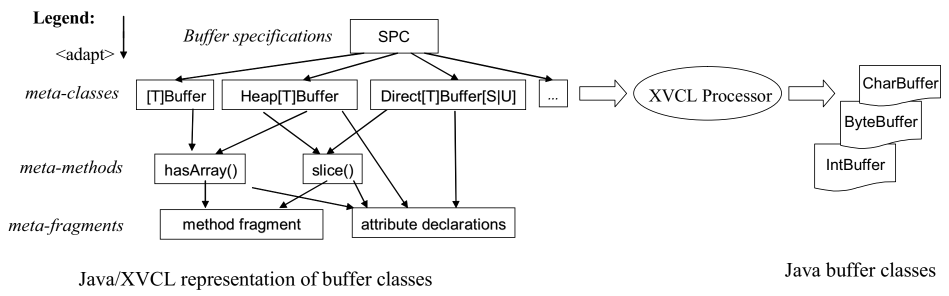


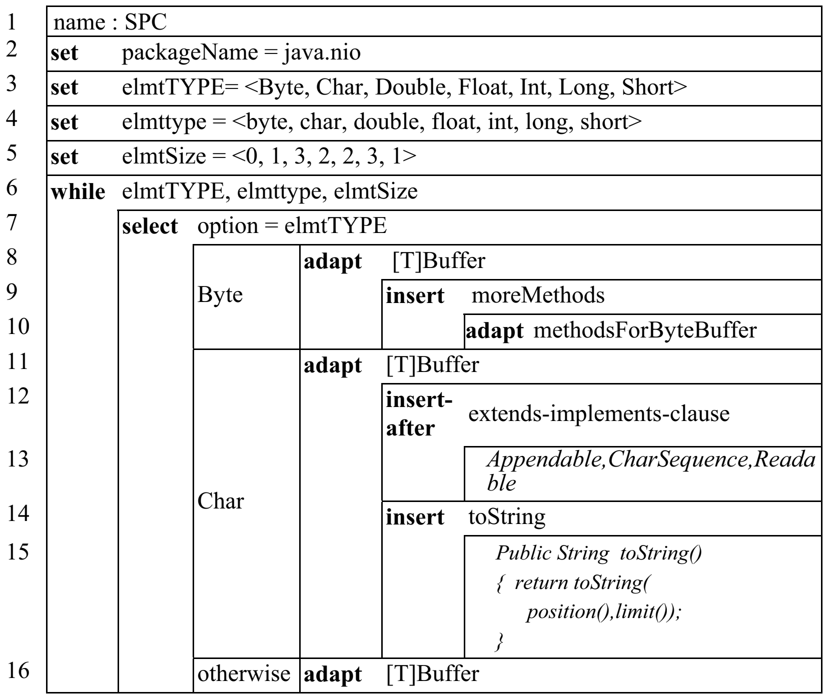
Figure 14. An overview of the Java/XVCL mixed-strategy solution.

XVCL defines a number of mechanisms to achieve flexible composition and adaptation of meta-components. XVCL variables and expressions provide a basic parameterization mechanism. They are a means to create generic names and to control the meta-component processing order. Typically, class or method names, data types, keywords, operators or even short algorithmic fragments are represented in parameterized form by expressions. During XVCL processing, values of variables propagate across visited meta-components. While each meta-component usually can set default values for its variables, values assigned to meta-variables in higher-level meta-components take precedence over the locally assigned default values. Thanks to this scoping mechanism, meta-components become generic and adaptable, with potential for reuse in many contexts. Other XVCL commands that help us design generic and adaptable meta-components include <select>, <insert> into <break> and <while>. We use the <select> command for direct processing into one of the many pre-defined branches (called options), based on the value of a meta-variable. With <insert> command, we can modify meta-components at designated <break> points in arbitrary ways. A <while> command allows us to iterate over certain sections of a meta-component, with each iteration generating custom output. For example, a <select> command in the <while> loop in the SPC of Figure 15, shown in the style of a Chapin chart, allows us to generate classes in each of the seven groups discussed at the beginning of this section.

We now examine excerpts from the meta-solution for the Buffer library for generating seven [T]Buffer classes at Level 1 in Figure 1. Meta-class [T]Buffer (Figure 16) is a template that defines a common structure and code (shown in italics) for [T]Buffer classes. SPC (Figure 15) iterates over the <while> loop seven times. Each iteration <adapt>s the meta-class [T]Buffer to generate one class from the group.

