

Chapter 10 Mixed-strategy approach to generic design

Summary of this Chapter:

In the last Chapter, we discussed software similarity phenomenon, and reasons why it often leads to software clones – program structures recurring within and across programs, in many variant forms, in many places. We find clones of different granularity levels, occurring at all levels of program abstraction, from subsystem level to implementation, taking many forms, from similar program fragments to patterns of collaborating components. The reasons why repetitions occur are common. As cloning problem must be viewed in the context other design goals, many clones are difficult to avoid with conventional techniques for generic design.

As there is much similarity across systems released during evolution, effective treatment of clones is essential for managing software evolution in a cost-effective way. It is in the center of mixed-strategy approach to reuse-based evolution.

In this Chapter, we elaborate on mixed-strategy approach to unifying similarity patterns with generic mixed-strategy representations. We build such representations by applying XVCL on top of conventional component design and code written in a programming language such as Java or C++. In a mixed-strategy solution, we represent each of the important similarity patterns in a unique generic, but adaptable form, along with information necessary to obtain specific program structures, instances of the generic form. Mixed-strategy is capable of unifying a wide range of similarity situations we find in software, considerably enhancing the power of generic design of conventional methods.

To illustrate the approach, we discuss mixed-strategy solutions to similarity patterns found in the Buffer library. In Chapter 13, we analyze mixed-strategy solutions developed in yet other studies discussed in Chapter 9.

The reader should briefly revisit XVCL notation described in Chapter 7 and 8.

10.1 Buffer library in Java/XVCL mixed-strategy representation

We now describe and analyze a mixed-strategy representation for the Buffer library. As a mixed-strategy representation is a combination of Java and XVCL, we call it Java/XVCL mixed-strategy solution. For the sake of comparison, we design a mixed-strategy representation so that XVCL Processor can produce original buffer classes from it.

The sketch of a Java/XVCL x-framework for buffer classes is shown on the left-hand-side of Figure 1. Having identified similarity patterns, the Java/XVCL x-framework represents each significant similarity in a generic form as follows:

For each of the seven groups of similar classes, such as [T]Buffer or Heap[T]Buffer, we have x-frames to generate classes in a given group. A meta-class, e.g., [T]Buffer.gen, is an x-frame defining a common structure for classes in a given group. Meta-classes correspond by name to seven groups of similar classes. For example, x-frame [T]Buffer.gen defines a common structure of seven classes [T]Buffer classes, Heap[T]Buffer.gen – seven Heap[T]Buffer

classes, and so on. In Figure 1, we show only two out of the seven class specification and meta-class x-frames.

Each meta-class has a corresponding *class specification* x-frame. Class specification x-frames also correspond by name to seven groups of similar classes. For example, [T]Buffer.s specifies global parameters and controls to construct seven [T]Buffer classes.

Generic representation of smaller granularity similarity patterns such as methods or attribute declaration sections appear as meta-fragments. Meta-fragments are reusable building blocks for classes. They are customized for reuse in different classes, as specified in x-frames above.

The top-most SPC x-frame specifies global parameters and controls to construct all the buffer classes.

The overall Java/XVCL x-framework has been “normalized” for non-redundancy to represent each of the meaningful similarity patterns in a generic way.

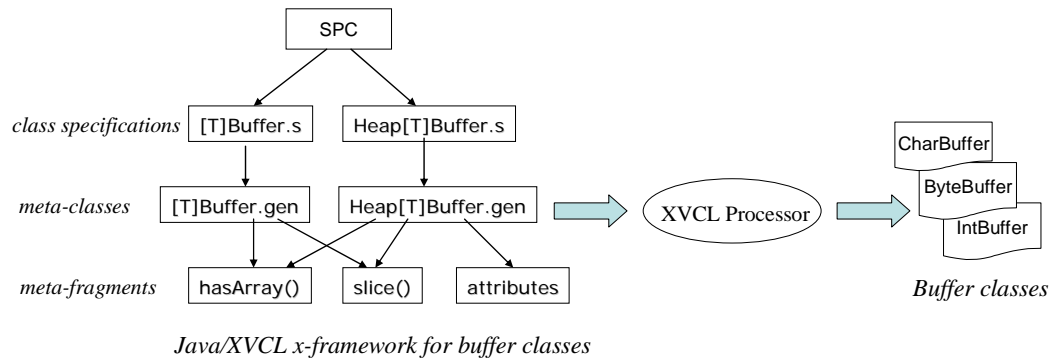


Figure 1. An overview of a Java/XVCL mixed strategy solution to the Buffer library

We now examine examples of generic solutions from the Java/XVCL buffer x-framework.

