Enhancing the Visibility of Tightly Coupled Concerns with Generic Design

Stan Jarzabek
Department of Computer Science, School of Computing
National University of Singapore
Lower Kent Ridge Road, Singapore 117543
stan@comp.nus.edu.sg

ABSTRACT
Separation of concerns (SoC) and generic design are two important principles to better control software complexity during development, maintenance and reuse. Some of the concerns identified at the concept level, can be separated at design/implementation level by modularization or unconventional techniques such as AspectJ or Hyper/J. Yet other concerns, however, may not be easily separable due to complex interactions with the rest of a program. In this paper, we show that generic design, in addition to simplifying program representation by avoiding repetitions, can also enhance the visibility of inseparable concerns, offering a weaker, but still useful form of SoC. The reason why generic design can penetrate software deeper than SoC is that generic design is based on the notion of similarity of program structures which is less formal and rigorous than separation of concerns. Paper’s contribution lies in explaining the inter-play between the two principles. In addition to the above observations, we show that there is an overlapping area where the goals of SoC and generic design, as well as means to achieve those goals, are the same.

Categories and Subject Descriptors
D.1.5 [Programming Techniques]: Object-Oriented Programming; D.2.2 [Software Engineering]: Design Tools and Techniques – Object-oriented design methods, Software libraries; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement; D.2.10 [Software Engineering]: Design – Representations; D.2.13 [Software Engineering]: Reusable Software - domain engineering;

General Terms
Design, Languages, Experimentation

Keywords
generic design, separation of concerns, software reuse, maintenance, component-based development, generative programming, meta-programming

1. Introduction
Software similarity patterns trigger look-alike program structures of varying type and size, spreading within and across programs. Generic design can help avoid that. Similarities create possibilities for conceptual simplification and reduction of programs’ physical size. STL [34] is a premier example of engineering benefits of generic design.

Generic solutions are a central theme in software reuse, component-based, pattern-driven development (e.g., facilitated by .NET™ or J2EE™), and architecture-centric Product Line [10][12] approaches.

Generic design, as understood in this paper, aims at achieving non-redundancy, by unifying similarity patterns at design and code levels, whenever this is beneficial for program simplification, easier maintainability or better reuse.

We do not make any specific assumptions about the type, granularity of similar program structures, what they represent, or the nature of differences among them. We also consider any mechanisms for defining generic representations to unify similar structures. The importance of generic design in managing software complexity has been recognized for long. Macros were one of the early attempts to make programs more generic. Goguen popularized ideas of parameterized programming [23]. Among programming language constructs, type parameterization [22] (called generics in Ada, Eiffel, Java and C#, and templates in C++), higher-order functions [42], and inheritance can help avoid repetitions in certain situations. Design techniques such as iterators, design patterns [21], table-driven design (e.g., in compiler-compilers), and modularization with information hiding [35] are supportive to building generic programs. Generative programming techniques, such as XVCL [44], build a generic program representation at the meta-level, and derive concrete programs, with possible redundancies, from the generic meta-level representation.

We can conceive a “generic design solution” as a parameterized structure that can be turned into a concrete, custom program solution by instantiating the parameters. The form of a parameterized structure, the nature of parameters, the mechanism for instantiating parameters, and the overall process that leads to instantiating a concrete program solution from its generic counterpart depends on the techniques used for generic design. Parameterized structures can be as simple as generics or templates, or as complex as an OO framework or a generic parser. In an OO framework, parameters are abstract classes and design patterns. Parameters for generic parser is a definition of a programming language syntax in BNF.

A concern is any area of interest in a program solution, pertinent to functional features, quality requirements, software architecture, detail design or implementation. The idea of separation of concerns (SoC) is to break a program into distinct concerns in order to deal with them separately. As we do so, we try to limit interactions between concerns as much as it is possible.

SoC principle states that “a given problem involves different kinds of concerns, which should be identified and separated to cope with complexity, and to achieve the required engineering quality
We believe the reason why generic design can penetrate software deeper than SoC is that generic design is based on the notion of similarity of program structures which is less formal and rigorous than separation of concerns.

As we argue in support for the above hypothesis, we also discuss the general inter-play between the two principles, and observe that there is an overlapping area where by means of separating with the goal of generic design. We discuss both engineering goals addressed by the two principles and technical means to achieve these goals. We compare engineering qualities of software designed based on the principles of SoC and generality, and hint at the possibility of using both principles in synergistic way.