SPC <set>s the value of variable packageName to ‘java.nio’. Values of variables elmtTYPE, elmttype and elmtSize are lists of constants enclosed between brackets ‘<’ and ‘>’. Variable elmtSize indicates the size of a buffer element: 2^{elmtSize} = (the number of bytes per buffer element). Values of variables propagate from the <set> command reaching subsequently processed lower-level meta-components.

Command <while> in SPC is controlled by three variables, namely elmtTYPE, elmttype, elmtSize. In the ith iteration, these variables accept the ith value from their respective lists. Based on the value of elmtType, a suitable option (such as Byte, Char or otherwise) under <select> generates code for the appropriate class.



```
name : SPC
set packageName = java.nio
set elmtTYPE= <Byte, Char, Double, Float, Int, Long, Short>
set elmttype = <byte, char, double, float, int, long, short>
set elmtSize = <0, 1, 3, 2, 2, 3, 1>
while elmtTYPE, elmttype, elmtSize
    select option = elmtTYPE
        Byte
            adapt [T]Buffer
                insert moreMethods
                    adapt methodsForByteBuffer
        Char
            adapt [T]Buffer
                insert-after extends-implements-clause
                    Appendable,CharSequence,Readable
                insert toString
                    Public String toString()
                    { return toString(
                        position(),limit());
                    }
        otherwise
            adapt [T]Buffer
```

Figure 15. A fragment of SPC to construct [T]Buffer classes.

Class ByteBuffer requires extra methods. These methods are stored in a separate meta-component called ‘methodsForByteBuffer’ and are <insert>ed into a <break> ‘moreMethods’ in the meta-class [T]Buffer. To generate class CharBuffer, we override the default implementation of method **toString()** defined in the meta-class [T]Buffer. All of the remaining classes do not require any adaptations, and are generated in the <otherwise> option of <select>.

The XVCL Processor concatenates values of variables, such as @elmtTYPE, with surrounding code (shown in italics for readability) and emits the result to the output. The first line in [T]Buffer instructs the Processor to emit output for respective classes to files ByteBuffer.java, CharBuffer.java and so on.

Exact clones, such as method **hasArray()** shown in Figure 13, are included ‘as is’ and do not require adaptation. Parameterized method **slice()** (Figure 7) is shown in Figure 17. An expression Direct@elmtTYPEBuffer@byteOrder allows us to generate names for all of the Direct* classes, such as DirectByteBuffer, DirectIntBufferU and DirectIntBufferS. Variable byteOrder is <set> to an empty string, ‘S’ or ‘U’, in a respective <set> command placed in an upper-level meta-component (not shown in our figures).

Class building blocks that differ only in simple parameters across the seven [T]Buffer classes, are stored in meta-components ‘commonAttributes’, ‘commonConstructors’ and ‘commonMethods’. An excerpt from meta-component ‘commonMethods’ is shown in Figure 18.

1	meta-class: [T]Buffer **outfile**:@elmtTYPE*Buffer.java*	
2		*package* @packageName; *public abstract class* @elmtTYPE*Buffer extends* *Buffer implements Comparable*
3	**break**	*extends-implements-clause*
4		*{ // attributes and methods here*
5	**adapt**	commonAttributes
6	**break**	moreAttributes
7	**adapt**	commonConstructors
8	**break**	moreConstructors
9	**adapt**	commonMethods
10	**break**	moreMethods
11	**break**	toString

12		*public String toString() {*
		StringBuffer sb = new StringBuffer();
		sb.append(getClass().getName());
		sb.append("[pos="+ position());
		sb.append(" lim="+ limit());
		sb.append(" cap="+ capacity()+"]");
		return sb.toString(); }

		}

Figure 16. A fragment of meta-class [T]Buffer.

meta-method: **slice**

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

Figure 17. Meta-method **slice**.

```
1  meta-component: commonMethods
2  comments   contains definitions of the methods at Level 1
3                   … //other methods
4                   //Tells whether or not this buffer is equal to another object.
                    public boolean equals(Object ob){
                       if (!(ob instanceof @elmtTYPEBuffer)) return false;
                       @elmtTYPEBuffer that = (@elmtTYPEBuffer)ob;
                       if (this.remaining() != that.remaining()) return false;
                       int p = this.position();
                       for (int i=this.limit()-1, j=that.limit()-1; i >= p;i--,j--) {
                          byte v1 = this.get(i);
                          byte v2 = that.get(j);
                          if (v1 != v2) {
                        if ((v1 != v1) && (v2 != v2))  // For float and double
                             continue;
                             return false;
                          }
                       }
                       return true;
                    }
5                   … //other methods
```

Figure 18. An excerpt of common methods for [T]Buffer classes.