Program structures with type-parametric variations are the simplest form of similarity. Generics (or templates in C++) are language features meant for representing such program structures in a generic form. For example, five of the [T]Buffer classes of numeric element type, namely IntBuffer, ShortBuffer, DoubleBuffer, FloatBuffer and LongBuffer, differ only in the data type definitions and type names in some of the method names (differences shown in blue font in Table 1).

Table 1. Excerpts from classes ShortBuffer and DoubleBuffer

Class ShortBuffer	Class DoubleBuffer
<pre> public abstract class ShortBuffer extends Buffer implements Comparable { final short[] hb; final int offset; boolean isReadOnly; // Valid only for heap buffers ShortBuffer(int mark, int pos, int lim, int cap, short[] hb, int offset) { super(mark, pos, lim, cap); this.hb = hb; this.offset = offset; } ShortBuffer(int mark, int pos, int lim, int cap) { this(mark, pos, lim, cap, null, 0); } public static ShortBuffer allocate(int capacity) { ... </pre>	<pre> public abstract class DoubleBuffer extends Buffer implements Comparable { final double[] hb; final int offset; boolean isReadOnly; DoubleBuffer(int mark, int pos, int lim, int cap, double[] int offset) { super(mark, pos, lim, cap); this.hb = hb; this.offset = offset; } DoubleBuffer(int mark, int pos, int lim, int cap) { this(mark, pos, lim, cap, null, 0); } public static DoubleBuffer allocate(int capacity) { ... </pre>

At times, parametric differences among similar program structures are not be confined to type names. For example, Figure 2 shows a method **slice()** that recurs 13 times in the `Direct[T]Buffer[S|U]` classes with various combination of values highlighted in bold. In Figure 2, we see method **slice()** from class `DirectByteBuffer`.

```

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