Section 2 discusses the relation between SoC and generic design. In Section 3, we show an example of concerns that are difficult to separate. In Section 4, we show generic design solution for the same example. Section 5 discusses yet another example, from application software. We analyze observations in Section 6. Related work and conclusions end the paper.

2. Forms of SoC and links to generic design
The main goal of SoC is to deal with a concern separately from other concerns, e.g., during maintenance. SoC at the concept level is useful, but the benefits magnify if we can apply some degree of SoC also at the level of software design and implementation. For this, program modularization is the most natural conventional technique to consider. Sometimes, a concern can be nicely aligned with modular decomposition of a program. In such case, a concern can be localized to a single module (a component, class or function) or a group of modules (e.g., component layer), exposing an abstract program interface (API) to its clients. A concern can also become a hidden part of a module. The details of a concern implementation become hidden behind the API [35]. This is an ideal situation from the engineering point of view.

To provide full localization of a concern, management of any variability within a concern should be either hidden part of a module or supported by suitable API operations.

A modularized and localized concern can be easily added to or taken out from programs, making programs more generic.

Modularization is also a simple form of generic design. Here, a similarity pattern is reflected by API. Hidden implementation part of module definition plays a role of a parameter that makes a module generic. Instantiation of such a “generic module” is done by defining a specific data representation, and implementing API operations in terms of the chosen data representation.

In the above case, SoC enhances genericity and we achieve SoC and generic design using the same technical means.

Concerns that cannot be localized in the above sense have a crosscutting effect on other concerns (and modules of a primary program decomposition in AOP [28]). Some of the crosscutting concerns can be modularized at the extra meta-level plane using unconventional approaches such as AOP, AHEAD [6], MDSOC [41] or XVCVL [44].

In AOP, ‘introductions’ and ‘advice’ play the role of parameters of modules of primary decomposition. By separating aspect code and providing a mechanism to easily inject or take out some of the aspect code from modules, we make modules more generic, as we may have a module with or without aspect code. The more module’s code can we place in aspects, the more combinations of aspects can we legally and meaningfully weave into a module, the more generic a module.

A similarity pattern that we unify with AOP is a functional module that can appear in multiple contexts, with or without aspects. This interpretation of AOP is in tune with goals of generic design, and we can view AOP as a kind-of generic design mechanism. In fact, AOP has been considered as a technique for building Product Line architectures [10][12], which justifies the above interpretation.

MDSOC [41] and AHEAD [6] aim at building programs by composing independently defined concerns. In MDSOC, there is no primary decomposition, meaning that all the concerns are treated as equal. AHEAD promotes feature-oriented programming in which features are modeled as mathematical functions, and then programs are built and evolved by refining those functions. In both cases, an architecture of concerns from which we can build specific programs by composing concerns must necessarily have the characteristics of generic design.

Component platforms hide implementation of some of the potentially crosscutting concerns, providing transparent access to them via APIs. In JEE™, containers provide a general mechanism to access, via APIs, services whose implementation crosscuts code in the containers. Examples of such services include transaction management, persistence, security, authentication/authorization and session management, depending on a container used [32][33]. While not completely eliminating, the JEE infrastructure makes crosscutting effect more visible and reduced to calls to container’s API operations.

The above examples illustrate that whether a given concern has a crosscutting effect or not depends on a technology used. It also
depends on design decisions regarding modular decomposition and other major mechanisms used in the design of a particular program.

### 3. An example of “difficult” concerns

Discussion in the last section suggests that SoC achieved “physically” at the levels of program design and code, is also likely to contribute to the goals of generic design.

If we could find a way to separate most of the concerns that matter using some technique, no doubt our software engineering problems would be much less than they are today. However, some of the concerns, are so tightly coupled one with another (or with modules of primary decomposition [28]) that their physical separation becomes unthinkable. These couplings may not be fully anticipated at the concept level, but as analysis of the exception handling concern show [18] “the devil is in the details.” Exception handling is an example of a “difficult” concern [1]: “The main problem is that realistic software systems exhibit very intricate relationships involving the normal-processing code and error recovery concerns.” Our experiments with EHAB (Exception Handling Application Block) on .NET [17] also revealed difficulties to separate exception handling from the rest of the code.

Performance concern in real-time systems is yet another example of a “difficult” concern. Performance has pervasive impact on many design decisions. While we can express and conceive performance concern conceptually (e.g., by documenting design decisions that have to do with performance), “physical” separation of performance concern from functional modules or yet other concerns that interact with performance may not be feasible. In other systems, where performance strategies are simpler, it may be possible to localize the performance concern in certain modules, or separate it by means of AOP.

