Computer Science and Information Systems 13(3):731–758 DOI: 10.2298/CSIS160129028J

On Interplay between Separation of Concerns and Genericity Principles: Beyond Code Weaving

Stan Jarzabek[1] and Kuldeep Kumar[2*]

[1] Faculty of Computer Science,
Bialystok University of Technology, Poland
s.jarzabek@pb.edu.pl
[2] Department of Computer Science and Information Systems,
Birla Institute of Technology and Science (BITS), Pilani, India
kuldeep.kumar@pilani.bits-pilani.ac.in
*Corresponding Author

Abstract. Ideally, we would separate concerns by designing a program so that each concern is contained in a module. Unfortunately, we often have to deal with concerns that cannot be modularized, but instead cross-cut modules of our primary decomposition. Some of the cross-cutting concerns can be separated using compositional techniques such as Aspect-Oriented Programming (AOP) that weave code into modules at specified program points. Here, we focus on cross-cutting concerns that would not be easily separable with code weaving compositional techniques due to their frequent and complex interactions with the modules of primary decomposition. Separation of Concerns (SoC) and genericity are two important Software Engineering principles to better control software complexity during development, maintenance, and reuse. In this paper, we study the interplay between these two principles, showing that there is an overlapping area where the goals of SoC and genericity, as well as means to achieve these goals, are the same. We make a case that by integrating the principles of SoC and genericity we can achieve non-redundancy, and at the same time enhance the visibility of inseparable concerns, offering a weaker, but still useful form of SoC. We illustrate the points we make with examples of program representations built with the Adaptive Reuse Technique (ART) that supports both SoC and generic mechanisms.

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

1. Introduction

1.1. Background

Recurring patterns in software requirements and design spaces, standardization of design solutions, as well as ad hoc copy-paste-modify practice lead to software similarity patterns of varying size and type, spreading within or across programs [1]. *Genericity* is a common way to avoid these redundancies. It is a central theme in software reuse, component-based, pattern-driven development (e.g., facilitated by

.NET™ or JEE™), and architecture-centric Software Product Line (SPL) approaches [2][3][4]. The Standard Template Library (STL) [5] is a premier example of engineering benefits of generic program representations.

Genericity, as understood in this paper, aims at achieving non-redundancy, by unifying software similarity patterns with generic program representations to achieve program simplification, reusability, or maintainability.

In this paper, we do not make any specific assumptions about the type, granularity of software similarity patterns, or the nature of differences among them. The importance of genericity in managing software complexity has been recognized for long. Macros were an early attempt to make programs more generic. Goguen popularized the idea of parameterized programming [6]. Among programming language features, type parameterization [7] (called generics in Ada, Eiffel, Java and C#, and templates in C++), higher-order functions [8], and inheritance can help avoid repetitions in certain situations. Design techniques such as iterators, design patterns [9], table-driven design (e.g., in compiler-compilers), and modularization with information hiding [10] can help building generic programs. Generative programming techniques, such as XML-based Variant Configuration Language (XVCL) [11], 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 program representation" as a parameterized structure that can be turned into a concrete, custom program solution by instantiating the parameters. 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 depend on the techniques used for generic design. Parameterized structures can be as simple as generics or templates, or as complex as an Object-Oriented (OO) framework or a generic parser. In an OO framework, parameters are abstract classes and design patterns. Parameters for a generic parser are encoded in BNF (Backus Normal Form) definition of a programming language syntax.

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. The aim is to limit interactions between concerns as much as it is possible.

The term "separation of concerns" in software engineering was introduced by Dijkstra in 1974 as a conceptual tool to tackle software complexity [12]. SoC principle can be applied at the levels of program analysis, design, and implementation [13].

1.2. Problem Statement

Ideally, we would like to separate concerns by designing a program so that each concern is contained in a module. Indeed, some of the concerns can be nicely aligned with modular decomposition. Unfortunately, we also have to deal with concerns that cannot be modularized, but spread through the modules of our primary decomposition instead. Delocalized concerns that cannot be modularized within a given primary modular

decomposition structure are called *cross-cutting concerns* [14]. It should be noted that cross-cutting need not be an inherent property of a concern: A concern that is cross-cutting in one modular decomposition, might not be cross-cutting in another decomposition.

Still, some of the cross-cutting concerns can be defined separately and then weaved into the code of the primary decomposition modules at specified program points (e.g., before or after a program function is called) using meta-program-level compositional approaches such as Aspect-Oriented Programming (AOP) [14]. A number of other compositional approaches for handling cross-cutting concerns have been proposed in academic research (Algebraic Hierarchical Equations for Application Design (AHEAD) [15], Multi-Dimensional Separation of Concerns (MDSOC) [13], or XVCL [11]), and in industrial practice ([16][17][18][19]). Compositional approaches provide a useful way to separating concerns when interaction between a concern and primary decomposition code are infrequent and occur at well-defined program points. However, they soon reach their limits when concerns become tightly coupled with modules of primary decomposition, that is, interactions between concern and primary decomposition code are many and occur at arbitrary program points. As it has been convincingly demonstrated in a study by Kästner et al. [20], an attempt to separate such tightly coupled concerns with code weaving in AOP style leads to overly complex, unworkable program representations. In their study, authors used AspectJ to separate application functional concerns (features) in a way that they could be composed together in various combinations to fit application reuse contexts.

1.3. Hypothesis

In this paper, we focus on tightly coupled cross-cutting concerns that are not easily separable with compositional weaving techniques due to their frequent and complex interactions with modules of primary decomposition. We propose to look at the problem from a perspective that integrates SoC and genericty principles into a unified framework that helps understand interactions among tightly coupled concerns. In addition to dealing with the problem by weaving the concern code that can be conveniently separated, we propose generic mechanisms that keep inseparable concerns together with primary decomposition modules, as generalized parameters.

We hypothesize that there is an overlapping area where the goals of SoC and genericity, as well as means to achieve these goals, are the same. Therefore, both principles can be neatly integrated to exploit the strength of each principle and avoid its pitfalls. By integrating the principles of SoC and genericity we can achieve non-redundancy, and at the same time enhance the visibility of inseparable concerns, offering a weaker, but still useful form of SoC. We hypothesize that genericity is a natural extension to the principle of SoC into the areas where SoC tends to show its limits. Hence, both principles are intimately interrelated and synergistic. We believe the reason why genericity can penetrate software deeper than SoC is because it is based on the notion of unifying similar program structures, which is less formal and rigorous than SoC.

We further analyze and argue in support of the above hypothesis in the remaining paper illustrating our points with examples from lab studies and industrial projects. We consider this analysis the main contribution of our paper. *To our best knowledge, our*

study is the first attempt to investigate the relation between SoC and genericity. We communicate our findings in the form of observations (or a hypothesis, at best), not claims.

We also discuss the engineering goals addressed by the two principles, and technical means to achieve these goals. We use our own meta-programming technique and tool, the Adaptive Reuse Technique (ART) [21][22] to demonstrate the points we make in the paper regarding SoC and genericity, and possible ways to handle tightly coupled concerns. The ART supports both SoC and generic mechanisms. It is amenable to automation—i.e., concerns separated or represented generically—can be selectively included into the code of modules of primary decomposition.

SoC cannot be regarded as a purely theoretical problem, but rather as a practical problem whose solution should bring specific engineering benefits in terms of program simplification, improved maintainability, or reusability. Therefore, solutions that are theoretically possible but do not bring any desirable engineering benefits are not worth considering. We evaluated engineering properties of program representations built with the ART and its predecessor XVCL [11] in previous papers [4][22][23][24][25][26][27][28], and we refer interested readers to these earlier publications. The essential novelty and contribution of this paper is our analysis of SoC and genericity principles. In this paper, we use the ART merely to illustrate the interplay between SoC and genericity principles. One might use another technique if it allowed to better illustrate the point we make about SoC and genericity.

This paper is an extended version of our work originally presented in the PTI KKIO Software Engineering Conference held at Miedzyzdroje, Poland in 2015 [29]. Based on comments from reviewers of the first round of revisions, we changed the way we positioned the paper, clearly identifying the class of tightly coupled concerns in the context of other concerns. We extended discussion of related work on cross-cutting concerns, including academic research as well as industrial solutions to the problem.

The paper is organized as follows: Section 2 discusses various forms of SoC and their links to genericity. In Section 3, we show examples of concerns that are difficult to separate. After providing brief overview of the ART in Section 4, we show unified program representations for one of such examples. Section 5 discusses yet another example, from application software. We analyze observations in Section 6. Related work is presented in Section 7. Section 8 concludes the paper.