```

Figure 2. Method **slice()** recurring with small changes in 13 `Direct[T]Buffer[S|U]` classes
 All kinds of a parametric differences are easily unified with XVCL variables or expressions. An x-frame representing method **slice()** in a generic way is shown in Figure 3.

```

Slice // meta-method x-frame
/* 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 3. An x-frame for generic method `slice()`

As XVCL expressions, in particular references to XVCL variables, can be intermixed with Java code, for clarity, we use italics font for Java code and bold font for XVCL expressions. References to XVCL variable `@elmTTYPE`, are replaced by the XVCL variable's value during processing. Variables referred to in x-frame `Slice` are assigned values in x-frames that adapt x-frame `Slice`, directly or indirectly. For example, the value of XVCL variable `byteOrder` is <set> to an empty string, "S" or "U". To produce method `slice()` for class `DirectByteBuffer`, we <set> the value of XVCL variable `elmTTYPE` to "Byte" and `byteOrder` – to an empty string. From `Direct@elmTTYPEBuffer@byteOrder` in Figure 3 XVCL Processor generates names for all the `Direct[T]Buffer[S|U]` classes such as `DirectByteBuffer`, `DirectIntBufferU` and `DirectIntBufferS`.

Parameterization via XVCL variables and expressions plays an important role in building generic components. It provides the means for creating generic names and controlling the traversal and adaptation of x-frames.

Program structures participating in other similarity patterns differ from each other in more diverse ways than we've seen in the above examples. For example, classes in each of the seven groups of similar classes differ in method signatures, method implementations, and attribute declarations. Some classes in a group may have extra methods that are not needed in other classes in the same group. Then, it becomes difficult to build a generic solution unifying such differences among similar program structures with simple parameters. Instead, we use the following technique to define a generic solution:

Suppose we have N similar program structures, e.g., seven `[T]Buffer` classes, that we wish to unify with a generic solution. We represent a common part of similar program structures as a generic adaptable x-frame `[T]Buffer.gen` that plays the role of a template. Then, we iterate over the `[T]Buffer.gen` N times, in each iteration adapting the `[T]Buffer.gen` in different way, to produce a required instance, a specific program structure.

Figure 4 shows x-frames used in generating seven `[T]Buffer` classes from a generic x-frame `[T]Buffer.gen`. Sections of common attributes, constructors and methods used as building blocks of `[T]Buffer` classes are defined in generic x-frames adapted from `[T]Buffer.gen`. The names of these x-frames are `commonAttributes.gen`, `commonConstructors.gen` and `commonMethods.gen`, respectively (Figure 4). These x-frames are shown as meta-fragments in Figure 1.

X-frame `[T]Buffer.s` iterates over `[T]Buffer.gen` seven times. For each iteration, `[T]Buffer.s` executes different XVCL customization commands to adapt `[T]Buffer.gen` and meta-fragments to generate a suitable `[T]Buffer` class.

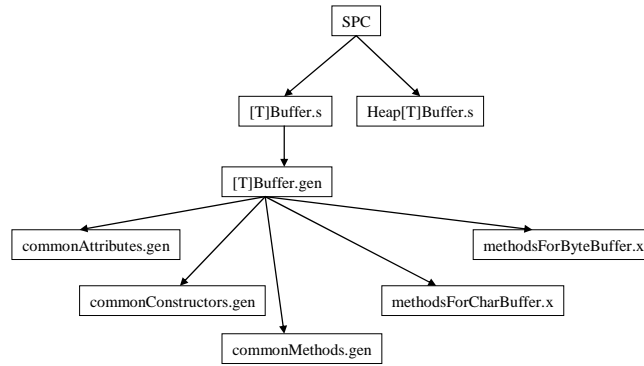


Figure 4. A fragment of an x-framework for generating seven [T]Buffer classes

The SPC of Figure 5 sets general parameters used in generating all the buffer classes. First, the SPC <set>s the value of XVCL variable **packageName** to “java.nio” and the value of XVCL variable **elmtTYPE** to <Byte,Char,Double,Float,Int,Long,Short>, the value of **elmttype** to <byte,char,double,float,int,long,short> and the value of **elmtSize** to <0,1,3,2,2,3,1>. XVCL variable **elmtSize** indicates the size of a buffer element:

2^{elmtSize} = (the number of bytes per buffer element).

Command <while> is controlled by a variable whose values are names of x-frames specifying how to generate classes in each of the seven groups of similar classes (we listed only two out of the seven x-frame names).

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 classGroup = [T]Buffer.s,Heap[T]Buffer.s, ... />
// values of classGroup are names of x-frames for seven groups of similar buffer classes
  <adapt classGroup />
</while />
  
```

Figure 5. A fragment of SPC

```

[T]Buffer.s
<while elmtTYPE, elmttype, elmtSize >
  <select option = elmtTYPE >
    <option = Byte >
      <adapt [T]Buffer.gen >
        <insert moreMethods >
          <adapt methodsForByteBuffer.x />
        <adapt />
    <option = Char >
      <adapt [T]Buffer.gen >
        <insert moreMethods >
          <adapt methodsForCharBuffer.x />
        <insert-after extends-implements-clause >
          Appendable,CharSequence,Readable
        <insert toString >
          Public String toString()
          { return toString(
            position(),limit()); }
        <adapt />
    <otherwise>
      <adapt [T]Buffer.gen />
  <select />
</while />

```

Figure 6. A fragment of specs to generate seven [T]Buffer classes

In x-frame [T]Buffer.s (Figure 6), <while> loop iterates over its body seven times, generating one of the seven [T]Buffer classes in each iteration. <while> loop is controlled by three multi-value variables, namely **elmtTYPE**, **elmttype**, **elmtSize**. In each iteration, each of the three variables accepts one value from their 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 x-frame [T]Buffer.gen.

```

[T]Buffer.gen outfile:@elmtTYPEBuffer.java
package @packageName;
public abstract class @elmtTYPEBuffer extends
    Buffer implements Comparable<@elmtTYPEBuffer> <break extends-implements-clause />
{ // attributes and methods here
<adapt commonAttributes.gen />
<break moreAttributes />
<adapt commonConstructors.gen />
<break moreConstructors />
<adapt commonMethods.gen />
<break moreMethods />
<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(); }
</break>
}

```

Figure 7. A fragment of generic x-frame [T]Buffer

We now comment on x-frame [T]Buffer.gen (Figure 7). An expression `@elmtTYPEBuffer.java` in the *outfile* attribute at the top of the x-frame [T]Buffer specifies the names of files where XVCL Processor is to emit the code for the respective classes. Rows below define the package name and the headers for each of the seven classes [T]Buffer in turn. Then, x-frame [T]Buffer defines a common structure of [T]Buffer classes in terms of attribute, contractor and method definitions, with suitable `<break>`s for extensions.