We end this section by summarizing the process of designing the mixed-strategy solution for buffer classes. We gained a general understanding of the buffer classes first. Then, we identified groups of classes that we suspected would be similar. We examined methods and attribute declarations within each group which confirmed that there was much similarity among classes. Top-down domain analysis gave us insights into an overall structure of the mixed-strategy solution. The actual process of building the solution was bottom-up: For each group of similar methods, we created a suitable meta-method. As for generic methods whose instances appeared in different contexts with changes, we parameterized them with XVCL commands to cater for required variations. Then, we designed a meta-class for each of the seven groups of similar classes, and wrote an SPC. It took 10 days for one person, with no prior knowledge of the Buffer library but with good knowledge of XVCL, to complete the job.

6. EVALUATION OF THE BUFFER LIBRARY EXPERIMENT

For fair comparison, our Java/XVCL mixed-strategy solution facilitates the generation of buffer classes in their original form. Our qualitative analysis includes an experiment in which we extended the Buffer library with a new type of buffer element—Complex. We discuss general trade-offs involved in the application of techniques such as XVCL.

Table II. Original Buffer library versus the Java/XVCL solution.

	Original Buffer library			Buffer library in Java/XVCL		
Classes	Fragments	LOC[a]	Java code[b]	Meta-components	LOC[c]	Java code[d]
Level 1 (7 classes)	258	3720	871	79	1400	320
Level 2 Heap* (7 classes)	159	914	802	52	313	291
Level 2 Direct* (13 classes)	337	2428	2249	85	689	665
Level 3 Heap* read-only (7 classes)	112	521	444	35	226	209
Level 3 Direct* read-only (13 classes)	187	979	895	42	378	367
Subtotal (47 classes)	1053	8562	5261	293	3006	1852
Other classes at level 2 (12 classes)	196	1014	952	18	139	133
Other classes at level 3 (12 classes)	136	556	506	13	100	95
Subtotal for others (24 classes)	332	1570	1458	31	239	228
Total	1385	10 132	6719	324	3245	2080

[a] Including Java code and comments.
[b] Only including Java code.
[c] Including Java code, comments, and XVCL instructions.
[d] Including Java code and XVCL instructions.

6.1. Quantitative analysis of the Java/XVCL solution

Table II contrasts the original Java code with Java/XVCL representation for the buffer classes.

Table II shows the number of conceptual elements in each of the two solutions. A conceptual element in a Java solution is a class, method or even smaller fragment that plays a role in the Buffer domain or in class construction. The columns on the left-hand side of Table II show the number of conceptual elements (column 'fragments') per level in the class hierarchy, along with fragments' sizes. Conceptual elements in Java/XVCL solution are meta-components. The columns on the right-hand side show the number of conceptual elements (column 'meta-components') and their sizes in the Java/XVCL solution.

```
//Complex.java
package java.nio;
public class Complex extends Number
{
  private double re = 0;
  private double im = 0;

  public Complex (double re, double im) {
       this.re = re;
       this.im = im;
  }
  public int getReal() {
       return re;
  }
  public int compareReal(Complex c) { //compare real parts
       return re – c.getReal();
  }
}
```

Figure 19. Class Complex.

Columns 'LOC' and 'Java Code' show physical lines of code without blanks, with and without comments, respectively. Unification of similarities with XVCL on top of Java code eliminates 68% of the code as compared with the original buffer classes. We observe a similar code reduction if we count code lines with comments. It is possible and useful to manage both executable code and comments with XVCL.

