Separation of Concerns and 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
Generic design and separation of concerns are two important principles to better control software complexity, during development, maintenance and reuse. This paper explores the interplay between these two principles. While there is an overlapping area where the goals, and to some extent technical means to achieve these goals, meet, there are also areas that belong exclusively to the domain of one of the principles only.

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, 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 [32] is a premier example of engineering benefits of generic design.

Generic design, as understood in this paper, aims at achieving non-redundancy, by unifying similarity patterns at design and code levels, whenever there is an engineering benefit of doing so for the purpose of program complexity reduction or reuse.

The importance of generic design in managing software complexity have been recognized for long. Macros were one of the early attempts to parameterize programs and make them more generic. Goguen popularized ideas of parameterized programming.

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.

AOD ’07, Month 1–2, 2007, City, State, Country.
Copyright 2007 ACM 1-58113-000-0/00/0004…$5.00.

[21]. Among programming language constructs, type parameterization [20] (called generics in Ada, Eiffel, Java and C#, and templates in C++), higher-order functions [20][21][40], and inheritance can help avoid repetitions in certain situations. Design techniques such as iterators, design patterns [19], table-driven design (e.g., in compiler-compliers), and modularization with information hiding [33] are supportive to building generic programs. Finally, generic solutions are a central theme in software reuse, component-based, patter-driven development (e.g., facilitated by .NET™ or J2EE™), and architecture-centric Product Line [10][12] approaches.

We can conceive a “generic design solution” as a parameterized structure that can be turned into a concrete program solution by instantiating the parameters. The nature of the parameterized structure, parameters, the mechanism of instantiating parameters, and the overall process that leads to building a concrete solution 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 factors such as robustness, adaptability, maintainability, and reusability.” [1] We can apply the SoC principle at the levels of program analysis, design and implementation [39].

First proposed by Dijkstra in early 1980’s [16] as a conceptual tool to tackle software complexity, only recently have we seen proposals of technical means to directly support SoC at the design and implementation levels [7][26][39].

The interplay between generic design and SoC is the theme of this paper. We are interested in both engineering goals addressed by the two principles and technical means to achieve these goals.

We concentrate on cases where physical (e.g., with AOP) or even conceptual separation of concerns becomes difficult. We show that in such cases generic design still can a viable strategy for managing complexity. We compare engineering qualities of software designed based on the principles of SoC and genericity, and hint at the possibility of using both principles in synergistic way.
2. Forms of SoC and links to generic design

The main goal of SoC, and AOP in particular, is the ability to deal with a concern (aspect) separately, e.g., during maintenance. Sometimes, a concern is 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. The details of a concern implementation become hidden behind the API [33]. This is an ideal situation from the engineering point of view.

To provide full localization, management of any variability within a concern should be catered for by suitable API operations. Modularization with information hiding is also a simple form of generic design. Here, a similarity pattern is reflected by API. A hidden implementation 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 trivial cases, we achieve SoC and generic design using the same technical means.

Concerns that cannot be localized in the above sense interact with other concerns - they have a crosscutting effect on other concerns, including modules of a primary program decomposition. Some of the crosscutting concerns can be modularized using AOP [26]. We can look at this situation as if ‘introductions’ and ‘advices’ were parameters of modules of primary decomposition. By separating aspect code and providing a mechanism to easily inject or take out 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 [39] and AHEAD [6] aim at building programs by composing independently defined concerns. In MDSOC, there is no primary decomposition and all concerns play the same role. 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 all the characteristics of generic design.

Component platforms hide implementation of some of the potentially crosscutting concerns, providing transparent access to them via APIs. In J2EE, containers provide general mechanism to access via APIs services whose implementation would otherwise lead to crosscutting concerns. Examples of such services include transaction management, persistence, security, authentication/authorization and session management, depending on a container used [30][31]. Without J2EE infrastructure or AOP support, these concerns have crosscutting effect.

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 any form of 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 lesser than they are today. However, some of the concerns, are so tightly coupled one with another (or with modules of primary decomposition) that their physical separation becomes unthinkable. For example, performance concern in some real-time systems 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.

We believe many concerns in application domain-specific areas, often called features [6][24], 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 and their possible limits, but also as they allow us to observe a link between SoC and generic design principles. More precisely, we hypothesize that generic design is a natural extension of SoC concepts into the areas where separation of concerns tends to show its limits. Therefore, both principles are intimately interrelated and synergistic. We illustrate our points with examples from our lab studies and industrial projects.

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. In Figure 1, shown as feature diagram [24], we see five feature dimensions, with specific variant features listed below a 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 [22].

![Figure 1. Features in the Buffer library](image)

The feature dimensions are some of the “concerns” in the Buffer library domain. For example, a developer 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 in Figure 1 would result in some “core structures” and five separately defined concerns. By composing specific features from each of the concerns into 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.