In our experience, many concerns in application domain-specific areas, often called features [6][26], are inseparable just as performance concern is inseparable in time-critical systems.

Situations where concerns become difficult to separate are interesting not only as they shed light on requirements for effective SoC techniques, but also as they show that SoC and generic design are intimately interrelated and synergistic one with another. An interesting observation is that generic design takes over SoC concepts into the software areas where separation of concerns tends to be difficult. We illustrate our points with examples from our lab studies and industrial projects.

#### Figure 1. Features in the Buffer library

Buffer library, a part of java.nio.* packages in JDK 1.5, implements containers for data in a linear sequence for reading and writing. Buffer classes differ in buffer element type, memory allocation scheme, byte ordering and access mode. Figure 1 shows a feature diagram [26], with five feature dimensions, with specific variant features listed below a corresponding feature dimension box. Each legal combination of variant features yields a unique buffer class. We end up having many buffer classes with much similarity among them [24].

Feature dimensions are some of the “concerns” in the Buffer library domain. A developer or maintainer of the Buffer library may be interested to know: “how does an element type (or access mode) affect implementation of classes?”, “can I separate certain concerns so that specific features can be incorporated into classes, and relevant code maintained, in separation from other concerns?”.

Class names reflect combination of specific features implemented into a given class. For example, DirectIntBufferR is a Read-Only buffer of integers, implemented using direct memory scheme. Classes whose names do not include ‘R’, by default are ‘W’ – Writable. The Buffer library contains classes whose names are derived from a template: [MS][T]Buffer[AM][BO], where MS = Memory Allocation Scheme: Heap or Direct; T = Element Type: Int, Double, Float, Long, Short, Byte or Char; AM = Access Mode: W – Writable (default) or R - Read-Only; BO = Byte Ordering: S – non-native or U – native; B – BigEndian or L – LittleEndian. For simplicity, we can ignore VB = View Buffer, which is, in fact, yet another concern that allows us to interpret byte buffer as Char, Int, Double, Float, Long, or Short.

If successful, separation of five concerns, shown as boxes in Figure 1, would result in some “core structures” and five separately defined concerns. By composing specific features from each of the concerns into the “core structures”, we would obtain a specific buffer class implementing these features.

The number of “core structures” should be considerably smaller than the number of specific buffer classes (around 100) to make separation of concerns worthwhile. Also, we would expect that the complexity of buffer classes represented by “core structures” plus separated five concerns would have some attractive engineering qualities, such as reduced conceptual complexity or reduced maintenance effort, over the original buffer classes in which the concerns remain intermingled.

The above view of a solution that achieves SoC again reminds generic design solution, with “core structures” playing the role parameterized representation, comprising design and code of buffer classes, and concerns playing the role of parameters that instantiate the “core structures”.

The nature of “core structures”, concerns and composition mechanism depends on a SoC technique. For example, in AOP, “core structures” correspond to some classes of a primary decomposition, and concerns are introductions and advises to be woven into primary classes. In MDSOC, “core structures” would be treated as just yet another concern. In AHEAD, concerns are groups of features just as we described above, and “core structures” correspond to classes that are subjected to refinements.

Now we look into issues involved in trying to separate concerns in the Buffer library. To separate a concern, we must first see how a given concern affects the structure of the library and implementation of classes that have to do with a given concern. Class naming conventions, described above, make the task of finding classes relevant to different concerns easy.

Let’s focus on the concern “buffer element type” T and observe its impact on buffer classes.
We have no problem to do so in five classes [T]Buffer, where T is restricted to five numeric types: int, double, float, long, or short. These classes are the same except the respective names affected by element type, highlighted in bold in Figure 2.

In the scope of five numeric types, the “buffer element type” concern can be separated by means of type parameter with Java generics [24]. (In fact, certain limitations of Java generics make type parameterization difficult even in this simple case, but here we are not concerned with language-specific limitations of Java generics. The reader can find more details in [24] and in case studies posted on our Web site [44].)

```java
public abstract class IntBuffer
{
    final int[] hb; // Non-null only for heap buffers
    IntBuffer(int mark, int pos, int lim, int cap, // package-private
              int[] hb, int offset)
    {
        ...}
    IntBuffer(int mark, int pos, int lim, int cap) {
        // package-private
        this(mark, pos, lim, cap, null, 0);
    }
    public static IntBuffer allocate(int capacity)
    {
        return new HeapIntBuffer(capacity);
    }
    ... public static IntBuffer wrap(int[] array, int offset, int length)
}
```

**Figure 2. Fragment of class IntBuffer**

Could we make “buffer element type” T an aspect, in the sense of AOP?