Five numeric [T]Buffer classes differ only in parameters that are catered for by XVCL variables defined in the SPC. These five classes do not require any further adaptations of the common class representation provided by the x-frame [T]Buffer.gen. The five classes are generated in five iterations over the `<otherwise>` clause, where `<adapt [T]Buffer.gen />` does not specify any further customization commands.

However, classes `ByteBuffer` and `CharBuffer` require some specific customizations of the template [T]Buffer.gen. These customizations are handled by `<insert>` into `<break>` commands.

Class `ByteBuffer` needs 35 extra methods in addition to those found in other [T]Buffer classes. These extra methods are in x-frame `methodsForByteBuffer.x` and are `<insert>`ed at a suitable `<break>` point in [T]Buffer.gen.

Extra methods for class `CharBuffer` are `<insert>`ed in a similar way as in the case of `ByteBuffer`. However, class `CharBuffer` requires yet other customizations:

The `<insert-after>` under `<option>` `Char` extends the `implements-clause` in a way that is required only for class `CharBuffer`. Figure 8 and Figure 9 show the `implements-clauses` in `ByteBuffer` and `CharBuffer` classes, respectively.

Method **toString()** converts a buffer element to a character string. For CharBuffer such conversion is trivial. For other buffer element types, implementation of **toString()** requires some interpretation. As six of the buffer classes need the same implementation of **toString()**, while only one needs a different implementation, we define more commonly used implementation as a default in x-frame [T]Buffer.gen (Figure 7). Method **toString()** for CharBuffer is handled by the last **<insert>** specified at the adaptation point in **<option>** Char (Figure 6). This **<insert>** overrides the default implementation of **toString()** defined **<break>** in [T]Buffer.gen.

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

Figure 8. Declaration of class ByteBuffer

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

Figure 9. Declaration of class CharBuffer

As already observed, inserting code and specifications at designated **<break>** points is also a simple yet powerful means to handle unexpected changes arising during software evolution. The implement-clause (Figure 8 and Figure 9) illustrates this point. As we add new features and the number of feature combination increases, we may need specify more interfaces a given class *implements*. We often cater for such unexpected extensions with **<insert>** into **<break>** commands. In the x-frame [T]Buffer (Figure 7), the **<break>** named extends-implements-clause marks the point into which **<insert>** commands from higher-level x-frames can inster custom code. In particular, when generating class CharBuffer, we **<insert>** text “,CharSequence” at the **<break>** extends-implements-clause. Notice that the implements-clauses of other classes are not affected. In general, a **<break>** point can have a default content (Java code with XVCL commands) and higher-level x-frames can replace the default content or insert extra content after/before many **<break>** points in **<adapt>**ed x-frames. If no **<insert>** command from higher-level x-frames affects the **<break>** point, then the XVCL processor emits the **<break>**’s default content to the output.

Finally, we have a brief look at the bottom level of the x-framework hierarchy where we find so-called meta-fragments – x-frames representing generic class building blocks. Excerpts from generic constructor and method definitions are shown in Figure 10and Figure 11, respectively.

```

commonConstructors.gen
// contains constructor definitions for all [T]Buffer classes
.....//other constructors
@elmTTYPEBuffer(int mark, int pos,
    int lim, int cap,@elmTtype [] hb, int offset)
{
    super(mark, pos, lim, cap);
    this.hb = hb;
    this.offset = offset;
}
.....//other constructors

```

Figure 10. An excerpt from generic constructor definition x-frame

```

commonMethods.gen
// contains method definitions for all [T]Buffer classes
.....//other omitted methods
//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;
}
.....//other methods

```

Figure 11. An excerpt from generic method definition x-frame

We only highlighted the structure of the solution. Complete documentation and XVCL code for the Buffer library case study can be found at our Web site [4], under “Case Studies”. This study was also described in [2][3].

