

# Eliminating Redundancies with a “Composition with Adaptation” Meta-programming Technique

Stan Jarzabek

Department of Computer Science, School of Computing,  
National University of Singapore  
Lower Kent Ridge Road  
Singapore 117543  
Tel: (65) 6874 2863  
stan@comp.nus.edu.sg

Li Shubiao

Department of Banking Information Engineering  
School of Economics and Finance  
Xi'an Jiaotong University  
Xi'an 710061, China  
Tel: (86) 295276949  
li\_shubiao@163.net

## ABSTRACT

Redundant code obstructs program understanding and contributes to high maintenance costs. While most experts agree on that, opinions - on how serious the problem of redundancies really is and how to tackle it - differ. In this paper, we present the study of redundancies in the Java Buffer library, JDK 1.4.1, which was recently released by Sun. We found that at least 68% of code in the Buffer library is redundant in the sense that it recurs in many classes in the same or slightly modified form. We effectively eliminated that 68% of code at the meta-level using a technique based on “composition with adaptation” called XVCL. We argue that such a program solution is easier to maintain than buffer classes with redundant code. In this experiment, we have designed our meta-representation so that we could produce buffer classes in exactly the same form as they appear in the original Buffer library. While we have been tempted to re-design the buffer classes, we chose not to do so, in order to allow for the seamless integration of the XVCL solution into contemporary programming methodologies and systems. This decision has not affected the essential results reported in this paper.

## Categories and Subject Descriptors

D.2.2 [SOFTWARE ENGINEERING]: Design Tools and Techniques - *Software libraries*; D.2.13 [SOFTWARE ENGINEERING]: Reusable Software - *Reusable libraries*

## General Terms

Performance, Design, Languages

## Keywords

meta-programming, generative programming, class libraries, Object-Oriented methods

## 1. INTRODUCTION

As early as in 1993, Batory et al. [2] described the “feature combinatorics” problem hampering the scalability of class/component libraries, reuse and programmers’ productivity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESEC/FSE'03, September 1–9, 2003, Helsinki, Finland  
Copyright 2003 ACM 1-58113-743-5/03/0009...\$5.00.

in general. Batory studied C++ data structure class libraries, where typical features relate to data structure, memory allocation scheme, access mode, concurrency, etc. In general, features may relate to any characteristic of a program such as functionality, design solution, platform, etc. 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 concludes 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 applied a generation technique of GenVoca [3] to produce classes with 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 limits of concrete component reuse caused by “feature combinatorics” [4].

The “feature combinatorics” problem also contributes to high maintenance cost. As the number of feature combinations increases, classes in a library not only grow in number, but also become polluted with numerous redundant code fragments. Redundant code obstructs 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 redundancies be removed, and (2) when one logical source of change affects many replicated code fragments scattered throughout a program, to implement a change, a programmer must find and update all the instances of the replicated fragment. The situation is further complicated if an affected fragment must be changed in slightly different ways, depending on the context.

The above arguments apply to components and programs in general, as well as classes. As components usually contain more functionality than classes, the number of feature combinations in components is even larger than in classes. Therefore, we shall see an explosion of look-alike components that, despite similarities, have to be implemented and maintained as separate products containing much redundancy.

In this paper, we address one specific symptom of the “feature combinatorics” problem, namely that of the redundant code. In the first part of the paper, we present results of the redundant code analysis in the Java Buffer library, JDK 1.4.1, pointing to common sources of redundancies. In the second part of the paper, we describe an alternative way of building a class library, with a

meta-programming technique called XVCL<sup>1</sup>. We developed XVCL to facilitate building flexible, adaptable and reusable software. The XVCL solution consists of class building blocks (meta-components) and an automated class construction process during which meta-components are composed incorporating possible adaptations. Our experiment shows that at least 68% of the executable code in the Buffer library is redundant and can be eliminated at the meta-level. If we count both executable code and comments, then we can eliminate 72% of code. Due to the smaller base of non-redundant code and better traceability from features to code, the Buffer library in XVCL meta-representation is easier to maintain than the original Buffer library in the source form.

In the experiment, we produced buffer classes in exactly the same form as they appear in the original Buffer library. While we have been tempted to re-design the buffer classes, we chose not to do so, to allow for seamless integration of the XVCL solution into contemporary programming methodologies and systems. Our solution can serve those who develop and maintain libraries, without affecting programmers using libraries. The fact that we produced classes in their original form did not affect essential results reported in this paper.

It seems that some of the design concerns, such as the reuse or handling of scattered code, are much easier to deal with at the program meta-level rather than at the level of concrete programs. In recent years, a number of approaches have been proposed to address such design concerns [3][8][10][14]. These approaches can also be applied to alleviate some of the effects of the “feature combinatorics” problem. Many of these approaches are described in [5] as “generative programming techniques”. A common motivation for these approaches may be phrased as follows: The number of concrete classes (or components) is potentially large and we cannot help it. But we can ease the problem by providing a suitable construction-time mechanism to synthesize concrete classes on demand from a relatively simpler base of generic meta-components. While feature combinations are inevitable in concrete classes, at the meta-level we can strive to separate different feature dimensions (also called aspects or concerns) and provide a meta-level mechanism to produce classes/components with the required combination of features on demand. In this paper, we show that, along those lines of thinking, interesting results can be achieved with a simple “composition with adaptation” technique.