2. Relation between SoC and Genericity

In this section, we illustrate different forms of SoC and discuss their relation with the principle of genericity. We show how the both principles are intimately interrelated. Principle of SoC aims at dealing with each concern separately from other concerns. Separating concerns at the concept level is useful, but the benefits amplify if we can also separate a concern at the software design and implementation levels.

Modularization is one of the most natural conventional ways to achieve SoC [10]. Some concerns can be nicely aligned with modular decomposition. In such cases, the concern is localized to a single module (a component, class, or function, for example) or a group of modules (e.g., a component layer), and an Abstract Program Interface (API) is exposed to its clients. The implementation details of the concern become hidden

behind that API. But, this is an ideal situation from the engineering point of view. To provide full localization of a concern, management of any variability within the concern should be either a hidden part of the concerned module, or should be supported by suitable API operations. A modularized and localized concern can be easily added to or taken out from programs. It makes the programs more generic.

Modularization is also a simple form of generic design. Here, a similarity pattern is reflected by an API. Design decisions hidden in the module (e.g., data representation) play a role of parameters that make a module generic. Instantiation of such a "generic module" is done by choosing specific design decisions (e.g., data structures), and implementing API operations in terms of this particular choice. By localizing concerns within modules, we achieve SoC and genericity at the same time.

Concerns that cannot be localized in the above sense have a crosscutting effect on other concerns. Some of the crosscutting concerns can be modularized at the extra meta-level plane using various techniques such as AOP [14], AHEAD [15], or MDSOC [13].

In AOP, 'introductions' and 'advices' play the role of parameters for modular decomposition. We can easily inject or take out some of the aspect's code from modules. This makes modules more generic. 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 genericity, and we can view AOP as a kind-of generic design mechanism. In fact, AOP has been considered as a technique for building SPL architectures [1][3], which justifies the above interpretation.

MDSOC [13] and AHEAD [15] aim at building programs by composing independently defined concerns. In MDSOC, there is no primary decomposition. It means that all the concerns are treated equally. 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 is a generic program representation.

Component platforms hide implementation details of some of the potentially crosscutting concerns and provide 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. Depending on the container used, such services include transaction management, persistence, authentication/authorization, security, and session management [30][31]. While not completely eliminating, the JEE™ infrastructure makes crosscutting effect more visible and reduced to calls to the container's API operations.

Another example situation [16] considers a case of separating concerns in User Interfaces (UIs). Consider a situation when designer wants to builds a UI page. One of the designer philosophies is that the designer wants to see all page-specific information, such as field presentation, security, and layout, at one place as it centralizes the perspective. Another designer philosophy may focus on concern centralizations. In that case, the designer may be interested in seeing all the field presentation at one place and similarly for other concerns. While the first designer philosophy may match to Ruby on Rails or Django design, the second brings development and maintenance benefits. This example suggests that there exist situations where design philosophy violates the SoC. The above examples illustrate that whether a given concern has a crosscutting effect or

not may depend on many factors such as the technology, design philosophy, language instruments, and other major mechanisms used in the design of a particular program.

3. Examples of "Difficult" Concerns

Discussion in Section 2 suggests that principle of SoC contributes to the goals of genericity. Some of the concerns are easy to separate "physically" at the levels of program design and code. However, some concerns are so tightly coupled with one another or modules of primary decomposition that their physical separation becomes difficult. These couplings may not be fully perceived at the concept level, but as analysis of the exception handling concern shows "the devil is in the details" [32]. Exception handling is an example of a "difficult" concern. "The main problem is that realistic software systems exhibit very intricate relationships involving the normal-processing code and error recovery concerns" [32]. Experiments with EHAB (Exception Handling Application Block) on .NET™ [33] also revealed difficulties to separate exception handling from the rest of the code.

Performance in real-time systems is another example of a "difficult" concern. It has pervasive impact on many design decisions. While we can conceive and express 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 [15][34], are difficult to separable just as performance concern is difficult to separable in time-critical systems.

Our next example is from the Java Buffer library. The Java Buffer library is a part of java.nio.* packages in JDK since version 1.4.1. It implements containers for data in a linear sequence for reading and writing. It consists of buffer classes that differ from each other based on possible values of the involved features (buffer element type, for example). Fig. 1 shows a feature diagram [34] for the Java Buffer library with such five feature dimensions. Specific variant features are listed below the 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 [25].

Each class name reflects combination of specific features implemented into the given class. Class 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 (Big-Endian) or L (Little-Endian). 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. For example, class name "DirectCharBufferRS" refers to a Read-Only buffer of characters, implemented with Native byte ordering using Direct memory allocation scheme. Classes whose names do not include 'R', by default are 'W'—Writable.

Feature dimensions are some of the "concerns" in the Java Buffer library domain. A developer or maintainer of the library may be interested to know: "how does an element

type (or memory allocation scheme, for example) affect implementation of buffer classes?", "can I separate certain concerns so that specific features can be incorporated into buffer classes, and relevant code maintained, in separation from the other concerns?".

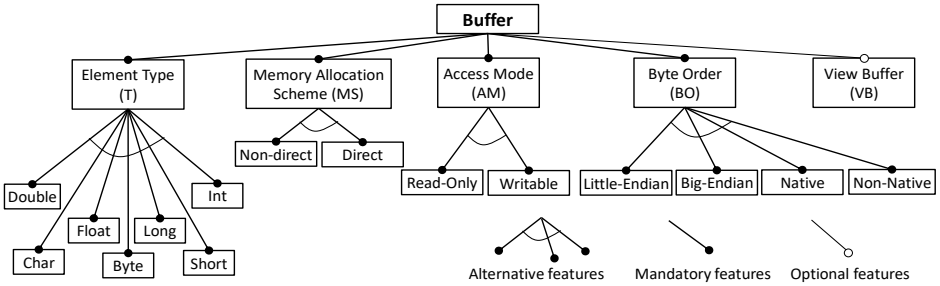

Fig. 1. Features in the Java Buffer library. [Buffer class names are derived from a template: [MS][T]Buffer[AM][BO], where MS – memory allocation scheme: Heap or Direct; T – element type: Double, Char, Float, Byte, Long, Short, or Int; AM – access mode: W - Writable (default) or R - Read Only; BO – byte ordering: S - non native or U - native, B - Big Endian or L - Little Endian; VB – view buffer: Char, Int, Double, Float, Long, or Short].

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

To make SoC worthwhile, the number of "core structures" should be considerably smaller than the number of specific buffer classes (around 100). 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 of parameterized representation, comprising design and code of the 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 the SoC technique used. For example, in AOP, "core structures" correspond to some classes of a primary decomposition, and concerns are 'introductions' and 'advises' to be weaved into the 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.

Let us now look into the issues involved in trying to separate concerns in the Java Buffer library. To separate a concern, we must first see how a given concern affects the structure of the library and implementation of the 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.

We focus on the concern "buffer element type" T and observe its impact on the buffer classes. There is no problem to do so in five classes [T]Buffer, where T is restricted to

five numeric types: Int, Double, Float, Long, and Short. These classes are almost same except with the respective names affected by element type, highlighted in bold in Fig. 2.

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

Fig. 2. Code Snippet for buffer class LongBuffer.java

In the scope of these five numeric types, the "buffer element type" concern can be separated by means of type parameter with Java generics [25]. However, there are certain limitations with Java generics that make type parameterization difficult even in this simple case. But, here we don't aim to worry about *language-specific limitations of Java generics*. Interested readers can find more details in [25].

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 must be executable on their own, then the answer is *no*. "Buffer element type" is an integral part of any possible primary decomposition in the above sense, and we wouldn't have buffer classes without mentioning "buffer element type", in either specific (such as Int or Short) or generic form. However, if we relax the requirement that modules of primary decomposition must be executable on their own, then we could consider "buffer element type" as an aspect, provided that we can weave code related to the type at specified join points in classes of primary decomposition. But, points of variations among the buffer classes as highlighted in bold in Fig. 2 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, but it seems that such a solution 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 our analysis to the two remaining features, namely 'Char' and 'Byte', in the concern "buffer element type". We found that class CharBuffer.java has different implementation for the method *toString()* than any of the numeric buffer classes. Method *toString()* converts a buffer element to a character string. In CharBuffer.java, method *toString()* is trivial and just returns the buffer element. While in numeric buffer classes, this method performs a proper conversion. In addition, CharBuffer.java has a number of extra methods that are not needed in numeric buffer classes. Situation in ByteBuffer.java is also analogical to CharBuffer.java. There are a few extra methods in ByteBuffer.java that do not appear in numeric buffer classes or CharBuffer.java.