If we require that classes of primary decomposition are complete and can be executed, then the answer is no. Buffer element type is an integral part of any conceivable primary decomposition in the above sense, and we can’t have a buffer class without mentioning buffer element type, in either specific (such as int or short) or generic form.

If, on the other hand, we relax the requirement that modules of primary decomposition must be executable on their own, then we could consider a buffer element type as an aspect, provided that we can weave code related to type at specified join points in classes of primary decomposition.

The exact points where differences among buffer classes (highlighted in bold in Figure 2) occur do not correspond to what is considered a join point in AOP. While we could place all the declarations affected by type name into ‘introductions’, and extend AOP to weave also method headers, it seems that such a solutions would not be in sync with the spirit of AOP. We rather conclude that the discussed situation is not aspect-friendly. Current form of AOP is not meant to deal with concerns that affect code in ad hoc way, at arbitrary program points. We try to strengthen this point in our further discussion.

We now extend analysis to two remaining features in the “buffer element type” concern, namely ‘char’ and ‘byte’. Inspection of code reveals that class CharBuffer has a different implementation of method `toString()` than any of the numeric buffer classes. Method `toString()` converts a buffer element to a character string. In class CharBuffer, method `toString()` is trivial, just returns the buffer element, while in numeric buffer classes this method must do a proper conversion. In addition, class CharBuffer has a number of extra methods that are not needed in numeric buffer classes.

The situation in ByteBuffer is analogous to CharBuffer. We see many extra methods that do not appear in numeric buffer classes or CharBuffer.

At this point, we can recap what it takes to separate concern “buffer element type” in seven classes [T]Buffer, where T is Int, Double, Float, Long, Short, Byte or Char:

1. We must deal with varying type names and method names (e.g., ‘Int’ is part of method names in IntBuffer, while ‘Char’ is part of method names in CharBuffer).

2. We must selectively insert extra methods into certain classes.

Extra methods can be easily separated (also aspectized) and weaved into relevant classes, therefore addressing the remaining two buffer element types char and byte does not raise further complications for separation of concerns. However, it creates a challenge for generics as extra methods cannot be represent by generic type. In Section 5, we describe generic design that lifts this limitation.

The situation in groups of classes Heap[T]Buffer and Heap[T]BufferR is the same as in the group [T]Buffer.