6.2. Qualitative analysis of the Java/XVCL solution

Being conceptually and physically smaller than original classes in Java, not only does Java/XVCL solution contain complete Java code for buffer classes, but also emphasizes important relationships among program elements that help programmers understand, modify and reuse the classes. Here, we mean the visibility of similarities and differences among classes, and the ability to trace how various features affect code. Instead of dealing with each class in separation from the others, we can understand classes in groups (we refer to the seven groups of similar classes shown in Section 3). In the Java/XVCL representation, we can see what is similar and what is different across specific classes in a group, across methods/constructors, down to every detail of the code. If we want to change one class, we can check whether the change also affects other similar classes. If we want to change a class method, we can analyze the impact of change on all of the classes that use that method in the same or similar form. The information contained in the Java/XVCL representation reduces the risk of update anomalies, and helps in reusing existing classes when building new classes.

The above relationships are implicit in the Java buffer classes, as well as in most other conventional programs. A programmer must recover them whenever a program must be understood for change.

name : SPC	
set	packageName = java.nio
set	elmtTYPE = <Byte, Char, Double, Float, Int, Long, Short, **Complex**>
set	elmttype = <byte, char, double, float, int, long, short, **Complex**>
set	elmtSize = <0, 1, 3, 2, 2, 3, 1, **4**>
	the rest is the same as in SPC of Figure 15

Figure 20. Modified SPC.

We conducted a small experiment to illustrate and substantiate the above observations. Suppose that we need buffers of complex numbers. We concentrate only on three among many classes that must be implemented, namely ComplexBuffer, HeapComplexBuffer and HeapComplexBufferR. A Complex buffer is similar to other numeric buffers. A sample implementation is given in Figure 19.

We represent a complex number as two double numbers. As each double occupies 2^3 bytes, the size of a Complex number is 2^4 bytes and the value of *elmtSize* for a Complex number is 4. To generate class ComplexBuffer, we need add an extra iteration of a while loop <adapt>ing meta-class [T]Buffer with values of meta-variables *elmtTYPE*, *elmtype* and *elmtSize* set to **Complex**, **Complex** and **4**, respectively. Figure 20 shows a part of the revised SPC to generate classes ComplexBuffer, HeapComplexBuffer and HeapComplexBufferR.

The new type Complex also affects definitions of methods **hashCode()** and **CompareTo()** in commonMethods, as follows: in **hashCode()**, the buffer element must be casted to type int in all classes except ComplexBuffer, in which the buffer element must be casted to type Complex. The <select> command in method **hashCode()** discriminates between these two situations, as shown in bold in Figure 21.

We generated sample classes from the modified meta-components and tested the result.

Now, we address Complex type in the original Buffer library, in the scope of the same three classes as we did before for the Java/XVCL solution. Since Complex buffer is analogical to other numerical buffers, we could start with integer buffer and copy selected code from classes IntBuffer.java, HeapIntBuffer.java and HeapIntBufferR.java to the three new classes. However, adaptation of the copied code is not simple. For example, there are 42 places in class IntBuffer.java (which is only 105 lines) that must be changed from **Int** or **int** to **Complex**. This replacement cannot be done automatically as not all of the occurrences of 'int' should be changed. Methods **hasCode()** and **compareTo()** mentioned before must be also changed. Adaptation of code from classes HeapIntBuffer.java and HeapIntBufferR.java for classes HeapComplexBuffer.java and HeapComplexBufferR.java, respectively, requires similar actions. Changes involved are quite tedious and error-prone.

Table III summarizes changes involved in adding element type Complex. In the original Buffer library, implementing class ComplexBuffer.java requires 25 replacements of 'Int' by 'Complex' that can be automated by an editing tool. It further requires 17 replacements of 'int' to 'Complex' that must be done manually, and yet another two manual changes. On the other hand, in the Java/XVCL solution, all of the changes must be done manually, but only five modifications are required.