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 weaved 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 subject of 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 Buffer[T], where T is restricted to five numeric types: Int, Double, Float, Long, or Short. These classes are the same except of respective type name. In group of these five classes, and in the scope of five numeric types, the “buffer element type” concern is separated by means of type parameter with Java generics [22]. (In fact, certain limitations of Java generics make type parameterization difficult even in this simple case, but here we a

are not concerned with language-specific limitations of Java generics. The reader can find more details in [22] and in case studies posted on our Web site [42].)

However, the above solution does not apply to two remaining features in the “buffer element type” concern, namely Char and Byte. For example, 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 Buffer[T], 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.

The situation in the each of the two groups of classes HeapBuffer[T] and HeapBufferR[T] is analogous to what we’ve seen in the [T]Buffer group: Five numeric classes in each of the two groups are the same except of respective type names. In the scope of five numeric types, the “buffer element type” concern is separated by means of type parameter with Java generics. In the scope of all seven buffer element types, we must use means described in points 1 and 2 above to achieve separation of concerns.

Looking beyond 21 classes in groups [T]Buffer, HeapBuffer[T] and HeapBufferR[T] considered above, we see more subtle code dependencies on “buffer element type” concern. For example, in method slice() (Figure 2) a constant reflects the length of a buffer element.

/* 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() 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. 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. Other concerns in the Buffer library are even more difficult to trace and separate for the above reasons.

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. Concept of similarity is less formal than a concept of cleanly separated concern. It is easier to find similarities than to spot the exact impact of concerns. Focusing on similarities, is more pragmatic as 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. We can use a clone detector [2][23] together with top-down domain analysis to zoom into similarity areas that are significant.

As we shall see from the examples, we are still doing a fair amount of SoC when unifying similarity patterns. However, we are doing this in an approximate way, only as far as it is practical.

We have the following seven groups of similar classes in the Buffer library [22]:

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 still similar classes.

The reader should notice 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 unify similarity patterns with help of a generative technique of XVCL [42]. We define generic solutions using conventional programming technologies (languages and platforms) together with XVCL. We call this method for generic design mixed-strategy.

In mixed-strategy approach, 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 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 constructs.

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 a mixed-strategy representation. An executable program synthesized from a mixed-strategy representation may still contain repetitions, if it is desire. Sometimes repetitions are unavoidable given a programming technology used (e.g., on J2EE or .NET platforms [43], see also [27]), and/or taking into account possibly yet other design goals a program must meet [22].

A building block of a mixed-strategy generic 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 for Buffer classes, shown in Figure 3. An arrow between two x-frames: X → Y means “X adapts Y”.

![Java/XVCL x-framework for buffer classes](image)

**Figure 3. A Java/XVCL mixed-strategy solution for Buffer classes**
We synthesize all the classes in each of the seven groups of similar classes from the x-framework shown in Figure 3. 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.

In our example, for the sake of comparison, we designed x-framework so that classes produced by the XVCL Processor are no different from the original classes in the Buffer library.

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 a mixed-strategy 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 4 blows up the a fragment of the Java/XVCL x-framework shown on the left-hand-side of Figure 3.

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 multiple contexts.

For example, names and other parameters of the seven similar classes [T]Buffer are represented by XVCL expressions in the x-frame [T]Buffer.gen (Figure 4).

A <set> command assigns a value to a variable. For example, a second command in the SPC of Figure 4 assigns a values listed on the right-hand-side to a variable named elmType. Expression @elmType 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, elmType is one of the variables that mark customizations related to “buffer element type” concern.

![Figure 4. A Java/XVCL x-framework for seven [T]Buffer classes (partial)](image-url)
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>` `<insert>` into `<break>` are analogous to AOP’s mechanism for weaving advices at specified join points. The difference is that while AOP specifies join points in a descriptive way, `<insert>` modify x-frames in arbitrary ways, at any explicitly designated `<break>` points.

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 4 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 5 shows generic method `slice()` from Direct[T]Buffer[S|U] classes (a specific instance of method `slice`) is shown in Figure 2). 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).

```
slice.gen // generic method slice()
/*Create a new byte buffer containing a shared subsequence of this buffer's content. */
public @elmtType Buffer slice() {
    int pos = this.position();
    int lim = this.limit();
    assert (pos <= lim);
    int rem = pos <= lim ? lim - pos : 0;
    int off = pos < <= @elmtSize);
    return new Direct[@elmtType]@[elmtSize]
    (this, -1, 0, rem, rem, off);)
```

**Figure 5. 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 XVCL 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.

5. Another example of “difficult” concerns

A Domain Entity Management System (DEMS) is contributed by ST Engineering 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`).

There were many similarities in how the same operation was to be designed and implemented across entities. However, there were also differences 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.

The design and implementation of operations for various entities were characterized by a similar pattern of collaborating classes across GUI, service and database layers. Each box in Figure 6 represents a number of classes, with much similarity across classes implementing similar concepts in the same types of operations for different entities.

DEMS was implemented with 18,823 LOC of C# code, contained in 117 classes covering GUI, service and database layers.

Some of the concerns in DEMS are domain entities, operations and the four system layers shown in Figure 6.

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.