We would like to end this section by summarizing the process of designing the Java/XVCL buffer x-framework. We gained general understanding of 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. We designed x-frames 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 XVCL variables, `<break>` points, `<select>`, `<insert>`, `<adapt>` and other XVCL commands to cater for required variations. Then, we created a template x-frame (e.g., [T]Buffer.gen) for each of the seven groups of similar classes, along with a corresponding specification x-frame (e.g., [T]Buffer.s). Finally, we created SPC with global controls of the generic Java/XVCL mixed-strategy representation of the Buffer library.

In this and other case studies in which we applied XVCL, the Object-Oriented structure of the programs to be generated has always been a starting point for building a meta-solution. It is not surprising, as the structure of a mixed-strategy solution depends on and implicitly defines the intended structure of programs to be generated.

10.2 Evaluation of the mixed-strategy Buffer library solution

Buffer library in representation described in this Chapter is meant for developers and maintainers of the Buffer library. Java/XVCL x-framework contains target code in generic form, along with the information about how to derive concrete buffer classes from their respective generic representations. Java/XVCL x-framework also contains yet other information that aids in understanding classes, and helps developers maintain Buffer classes. For example, Java/XVCL x-framework explicates the impact of various features on code.

Developers have full control over every detail of the class structure and code that XVCL Processor generates from the Java/XVCL x-framework. Programmers using 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 main-stream programming work. Especially, programmers working in unstable domains, where change is pervasive, may see good reasons to do so. In such cases, x-frames of the class library can be neatly integrated with programs using those libraries at the mixed-strategy.

To enable fair comparison, in this experiment, a class library generated from the Java/XVCL x-framework was no different from the original class library. In another study, we re-designed the x-framework to generate buffer classes optimized for performance, as required, for example, in embedded systems.

In the following sections, we conduct quantitative and qualitative analysis of the presented Java/XVCL mixed-strategy solution for the Buffer library. Our qualitative analysis includes an experiment (Section 10.2.3) in which we extend the Buffer library with a new type of buffer element – Complex, comparing the maintenance effort involved in each, the original and the mixed-strategy solutions. The reader can find a comprehensive evaluation of strengths and weaknesses of the mixed-strategy approach on general grounds in Chapter 14, once we have discussed yet other mixed-strategy projects.

10.2.1 Quantitative comparison of Buffer library solutions

Table 2 and Figure 12 show the results of comparing the original Java Buffer classes with the Java/XVCL mixed-strategy solution.

Table 2. Original Java Buffer library vs. Java/XVCL mixed strategy solution

classes	original Java Buffer library			Java/XVCL mixed-strategy representation of Buffer library		
	1	2	3	4	5	6
	# class building blocks	LOC ¹	Java Code ²	# x-frames	LOC ³	Java Code ⁴
[T]Buffer (7 classes)	258	3720	871	79	1400	320
Heap[T]Buffer (7 classes)	159	914	802	52	313	291
Direct[T]Buffer[S U] (13 classes)	337	2428	2249	85	689	665
Heap[T]BufferR (7 classes)	112	521	444	35	226	209
Direct[T]BufferR[S U] (13 classes)	187	979	895	42	378	367
subtotal (47 classes)	1053	8562	5261	293	3006	1852
other classes (24 classes)	332	1570	1458	31	239	228
total	1385	10132	6719	324	3245	2080
¹ physical lines of Java code with comments, excluding blanks ² physical lines of Java code without comments, excluding blanks ³ physical lines of Java code with comments and XVCL commands, excluding blanks ⁴ physical lines of Java code and XVCL commands, without comments, excluding blanks						

Columns 1 and 4 show the number of conceptual elements in each of the solution spaces, Java and Java/XVCL, respectively. Columns 2 and 3 show sizes of the elements in Java solution, in terms of physical lines of Java code **with** and **without** comments, excluding blanks, respectively. Columns 5 and 6 show sizes of the elements in Java/XVCL solution, in terms of physical lines of Java code and XVCL commands, **with** and **without** comments, excluding blanks, respectively.

A conceptual element in the Java program is a class or a program “fragment” such as a method/constructor, or yet smaller code fragment representing declaration section or a fragment of method/constructor implementation. We pay attention only to fragments that, in our understanding, play a specific role in the Buffer domain or in class construction, and therefore, are important for program understanding.

Figure 12 shows the overall code size of each solution. Unification of similarities with XVCL on top of Java code eliminates 68% of the code as compared to 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.