Being a modern and versatile version of Bassett’s frames [1], a technology that has achieved substantial productivity improvements in industry, the underlying principles of XVCL have been thoroughly tested in practice. Unlike the original frames, however, XVCL blends with contemporary programming and design paradigms, offering an effective reuse mechanism on top of mechanisms supported by those paradigms. XVCL works on the principle of adapting generic, reusable meta-components into concrete components. Any location or structure in a meta-component can be a designated variation point, available for adaptation by ancestor meta-components. Program generation is 100% transparent to the programmer, who can fine-tune and re-

<sup>1</sup> XVCL: XML-based Variant Configuration Language, is a public domain method and tool available at: [fxvcl.sourceforge.net](http://fxvcl.sourceforge.net)

generate code without losing prior customizations. We developed and experimented with XVCL in a collaborative project involving two universities and two industrial partners<sup>2</sup>. We applied XVCL in two medium-size product line projects [16] and a number of smaller case studies [6][13]. In earlier papers, we described experimentation with the original frame technology in the context of product line architectures and component-based systems.

This paper is organized as follows: In Sections 2-4, we provide statistics of the redundant code in the Buffer library and discuss the reasons why the redundancies arose. In Section 5, we describe the XVCL solution. In Section 6, we compare the original Buffer library with the XVCL solution and in Section 7, we discuss the results. In the remaining sections, we discuss related work and conclude 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 1.4.1. Buffer classes differ in features such as the buffer element type, memory allocation scheme, byte ordering and access mode (Table 1). Each legal combination of features yields a unique buffer class. That is why, even though all the buffer classes play essentially the same role, there are 74 classes in the Buffer library.

**Table 1. Features in the Buffer library**

Level in class hierarchy	Feature dimension	Features
Level 1	buffer data element type	byte, char, int, float double, long, short
Level 2	memory allocation scheme	direct, nondirect
	byte ordering	native, non-native, <b>Big_endian</b> , <b>Little_endian</b>
Level 3	access mode	writable, read-only

Figure 1 shows a part of the Buffer library with 49 classes that we discuss in detail in the paper. Class sub-hierarchies for classes `DoubleBuffer`, `FloatBuffer`, `LongBuffer`, `ShortBuffer` are analogical to sub-hierarchies for classes `CharBuffer` and `IntBuffer`, so for the sake of brevity we do not depict them in Figure 1. Below, we briefly describe features addressed in the Buffer library and explain how those features are reflected in classes.

At Level 1 in the class hierarchy, 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

<sup>2</sup> Project funded by Singapore National Science and Technology Board and Canadian Ministry of Energy, Science and Technology, involving National University of Singapore, SES Systems Pte Ltd, University of Waterloo and Netron, Inc. (Toronto).

access methods to read and write buffer elements, using a native or non-native byte ordering scheme. On the other hand, in the nondirect memory allocation scheme, we access a buffer through Java array accessor methods.

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

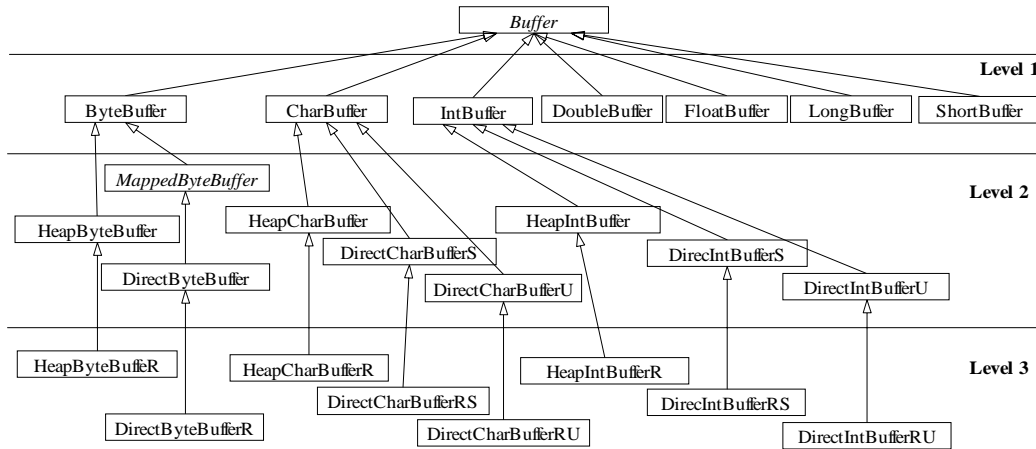


Figure 1. A fragment of the Buffer library

When using the direct memory access 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 access 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 nondirect memory access scheme for that buffer. Classes with suffixes ‘U’ and ‘S’ implement direct memory access 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. STATISTICS OF REDUNDANCIES IN BUFFER CLASSES

Code fragments that we studied typically contain definitions of classes, class attributes, constructors or methods. Redundancy occurs among similar code fragments. It is difficult to define redundancy in general and descriptive terms. Therefore, we shall accept the following pragmatic definition of redundancy for the purpose of this paper: Redundancy occurs in a group of similar code fragments whenever we can unify all the differences among those fragments at the meta-level. We also require that the result is beneficial for maintenance, that is, such a unified meta-representation should be easier to understand and maintain than the original program with redundant code fragments.

Among the classes at Level 1, we identified 30 code fragments, recurring in either the same form or with slight changes. In Figure 2, circles marked with (a), (b) and (c) represent classes as indicated in the diagram. Notice that Circle (b) represents five classes. In the overlapping areas, we indicate fragments recurring in the respective classes. We refer to the overlapping areas by

listing the circles involved such as (ab), (abc), etc. In Figures 2 and 3, ‘A’ refers to class attributes, ‘C’ – to constructors and ‘M’ – to methods. Symbol ‘s’ denotes fragments recurring in the same form, ‘c’ - fragments recurring with slight changes.