Now we look at the problem from the generic design perspective. Figure 7 shows an outline of DEMS as a generic C#/XVCL x-framework. At Level 3, each group of operations such as CreateUser, CreateTask, … has been represented by one generic operation parameterized by a respective domain entity. Similarities among different operations for the same entity (e.g., CreateUser, UpdateUser, …) are unified at Level 4. X-frames at Level 4 represent generic classes, building blocks for DEMS operations, as indicated by x-frames referenced from more than one operation (e.g., generic classes labeled with CU are reused in construction of Create and Update for various entities).

**Figure 7. Hierarchical unification of similarities**

The top-most SPC x-frame contains global controls and parameter settings that specify the overall process of constructing DEMS from x-frames below. “DEMS template” at Level 2 defines the structure of DEMS architecture, that is the organization of component patterns implementing various operations plus any other functions supported by DEMS, not discussed in this example.

The C#/XVCL mixed-strategy representation contains C# code needed to produce all the DEMS operations, and also information helpful in maintenance/reuse, such as the record of similarities and differences among operations for different entities. The size of the C#/XVCL representation is 68% smaller (in terms of lines of code, without blanks or comments) and conceptually simpler than its C# counterpart. Size reduction also leads to improved maintainability. Typical enhancements (such as change of business rules for domain entities or adding/deleting a new entity or operation) require less modifications with smaller impact in the C#/XVCL DEMS as compared to its C# counterpart. For example, it is eight times faster to create a new domain entity in the C#/XVCL DEMS (133 LOC and 2 man-hours), as compared to C# DEMS (1440 LOC and 16 man-hours). Finally, it is quite easy to reuse the C#/XVCL DEMS in other similar command and control systems.

Adding more operations and entities to DEMS in C# increases the complexity by a constant value, independently of the similarity of new operations/entities to existing ones. In mixed-strategy C#/XVCL, complexity grows proportionally to the level of novelty that a new operation/entity brings. New operations/entities that differ little from existing ones require little new code to be written.

It is interesting to note that, despite striking similarities, using conventional OO and component-based design methods, it is not possible to build a generic solution for groups of operations such as CreateUser and CreateTask. To implement such generic operations, one would have to first unify groups of classes in GUI, Service, Entity and DB layers such as Create[entity-type]Form, Update[entity-type]Form, etc. However, the nature of variations in business logic across operations for different entity types is such that neither inheritance nor type parameters (such as in generics proposed for C# [25]) fail to implement operations in generic way. Therefore, each operation must be implemented as a separate instance of the component pattern, complicating DEMS.

### 6. Analysis

Interesting question is how are generic design solutions described in last two sections related to SoC?

In case of separable concerns, SoC achieves goals of generic design, and generic design reflects SoC. We observe this in the case of localized concerns or concerns that can be separated by supporting techniques such as AOP, MDSOC, AHEAD, J2EE containers or XVCL. Our tutorial example of Notebook Product Line (at XVCL web site xvcl.comp.edu.sg) illustrates such separable concerns. SoC and generic design goals are in sync and go hand-in-hand.

In case of “difficult” concerns, the situation is different. In both examples discussed in Section 4 and 5, we can see an element of SoC, however we give priority to one concern on 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 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 do selective and imperfect separation of concerns.

### 7. Related work

Modular decomposition with information hiding [33], macros, generics in Ada or Java [20], templates in C++, other forms of parameterization such as higher order functions [40], inheritance with dynamic binding, and design patterns [19] are some of the conventional design techniques to achieve generivity. Aspect-Oriented Programming (AOP) [26] and MDSOC [39] support generivity by separating cross-cutting concerns. In AHEAD [6] (based on the earlier Batory’s work on GenVoca), generivity is supported by feature composition and refinement. Many techniques described under the umbrella of generative techniques [14], notably meta-programming with C++ templates, achieve generivity as well as certain forms of separation of concerns. Domain analysis [35] 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 [37] and patterns [12] help developers avoid repeatedly designing the same solution by providing component plug-in plug-out capability. Component platforms such as J2EE™ 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 genericity, we discuss them in this section. Cloning has been studied in the context of re-engineering [8], refactoring [17] and clone detection [8][23][2]. In an empirical study of cloning practices Kim et al. [27] observed that “limitations of particular programming languages produce unavoidable duplicates in a code base”.

8. Conclusions

We presented empirical and analytical comparison between separation of concerns and generic design. We have shown the areas where the two principles meet. We zoomed into situation where separation of concerns becomes difficult, and how generic design can pick up from there, still contributing useful engineering solutions. In conclusions, both principles can be applied in a synergistic way, complementing each other.

The results described in this paper are preliminary and should be treated as a hypothesis. We are in the process of conducting experiments that zoom deeper into the interplay between SoC and generic design principles. 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.

References

[10] Bosch, J. Design and Use of Software Architectures – Adopting and evolving a product-line approach, Addison-Wesley, 2000
[17] Fowler M. Refactoring - improving the design of existing code, 1999, Addison-Wesley
[18] Fowler, M. Analysis Patterns: Reusable Object Models, 1997, Addison-Wesley

8


[31] Private communication with Ali Mesbah and Arie van Deursen, authors of [30]