The above experiment is small, but it gives a flavor of the changes required in each case.

```
1   meta-component : commonMethod
2           //other methods
3            // Returns the current hash code of this buffer.
              public int hashCode() {
                int h = 1;
                int p = position();
                for (int i = limit() - 1; i >= p; i--)
4   select  option = elmtTYPE
5           Complex    h = 31 * h + ((Complex)get(i)).getReal();
6           otherwise  h = 31 * h + (int)get(i);
7               return h;
            }
8            // Compares this buffer to another object.
             public int compareTo(Object ob) {
                @elmtTYPEBuffer that = (@elmtTYPEBuffer)ob;
                int n = this.position() + Math.min(this.remaining(), that.remaining());
                for (int i = this.position(), j = that.position(); i < n; i++, j++) {
                    @elmtTYPEBuffer v1 = this.get(i);
                    @elmtTYPEBuffer v2 = that.get(j);
                      if (v1 == v2)  continue;
                      if ((v1 != v1) && (v2 != v2))     // For float and double
                          continue;
9   select  option = elmtTYPE
10          Complex    if (v1.compareReal(v2) < 0)
11          otherwise  if (v1 < v2)
12                          return -1;
                     return +1;
                }
                return this.remaining() - that.remaining();
              }
13          //other constructors
```

Figure 21. Modified methods hashCode() and CompareTo().

6.3. Trade-offs

Despite the advantages described above, applying XVCL is not without pitfalls. In general, it is more difficult to develop generic, reusable and maintainable programs than concrete programs: a concrete program is a prerequisite to the understanding of areas that can be generalized for reuse and/or prepared for better changeability. A mixed strategy integrates two distinct program perspectives. The first perspective is concerned with the program runtime structure (in terms of components),

Table III. The impact of addressing type Complex in the original Buffer library versus the Java/XVCL solution.

		Changes in original Buffer library			Changes in Java/XVCL solution		
New classes	Copied classes	Type of changes	Number	Type	Type of changes	Number	Type
ComplexBuffer	IntBuffer	text: Int–Complex	25	automatic	values of multi-variables	3	manual
		text: Int–Complex	17	manual			
		specific changes	2	manual	specific changes	2	manual
HeapComplexBuffer	HeapIntBuffer	text: Int–Complex	21	automatic	values of multi-variables	3	manual
		text: Int–Complex	10	manual			
HeapComplexBufferR	HeapIntBufferR	text: Int–Complex	16	automatic	values of multi-variables	3	manual
		text: int–Complex	5	manual			

behavior (in terms of user interface, business logic and database structures) and software properties such as performance or reliability. This perspective is the domain of a programming language (or languages). The other perspective is concerned with issues of genericity and changeability. This is a domain of XVCL. Properly relating and integrating these two perspectives is a challenge. XVCL is not a substitute for thinking. On the contrary, it requires more thoughtful and skillful design than the conventional object-oriented and component-based program design. As a concrete program is only a prerequisite for designing a mixed-strategy solution, development with XVCL must take longer than the development of a concrete program. We do not use XVCL for quick gains. XVCL mostly benefits long-lived programs that are subject to frequent changes or offer reuse opportunities.

A mixed-strategy solution is expressed in, at least, two languages, namely base programming language(s) and XVCL. This creates extra difficulties. As compensation for this, a mixed-strategy solution contains much useful information for maintenance and reuse, in addition to complete information about the subject program(s) itself.

As we relax the coupling between the parameterization mechanism and the rules (syntax and semantics) of the underlying programming language, the power of the parameterization mechanism increases. However as we apply less-restrictive parameterization mechanisms, we also decrease the type-safety of parameterized program solutions. Skillful design of generic structures, informal documentation and tools can mitigate problems to some extent. A mixed-strategy solution can be organized based on the usual principles of the abstraction and separation of concerns. Tools may further simplify application of XVCL. In our lab, we are working on an XVCL Workbench that helps in the editing, visualization, debugging and static analysis of XVCL solutions.

The feedback from our industry partner indicates that, in practice, the benefit of a mixed strategy may outweigh the cost of the added complexity [10]: the learning curve and development effort of an XVCL solution can be reasonable even for large programs (provided that an XVCL expert is also familiar with an application domain and program itself). At the same time, the return on investment may be quick and substantial. We are collecting further empirical evidence to better assess the trade-offs involved.

A mixed strategy affects conventional development processes in a similar way to systematic reuse via a product line. Changing the way people think about software, and changing existing processes and company structures is always a challenge. At this point, we do know how XVCL can raise the productivity of small teams of highly-skilled expert software developers. We are yet to learn what it takes to inject mixed-strategy methods into large-scale team-based industrial development processes.