For example, Area (ab) contains one method recurring in the same form (short form M:1s) in all classes but *CharBuffer*. The central Area (abc) contains 28 fragments that recur in all seven classes. These fragments include:

- o 2 attribute definition sections recurring in the same form and 1 attribute definition section recurring with slight change (in short notation: A:2s, 1c)
- o 2 constructor definitions recurring with slight change (in short notation: C:2c)
- o 23 method definitions recurring in the same form or with slight change (in short notation: M:4s, 19c).

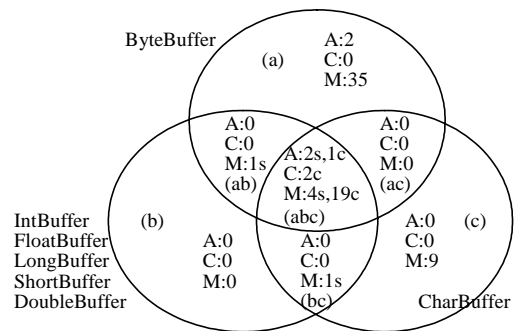


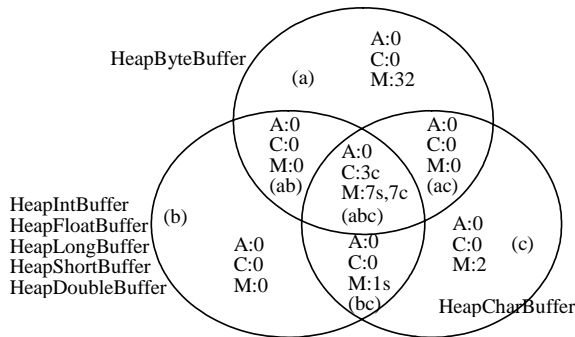
Figure 2. Distribution of code fragments in classes at Level 1

Non-overlapping areas represent unique fragments in the respective classes. For example, *ByteBuffer* class has 37 unique fragments including two attribute definition sections and 35 method definitions (A:2 C:0 M:35), while *IntBuffer* class has no unique fragments.

The five classes in Circle (b) are very similar, in the sense, that they are built from virtually the same group of fragments, with only slight differences among fragments recurring in different classes. The reader should refer to relevant entries in Table 2 for statistics of classes at Level 1.

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 sub-classes of ByteBuffer and three new sub-classes for each of the remaining classes at Level 1.

Figure 3 shows the distribution of both unique and redundant code fragments in classes implementing the nondirect memory allocation scheme (seven classes named Heap\* in Figure 1). In the central Area (abc), there are three constructor definitions recurring in all seven classes with small changes. Also, there are seven methods recurring in all the classes without any change, one method recurring in six classes except HeapByteBuffer (Area (bc)), and seven methods recurring with small changes in all seven classes.



**Figure 3. Distribution of code fragments in Heap\* classes at Level 2**

Classes implementing the direct memory allocation scheme (named Direct\* in Figure 1) also contain many redundant code fragments. Due to the lack of space, we skip the details of the analysis of the remaining classes. Table 2 shows the distribution of the redundant code fragments across the different slices of the Buffer library and the frequency in which they recur.

#### 4. WHY DO REDUNDANCIES ARISE?

To simplify the use of the Buffer library, the designers decided to reveal to programmers only the top eight classes (Figure 1). Functionalities related to lower-level concrete classes can be accessed via methods provided in these classes. While this and other concerns related to usability and performance affected the design of the buffer classes and introduced some extra code, we did not find evidence that those concerns led to redundant code. On the other hand, we found that addressing feature combinations led to redundancies in the respective classes. As features cannot be implemented independently of each other in separate implementation units (e.g., class methods), code fragments related to specific features appear with many variants in different classes, depending on the context. Whenever such code cannot be parameterized to unify the variant forms, and placed in some upper-level class for reuse via inheritance, a redundancy arises.

**Table 2. Buffer library statistics**

Classes	Fragments					Unique	LOC	
	Recurring fragments							
	times	2	6	7	12			13
<b>level 1</b> (7 classes)	same form		2	6			46	4534
	small changes			22				
<b>level 2 Heap*</b> (7 classes)	same form		1	7			34	1151
	small changes			10				
<b>level 2 Direct*</b> (13 classes)	same form	1				8	50	2320
	small changes	1			2	11		
<b>level 3 Heap*</b> read-only (7 classes)	same form		1	1			22	704
	small changes			11				
<b>level 3 Direct*</b> read-only (13 classes)	same form	1				2	30	1207
	small changes	1			2	8		
<b>subtotal for 47 classes</b>	same form	2	4	14		10	182	9916
	small changes	2		43	4	19		
<b>other classes at level 2</b> (12 classes)	same form	1				4	0	1244
	small changes	1			12			
<b>other classes at level 3</b> (12 classes)	same form	1				2	0	764
	small changes	1			9			
<b>subtotal for other classes</b>	same form	2				6	0	2008
	small changes	2			21			
<b>total</b>	same form	4	4	14	6	10	182	11924
	small changes	4		43	25	19		

To observe the impact of feature combinations on redundancies, 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 redundant code 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 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. But 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 another situation, a certain method is needed in all the classes derived from class A, but in some of those classes the method requires different parameters, return type or implementation that 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 redundant code.