Now, 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.java, while 'Long' is part of method names in LongBuffer.java).
2. We must selectively insert extra methods into certain buffer 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 SoC. However, it creates a challenge for generics as extra methods cannot be represented by generic types.

With regard to the concern "buffer element type" T, the groups of buffer classes Heap[T]Buffer and Heap[T]BufferR have the same situation as the group [T]Buffer has. But, separation of "buffer element type" concern becomes more problematic when we look beyond the 21 classes in these three groups (i.e., [T]Buffer, Heap[T]Buffer, and Heap[T]BufferR). In other buffer classes, we found 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 Fig. 3, a constant in bold is equal to the length of the buffer element minus one, so the constant is 0 for 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);
}
```

Fig. 3. Method *slice()* in DirectByteBuffer.java

We further analyze the impact of concerns other than "buffer element type" on implementations of the buffer classes. We found that classes implementing 'Direct' memory allocations scheme differ a lot from analogical classes implementing 'Heap' memory allocation scheme. Similarly, 'Writable' classes differ from analogical 'Read-Only' classes significantly. With that, the visibility of concerns becomes blurred. Hence, 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 SoC difficult.

4. Switching Perspectives

Section 3 concludes that separation of "buffer element type" concern (and similarly other concerns) becomes "difficult" due to:

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 software similarity patterns in program structures that result in interactions among combinations of concerns implemented into classes. Instead, 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 Java Buffer library [25]:

1. [T]Buffer: seven classes that differ in buffer element type, T: Byte, Char, Int, Double, Float, Long, Short
2. Heap[T]Buffer: seven classes, with memory allocation scheme 'Heap', that differ in buffer element type, T
3. Heap[T]BufferR: seven 'Read-Only' classes, with memory allocation scheme 'Heap', that differ in buffer element type, T
4. Direct[T]Buffer[S|U]: 13 'Direct' classes for combinations of buffer element type, T, with byte orderings: S—Non-native or U—Native (note that byte ordering is not relevant to buffer element type 'Byte')
5. Direct[T]BufferR[S|U]: 13 'Read-Only' and 'Direct' classes for combinations of parameters T, S and U (byte ordering is not relevant to buffer element type 'Byte')
6. ByteBufferAs[T]Buffer[B|L]: 12 classes for combinations of buffer element type, T, with byte orderings: B—Big-Endian or L—Little-Endian. T here denotes all seven buffer element types except 'Byte' (i.e., equivalent to VB—View Buffer)
7. ByteBufferAs[T]BufferR[B|L]: 12 'Read-Only' classes for combinations of parameters T (except 'Byte'), B and L.

We see that similarities among buffer classes manifest themselves as methods and attribute declarations that appear in different classes in similar form. However, some classes contain extra methods that do not appear in other still similar classes.

We 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 the help of a generative technique of the ART. As we define generic solutions using conventional programming technologies (languages and platforms) together with the ART, we call the approach *mixed-strategy*. We first present brief overview of the ART which is followed by our *mixed-strategy* solution of the Java Buffer library.

4.1. An Overview of the ART

The ART is a meta-programming technique and tool that works on the principle of representing each group of similar program structures found in the software in forms of non-redundant, adaptable, and reusable meta-components, we called ART templates. An ART template is a file with original program code (i.e., native language of the software, for example Java in the Buffer Library) instrumented with ART commands for ease of customization. Table 1 gives summary of the selected ART commands.

Table 1. Summary of Selected ART Commands

Syntax	Command Definition
#adapt template_name or: *#adapt*: template_name <customizations> *#endadapt*	*#adapt* command instructs the ART processor to adapt the named template and its descendants. *#adapt* may also allows to specify customizations that should be applied to the adapted template. Customizations may include any ART commands.
#output pathname	*#output* command specifies the path of the output file where the source code should be placed. The pathname can be absolute or relative path. If the output file is not specified, then the ART Processor emits the code to an automatically generated default file named *defaultOutput.txt* in the main folder of the installed ART processor.
#set var_name = val1[,val2,val3, …]	*#set* command declares an ART variable "var_name" and sets its value to a single or multi-values.
?@var_name?	A direct reference to the value of variable "var_name". Each extra '@' symbol in the front of a variable name indicates an extra level of indirection.
#break breakX or: *#break*: breakX default content *#endbreak*	*#break* marks a breakpoint at which changes can be made by ancestor template via *#insert*, *#insert_before*, *#insert_after* commands. The content under *#break* is the default content. If no *#insert* matches a *#break*, then the break's default content is processed.
#insert breakX content_body *#endinsert* *#insert-before* breakX content_body *#endinsert* *#insert-after* breakX content_body *#endinsert*	*#insert* command replaces all matching *#break*s with its content. Matching is done by a name (breakX in the example). *#insert-before* and *#insert-after* add their content before or after the matching *#break*s, without deleting their content. A single *#break* may be simultaneously extended by *#insert*, *#insert-before* and *#insert-after* commands.
#while var1[,…,varN] content_body *#endwhile*	*#while* is a generation loop that iterates over its body and generates custom text at each iteration.
#select control_var *#option* option option_body *#endselect*	*#select* allows us to choose one of the many customization options.
% comment %> comments <%	Single line comment Multiple lines comments

Despite a large fraction of code common to the group (e.g., exact code fragments or methods in the corresponding similar program structures), there can be mainly three types of differences among corresponding program structures: parametric differences (code with parametric changes), alternatives (code modifications), and extras (code insertions and deletions). For each group of similar program structures, we distill the common code into ART templates and mark the locations of variation points using ART

commands. ART commands can be used systematically to mark these variation points as discussed below:

- *Handling exact code fragments/methods:* Identical code fragments or methods can be used directly as-it-is in the corresponding ART templates.
- *Handling parametric differences:* Parametric differences such as variations in user-defined identifiers, literals, layout, types, etc. can be dealt with systematically using ART multi-value variables. Such multi-value variables can be declared using *#set* command. The value of an ART variable say varName can be referred by using expression "?@varName?".
- *Handling alternatives:* ART command *#select* allows choosing one among alternatives. Each of the alternatives is represented by a *#option* clause under *#select* command.
- *Handing extras:* ART's *insert-break* mechanism allows handling additions and deletions of extra code. ART command *#break* marks the location in the ART template where the extra code needs to be inserted. That extra code can be then injected at the marked variation point using *#insert*, *#insert-before*, and/or *#insert-after* commands.

The ART is supported by a tool, we-called the ART Processor, which interprets the ART and provides a semi-automated support for the customizations. The ART Processor is implemented in Java. It is open-source and is available in a ready-to-use form (available at https://sourceforge.net/projects/vclang). The ART Processor can be run from command-line mode as well as in using graphical user interface mode. It is also supported by editor plug-ins for Notepad++.

In the remaining section, we show how we can apply generative technique of the ART to unify similarity patterns by continuing our Java Buffer library example.

4.2. Unified Program Representation using the ART

In order to have a unified program representation for a program, we start with the concrete program, or at least with some idea of a program's component/class architecture, and its partial implementation. In case of our example, 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 the ART applied on top of Java.

We can imagine that the ART decomposes a conventional program in its own way, wrapping structures of a subject program (of any granularity and type) within ART templates to make them generic. In case of the Java Buffer library, we build a generic program representation in combination of Java and the ART. Therefore, we call our overall solution a mixed-strategy Java/ART-template solution.