7. RELATED WORK

Clones are usually described in the literature as multiple copies of the same or similar code fragments scattered throughout a program [5,11,12]. In our studies and in this paper, we consider any program structures that exhibit significant similarity as clones. We use the term *simple clone* to mean similar segments of contiguous code such as class methods or implementation fragments. We use the term *structural clones* [13] to mean larger chunks of similar code, with more complex internal structure than simple clones. Examples include similar classes or components and patterns of inter-related components/classes—any program structures that often result from repeatedly applied similar design solutions. Analysis patterns [14], design patterns [15] and 'mental templates' [5] exemplify situations that often lead to structural clones.

The many reasons why clones occur in software have been discussed in the literature [5,6,11–13]. Practitioners often mention the negative impact of clones on maintainability (because of increased complexity of a program and risk of update anomalies), but some also hint at technical difficulties in eliminating clones [16]. Cordy points at the risks involved in changing programs as a factor that can rule out clone elimination as a viable strategy for improving code quality [17].

The most common reason for cloning is to boost productivity during development and maintenance by *ad hoc* 'copy–paste–modify' reuse of code. At times, clones are created on purpose, for example, to increase the robustness of life-critical systems, for better performance or to minimize dependencies among developers in large projects. Pattern-driven development encouraged by modern component platforms, such as .NET^{TM} and J2EE^{TM}, leads to uniform, similar design solutions. It can also result in the fragmentation of functions in application-domain-specific areas, and substantial cloning that may complicate maintenance [18].

Automatic detection of clones is essential in analysis (or re-engineering) of large programs. Techniques for clone detection operate on programs represented as text [11], tokens [12] or abstract syntax trees [5], applying algorithms developed in string research [19] or metrics [20].

In generic programming, we achieve genericity and non-redundancy through parameterization [21]. Programming languages such as Ada, C++, Eiffel and recently Java (JDK 1.5) support some form of generics [7,22,23]. There are proposals to incorporate generics into C# [24]. In [25], generics are compared in six programming languages. Generics use type parameters to implement type-safe polymorphic container classes. Generics show limitations when we need to unify program structures other than classes, and when similar program structures differ in non-type parameters. C++ templates [7], being less restrictive than other generics, can be stretched to address non-type variations across classes or methods. We believe that the C++ template meta-programming described in [8] provides a powerful means for program manipulation, but we are not aware of studies that would evaluate the practical value of such non-standard applications of templates.

Simple clones can often be replaced by macros or functions [5]. Refactorings [26] have to do with eliminating clones using object-oriented design mechanisms. Design patterns [15] increase the genericity and flexibility of programs, helping one to avoid clones. Design patterns can be a target for refactorings. We did not try to apply design patterns to eliminate clones in the Buffer library in a systematic way. Such an experiment would be an interesting extension of our study and would be best conducted by design-pattern experts.

To address the unwanted symptoms of the 'feature combinatorics' problem, a suitable technique must achieve some degree of 'separation of concerns' in software design and implementation. In recent years, a number of approaches have been proposed along these lines of thinking, many of which are described in [8] as generative programming techniques. Aspect-oriented programming (AOP) from Xerox [27] and hyperspaces from IBM postulate that different types of program decomposition are needed to address 'cross-cutting concerns', in order to make programs easier to change. In AOP, various computational aspects are programmed separately and weaved (in terms of this paper, composed) into the base of functional program components, which are designed in the usual way. As AOP lacks a parameterization mechanism and because of its constrained composition rules, we believe AOP in its pure form is not meant for solving programming problems such as elimination of redundancies or supporting product lines. Hyperslices [28] encapsulate specific concerns and can be composed in various configurations to form custom programs. Hyperslices are written in the underlying programming language and can be composed by merging or overriding program units by name and in many other ways. To the best of our knowledge, the technique is not meant for solving problems such as that we study in this paper. In AHEAD [29] (based on the earlier work on GenVoca [2]), features are added to existing programs incrementally, through refinements.

Like AOP, the hyperspace approach and AHEAD, XVCL offers a mechanism to define alternative program decompositions at the meta-level. In XVCL, groups of inter-related meta-components may correspond to aspects (or concerns [30]). However, unlike in other approaches, XVCL's compositions are defined in an operational way and take place at designated program points marked with XVCL commands. XVCL's aspects are unconstrained in the sense that they may overlap each other or form aspect hierarchies, as one aspect may contain other aspects. XVCL's aspects can be parameterized (e.g., via meta-expressions or <select> commands), which further enhances programmer's ability to define variations in code at any level of granularity that is required, from a single program statement to a component or subsystem. This flexibility in configuring variants allowed us to define generic representations for similarity patterns of any type or granularity, and at any abstraction level.