Method `hasArray()` shown in Figure 4 illustrates a simple yet interesting case. This method is repeated 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` – `int`.

```

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

```

**Figure 4. Recurring method `hasArray()`**

Many redundancies 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 or constants. Such situations are easily handled by templates. For example, 15 classes represented as Circles (b) in Figure 2 and 3 can be implemented by templates. The current release of Java does not support templates, but JSR-14 is likely to become a part of Java soon (<http://www.jcp.org/en/jsr/>). In the related project, we built JSR-14 templates for classes that differed only in the buffer element type, eliminating 27% of code. However, we found it impossible to integrate template-based classes with the rest of the buffer classes because of tight coupling among classes across the library.

```

/*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 5. Method `slice()`**

In some situations, to unify similar fragments, we would need parameters representing algorithmic elements rather than data types. In yet other situations, we have to do with many small differences across implementations of the same method in different classes. We observe this in the classes in Circles (a) and (b) in Figure 2 and 3. For example, method `slice()` recurs 13 times in all the `Direct*` classes with small changes highlighted in bold (Figure 5). This happens when the impact of various features overlaps in code fragments, affecting data type names, constant values or details of algorithms. Template mechanism supported by JSR-14 is not meant to unify this kind of differences in classes. In JSR-14, template parameters cannot be primitive types such as `int` or `char`. This is a serious limitation, as one has to create wrapper classes just for the purpose of parameterization. Wrapper classes introduce extra complexity and hamper performance, so it is unlikely that we shall see the implementation of the Buffer library using JSR-14. Templates in other languages, for example, C++ are free of these limitations.

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 adequate (that is a sufficiently small) granularity level. Therefore, the impact of a small variation on a program solution is often not proportional to the size of the variation.

## 5. CONSTRUCTION OF BUFFER CLASSES WITH XVCL

We shall now show how the Buffer library can be produced from a much smaller base of meta-code using XVCL. Our objective in this experiment is to construct classes in the same form as they appear in the original Buffer library.

Based on the analysis of redundancies described in the previous sections, we identified seven groups of similar classes, namely:

1. `[T]Buffer`: 7 classes at Level 1 that differ in buffer element type, **T**: `Byte`, `Char`, `Int`, `Double`, `Float`, `Long`, `Short`
2. `Heap[T]Buffer`: 7 classes at Level 2, that differ in buffer element type, **T**
3. `Heap[T]BufferR`: 7 read-only classes at Level 3
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 (notice 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 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 for combinations of parameters **T**, **B** and **L**, as above.

For each of the above groups, we designed meta-components to generate classes in a given group. The overall structure of meta-components is outlined in Figure 6. The top-most meta-component, called *SPC*, specifies how to construct all the buffer classes. Below, we see a layer of meta-components called *meta-classes*. Meta-classes correspond by name to seven groups of similar classes and each meta-class facilitates generation of classes in its group. (In Figure 6, we showed only three out of seven meta-classes.)

The rest of the meta-components, called *meta-fragments*, are class building blocks, “normalized” to eliminate redundancies. Meta-fragments represent both unique and recurring fragments of class definitions, related to various features.

Meta-components contain Java code inter-mixed with XVCL commands. XVCL commands indicate how a meta-component can adapt meta-components below it (Figure 6), and how a meta-component can be adapted by meta-components above it. After adaptation, a child meta-component is included into the parent meta-component (as indicated by `<adapt>` arrows). For example, the *SPC* includes after possible adaptations meta-classes `[T]Buffer`, `Heap[T]Buffer` and `Direct[T]Buffer[S|U]`.

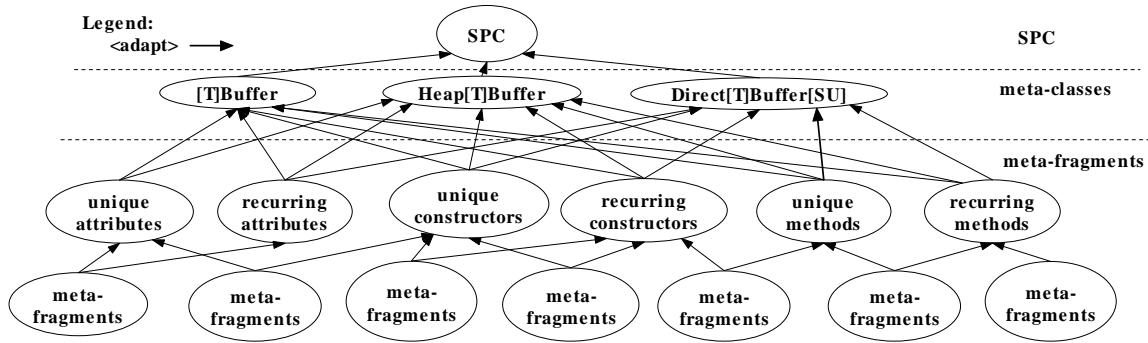


Figure 6. A fragment of meta-component architecture for the Buffer library

The XVCL processor traverses the meta-component architecture as indicated by `<adapt>` commands in depth-first order, starting with the SPC. (For readability, we enclose XVCL commands in angle brackets.) For each visited meta-component, the processor interprets XVCL commands embedded in that meta-component and generates source code for classes (Figure 7).

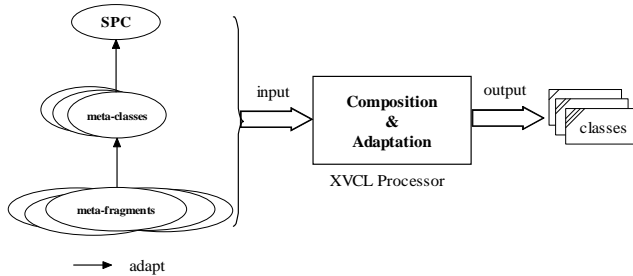


Figure 7. Class construction with XVCL

Code fragments that recur in buffer classes without changes, such as method `hasArray()` in Figure 4, are included “as is”. However, other meta-fragments must be adapted for reuse in a given context. Adaptations are achieved by means of parameterization via meta-variables and meta-expressions, insertions of code and specifications at designated break points, selection among given options based on conditions, code generation by iterating over sections of meta-components, etc.