Fig. 4 outlines the solution, which consists of an ART-template hierarchy in which ART templates at the lower-level serve as building blocks for the higher-level ART templates. As shown by arrows in the figure, the ART templates in the hierarchy are linked by *#adapt* commands. An arrow from an ART template A to another ART template B indicates that template B is used, after possible adaptations, to build A.

Using the ART Processor, we can derive all the buffer classes in each of the seven similarity groups mentioned above from the ART-template solution shown in the left-hand-side of Fig. 4.

The process of generating specific buffer classes from the ART-template solution is governed by ART templates defined at Level 1 and Level 2. The top-most template (Level 1) called SPC sets up global parameters, and exercises the overall control over the generation process. ART templates at Level 2 specify controls for each of the seven groups of similar classes.

Each of the ART templates at Level 3 plays the role of a template defining a common part for all the buffer classes in the respective group. For example, seven buffer classes in the group [T]Buffer are derived using ART template [T]Buffer.art. ART template [T]Buffer.spc contains specifications instructing the ART Processor how to adapt [T]Buffer.art and other ART templates at levels below it to derive classes in the [T]Buffer group. We have analogical solutions in parts of the buffer ART-template solution for other six groups of similar classes. Smaller granularity building blocks for buffer classes are defined at Level 4 (methods) and Level 5 (fragments of method implementation or attribute declaration sections).

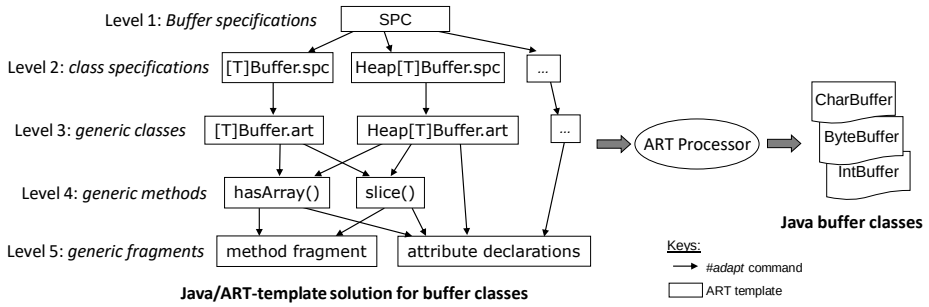

Fig. 4. A Java/ART mixed-strategy solution for the Java Buffer library

The essence of an ART template is that it can be adapted to produce its instances (e.g., specific classes in a group). Therefore, small-granularity generic solutions (represented by the lower-level ART templates) are composed, after possible adaptations, to construct required instances of higher-level generic solutions (represented by higher-level ART templates).

In our example, for the sake of comparison, we designed ART-template solution so that classes produced by the ART Processor are no different from the original classes in the Java Buffer library. The ART Processor interprets an ART-template solution starting from the SPC, traverses ART templates below, adapting visited ART templates, and emitting the custom program. By varying specifications, we can instantiate the same ART-template solution in different ways, deriving different, but similar, program components from it. In that sense, an ART-template solution forms a generic program representation that enables reuse within a single program or across programs. In the latter case, an ART-template solution implements a concept of the SPL architecture [3].

In that way, the proposed mixed-strategy approach provides a two-fold view of the software system: One is an ART-enabled generic program solution that consists of software code instrumented with the ART commands in the form an ART-template hierarchy. Another is the software system that can be generated automatically from the template-hierarchy using the ART Processor. To better see the nature of an ART-enabled generic solution and its relation to SoC, we now explain the parameterization and

adaptation mechanism, which is the "heart and soul" of how the ART achieves genericity.

4.3. Relation between ART-enabled Generic Solution and SoC

Fig. 5 shows the details of a fragment of the Java/ART-template solution shown on the left-hand-side of Fig. 4.

ART variables and expressions in the ART templates correspond to parametric differences. Typically, names of program elements manipulated by the ART, such as components, source files, classes, methods, data types, operators, or algorithmic fragments, are represented by ART expressions. Using such parameters, rather than concrete names, makes ART templates more generic, adaptable to fit into multiple contexts. For example, names and other parameters of the seven similar classes [T]Buffer are represented by ART expressions in the ART template [T]Buffer.art (Fig. 5). An ART expressions can appear anywhere in ART templates. An ART expression is enclosed between question mark "?" symbols. Expressions can be used to refer the value of corresponding ART variable. For example, expression "?@elmtType?" (line 2 in [T]Buffer.art] refers to the value of the ART variable elmtType (further details to follow).

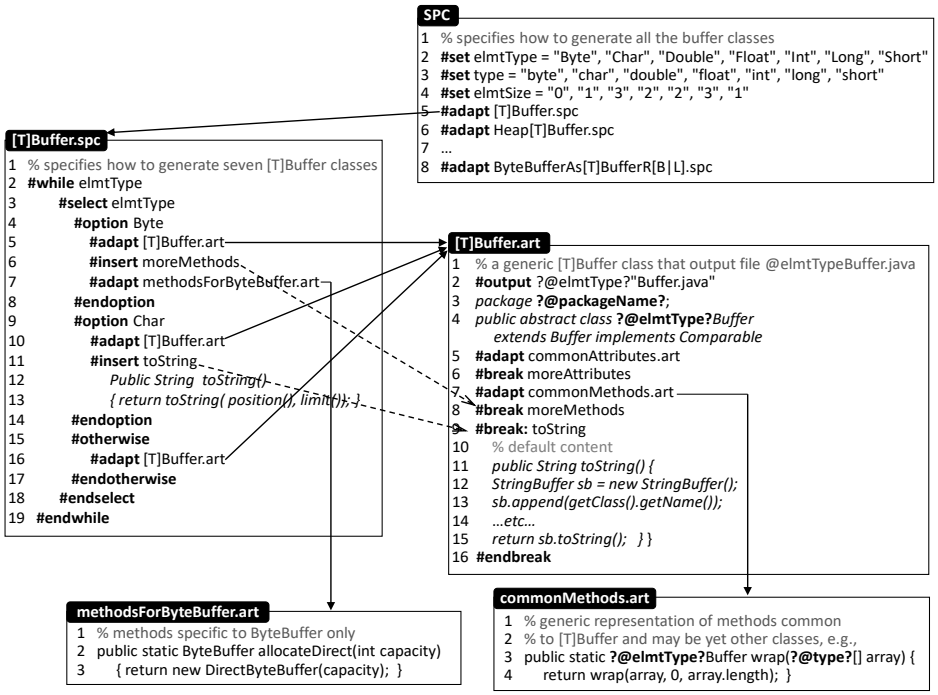

Fig. 5. A Java/ART-template solution for seven [T]Buffer classes (partial)

ART parameters play an important role of control elements that mark traces of customization changes related to a single *source*, that span across multiple ART templates. *This "source" often represents a concern or a specific feature within a concern*. For example, 'elmtType' is one of the ART variables that marks customizations related to "buffer element type" concern.

The ART Processor propagates variable values from an ART template where the value of a variable is set, down to the adapted ART templates. While each ART template usually sets default values for its variables, values assigned to variables in higher-level templates take precedence over the locally assigned default values. Thanks to this overriding rule, ART templates become generic and adaptable, with potential for reuse in unifying similarity patterns in many contexts.

Other ART 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 of concerns. Using *#select* command, depending on the value of a control variable, we can select one of many options. Options are selected based on the value of the control variable specified as attribute in *#option* clause. *#insert* command allows us to modify ART templates at designated *#break* points in arbitrary ways.

#while command iterates over ART template(s), with each iteration generating similar, but with minuscule differences, program structures. A #while loop can be controlled by using one or more multi-valued ART variables. *#select* command nested in the *#while* loop allows us to derive specific classes in each of the seven similarity groups discussed above. This is a key element of the ART strategy that allows us to unify similarity patterns at the level of mixed-strategy representation (i.e., in an ART-template solution), and still have repetitions in a program that ART Processor derives from an ART-template solution.

Now, we comment on the above mechanisms in more details, referring to Fig. 5 that shows a partial Java/ART-template solution for the Buffer classes.

ART commands and references to ART variables are shown in bold. *#set* command assigns values to a variable. For example, *#set* command in line 2 of the SPC assigns values listed on the right-hand-side to a variable named elmtType. References to ART variables (highlighted in bold) can be embedded in the code. For example, a reference to ART variable elmtType is written by an ART expression ?@elmtType? (line 4 in [T]Buffer.art), which is replaced by the variable's value during processing. Having set values for the ART variables, the SPC initiates generation of classes in each of the seven groups of similar classes via suitable *#adapt* commands (lines 5–8). The *#while* loop in [T]Buffer.spc (lines 2–19) is controlled by a multi-value variable, namely elmtType. The i'th iteration of the loop uses i'th value of the variable. In each iteration, the *#select* command uses the current value of elmtType to choose a proper *#option* for processing. *#select* command nested in the *#while* loop (lines 3–18) allows us to specify control for the seven buffer classes in the [T]Buffer similarity group.

#output command in [T]Buffer.art (line 2) defines the name of a file where the ART Processor will emit the code for a given buffer class. ART template [T]Buffer.art further defines common elements found in all seven buffer classes in the group. Five of those buffer classes, namely DoubleBuffer, IntBuffer, FloatBuffer, ShortBuffer, and LongBuffer differ only in type parameters (as in the sample method wrap() shown in ART template commonMethods.art). These differences are unified by ART variables, and no further customizations are required to generate these five buffer classes from [T]Buffer.art. These five buffer classes are catered for in *#otherwise* clause under

#select (lines 15–17 in [T]Buffer.spc). However, buffer classes ByteBuffer.java and CharBuffer.java have some extra methods and/or attribute declarations. In addition, method toString() has different implementation in CharBuffer.java than in the remaining six classes. Customizations specific to classes ByteBuffer.java and CharBuffer.java are listed in the *#adapt* commands, under *#option* Byte and *#option* Char, respectively.

Further, Fig. 6 shows generic method slice() from Direct[T]Buffer[S|U] classes (a specific instance of method slice() is shown in Fig. 3). Values of variables set in SPC reach all their references in adapted ART templates. The value of variable byteOrder is set to an empty string, "S" or "U", in a respective *#set* command placed in one of the ART templates that *#adapt*s ART template slice.art (not shown in our figures).

slice.art