Separation of type concern becomes more problematic when we look beyond 21 classes in groups [T]Buffer, Heap[T]Buffer and Heap[T]BufferR. We see more subtle code dependencies on “buffer element type” concern. For example, in method `slice()` buffer element type causes changes of algorithmic details as shown in Figure 3 (a constant in bold is equal to the length of a buffer element minus one, so the constant is 0 for type byte).

```
/*@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 3. Method slice() in DirectByteBuffer**

We also see more drastic impact of other concerns on class implementation. For example, classes implementing Direct memory allocations scheme differ a lot from analogical classes implementing Heap memory allocation scheme. Writable classes differ from analogical Read-Only classes. the visibility of concerns becomes blurred. Trying to look for exact impact of “buffer element type” concern on class implementation becomes most difficult task, not mention separating the concern.

Still, the “buffer element type” concern seems to be the simplest case. Other concerns are even more difficult to trace and separate. Interactions between concerns are not clearly visible in class implementation. Class implementation seems to reflect the net result of concern interactions in the form that makes separation of concerns difficult.
4. Switching perspectives

What makes separation of “buffer element type” concern difficult is (1) much variation in the impact of different buffer element types on class implementation, and (2) subtle, ad hoc interactions between “buffer element type” and other concerns.

When dealing with “difficult” concerns, a change of the perspective from SoC to generic design is quite refreshing. Rather than looking for ways to separate concerns, we look for similarity patterns in program structures that result from interactions among combinations of concerns implemented into classes. We are still doing a fair amount of SoC, but in an approximate way, only as far as it is practically achievable.

We have the following seven groups of similar classes in the Buffer library [24]:

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.

Similarities among classes manifest themselves as methods and attribute declarations that appear in different classes in similar form. Some classes contain extra methods that do not appear in other similar classes.

It should be noticed that seven groups of similar classes are organized around concerns: each group is characterized by concerns that vary across classes in a group, and yet other concerns that are fixed.

We now proceed to the part where we apply generic design to unify similarity patterns with help of a generative technique of XVCL [44]. As we define generic solutions using conventional programming technologies (languages and platforms) together with XVCL, we call the approach mixed-strategy.

We start with a concrete program, or at least with some idea of a program’s component/class architecture, and its partial implementation. In case of our experiment, we start with the existing Java buffer classes. We represent each group of similar program structures (methods or classes), with unique, generic customizable structure built with XVCL applied on top of Java.

We can imagine that XVCL decomposes a conventional program in its own way, wrapping structures of a subject program (of any granularity and type) within XVCL constructs to make them generic. It is important to notice that unification of similarity patterns occurs only at the level of an XVCL representation (left-hand-side of Figure 4). An executable program synthesized from the XVCL representation may still contain repetitions, if that’s required or unavoidable. Sometime repetitions are required for performance or reliability reasons. Yet other may be unavoidable given a programming technology used (e.g., on JEE or .NET platforms [45], see also [29]), and/or taking into account possibly yet other design goals a program must meet [24].

A building block of an XVCL generic program representation is called an x-frame. An overall solution, a hierarchy of generic design structures, is called an x-framework. In case of the Buffer library, we build a generic representation in combination of Java and XVCL, therefore we call it a mixed-strategy Java/XVCL x-framework, shown in Figure 4. An arrow between two x-frames: X → Y means that Y is used, after possible adaptations, to build X.

We synthesize all the classes in each of the seven groups of similar classes from the x-framework shown in Figure 4. Each of the Level 3 x-frames plays the role of a template defining a common part for all the classes in the respective group. For example, seven classes in group [T]Buffer are synthesized using x-frame [T]Buffer.gen as a template. X-frame [T]Buffer.s contains specifications instructing the XVCL Processor how to adapt [T]Buffer.gen and x-frames at levels below it to synthesize classes in [T]Buffer group.

We have analogical solutions in parts of the buffer x-framework for other six groups of similar classes.

Figure 4. A Java/XVCL mixed-strategy solution for Buffer classes