```

meta-fragment name: slice
text
/*Creates a new byte buffer containing a shared
subsequence of this buffer's content.*/
public @elmtTypeBuffer 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@elmtTypeBuffer@byteOrder
        (this, -1, 0, rem, rem, off); }

```

Figure 8. Meta-fragment slice.xvcl

Parameterization via meta-variables and meta-expressions plays an important role in building generic, reusable programs. It provides the means for creating generic names and controlling the traversal and adaptation of a meta-component architecture. For example, we parameterized method `slice()` (Figure 5) with meta-variables as shown in Figure 8.

A reference to a meta-variable, such as `@elmtType`, is replaced by the meta-variable’s value during processing. The value of meta-variable `elmtType` may be `<set>` to `Byte`, `Int`, `Char`, etc., as required at the adaptation point. For example, to produce method `slice()` for class `DirectByteBuffer`, we `<set>` the value of meta-variable `elmtType` to “Byte”, and for classes `DirectIntBufferS` and `DirectIntBufferU` - to “Int”. A meta-expression `Direct@elmtTypeBuffer@byteOrder` in Figure 8 allows us to generate names for all the `Direct*` classes at Level 2 (Figure 1).

```

meta-class name: [T]Buffer
text
package @packageName;
public abstract class @elmtTypeBuffer extends
    Buffer implements Comparable
text
{ // attributes and methods here
break
toString
    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(); }
text
}

```

Figure 9. Meta-class [T]Buffer

Meta-class `[T]Buffer` facilitates generation of all the 7 buffer classes at Level 1 (Figure 1). As the reader may recall from Section 4, class `CharBuffer` requires different implementation of method `toString()` than the remaining 6 classes at Level 1. This situation is handled by a `<break>` point and `<insert>` command. The `<break>` point `toString` contains default implementation of method `toString()` that is used in 6 classes. When generating class `CharBuffer`, we override the default implementation of method `toString()`. This is done in the SPC (Figure 10) - `<adapt>` command for option “Char” contains a proper `<insert>` command. In general, higher-level meta-components can replace the default code or insert extra code after/before the `<break>` points in `<adapt>`ed meta-components.

The SPC of Figure 10 controls the overall process of generating all the buffer classes. First, the SPC `<set>`s the value of meta-variable `packageName` to “java.nio” and the value of meta-variable `elmtType` to `<Byte,Char,Double,Float,Int,Long,Short>`. Values of those meta-variables are propagated down to the `<adapt>`ed meta-components.

<i>name</i> : SPC	
<b>set</b>	<i>packageName</i> = "java.nio"
<b>set</b>	<i>elmType</i> =<Byte,Char, Double, Float, Int, Long, Short>
<b>while</b>	<i>using-items-in=elmType</i>
<b>select</b>	<i>option="elmType"</i>
	Byte <b>adapt</b> [T]Buffer
	<b>adapt</b> [T]Buffer
	<b>insert</b> <i>toString</i>
	<i>Public String toString()</i>
	<b>text</b> { <i>return toString(</i>
	<i>position(),limit());</i>
	}
	otherwis e <b>adapt</b> [T]Buffer

Figure 10. SPC to construct classes at Level 1

The value of *elmType* is a list. Command <while> iterates over its body seven times. In each iteration, *elmType* accepts one value from the list, in the left-to-right order. Based on that value, the processor <select>s a suitable option (such as Byte, Char or otherwise) and generates code for appropriate class(es) (**ByteBuffer**, **CharBuffer** and all the remaining classes, respectively). Generation is done by <adapt>ing the meta-class [T]Buffer. To generate class **CharBuffer**, we override the default implementation of method **toString()**. Class **ByteBuffer** requires extra methods that are <insert>ed at the adaptation point (details not shown in Figure 10). All the remaining classes at Level 1 require the same adaptations, as shown in otherwise option of <select>.

Due to space limitation, we only highlighted the structure of the solution and basic concepts of the XVCL's "composition with adaptation" mechanism. In XVCL, meta-components are encoded as XML files, with XVCL commands expressed as XML tags [15]. In the above examples, for readability, we showed XML-free, tabular views of meta-components as produced by the XVCL Workbench, a productivity tool being developed at NUS.

The reader may find a description of the XVCL and its implementation in [15]. Full specifications of XVCL and its processor's source code can be downloaded from [fxvcl.sourceforge.net](http://fxvcl.sourceforge.net). A tutorial paper [13] gives a friendly introduction to XVCL concepts. Complete documentation and code for the Buffer library case study can be found at [fxvcl.sourceforge.net](http://fxvcl.sourceforge.net) in "Case Studies".

We would like to end this section by summarizing the process of designing the meta-component architecture for buffer classes. We started by analyzing types of the redundant code fragments in buffer classes, which led us to identifying seven major groups of similar classes. We designed meta-components for each group of classes, eliminating redundant code fragments as follows: For each group of similar fragments, we created a suitable meta-fragment. As for meta-fragments that appeared in different contexts with changes, we parameterized them with meta-variables, <break> points, <select>, <insert>, <adapt> and other XVCL commands to cater for required variations. Finally, we incorporated suitable adaptation commands to the corresponding meta-classes and SPC.

## 6. THE ORIGINAL BUFFER LIBRARY VERSUS THE XVCL SOLUTION

Table 3 and Figure 11 show the results of comparing the original Buffer classes with the XVCL solution.

Table 3. Original Buffer library vs. XVCL solution

classes	original Buffer library			Buffer library in XVCL		
	fragments	LOC <sup>1</sup>	Java Code <sup>2</sup>	meta-components	LOC <sup>3</sup>	Java Code <sup>4</sup>
<b>level 1</b> (7 classes)	254	4534	1108	76	1008	308
level 2 Heap* (7 classes)	159	1151	948	52	399	364
<b>level 2 Direct</b> * (13 classes)	325	2320	2104	73	753	729
<b>level 3</b> <b>Heap* read-only</b> (7 classes)	112	704	578	35	306	282
<b>level 3</b> <b>Direct * read-only</b> (13 classes)	188	1207	1113	44	503	494
<b>subtotal</b> (47 classes)	1038	9916	5851	280	2969	2177
<b>other classes at level 2</b> (12 classes)	196	1244	1146	18	193	184