XVCL has its roots in Frame Technology^{TM}, by Netron, Inc. Frames have been extensively applied to maintain multi-million-line COBOL-based information systems and to build reuse frameworks in companies [31].

8. CONCLUSION

We have presented a study of cloning in the Java Buffer library JDK 1.5. We have found that at least 68% of code in the Buffer library was contained in cloned classes or class methods. Close analysis of program situations that led to cloning has revealed difficulties in eliminating clones with conventional program design techniques. As a possible solution, we have applied a generative technique of XVCL to represent similar classes and methods in generic, adaptable form. Concrete buffer classes could be automatically produced from the generic structures. We argued, on analytical and empirical grounds, that unifying clones with XVCL reduced conceptual complexity, and enhanced the changeability of the Buffer library at rates proportional to code size reduction (68%). We have evaluated our solution in qualitative and quantitative ways, and conducted a controlled experiment to support this claim.

The approach presented in the paper can be used to enhance the genericity and changeability of any program, independently of an application domain or programming language. As the solution is not without pitfalls, we discussed the trade-offs involved in its project application.

In future work, we plan to conduct a systematic study of cloning in programs written in different programming languages, using different design techniques, and for different types of applications. Our studies will include both newly written and old programs. We also plan to conduct comparative studies to evaluate the effectiveness of various techniques in handling clones. We hope that others will conduct studies applying their favorite techniques to problems such as the Buffer library. In the project described in this paper, we manually analyzed code to find similar fragments. It was a tedious process. We are working on techniques to automate the search for structural clones (i.e., design-level similarities). A Clone Miner developed in our lab [13] applies data-mining heuristics to infer possible design-level similarities, as candidates for generic solutions with XVCL. We obtained some promising results, but much work lies ahead to validate initial findings.

XVCL complements rather than competes with conventional programming and design techniques. We believe that the full potential of its simple yet powerful generic design mechanism are yet to be discovered.

ACKNOWLEDGEMENTS

Thanks are due to the following NUS students: Hong Ruiling who developed XVCL solution for optimized buffer classes; and Damith Chatura Rajapakse, Pavel Korshunov and Hamid Abdul Basit who re-designed the Buffer library with JSR-14. We are grateful to the reviewers who have given us insightful suggestions that helped us better interpret and position our results, and improved the quality of the paper.

REFERENCES

1. Batory D, Singhal V, Sirkin M, Thomas J. Scalable software libraries. *Proceedings 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*. ACM Press: New York NY, 1993; 191–199.
2. Batory D, O'Malley S. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology* 1992; **1**(4):355–398.
3. Biggerstaff TJ. The library scaling problem and the limits of concrete component reuse. *Proceedings 3rd International Conference on Software Reuse (ICSR 1994)*. IEEE Computer Society Press: Los Alamitos CA, 1994; 102–109.
4. Jarzabek S, Basset P, Zhang H, Zhang W. XVCL: XML-based Variant Configuration Language. *Proceedings International Conference on Software Engineering*. IEEE Computer Society Press: Los Alamitos CA, 2003; 810–811. Available at: http://fxvcl.sourceforge.net [25 March 2006].