The essence of an x-frame is that it can be adapted to produce its instances (e.g., specific classes in a group). Smaller granularity building blocks for classes are defined at Level 4 (methods) and Level 5 (fragments of method implementation or attribute declaration sections). Therefore, small-granularity generic solutions (represented by the lower-level x-frames) are composed, after possible adaptations, to construct required instances of higher-level generic solutions (represented by higher-level x-frames).
Specification x-frames – Level 1 and 2 - tell the XVCL Processor how to synthesize specific (buffer classes from the x-framework. The top-most x-frame called SPC sets up global parameters and exercises the overall control over the generation process. Specifications of controls for each of the seven groups of similar classes are at Level 2.

XVCL Processor interprets an x-framework starting from the SPC, traverses x-frames below, adapting visited x-frames and emitting the custom program. By varying specifications, we can instantiate the same x-framework in different ways, deriving different, but similar, program components from it. In that sense, an x-framework forms a generic design structure that enables reuse within a single program, or across programs. In the latter case, an x-framework implements a concept of a Product Line architecture [12].

To better see the nature of an XVCL-enabled generic solution and its relation to SoC, we now explain the parameterization and adaptation mechanism, which is the “heart and soul” of how XVCL achieves goals of generic design. Figure 5 shows the details of a fragment of the Java/XVCL x-framework shown on the left-hand-side of Figure 4.

XVCL variables and expressions are parameters. Typically, names of program elements manipulated by XVCL, such as components, source files, classes, methods, data types, operators or algorithmic fragments, are represented by XVCL expressions. Using such parameters, rather than concrete names, makes x-frames more generic, adaptable to fit into multiple contexts.

For example, names and other parameters of the seven similar classes TBuffer are represented by XVCL expressions in the a x-frame TBuffer.gen (Figure 5).

A <set> command assigns a value to a variable. For example, the second command in the SPC of Figure 5 assigns values listed on the right-hand-side to a variable named elmtType. Expression @elmtType refers to one of such values (further details to follow).

XVCL parameters also play an important role of control elements that mark traces of customization changes related to a single source, that span across multiple x-frames. This “source” often represents a concern or a specific feature within a concern. For example, elmtType is one of the variables that mark customizations related to “buffer element type” concern. XVCL Processor propagates variable values from an x-frame where the value of a variable is set, down to adapted x-frames. While each x-frame usually sets default values for its variables, values assigned to variables in higher-level x-frames take precedence over the locally assigned default values. Thanks to this overriding rule, x-frames become generic and adaptable, with potential for reuse in unifying similarity patterns in many contexts.

Other XVCL commands, such as <select>, <insert> into <break> and <while>, collectively help us design generic solutions. At the same time, they also contribute to enhancing the visibility concerns. A <select> command directs processing into one of the many pre-defined branches (called options), based on the value of its control variable. With <insert> command, we modify x-frames at designated <break> points in arbitrary ways. XVCL expressions, <select> and <insert> into <break> are analogous to AOP’s mechanism for weaving advices at specified join points. The difference is that while AOP specifies joint points in a descriptive way, <insert> modify x-frames in arbitrary ways, at any explicitly designated <break> points.

Figure 5. A Java/XVCL x-framework for seven [T]Buffer classes (partial)

A <while> command iterates over x-frame(s), with each iteration generating similar, but also different, program structures. A <select> command in the <while> loop allows us to synthesize classes in each of the seven groups discussed in Section 3. This is a key element of XVCL strategy that allows us to unify similarity patterns at the level of mixed-strategy representation (i.e., in an x-framework), and still have repetitions in a program that XVCL Processor synthesizes from an x-framework.

Now, we comment on the above mechanisms in more details, referring to Figure 5 that shows a partial Java/XVCL x-framework for buffer classes. X-frame names, XVCL commands and
references to XVCL variables are shown in bold, using simplified, XML-free syntax. References to XVCL variables (highlighted in bold) can be embedded in code. For example, a reference to variable @elmtType, is replaced by the variable’s value during processing. Figure 6 shows generic method slice() from Direct[T]Buffer[S|U] classes (a specific instance of method slice() is shown in Figure 3). Values of variables set in SPC reach all their references in adapted x-frames. The value of variable byteOrder is set to an empty string, "S" or "U", in a respective <set> command placed in one of the x-frames that <adapt>s x-frame slice.gen (not shown in our pictures).