<b>other classes at level 3</b> (12 classes)	136	764	666	13	153	144
<b>subtotal for others</b> (24 classes)	332	2008	1812	31	346	328
<b>total</b>	1370	11924	7663	311	3315	2505

<sup>1</sup>including Java code and comments. <sup>2</sup>only including Java code  
<sup>3</sup>including Java code, comments, and XVCL instructions  
<sup>4</sup>including Java code, and XVCL instructions

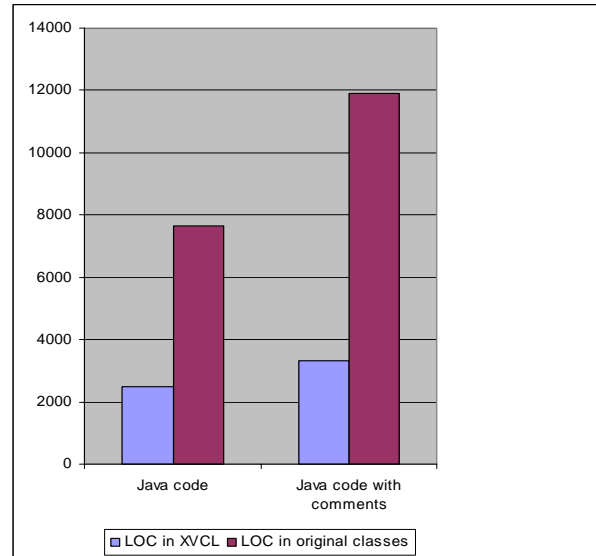


Figure 11. Buffer library: summary chart

The XVCL solution eliminates 68% of the code as compared to the original buffer classes. As both code and comments are needed to understand a program, and both must be maintained, the size of code with comments is a better indicator of maintenance effort than code alone. It is possible and useful to manage both executable code and comments with XVCL. If we count executable code with comments, the XVCL solution eliminates 72% of the code.

## 7. DISCUSSION OF THE RESULTS

The XVCL solution described in this paper is meant for developers and maintainers of complex, evolving class libraries. As a class library produced from meta-components is no different from the original class library, programmers reusing the library need not be concerned or even aware that the library is managed with XVCL. On the other hand, some programmers may also wish to incorporate elements of the XVCL technique into their mainstream programming work. Especially, programmers working in unstable domains, where change is pervasive, may see good reasons to do so. In such cases, meta-components of the class library can be neatly integrated with programs reusing those libraries at the meta-level. This option also opens the possibility of re-designing concrete classes, as we have done in another project (we discuss this in more details below).

The main return on investment in applying XVCL is from savings in program maintenance. The principle of XVCL design is clean separation of various sources of change affecting a program. Each source of change (e.g., variation of the features listed in Table 1) can be traced to codes affected by this change. Lack of the redundant code reduces the number of points at which affected classes must be modified. Changes done to one meta-fragment consistently propagate to all the contexts into which the meta-fragment is adapted. If the impact of change is not uniform in all such contexts, the exceptions can be handled at the specific adaptation point, without directly modifying the code fragment involved. The meta-component architecture explicitly reflects the impact of change on program elements. From each meta-class we can trace how different feature combinations affect the code.

Despite those benefits, design in terms of meta-components is not easy and different from the inheritance-based program design. For a skilled OO programmer familiar with the application domain (e.g., buffer classes), the development of an XVCL meta-component architecture takes longer than the development of pure OO class library. The development of a meta-component architecture starts by designing a program runtime architecture and developing a default, simplified program. A meta-component architecture is then developed in iterations, starting with the default program. Each iteration applies XVCL to extend the meta-component architecture with new features, refine existing meta-components and create new generic, reusable meta-components. It takes some time to adjust to this way of thinking about a program. But as this perspective so well addresses concerns that matter in maintenance, eventually this shift of the viewpoint pays off.

To better understand the results of this study, we conducted two related experiments:

1. In the first experiment, we produced buffer classes with the same functionality as the original ones, but optimized for memory consumption and speed. Each concrete class implementing a specific feature combination is complete in the sense that it can be used without any other classes. Therefore, each class includes all the required methods and classes are not related by means of inheritance. We envision application of this Buffer library solution in time-critical and embedded systems. As in the experiment described in this paper, we eliminated redundant code at the meta-level, obtaining above 60% reduction of the code size as compared to the original Buffer library. At least in those two

experiments, we observed similar code reduction at the meta-level independently of the structure of generated classes.

2. In the second experiment, we studied JSR-14, Java with templates, in the context of the Buffer library. Templates are meant mainly for defining generic containers. We re-designed 15 template-friendly buffer classes with JSR-14, eliminating 27% of the code. However, we found it impossible to integrate the re-designed, template-based classes with the rest of the buffer classes because of tight coupling among classes. The reader may find more details about the template solution in section 4. On the overall, we think that the practical value of templates in elimination of redundant code in tightly coupled classes is rather limited.

It would be interesting to compare our results with the results obtained with other techniques applied to the same Buffer library. We believe template-based generative techniques [5], GenVoca [3], Aspect-Oriented Programming [8] and multi-dimensions hyperspaces [14] may be effective in eliminating redundancies. Some redundancies could be eliminated with multiple inheritance, delegation, design patterns and other design techniques. But how effective would be these solutions in reducing program complexity and improving program maintainability? Another interesting problem, suggested by one of the reviewers, is to study how much redundancy is induced by the limitations of the underlying programming language. We are planning to address the above open problems in future research. We hope this paper will encourage others to conduct similar studies using their favorite techniques.

## 8. RELATED WORK

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. It is easier to achieve separation of concerns at the meta-level rather than at the level of concrete program components. Using meta-level techniques, custom components with the required combination of features are synthesized from a set of primitive meta-components. The techniques differ in the nature of meta-components and in how synthesis is done to produce a concrete component. Below, we briefly contrast XVCL with other meta-techniques.