```
1  public ?@elmtType?Buffer slice() {
2     int pos = this.position();
3     int lim = this.limit();
4     assert (pos <= lim);
5     int rem = (pos <= lim \? lim - pos : 0);
6     int off = (pos << ?@elmtSize?);
7     return new Direct?@elmtType?Buffer?@ByteOrder?(this, -1, 0, rem, rem, off);
8  }
```

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

The above described ART-template solution is meant to illustrate our points about relationship between genericity and SoC. The original Java Buffer Library consists of 74 buffer classes with 16,299 lines of code (LOC). However, the corresponding Java/ART-template solution consists of just 3,771 LOC with 74 ART template files and three buffer classes which are used intact in the constructed Java/ART-template solution. A brief discussion on engineering benefits of the ART-template solution is provided in Section 6. Detailed evaluation of engineering qualities of ART-template solution is not in the scope of this paper. We refer the reader to the papers and website for the discussion of trade-offs involved in applying the ART.

5. Another Example of a "Difficult" Concern

The Java Buffer library example discussed in the previous section is a very special type of a program. In this section, we show how a problem observed in the Java Buffer library occurs 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 was implemented in C#, with 18,823 LOC that contained in 117 classes covering GUI, service, and database layers. DEMS involved 13 domain entities (such as *User* or *Task*) with up to 10 operations per entity (such as *Create*, *Delete*, *Update*, or *Copy*). Each combination of entity-operation is implemented by a pattern of collaborating components, two of which are shown in Fig. 7. Each such pattern involves classes from four system layers as shown in Fig. 7. Each box in Fig. 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 Fig. 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. However, SoC along the "domain entity" or "operation" dimension is difficult. This is because of much differences in the 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 Java Buffer library, namely:

1. Much variation in the impact of different domain entities on operations, and
2. Subtle, ad hoc interactions between concerns.

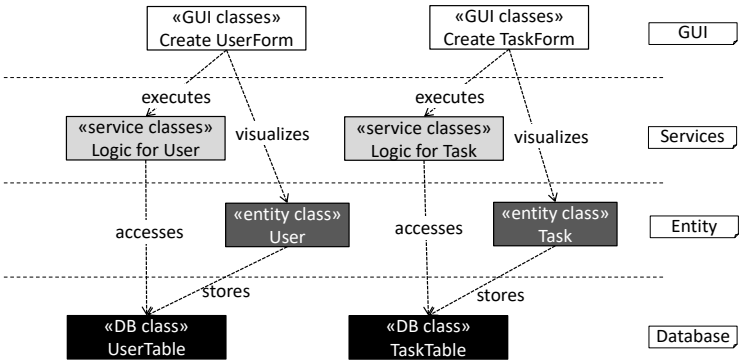

Fig. 7. A recurring pattern of components in Domain Entity Management System

Now we look at the problem from the *genericity* perspective. We found that there is much similarity among patterns of components implementing the same operation for different entities. Also, there are differences among the patterns caused by different meaning of domain entities: For example, operation *Create* for a *Task* requirs 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 the ART. In the next subsection, we show the ART-enabled generic solution for DEMS.

5.1. ART-enabled Generic Solution for Domain Entity Management System

Fig. 8 shows an outline of DEMS as a generic C#/ART-template solution. The top-most template SPC (Level 1) contains global controls and parameter settings that specify the overall process of constructing DEMS from the ART templates defined below it. ART templates at Level 2 (such as Create.spc, Update.spc) specify control for different operations applied to different domain entities. ART template DEMS_template.art at Level 3 defines the structure of the DEMS architecture, that is the organization of component patterns implementing various operations plus any other functions supported by DEMS, not discussed in example of Fig. 7.

At Level 4, each group of operations such as *CreateUser*, *CreateTask*, … has been represented by one generic operation parameterized by the respective domain entity (i.e., Create[E].art). Similarities among different operations for the same entity (e.g., *CreateUser*, *UpdateUser*, …) are unified at Level 5. ART templates at Level 5 represent generic classes, building blocks for DEMS operations, as indicated by ART templates referenced from more than one operation (e.g., generic classes labeled with CU are reused in construction of *Create* and *Update* for various entities).

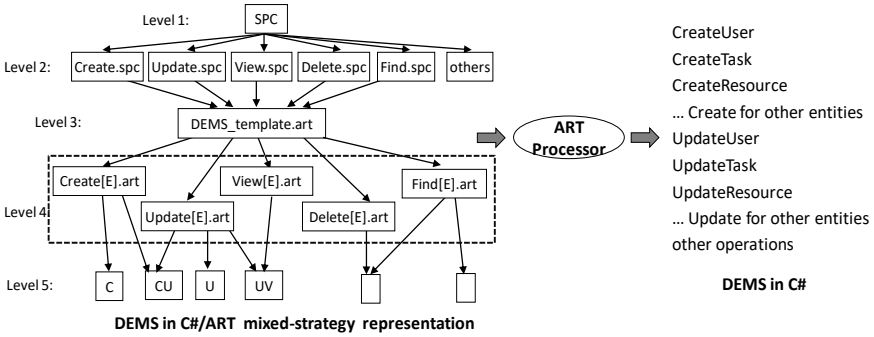

Fig. 8. A C#/ART mixed-strategy solution for Domain Entity Management System

Fig. 9 provides some of the details of the C#/ART-template solution for DEMS shown on the left-hand-side of Fig. 8. ART variables 'operation' (set in line 2 in SPC) and 'entity' (set in line 2 in DEMS_template.art) are generic names for the DEMS operations and domain entities, respectively. As mentioned previously in Section 4.3, these ART variables also play an important role in controlling traces of customization changes related to a single source—*representing a concern or a specific feature within a concern*—that span across multiple ART templates. For example, ART variable 'operation' marks customizations related to "operation" concern. On the other side, customizations related to "domain entity" concerns are marked using variable 'entity'.

The *#while* loop in SPC (lines 3–14) is controlled by an multi-valued ART variable, 'operation'. In each iteration of the *#while* loop, the SPC allows specifying control for initiating patterns of components implementing *Create*, *Update*, *View*, and others operations. Unique specifications related to specific operations are listed under a suitable option of the *#select* command (lines 4–13) nested inside the *#while* command. Similar to the SPC, ART template DEMS_template.art provides a similar solution that specifies unique customizations required for specific domain entities.

ART templates Create.spc and Update.spc specify control for *Create* and *Update* operations, respectively, applied to different domain entities via adapt mechanism (line 2 in Create.spc and Update.spc). Code specific to *Create* operation is defined in ART template C.art. Similarly ART template UV.art specifies code reused in *Update* and *View* operations for different entities. For example, with *#insert* command in Create.spc (lines 5–7), we insert code specific to *Create* operation at designated variation pointed using *#break* command as in Create[E].art (line 2). This example shows how we deal with ad hoc variations related to a specific operation in DEMS without actually affecting the other operations that should not be affected by these variations. Similar mechanisms will be followed for the other operations in DEMS.

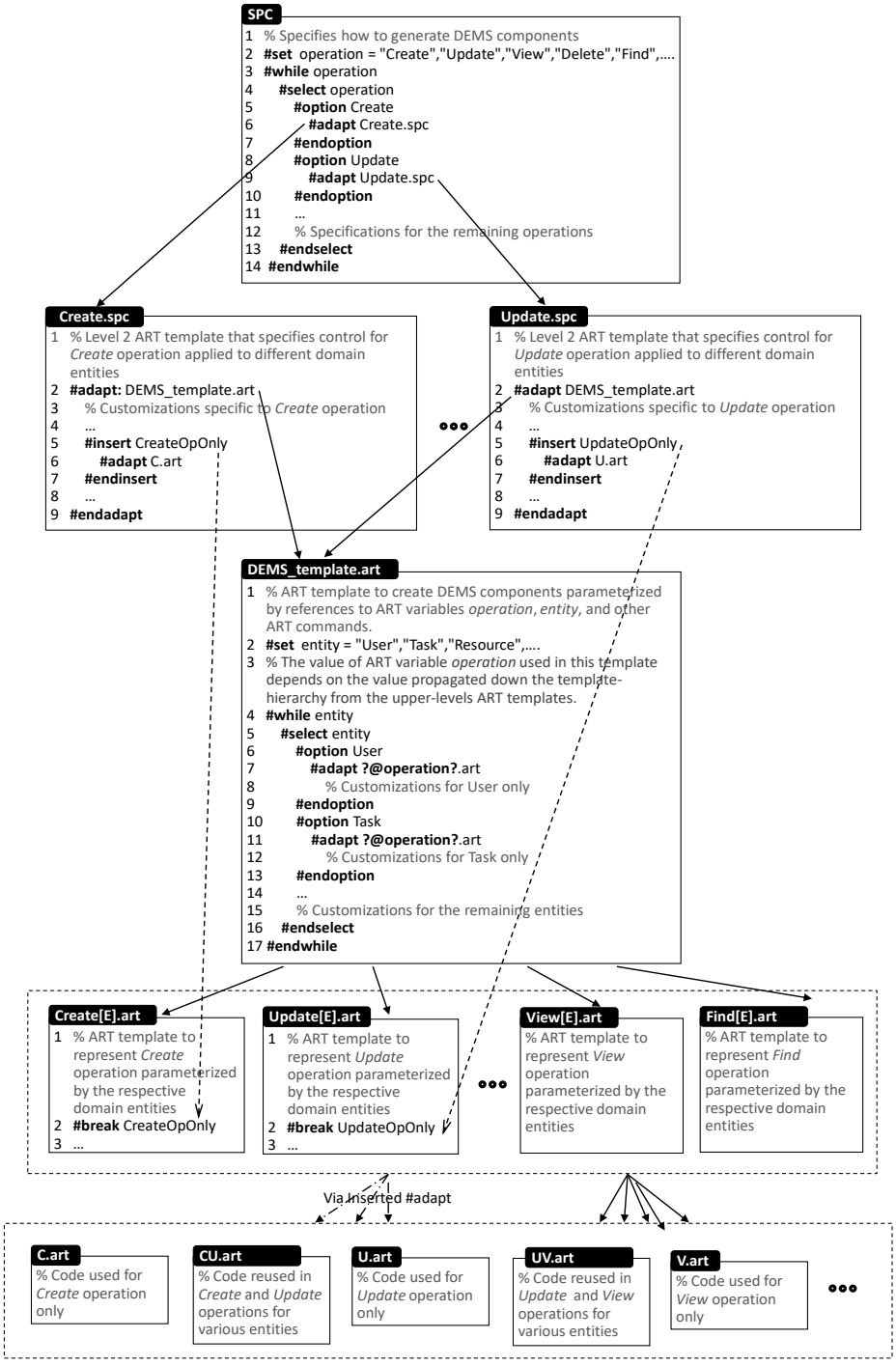

Fig. 9. Code snippet of C#/ART-template solution for Domain Entity Management System

The C#/ART-template solution consists of complete C# code required to generate all the DEMS operations, and also information helpful in maintenance/reuse, such as the record of similarities and differences among operations for different domain entities. Statistically, the C#/ART-template solution consists of 5,921 LOC (approximate 68% less compared to original C# code) and is conceptually simpler than its C# counterpart.

In the next section, we provide summary and analysis of observations based on the discussion followed in Sections 2–5.

6. Summary and Analysis of Observations

We discussed some examples that highlight some difficulties in archiving clean SoC. *We showed, how generic design, by looking at the problem from a different angle, achieves a weaker form of separating concerns*. In this subsection, we summarize observations, trying to distil observations that carry some more general message from those that are specific to our examples or to the use of the ART.

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 the ART try to overcome the problem at an extra meta-level plane.

SoC at the design and implementation levels 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, XVCL, or the ART.

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. We observe that our discussion of "difficult" concerns, becomes necessarily dependent on many factors such as the technology used, the design philosophy followed, language instruments, and other major mechanisms used in the design of a particular program.

In the 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 Java Buffer library, we bet on "buffer element type" concern. ART variables set in the top-most SPC are all related to this concern and they navigate the process of adapting ART templates below. These variables and ART constructs controlled by them enhance the visibility of the "buffer element type" concern. We can see the impact of buffer element types on the ART templates below the SPC and other ART templates adapted from there. 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.