```java
slice.gen // generic method slice()
/* Creates a new byte buffer containing a shared subsequence of this buffer's content. */
public Direct[T]Buffer<T> slice() {
    int pos = this.position();
    int lim = this.limit();
    int rem = (pos <= lim ? lim - pos : 0);
    int off = (pos << 2) - rem;
    return new Direct[T]Buffer<T>(this, -1, 0, rem, rem, off);
}
```

Figure 6. Generic method slice() recurring in 13 Direct[T]Buffer[S|U] classes

The <while> loop in [T]Buffer.s is controlled by two multi-value variables, namely elmtType and elmtSize. The i’th iteration of the loop uses i’th value of each of the variables. In each iteration of the loop, the <select> command uses the current value of elmtType to choose a proper <option> for processing.

Attribute outfile of [T]Buffer.gen defines the name of a file where XVL Processor will emit the code for a given class.

Having set values for XVCL variables, SPC initiates generation of classes in each of the seven groups of similar classes via suitable <adapt> commands. Generic class [T]Buffer.gen defines common elements found in all seven classes in the group. Five of those classes, namely DoubleBuffer, IntBuffer, FloatBuffer, IntBuffer, and LongBuffer differ only in type parameters (as in the sample method wrap) shown in x-frame commonMethods.gen). These differences are unified by XVCL variables, and no further customizations are required to generate these five classes from x-frame [T]Buffer.gen. These five classes are catered for in <otherwise> clause under <select>. However, classes ByteBuffer and CharBuffer have some extra methods and/or attribute declarations. In addition, method toString() has different implementation in CharBuffer than in the remaining six classes. Customizations specific to classes ByteBuffer and CharBuffer are listed in the <adapt> commands, under <option>s Byte and Char, respectively.

The above described XVCL solution is meant to illustrate our points about relationship between generic design and SoC. Evaluation of engineering qualities of XVCL solution is not in the scope of this paper. We refer the reader to other papers for the discussion of trade-offs involved in applying XVCL.

5. Another example of “difficult” concerns

Class library is a very special type of a program. In this section, we show how a problem observed in the Buffer library shows in an application software.

A Domain Entity Management System (DEMS) is contributed by ST Electronics Pte Ltd (STEE), an industrial partner in our projects. DEMS involved 13 domain entities (such as User or Task) with up to 10 operations per entity (such as Create or Delete). DEMS was implemented with 18,823 LOC of C# code, contained in 117 classes covering GUI, service and database layers.

![Diagram of DEMS](https://via.placeholder.com/150)

Figure 7. A recurring pattern of components

Each combination of entity-operation is implemented by a pattern of collaborating components, two of which we see in Figure 7. Each such pattern involves classes from four system layers. Each box in Figure 7 contains a number of classes pertaining to user interface, business logic, database communication or database table definition layer.

Some of the concerns in DEMS are domain entities, operations and the four system layers shown in Figure 7.

Separating domain entity concern would mean that any entity-specific code would have to be isolated in a form that could be injected into the rest of DEMS using some composition mechanism. Operation concern is symmetric to domain entity concern, and its separation would require a similar solution.

Separation of concerns along the domain entity or operation dimension is difficult because of much differences in requirements for specific domain entities (such as User or Task) operations that apply to different entities (such as CreateUser or CreateTask). The essence of difficulties is the same as in the case of Buffer library, namely (1) much variation in the impact of different domain entities on operations, and (2) subtle, ad hoc interactions between concerns.

![Diagram of Hierarchy](https://via.placeholder.com/150)

Figure 8. Hierarchical unification of similarities

Now we look at the problem from the generic design perspective. There are many similarities among patterns of components implementing the same operation for different entities. There are also differences among patterns caused by different meaning of domain entities: For example, operation Create for a Task required different types of data entry and data validation than Create for a User. Ad hoc, induced by real-world DEMS requirements, nature of difference among patterns makes it difficult to design “generic pattern” using conventional techniques, but such a solution can be built with XVCL.

Figure 8 shows an outline of DEMS as a generic C#/XVCL x-framework. At Level 3, each group of operations such as
Separation of concerns at the design/implementation level, is true for generic design. When conventional techniques fail to achieve clean separation of concerns, and how generic design, by looking at the problem from a different angle, enhances the visibility of concerns. Now, we summarize observations, trying to distill observations that carry some more general message from those that are specific to our examples or to the use of XVCL.

Both SoC and generic design are realized by a mixture of top-down and bottom-up activities.

In SoC, first intentions are conceived at the concept level, and then we try to separate concerns at the design and implementation levels. Moving from the concept level down to the design and implementation, we observe the nature of concern design/implementation and identify yet other “lower-level” concerns.

In generic design, first we identify similarity patterns inherent in application domain concepts. In case of platforms such as JEE or .NET, we also consider recurring patterns of program organization induced by a platform, as we can expect to see them in any program developed on a given platform. Then, as we design and implement a program (or work with an existing program as in our example), we observe similarities in the actual program structures. For significant groups of such similar program structures, we design generic, adaptable representation.

At times, SoC cannot be achieved at the actual program level, using features of conventional programming languages. The same is true for generic design. When conventional techniques fail to deliver a workable solution, AOP and XVCL try to overcome the problem at an extra meta-level plane.

Separation of concerns at the design/implementation level, increases genericity of program structures. We can view program structures as being “parameterized” by concerns. By composing concerns, we instantiate program structures in variant forms. In that sense, program structures gain genericity and reusability due to SoC. We observe this in the case of concerns that can be separated using conventional programming techniques (such as modularization or generics), as well as concerns that can be separated by supporting techniques such as AOP, MDSOC, AHEAD, JEE containers or XVCL.

In case of separable concerns, there may be still a room for generic design, as program structures parameterized by concerns may still exhibit similarity due to yet other reasons not related to given concerns. For example, we can apply AOP to separate certain aspects, but modules of primary decomposition may still contain similarities induced by similar user-level requirements. These similarities create opportunities for generic design to further simplify software solution.