Generators [11] produce custom programs from problem specifications in domain-specific languages. Domain-specific languages can be developed in well-understood and stable domains. As problem specifications can be very compact, generators yield higher productivity gains than XVCL. Generators are also more effective than XVCL when we require sophisticated domain-specific optimizations [3][10][11]. On the other hand, XVCL is an application domain-independent language, method and tool. XVCL performs best in immature, poorly understood and evolving domains and in domains where frequent changes occur at both large and small granularity levels. Unlike in many generators, a programmer can modify any detail of the program solution and the required code changes are always proportional to the change in the problem domain.

Macro systems are probably the oldest form of meta-programming. Macros handle variant features only at the implementation level, which causes well-known problems with

understanding programs instrumented with macros [7]. Even though XVCL commands, like macros, instrument programs for change, capabilities of XVCL in handling variants reach far beyond the capabilities of macro systems. XVCL is a full-fledged design method, in which variant features are directly addressed at both program design and implementation levels. Over time, an XVCL meta-component architecture emerges as a well-organized architecture that explicates the impact of variant features on components and automates production of custom components. XVCL has unique features to support reuse and evolution such as propagation of meta-variables across meta-components, meta-variable scoping rules that allow us to adapt generic meta-components at inclusion points, meta-expressions to formulate generic names, code selection or insertion at designated breakpoints and a while loop construct to implement generators.

Configuration Management (CM) systems have been applied to handle variant features in software. Like an OO library, for each legal combination of features, a CM system maintains a separate component version. CM systems are strong in handling variants at the file level but weak in handling small, inter-dependent variants, spreading over many components. In XVCL, we capture component variability specifications at the meta-level separately from the components themselves, and we can configure variants at any level of granularity.

In Aspect-Oriented Programming (AoP) [8], each computational aspect is programmed separately and weaved into the base code. AoP composition rules are specified in a descriptive, easy to grasp way, but compositions can occur only at join points supported by the AoP system. XVCL composition rules are defined in an operational way, and therefore more difficult to grasp, but compositions may occur at arbitrary break points. AoP was designed to deal with reasonably big chunks of functionality (aspects) and lacks a mechanism to handle small variations. While it is possible to define an aspect within another aspect, probably the result would be rather complicated. XVCL, on the other hand, can deal with variations at any granularity level, using a uniform, yet simple mechanism.

In the hyperspace approach [14], hyperslices encapsulating computational aspects can be composed in various configurations to form specific programs. In XVCL, we achieve separation of concerns by placing code related to different computational aspects into different meta-component layers [16].

## 9. CONCLUSION

It is well known that redundant code obstructs program understanding and maintenance. Yet, programs are often polluted by such code. In some cases, redundancy is created on purpose, for example, to increase the robustness of life-critical systems or to minimize dependencies among developers in large projects [12]. In other cases, redundancies do not play any positive role and are created during maintenance, new development (due to inadequacy of programming languages and design techniques) or generated by tools. Whatever the reason, redundancy obstructs program understanding and maintenance.

While we may not be able to eliminate all the redundancies in executable programs, the good news is that redundancy can be effectively dealt with at the meta-level. In this paper, we described the results of the study of the redundant code in the

Java Buffer library, JDK 1.4.1. We found that more than 68% of code in the Buffer library is redundant in the sense that it recurs in many classes in the same or slightly modified form. We effectively eliminated that 68% of code at the meta-level, using a technique based on “composition with adaptation”, called XVCL. We argued that such a program solution is easier to maintain than the buffer classes with redundant code. In this experiment, we designed our meta-representation so that we could produce buffer classes in exactly the same form as they appear in the original Buffer library. While we have been tempted to re-design the buffer classes, we chose not to do so, to allow for seamless integration of the XVCL solution into contemporary programming methodologies and systems. This decision did not affect essential results reported in this paper.

In other experiments, XVCL achieved code reductions of:

- o 60% when generating buffer classes optimized for memory consumption and speed,
- o 68% in n-tier application (C#), and
- o 61% in Data Access component of an n-tier application, developed using MS ADO in MS VC++.

In XVCL, we produce classes by composing meta-components with possible adaptations. In the paper, we described both the general concepts of the “composition with adaptation” technique and its realization with XVCL.