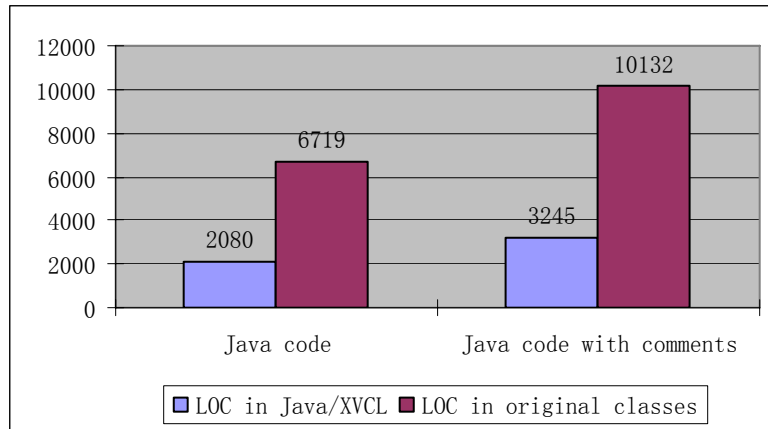


Figure 12. Buffer library: summary chart

10.2.2 Qualitative analysis of the Java/XVCL solution

As we can see in Table 2, in the Java/XVCL solution, a programmer deals with a smaller number of conceptual elements than in the original Java solution. Also, the sizes of conceptual elements in the Java/XVCL solution are smaller than the sizes of elements in the original Java solution. On the other hand, in the Java/XVCL solution, developers have to deal with added complexity of understanding generic representation.

The physical size is just one among many factors that collectively determine the complexity of a program solution in terms of its understandability, changeability, or maintainability. For example, by applying data compression techniques, we do not make a program any simpler for a programmer to understand. Likewise, unifying all possible similarity patterns in a program would complicate program understanding rather than simplify it. (Consider an extreme case in which we unify all references to a certain program variable or all occurrences of letter ‘a’ in a program!). Balance and common sense are irreplaceable in arriving at any good design solution.

A mixed-strategy solution represents each of the important similarity patterns in a unified, generic, but adaptable form, along with the information necessary to obtain its instances – specific classes or class methods. This is its main strength and the source of simplification of implementing changes.

Not only does such unification reduce the conceptual and physical size of the overall program solution, but also it emphasizes important relationships among program elements that matter to programmers who try to understand and modify the program. Instead of dealing with each class separately from others, we can understand classes in groups. Traceability allows us to see the exact differences among specific classes in a group, should we need this information. The same rule of genericity/traceability is applied below the class level, to methods/constructors and to other meaningful program fragments (such as method implementation fragments). In the Java/XVCL x-framework, we can see what’s similar and what’s different at each level. The x-framework also interconnects all the levels of program definition, so that we can see important relationships among them: If we want to change one class, we can check if the change also affects other similar classes. If we want to change a

class method, we can analyze the impact of change on all the classes that use that method in the same or similar form.

The above relationships are implicit in the Java buffer classes (and in most of conventional programs in general). A programmer must recover them whenever he/she needs understand a program during maintenance. The benefit of the Java/XVCL solution also lies in separating concerns: Defining a class architecture, to meet usability, performance and other runtime concerns, is the domain and responsibility of a programming language. Design-time concerns - such as how classes are built, what is similar and different among them, and what it takes to change them – are delegated to the mixed-strategy.

10.2.3 Extending the Buffer Library – an experiment in maintenance

We now describe a small experiment comparing maintainability of the Java/XVCL buffer x-framework versus the original buffer classes. Suppose we need extend the Buffer library with buffers of a new element type, say complex number (Complex). Many classes must be implemented to address this new element type, but here we concentrate only on three of them, namely ComplexBuffer, HeapComplexBuffer and HeapComplexBufferR. We start by illustrating what it takes to extend our meta-solution so that XVCL Processor can generate these three new classes.

A Complex buffer is similar to other numeric buffers, but as Complex is not a primitive type as int or float, we must implement it first. A sample implementation of class Complex is given in Figure 13.

```
//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 13. Class Complex

We represent a complex number as two double numbers: double re; double im;. 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 <while> loop <adapt>ing x-frame [T]Buffer with values of XVCL variables **elmtTYPE**, **elmttype** and **elmtSize** set to **Complex**, **Complex** and **4**, respectively. Figure 14 shows a

relevant part of the revised SPC. These changes in SPC are sufficient to generate classes HeapComplexBuffer and HeapComplexBufferR.