Further, *ART-enabled generic representation improves the visibility of other concerns, due to groupings of similar classes into groups, but here the SoC is less systematic.*

We believe the reason why genericity can penetrate software areas deeper than SoC is that genericity, based on the notion of unifying similar program structures, is less formal and rigorous than SoC: *Arbitrary software structures that exhibit enough similarity can be unified with generic program representations, using unconventional techniques such as the ART.* This makes genericity technically easier to achieve than SoC.

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 [35][36][37][38]). 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.

It is important to mention that unification of similarity patterns occurs only at the level of an ART-template representation (left-hand-side of Fig. 4 and Fig. 8). An executable program derived from the ART 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 [39], see also [40]), and/or taking into account possibly yet other design goals a program must meet [25].

The ART-enabled mixed-strategy representation offers semi-automated solution: The process of generation of native code from the templates is automated. Whereas, the actual construction of ART templates is a manual process that can be performed systematically using the ART commands. It is because, just like program design, ART template design requires expert judgment that cannot be easily replaced by automated decision making process. There is a choice of ART mechanisms such as parameterization, selection, or insertions of program structures (discussed in Section

4.1) at designated points in templates that can be used to tackle various redundancy situations. These ART template design choices have various desirable and undesirable outcomes just like a decision to use a certain design pattern during conventional program design may have positive and negative implications. However, the ART offers a very simple and systematic mechanism that consists of only few constructs (such as *#adapt*, *#while*, *#selcet*, or insert-break mechanism). In addition, the process of generation of code from the ART templates has been automated using the ART Processor.

The proposed ART-enabled mixed-strategy approach incurs cost of building ART templates and have additional layer of generating source files. The examples of ART templates shown in the paper may also look complex. We agree that at the first glance they do. But, the fact is that the ART is governed by only five important constructs (i.e., #adapt, #output, #insert-break mechanism, #while, and #select) that are neatly integrated to form a method that can be learned easily. Further, the ART is an enhanced and improved version of the XVCL. The ART and XVCL has already been applied in many case studies including industrial projects ([1][4][22][23][25][26][27][28]). In these industrial projects, productivity impact of applying the ART and XVCL was measured and evaluated. There are sufficient evidences from these projects that the overhead incurred by the application of the ART and XVCL is smaller than benefits incurred by these techniques.

A discussion of and comparison of the ART-enabled mixed-strategy solution with related techniques is elaborated in Section 7.

7. Related Work

Modular decomposition with information hiding [10], macros, generics in Ada or Java [7], templates in C++, other forms of parameterization such as higher order functions [8], inheritance with dynamic binding, and design patterns [9] are some of the conventional design techniques to achieve genericity. The ART-enabled mixed-strategy solution uses templates and code generation to achieve genericity. ART templates can represent any groups of similarity patterns (e.g., files, directories, or patterns of collaborating components) with arbitrary differences among them (as opposed to only type-parametric differences in C++ templates or Java generics). From the ART template solution of a similarity pattern, the ART Processor generates code for all the instances based on the specifications of deltas, i.e., the differences between the template and each of the instances in the similarity pattern.

AOP [14] and MDSOC [13] support genericity by separating cross-cutting concerns. In AOP, various computational aspects are programmed separately and weaved at specified join points into the base program. AOP can separate a range of programming aspects, such as persistence, synchronization, or authentication/authorization. Separated aspects can be easily modified, added, or deleted to/from the program modules. However, a study revealed some difficulties in using AspectJ [41] (an AOP extension for the Java programming language) to deal with features that have a chaotic impact on the base code [20]. While AOP deals with big chunks of functionalities (i.e., aspects) reasonably, it lacks a mechanism to handle variations at the lower-levels of granularity. The ART-enabled mixed-strategy generic solution, on the other hand, can handle variations at any level of the granularity. Also, there is a fixed set of join points defined

in AOP. Compared to this, breakpoints in the ART can be defined anywhere in the program whenever needed. Using breakpoints, we can explicitly mark the variation points where specific code to a variant can be easily inserted. However, there is also a disadvantage of the ART as compared to AOP. The ART requires additional cost in creating templates for the code before adaptation. Whereas in case of AOP, there is no need to modify the existing program before weaving begins. ART 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*s modify ART templates in arbitrary ways, at any explicitly designated *#break* points.

MDSOC permits separations of overlapping concerns along multiple dimensions of compositions and decompositions. MDSOC introduces hyperslices that encapsulate specific concerns, and can be composed in various configurations to form custom programs. But, unlike the ART, hyperslices are written in the underlying programming language, and can be composed by merging or overriding program units by name, and in many other ways. On the other hand, the ART-enabed mixed-strategy solution is independent of the underlying programming language. It does not rely on any type of the abstract specifications that are associated with the programming language of the native code. Actually, the proposed solution offers uniform mechanism to handle variability. It means that it can be used to handle variability in a variety of interrelated SPL assets such as architecture, code components, domain models, documentations, test cases, etc.

In AHEAD [15] (based on the earlier Batory's work on GenVoca), genericity is supported by feature modularization and composition. Feature modularization helps in understanding and maintaining the feature code. Feature composition extends the base program with the required features. AHEAD provides a powerful solution for feature management in many situations, but may not be geared for features that have complex mappings to the code [42]. Therefore, Kästner et al. [42] relaxed the requirement for feature modularization, and revisited the idea of keeping feature-related code together with the base code. They proposed a tool CIDE that provides a visual means for understanding and manipulating the features. CIDE represents a base program as an abstract syntax tree, which makes it language-dependent. Compared with these techniques, the ART is strictly language-independent. The ART's adaptations are defined in an operational way, and take place at designated variation points marked with the ART commands only.