5. Baxter I, Yahin A, Moura L, Sant'Anna M, Bier L. Clone detection using abstract syntax trees. *Proceedings International Conference on Software Maintenance*. IEEE Computer Society Press: Los Alamitos CA, 1998; 368–377.
6. Jarzabek S, Li S. Eliminating redundancies with a 'Composition with Adaptation' meta-programming technique. *Proceedings European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2003)*. ACM Press: New York NY, 2003; 237–246.
7. Musser D, Saini A. *STL Tutorial and Reference Guide: C++ Programming with Standard Template Library*. Addison-Wesley: Reading MA, 1996; 400pp.
8. Czarnecki K, Eisenecker U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley: Reading MA, 2000; 832pp.
9. Basit HA, Rajapakse DC, Jarzabek S. Beyond templates: A study of clones in the STL and some general implications. *Proceedings 27th International Conference on Software Engineering (ICSE 2005)*. ACM Press: New York NY, 2005; 451–459.
10. Pettersson U, Jarzabek S. Industrial experience with building a Web portal product line using a lightweight, reactive approach. *Proceedings European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2005)*. ACM Press: New York NY, 2005; 326–335.
11. Ducasse S, Rieger M, Demeyer S. A language independent approach for detecting duplicated code. *Proceedings IEEE International Conference on Software Maintenance (ICSM 1999)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 109–118.
12. Kamiya T, Kusumoto S, Inoue K. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 2002; **28**(7):654–670.
13. Basit AH, Jarzabek S. Detecting higher-level similarity patterns in programs. *Proceedings European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE 2005)*. ACM Press: New York NY, 2005; 156–165.
14. Fowler M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley: Reading MA, 1997; 357pp.
15. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading MA, 1995; 395pp.
16. Kim M, Bergman L, Lau T, Notkin D. An ethnographic study of copy and paste programming practices in OOPL. *Proceedings 2004 International Symposium on Empirical Software Engineering (ISESE 2004)*. IEEE Computer Society Press: Los Alamitos CA, 2004; 83–92.
17. Cordy JR. Comprehending reality: practical challenges to software maintenance automation. *Proceedings 11th International Workshop on Program Comprehension (IWPC 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 196–206.
18. Yang J, Jarzabek S. Applying a generative technique for enhanced reuse on J2EE platform. *Proceedings 4th International Conference on Generative Programming and Component Engineering (GPCE 2005)*. Springer: Berlin, 2005; 237–255.
19. Smyth W. *Computing Patterns in Strings*. Addison-Wesley: Reading MA, 2003; 440pp.
20. Kontogiannis K. Evaluation experiments on the detection of programming patterns using software metrics. *Proceedings 4th Working Conference on Reverse Engineering*. IEEE Computer Society Press: Los Alamitos CA, 1997; 44–54.
21. Goguen JA. Parameterized programming. *IEEE Transactions on Software Engineering* 1984; **10**(5):528–543.
22. Taft S, Duff R. *Ada 95 Reference Manual: Language and Standard Libraries ISO/IEC 8652*. Springer: New York NY, 1995; 548pp.
23. Meyer B. *Object-Oriented Software Construction*. Prentice-Hall: London, 1988; 534pp.
24. Kennedy A, Syme D. Design and implementation of generics for the .Net common language runtime. *Proceedings ACM SIGPLAN 2001 Conference on Programming Languages Design and Implementation*. ACM Press: New York NY, 2001; 1–12.
25. Garcia R, Jarvi J, Lumsdaine A, Siek JG, Willcock J. A comparative study of language support for generic programming. *Proceedings 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM Press: New York NY, 2003; 115–134.
26. Fowler M, Beck K, Brant J, Opdyke W, Roberts D. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading MA, 1999; 431pp.
27. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming*. Springer: Berlin, 1997; 220–242.
28. Tarr P, Ossher H, Harrison W, Sutton S. *N* degrees of separation: Multi-dimensional separation of concerns. *Proceedings 21st International Conference on Software Engineering (ICSE 1999)*. IEEE Computer Society Press: Los Alamitos CA, 1999; 107–119.
29. Batory D, Sarvela JN, Rauschmayer A. Scaling step-wise refinement. *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*. IEEE Computer Society Press: Los Alamitos CA, 2003; 187–197.
30. Zhang HY, Jarzabek S, Soe MS. XVCL approach to separating concerns in product family assets. *Proceedings 3rd International Conference on Generative and Component-based Software Engineering*. Springer: London, 2001; 36–47.
31. Bassett P. *Framing Software Reuse: Lessons From the Real World*. Prentice-Hall: Upper Saddle River NJ, 1997; 365pp.

AUTHORS' BIOGRAPHIES

Stan Jarzabek is an Associate Professor at the Department of Computer Science, School of Computing, National University of Singapore (NUS). He is interested in all aspects of software design, in particular techniques for the design of adaptable, easy to change (high-variability) software. He received Master and PhD degrees from Warsaw University.

Shubiao Li is an Associate Professor at the Department of Banking Information Engineering, School of Economics and Finance, Xi'an Jiaotong University, China. He received his PhD degree from Xi'an Jiaotong University in 2003. He focuses on object-oriented technologies, software engineering and management information systems, in particular applications in banking and finance industries.