We believe the above observations are general. Our discussion of “difficult” concerns, becomes necessarily dependent on a technique used for generic design in our experiments, that is XVCL. In both examples discussed in Sections 4 and 5, we can see an element of SoC, however we give priority to one concern at the expense of others. In the Buffer library, we bet on “buffer element type” concern. XVCL variables set in top-most SPC are all related to this concern and they navigate the process of adapting x-frames below. These variables and XVCL constructs controlled by them enhance the visibility of the “buffer element type” concern. We can see the impact of buffer element types on generic templates (x-frames below SPC), an other x-frames adapted from there.

XVCL representation improves the visibility of other concerns, due to groupings of similar classes into groups, but here the separation of concerns is less systematic.

In the DEMS example, we give priority to separating “operation” concern over “domain entity” concern. A criterion in making this decision is the extent of similarity in operations across domain entities as opposed to domain entities across operation. The perspective that raises more easy to spot similarities gets more attention than the other competing perspective where similarities are less evident.

Focusing on unifying similarity patterns with generic design representations, in both examples, we also achieve selective and imperfect separation of concerns.

Our technology-dependent experiences seem to point to observations of a general nature: The concept of similarity is less formal than the concept of cleanly separated concerns. We can identify similar program structures by top-down domain analysis, combined with bottom-up analysis of design and code (possibly supported by clone detector [2][25]). We can zoom into similarity areas that are significant. Having identified a group of similar program structures, we can always analyze the exact differences among them.

While it is relatively easy to find similarities, spotting the exact impact of “difficult” concerns is more difficult. Focusing on similarities, we do not even have to fully understand the exact nature of a given concern or complex interactions among the concerns. Instead, we stay at the level of observing the symptoms of net effect of concern interactions.

7. Related work

Modular decomposition with information hiding [35], macros, generics in Ada or Java [22], templates in C++, other forms of parameterization such as higher order functions [42], inheritance with dynamic binding, and design patterns [21] are some of the conventional design techniques to achieve genericity. Aspect-Oriented Programming (AOP) [28] and MDSOC [41] support genericity by separating cross-cutting concerns. In AHEAD [6] (based on the earlier Batory’s work on GenVoca), genericity is
supported by feature composition and refinement. Many techniques described under the umbrella of generative techniques [14], notably meta-programming with C++ templates, achieve generativity as well as certain forms of separation of concerns. Domain analysis [37] is essential in identifying high-level, large granularity patterns of similarity. Generic solutions unifying such patterns are most beneficial for programmer’s productivity as they can significantly reduce the size and complexity of the solution. Software architectures [10][12], architectural styles [39] and patterns [12] help developers avoid repeatedly designing the same solution by providing component plug-in plug-out capability. Component platforms such as JEE™ or .NET™, provide also an infrastructure for reuse of pre-defined common services.

Code cloning has received much attention in research. As clones are closely related to the notions of similarity patterns and generativity, we discuss them in this section. Cloning has been studied in the context of re-engineering [8], refactoring [19] and clone detection [8][25][2]. In an empirical study of cloning practices Kim et al. [29] observed that “Limitations of particular programming languages produce unavoidable duplicates in a code base”.

8. Conclusions
We zoomed into the situations where separation of concerns becomes difficult. We showed that analysis of similarities pays attention to concerns, and generic structures can enhance the visibility of inseparable concerns, offering a weaker, but still useful form of separation of concerns. We believe the reason why generic design can penetrate software areas deeper than separation of concerns (SoC) is that generic design is based on the notion of similarity of program structures which is less formal and rigorous than separation of concerns. In some cases, this makes generic solutions technically easier to achieve than SoC.

In the paper, we made yet other observations, in the form of hypothesis rather than claims, about the general inter-play between SoC and genericity: There is an overlapping area where the goals of SoC and generic design, as well as means to achieve them, are the same. For example, type parameterization or modularization with information hiding separates a concern and achieves generativity at the same time. We can view program structures as being “parameterized” by concerns. By composing concerns, we instantiate program structures in variant forms. In that sense, program structures gain generativity and reusability due to SoC. In case of separable concerns, there may be a room for generic design to further improve engineering qualities of a program solution, as program structures parameterized by concerns may still exhibit similarity due to yet other reasons not related to given concerns. For example, we can apply AOP to separate certain aspects, but modules of primary decomposition may still contain similarities induced by similar user-level requirements.

In our current and future work, we conduct comparative studies to zoom deeper into the interplay between SoC and generic design principles, and further test our observations. We hope the proponents of techniques that have to do with SoC and generic design will contribute their solutions to selected problems. No doubt comparison of solutions developed using different techniques would allow us to see clearer the potentials and limitation of each of the discussed principles, and their synergistic application to form maintainable and reusable software representations.

Concerns related to different areas of a software system have different properties. For example, user requirement-level concerns, reflected in user interface and business logic software layers, tend to be less separable than software functions typically addressed by aspects [28]. An interesting area of study is development of a concern ontology. A concern ontology would help one express research results on SoC and generic design in more precise terms. We plan to extend our study described in this paper to cover possibly wide range of concern types.

References
[10] Bosch, J. Design and Use of Software Architectures – Adopting and evolving a product-line approach, Addison-Wesley, 2000


[19] Fowler M. Refactoring - Improving the design of existing code, 1999, Addison-Wesley


[33] Private communication with Ali Mesbah and Arie van Deursen, authors of [32]


[38] Rajapakse, D. and Jarzabek, S. “An Investigation of Cloning in Web Applications,” Int. Conf. on Web Engineering, ICWE’05, July 2005, Sydney, pp. 252-262 (also poster at WWW’05)