Recent advancements in modeling and generation techniques led to Model-Driven Development (MDD) [43][44]. In MDD, domain specific abstractions can be expressed using multiple inter-related models. It considers 'model' as a central source of information and the rest of the system is generated from the model using transformation and template rules specific to a particular platform [17]. Although MDD allows combining multiple models together, yet it lacks with a generic, multi-model integration mechanism [45]. This restricts MDD to effectively deal with crosscutting concerns that can arise at model level [46]. Further, generally MDD allows transformations to be performed at compile time [44]. Cerny and Donahoo [17][47] proposed a solution that decompose and untangle various elements—they called particles— involved in the user interface assembly. Some of these particles may be platform-independent while others are not. They provided a solution that allows runtime composition of such particles that matches user demands, context, and target platform.

Many techniques described under the umbrella of generative techniques [48], notably meta-programming with C++ templates, achieve genericity as well as certain forms of SoC. Application Generators [49] build domain-specific solutions by formalizing the domain knowledge. A Generator encodes domain-specific abstractions in a generic, parameterized form. Based on requirements specification in a Domain-Specific Language (DSL), a generator instantiates the generic form to produce a custom program. Such DSL-based techniques address the SoC, but can introduce information replication [46]. In the situations where DSL specifications are compact and are in the scope of given domain, generators can have better yield than the ART-enabled mixed-strategy solution.

The ART-enabled mixed-strategy solution performs best in domains where frequent changes occur at both large and small granularity levels. The problem with model-based and DSL-based solution is the likelihood of *disconnecting models* during evolution, especially when multiple independent evolving versions of a program originate from a model or generator. This occurs when model-based and DSL-based solution cannot cater for unexpected evolutionary changes, and developers modify the generated code. Compared to this, the ART-enabled mixed-strategy solution allows programmers to modify any details of the program solution and the required code changes are always proportional to the changes in the problem domain. The ART is an application domain independent technique. However, considering many large-granular similarity patterns that represent domain-specific abstractions [35], the ART-enabled mixed-strategy solution enables realization of such abstraction in the design/implementation solution space. Thus, the ART can be considered as a domain-independent technique that captures some of the domain-specific abstractions.

Domain analysis [50] 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 [1][3], architectural styles [51], and patterns [3] help developers avoid repeatedly designing the same solution by providing component plug-in plug-out capability. Code inspection and transformation based techniques such as MetaWidget [18] and AspectFaces [52] provide possible solution to avoid similarity/information duplication in the user interface layer of software applications. Component platforms such as JEE™ or .NET™, provide also an infrastructure for reuse of pre-defined common services.

Preprocessors can also be used to separate code for variant features [53]. The ART adds a non-redundancy layer on top of separation of concerns achieved by preprocessors, without changing the way preprocessors are configured in native code. Non-redundant ART-template views of programs lessen variability management, as one variation point in an ART template represents 'n' variations points in instances of that template, where 'n' is the number of instances of the template in a program. The capability to deal with redundancies is what distinguishes the ART from the techniques proposed by others.

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 [54][55], refactoring [56] and clone detection [54][35][38]. In an empirical study of cloning practices Kim et al. [40] observed that "Limitations of particular programming languages produce unavoidable duplicates in a code base".

8. Conclusions

In this paper, we first made observations, in the forms of hypotheses rather than claims, about the general interplay between the principle of SoC and genericity. Next, we showed that generic design can enhance the visibility of inseparable concerns, offering a weaker, but still useful form of SoC.

With the help of experimentation and industrial case study, we proved that there is an overlapping area where the goals of SoC and genericity, as well as means to achieve them, are the same. For example, type parameterization or modularization with information hiding separates a concern and achieves genericity at the same time. In this case, program structures can be viewed as being "parameterized" by concerns. By composing concerns, we instantiate program structures in variant forms. In that perspective, program structures gain genericity and reusability due to SoC. In case of separable concerns, program structures parameterized by concerns may still exhibit similarity due to yet other reasons not related to given concerns. It may facilitate a room for generic design to further improve engineering qualities of a program solution. 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.

Further, we also considered situations where attempts to cleanly separate concerns fail. We showed that generic design can enhance the visibility of inseparable concerns by offering a weaker but still useful form of SoC. Genericity is based on the notion of unifying similar program structures and is less formal and rigorous than SoC. Due to this reason, we believe that the genericity can penetrate software areas deeper than SoC. To achieve genericity, we also presented the use of ART templates. ART-enabled generic program representations have been shown to be useful for unifying arbitrary software structures exhibiting enough similarity. It further makes genericity technically easier to achieve than SoC.

In future, we plan to extend our study to cover wide range of concern types. 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 [14]. Another interesting enhancement can be to develop a concern ontology. It would help in expressing research results on SoC and genericity in more precise terms.

References

1. Jarzabek, S.: Effective Software Maintenance and Evolution: A Reuse-based Approach. CRC Press, USA. (2007)
2. Bosch, J.: Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach. Addison-Welsey, USA. (2000)
3. Clements, P., Northrop, L: Software Product Lines: Practices and Patterns. Addison-Wesley, USA. (2002)
4. Jarzabek, S., Pettersson, U., and Zhang, H.: University-industry Collaboration Journey towards Product Lines. In Proceedings of 12th International Conference on Software Reuse, ICSR'2011, 223–237. (2011)
5. Musser, D., Saini, A.: STL Tutorial and Reference Guide: C++ Programming with Standard Template Library. Addison-Wesley, USA. (1996)

6. Goguen, J. A.: Parameterized Programming. IEEE Transactions on Software Engineering, Vol. SE-10(5), 528–543. (1984)
7. Garcia, R., Järvi, J., Lumsdaine, A., Siek, J., Willcock, J.: A Comparative Study of Language Support for Generic Programming. In Proceedings of 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'2003, California, USA, 115–134. (2003)
8. Thompson, S.: Higher Order + Polymorphic = Reusable. Unpublished manuscript available from the Computing Laboratory. University of Kent, UK. Available: http://www.cs.ukc.ac.uk/pubs/1997. (2015)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, USA. (1995)
10. Parnas, D.: On the Criteria To Be Used in Decomposing Software into Modules. Communications of the ACM, Vol. 15, No. 12, 1053–1058. (1972)
11. Jarzabek, S., Bassett, P., Zhang, H., Zhang, W.: XVCL: XML-based Variant Configuration Language, In Proceedings of 25th International Conference on Software Engineering, ICSE'2003, 810–811. (2003)
12. Dijkstra, E. W.: On the Role of Scientific Thought. Selected Writings on Computing: A Personal Perspective. Springer-Verlag, New York, 60–66. (1982)
13. Tarr, P. , Ossher, H., Harrison, W., Sutton, S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In Proceedings of International Conference on Software Engineering, ICSE'99, Los Angeles, 107–119. (1999)
14. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwin, J.: Aspect-Oriented Programming. In Proceedings on European Conference on Object-Oriented Programming, Finland, 220–242. (1997)
15. Batory, D., Sarvela, J. N., Rauschmayer, A.: Scaling Step-Wise Refinement. In Proceedings of International Conference on Software Engineering, ICSE'03, Portland, Oregon, USA, 187–197. (2003)