```
SPC // modified SPC of Figure 5; modifications shown in bold
<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 >
<while classGroup = [T]Buffer.s,Heap[T]Buffer.s, ... />
// values of classGroup are names of x-frames for seven groups of similar buffer classes
  <adapt classGroup />
<while />
```

Figure 14. Modified SPC for class Complex

The new type Complex also affects definitions of methods **hashCode()** and **CompareTo()** in the x-frame commonMethods.gen (Figure 4 and Figure 11). XVCL code in Figure 15 shown in bold results from addressing the new type Complex. In method **hashCode()**, the buffer element must be casted to type int in all classes but ComplexBuffer, in which the buffer element must be casted to type Complex. The **<select>** commands in methods **hashCode()** and **compareTo()** discriminate between these two situations.

```

commonMethod.gen
... // other methods
// 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--)
<select option = elmTYPE >
    <option = Complex > h = 31 * h + ((Complex)get(i)).getReal();
    <otherwise > h = 31 * h + (int)get(i);
</select>
return h }
// Compares this buffer to another object.
public int compareTo(Object ob) {
    @elmTYPEBuffer that = (@elmTYPEBuffer)ob;
    int n = this.position() + Math.min(this.remaining(), that.remaining());
    for (int i = this.position(), j = that.position(); i < n; i++, j++) {
        @elmTYPEBuffer v1 = this.get(i);
        @elmTYPEBuffer v2 = that.get(j);
        if (v1 == v2) continue;
        if ((v1 != v1) && (v2 != v2)) // For float and double
            continue;
<select option = elmTYPE >
    <option = Complex > if (v1.compareReal(v2) < 0)
    <otherwise > if (v1 < v2)
</select>
        return -1;
        return +1;
    }
    return this.remaining() - that.remaining(); }
.....// other methods

```

Figure 15. Modified methods **hashCode()** and **CompareTo()**

The above completes the description of modifications of the Java/XVCL x-framework to address class **Complex**. We generated sample classes from the modified x-framework and tested the result.

Now, let us try to address element type **Complex** in the original Java Buffer library, concentrating on the same three classes as 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 very tedious and error-prone. For example, there are 42 places in class **IntBuffer.java** (which is only 124 lines) that must be changed from **Int** or **int** to **Complex**. This replacement cannot be done automatically as not all 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.

Table 3 compares changes of the original Buffer library versus the Java/XVCL solution involved in adding element type Complex. For example, in the original Buffer library, implementing class ComplexBuffer.java based on the code of class IntBuffer 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. On the other hand, in the Java/XVCL solution all the changes must be done manually, but only five modifications are required.

Table 3. The impact of addressing type Complex in the original Buffer library vs. Java/XVCL solution

New classes	Copied classes	Changes in original Buffer library			Changes in Java/XVCL solution		
		Type of changes	No.	type	Type of changes	No.	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			

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

10.3 Conclusions of this Chapter

We described the principle of generic design with mixed-strategy approach, using a Buffer library as an example. We analyzed engineering qualities of the Java/XVCL mixed-strategy solution in quantitative and qualitative terms, and by conducting a controlled experiment.

A Java/XVCL mixed-strategy solution compressed the physical and conceptual solution size as compared to the original Java Buffer library. It also emphasized relationships among program elements that matter to programmers who try to understand and modify the program. Finally, it enhanced the conceptual integrity of the design, which Brooks calls “the most important consideration in system design” [1].

References

- [1] Brooks, P.B *The Mythical Man-Month*, Addison Wesley, 1995
- [2] Jarzabek, S. and Li, S. "Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique," *Proc. ESEC-FSE'03, European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, September 2003, Helsinki, pp. 237-246
- [3] Jarzabek, S. and Li, S. "Unifying clones with a generative programming technique: a case study," *Journal of Software Maintenance and Evolution: Research and Practice* John Wiley & Sons, Volume 18, Issue 4, July/August 2006, pp. 267-292
- [4] XVCL (XML-based Variant Configuration Language) method and tool for managing software changes during evolution and reuse, <http://fxvcl.sourceforge.net>