*Strengths:* XVCL allows us to develop and maintain class libraries from a small, non-redundant base of meta-components. The meta-component architecture provides a clear view on how feature combinations and other changeable requirements affect the code. We can eliminate redundancies at the meta-level, thus simplifying maintenance, and still keep redundant code in executable programs, if it is so required. One of the reasons for many redundant code fragments in class libraries is that inheritance does not support fine-grain reuse – small change in requirements may lead to many changes in code. XVCL supports reuse at any level of granularity that is needed - small changes in requirements trigger equally small amounts of re-work in meta-components. The XVCL solution is simple and transparent - all the codes that we see in the final Java classes also appear in meta-components. A programmer can intervene in any details of the transition from the meta-level to programs. We can re-design classes or produce classes in the same form as in the original library. XVCL complements rather than competes with programming languages and other design paradigms. A developer can switch from the OO paradigm to XVCL to deal with certain problems in a more efficient way. Therefore, the XVCL solution can be neatly integrated into other programming methodologies and environments. XVCL is a comprehensive design method, leading to compact program solutions that are structured to maximize flexibility, reuse and ease of change.

*Weaknesses:* There are well-known problems with understanding programs heavily instrumented with macros [7] and meta-programs in general. XVCL meta-components also contain code instrumented with commands. This problem is mitigated by the fact that, unlike macros that merely complement a programming language, XVCL is a full-fledged design method supported by tools. XVCL meta-components are first-class design concepts that facilitate change. Meta-components are organized into a layered architecture that strives to achieve separation of concerns. XVCL is supported by tools that produce adaptation traces and help

debug meta-component architectures: For a given SPC, one can follow the sequence of adapted meta-components and analyze detailed adaptations. We implemented a tool for “round trip” engineering that helps a programmer propagate changes which are made directly to the generated program, back to the affected meta-components. A big challenge in meta-programming is how to validate code generated from meta-components. Correctness of produced code is not guaranteed by XVCL. This problem is mitigated by the fact that lower levels of the meta-component architecture get pretty stable and reliable over time. Potential errors tend to be located only in top-most, context-specific and still fragile meta-components. In the project described in this paper, we manually analyzed code to find similar fragments. It was a tedious process. We plan to explore existing techniques in duplicated code detection in order to automate the search of groups of similar code fragments as candidates for meta-components.

Redundant code obstructs program understanding and contributes to high maintenance costs - the evidence abounds but is mostly anecdotal. We did not find recent studies on redundancies. A study conducted in 1984 reports that redundancy ranges from 50% to 85% [9]. Re-engineering experts say that around 40% of code in systems they examine is redundant [12]. In future work, we plan to conduct a systematic study on redundancies in programs written in different programming languages, using different design techniques, and for different types of applications. Such a study should include both newly written and old programs. We also plan to conduct comparative studies to evaluate the effectiveness of various techniques in handling redundancies. We hope that others will conduct studies applying their favorite techniques to problems such as the Buffer library.

Meta-techniques based on “composition with adaptation”, such as XVCL, complement rather than compete with the design techniques offered by programming languages. We believe that “composition with adaptation” is a simple yet powerful programming technique whose potentials have yet to be discovered. Our research group is strongly committed to exploring this potential.

## 10. ACKNOWLEDGEMENTS

Thanks are due to the following NUS students: Hong Ruiling who developed XVCL solution for optimized buffer classes; Damith Chatura Rajapakse, Pavel Korshunov and Hamid Abdul Basit who re-designed the Buffer library with JSR-14. We are grateful to reviewers who have given us insightful suggestions and pointed to interesting and important open problems related to our work. This research was supported by National University of Singapore Research Grant R-252-000-066-112.

## REFERENCES

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

- [2] 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
- [3] Batory, D. and Geraci, B.J. “Validating component compositions and subjectivity in GenVoca generators,” *Trans. on Software Engineering*, 23, 2, 1997, pp. 67-82
- [4] Biggerstaff, T. “The library scaling problem and the limits of concrete component reuse,” 3<sup>rd</sup> Int. Conf. on Software Reuse, ICSR’94, 1994, pp. 102-109
- [5] Czarnecki, K. and Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000
- [6] Jarzabek, S. and Zhang, H. “XML-based Method and Tool for Handling Variant Requirements in Domain Models”, *5th IEEE International Symposium on Requirements Engineering*, August 2001, Toronto, Canada, pp. 166-173
- [7] Karhinen, A., Ran, A. and Tallgren, T. “Configuring designs for reuse,” *Proc. International Conference on Software Engineering, ICSE’97*, Boston, MA., 1997, pp. 701-710.
- [8] Kiczales, G, Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J-M., Irwin, J. *Aspect-Oriented Programming*, *European Conference on Object-Oriented Programming*, Finland, Springer-Verlag LNCS 1241, 1997, pp. 220-242
- [9] Maginnis, N. “Specialist: Reusable code helps increase productivity,” in *Computerworld*, Nov. 1986
- [10] 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
- [11] Smaragdakis, Y. and Batory, D. “Application generators,” in *Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering*, J. Webster (ed.), John Wiley and Sons, 2000
- [12] Sneed, H. private communication
- [13] Soe, M.S., Zhang, H. and Jarzabek, S. “XVCL: A Tutorial,” *Proc. of 14<sup>th</sup> Int. Conf. on Software Engineering and Knowledge Engineering, SEKE’02*, ACM Press, July 2002, Italy, pp. 341-349
- [14] 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
- [15] Wong, T.W., Jarzabek, S., Soe, M.S., Shen, R. and Zhang, H.Y. “XML Implementation of Frame Processor,” *Symposium on Software Reusability, SSR’01*, Toronto, Canada, May 2001, pp. 164-172
- [16] Zhang, H.Y., Jarzabek, S. and Soe, M. S., 2001. 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