16. Cerny, T., Macik, M., Donahoo, M. J., Janousek, J.: On Distributed Concern Delivery in User Interface Design. Computer Science and Information Systems, Vol. 12, No. 2, 655–681. (2015)
17. Cerny, T., Donahoo, M. J.: On Separation of Platform-Independent Particles in User Interfaces. Cluster Computing, Vol. 18, No. 3, 1215–1228. (2015)
18. Kennard, R., Edmonds, E., Leaney, J.: Separation Anxiety: Stresses of Developing a Modern Day Separable User Interface. In Proceedings of the 2nd Conference on Human System Interactions, 225–232. (2009)
19. Java Interceptors. Available: https://java.net/downloads/interceptors-spec/interceptor-1-2-mrel.pdf. (2016)
20. Kästner, C., Apel, S., Batory, D.,: A Case Study Implementing Features using AspectJ. In Proceedings of 11th International Software Product Line Conference, SPLC'2007, 223–232. (2007)
21. Adaptive Reuse Technique (ART). Available: https://sourceforge.net/projects/vclang/. (2016)
22. Kumar, K., Jarzabek, S., Daniel, D: ART: A Meta-programming Language for Configuring Variants in Software, in Poster track of 12th Asian Symposium on Programming Languages and Systems, APLAS'2014, Singapore, 2p. (2014)
23. Pettersson, U., Jarzabek, S.: Industrial Experience with Building a Web Portal Product Line using a Lightweight, Reactive Approach. In Proceedings of 10th European Software Engineering Conference held jointly with 13th International Symposium on Foundations of Software Engineering, ESEC/FSE'2005, Portugal, 326–335. (2005)
24. Basit, H. A., Rajapakse, D. C., Jarzabek, S.: Beyond Templates: A Study of Clones in the STL and Some General Implications, In Proceedings of the 27th International Conference on Software Engineering, ICSE'2005, 451–459. (2005)

25. Jarzabek, S., Li, S.: Unifying Clones with a Generative Programming Technique: A Case Study. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 18, No. 4, 267–292. (2006)
26. Jarzabek, S., Shubiao, L.: Eliminating Redundancies with a "Composition with Adaptation" Meta-programming Technique. In Proceedings of 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE'2003, 237–246. (2003)
27. Asaithambi, S. P. R., Jarzabek, S.: Pragmatic Approach to Test Case Reuse-A Case Study in Android OS BiDiTests Library. In Proceedings of International Conference on Software Reuse, ICSR'2015, 122–138. (2015)
28. Lieh, O. E., Jarzabek, S.: An Adaptability-Driven Model and Tool for Analysis of Service Profitability. In Proceedings of 28th International Conference on Advanced Information Systems Engineering, CAiSE'2016, 393–408. (2016)
29. Jarzabek, S., Kumar, K.: Weak Separation of Tightly Coupled Concerns with Generic Program Representations. In Proceedings of PTI 17th KKIO Software Engineering Conference, KKIO'2015, Miedzyzdroje, Poland, 119–136. (2015)
30. Mesbah, A., Deursen, A. V.: Crosscutting Concerns in JEE Applications. In Proceedings of 7th IEEE International Symposium on Web Site Evolution, WSE'05, Budapest, Hungary, 14–21. (2005)
31. Private communication with Ali Mesbah and Arie van Deursen, authors of 30
32. Filho, F., Cacho, N., Figueiredo, E., Maranhao, R., Garcia, A., Rubira, C.: Exceptions and Aspects: The Devil is in the Details. In Proceedings of ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE'06, USA, 152–162. (2006)
33. Exception Management Architecture Guide ver 1.0. Microsoft Patterns & Practices, 2003. Available: http://www.usol.com/~joe/Exception%20Management%20-%20EntLib.pdf. (2016)
34. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, SEI, CMU, Pittsburgh, 1990. Available: http://www.sei.cmu.edu/reports/90tr021.pdf (2016)
35. Basit, H. A., Jarzabek, S.: A Data Mining Approach for Detecting Higher-Level Clones in Software. IEEE Transactions on Software Engineering, Vol. 35, No. 4, 497–514. (2009)
36. Sajnani, H., Lopes, C.: A Parallel and Efficient Approach to Large Scale Clone Detection. In Proceedings of 7th International Workshop on Software Clones, IWSC'2013, 46–52. (2013)
37. Kumar, K., Jarzabek, S.: Detecting Design Similarity Patterns using Program Execution Traces. In Proceedings ACM SIGPLAN conference on Systems, Programming, and Applications: Software for Humanity, SPLASH'2014, Portland, Oregon, USA, 55–56. (2014)
38. Kamiya, T., Kusumoto, S., Inoue, K.: CCFinder: A Multi-linguistic Token-based Code Clone Detection System for Large Scale Source Code, IEEE Transactions on Software Engineering, Vol. 28, No. 7, 654–670. (2002)
39. Yang, J., Jarzabek, S.: Applying a Generative Technique for Enhanced Reuse on JEE Platform. In Proceedings 4th International Conference on Generative Programming and Component Engineering, GPCE'05, Tallinn, 237–255. (2005)
40. Kim, M., Sazawai, V., Notkin, D., Murphy, G.: An Ethnographic Study of Code Clone Genealogies. In Proceedings of European Software Engineering Conference and International Symposium on the Foundations of Software Engineering, Portugal, 187–196. (2005)
41. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP'01, Budapest, Hungary, 327–353. (2001)
42. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In Proceedings of 30th International Conference on Software Engineering, ICSE'2008, Leipzig, Germany, 311–320. (2008)
43. Selic, B.: The Pragmatics of Model-driven Development. IEEE Software, Vol. 20, No. 5, 19–25. (2003)

44. Cerny, T., Song, E.: Model-driven Rich Form Generation. Inf. Int. Interdiscip., Vol. 15, No. 7, 2695–2714. (2012)
45. Sottet, J. S., Calvary, G., Coutaz, J., Favre, J. M..: A Model-driven Engineering Approach for the Usability of Plastic User Interfaces. In Engineering Interactive Systems, Springer Berlin Heidelberg, 140–157. (2008)
46. Macik, M., Cerny, T., Slavik, P.: Context-sensitive, Cross-Platform User Interface Generation. Journal on Multimodal User Interfaces, Vol. 8, No. 2, 217–229. (2014)
47. Cerny, T., Donahoo, M.J.: Separating Out Platform-Independent Particles of User Interfaces. Information Science and Applications, Springer, Berlin, 941–948. (2015)
48. Czarnecki, K, Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, USA. (2000)
49. Smaragdakis, Y. and Batory, D.: Application generators. In Software Engineering volume of the Encyclopedia of Electrical and Electronics Engineering, J. Webster (ed.), John Wiley and Sons. (2000)
50. Prieto-Diaz, R.: Domain Analysis for Reusability. In Proceedings of Annual International Computers, Software & Applications Conference, COMPSAC'87, Tokyo, Japan, 23–29. (1987)
51. Shaw, M., Garlan, D.: Software Architecture: Perspectives on Emerging Discipline. Prentice Hall, USA. (1996)
52. AspectFaces. Available: http://www.aspectfaces.com/. (2016)
53. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In Proceedings of International Conference on Software Engineering, ICSE'2010, Cape Town, South Africa, 105–114. (2010)
54. Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone Detection using Abstract Syntax Trees. In Proceedings of International Conference on Software Maintenance, ICSM'98, 368–377. (1998)
55. Asaithambi, S., Jarzabek, S.: Towards Test Case Reuse: A Study of Redundancies in Android Platform Test Libraries. In Proceedings of International Conference on Software Reuse, ICSR'2013, 49–64. (2013)
56. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison-Wesley, USA. (1999)

Stanisław Jarząbek is a Prof. PB at Bialystok University of Technology. He received MSc and PhD from Warsaw University. He works on software engineering (software reuse and maintenance), and mHealth – use of mobile technology to improve delivery of healthcare. He works with industries, and a reuse method developed in his lab have been applied in industry. He is an author of a book Effective Software Maintenance and Evolution: Reuse-based Approach, CRC Press, 2007, published over 100 papers, and gave tutorials at international forums.

Kuldeep Kumar is an Assistant Professor in the Department of Computer Science and Information Systems (CSIS) at Birla Institute of Technology and Science, Pilani (BITS-Pilani), Pilani Campus, Rajasthan, India. He received his PhD in Computer Science from School of Computing, National University of Singapore (NUS SoC), Singapore in 2016. He has several publications in reputed international journals/conferences. His current areas of interest include software engineering, machine learning, and information retrieval.

Received: January 29, 2016; Accepted: July 6, 2016.