Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems for, Advanced Separation of Concerns*

Peri Tarr[1], Maja D'Hondt[2], Lodewijk Bergmans[3], and Cristina Videira Lopes[4]

[1]IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown, NY 10598 USA
`tarr@watson.ibm.com`
[2]Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium
`mjdhondt@vub.ac.be`
[3]University of Twente, Dept. of Computer Science, P.O. Box 217, 7500 AE, Enschede, The Netherlands
`lbergmans@acm.org`
[4]Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA
`lopes@parc.xerox.com`

Workshop web site: `http://trese.cs.utwente.nl/Workshops/adc2000`

Abstract. This document describes the results of the two-day Workshop on Aspects and Dimensions of Concern at ECOOP 2000. The workshop produced the beginnings of a set of goals, requirements, and issues to be addressed by approaches to advanced separation of concerns. These goals and issues are encapsulated, in part, in a set of challenge problems that are intended to be solved by providers of advanced separation of concerns technologies. The challenge problems and requirements begin to define the boundaries of this problem domain and solution space, and they will help provide a basis for evaluating and comparing different solutions and solution approaches.

1 Introduction

The First Workshop on Aspects and Dimensions of Concern was held at ECOOP 2000, following several previous workshops on Aspect-Oriented Programming, Subject-Oriented Programming, and Multi-Dimensional Separation of Concerns. Whereas the goals of previous workshops had been to introduce and begin to explore the new subdomain of advanced separation of concerns, this workshop's purpose was instead to begin to define a set of concrete goals for, requirements on, and issues to be addressed by, approaches to advanced separation of concerns. The explicit representation of these goals, requirements, and issues represents the start of the transition of advanced separation of concerns from an infant subdiscipline to a coherent, mature research area. When completed, they will help to define the boundaries of the problem domain and solution space, and they will provide a basis for evaluation of, and comparison among, different solutions and solution approaches.

* This workshop was funded in part by Xerox PARC and IBM Research.

To accomplish the workshop goals, we brought together both practitioners and researchers who have experienced problems related to inadequate separation of concerns, and practitioners and researchers who have provided solution technologies or approaches that address these problems. It was our intention to establish a dialogue between these two groups, to reveal the needs of the former group and the capabilities of the latter, thus facilitating the identification of goals, requirements and issues for advanced separation of concerns. To achieve this, prospective participants were required to submit position papers describing either some problems they had encountered, or solutions they had developed, in the field of advanced separation of concerns. This resulted in more than 40 experienced researchers and developers participating in the workshop, and their position papers can be found on the workshop web site, `http://trese.cs.utwente.nl/Workshops/adc2000`.

As a highly interactive setting seemed most suitable, we allocated the larger amount of the two-day workshop to group work, only occasionally alternating it with invited talks from some of the experts in the field and authors who provided highly incisive, challenging problems. Seven heterogeneous groups were formed, joining problem providers with solution providers, and distributing co-authors or colleagues over as many groups as possible. This policy ensured that as wide a variety of submitted problems and solutions as possible were discussed in each group. Over the two days, the groups analyzed selected problems and evaluated some solutions by applying them to the problems. Each group presented both an interim report and their final results. The workshop concluded with a panel consisting of one representative per group, which proved to be a valuable source of general insights to the community.

The rest of this document describes some of the results of this two-day workshop. Depending on the viewpoint adopted by a group—either the problem perspective or the solution perspective—the group work resulted in the following kinds of results:

– *Challenge problems*: Challenge problems are software engineering scenarios that highlight one or more key problems and issues in advanced separation of concerns. The problem statement also describes any necessary requirements or constraints on solutions. Challenge problems, as their name suggests, are intended to serve as challenges to those researchers and developers in this subdomain who have produced, or who will produce, models of, or tools to support, software engineering using advanced separation of concerns. It is our hope that the producers of such models and technologies will propose solutions to the challenge problems using their models and technologies. As a result, we expect that some limitations or interesting features of existing models and technologies will be uncovered, prompting new research, and that the various solutions will be used as a basis for comparing and contrasting the relative strengths and weaknesses of different approaches.
– *Generalization of challenge problems*: Particular subsets of challenge problems suggest larger, abstract classes of problems. Such generalizations are particularly critical, as they help to identify and categorize key parts of the problem domain.
– *Requirements statements*: Some of the requirements on advanced separation of concerns approaches are already well defined; for example, an advanced separation of concerns approach must facilitate the encapsulation of multiple kinds of concerns, including those that cross-cut objects. Many requirements, however, are not well defined or widely accepted. Requirements statements highlight

requirements on approaches to advanced separation of concerns, and they are generally illustrated with examples. It is our intent that the producers of advanced separation of concerns models and technologies will indicate how their models and technologies address these requirements; if they do not, we hope that they will use the requirements to help them identify new areas for subsequent research.

Each result is classified accordingly.

The remainder of this document is organized as follows. Sections 2-REF8 describe, respectively, the results produced by a particular workshop subgroup. Finally, Section 9 presents some conclusions and future work.

Clearly, a single, two-day workshop is not sufficient time to produce a complete set of requirements, issues, or challenge problems, or even to produce a representative set. The results reported in this document are preliminary. Subsequent efforts may determine that some of the results are incomplete, inadequate, mutually inconsistent, or erroneous. It is our intention for these results to be discussed, revised and expanded by the advanced separation of concerns community. Beyond all else, this document is intended to provide food for thought.

1.1 Contributors

The results described in this paper are due to—and in many cases, adapted from reports written by—the workshop participants and organizers: Mehmet Aksit, Jean Paul Arcangeli, Federico Bergenti, Lodewijk Bergmans, Andrew Black, Johan Brichau, Isabel Brito, Laurent Bussard, Lee Carver, Constantinos Constantinides, Pascal Costanza, Krzysztof Czarnecki, Lutz Dominick, Maja D'Hondt, Wolfgang De Meuter, Kris de Volder, Erik Ernst, Robert Filman, Eric Hilsdale, Mathias Jung, Pertti Kellomäki, Mik Kersten, Gregor Kiczales, Thomas Kühne, Donal Lafferty, Cristina Videira Lopes, Mira Menzini, Tommi Mikkonen, Blay Mireille, Oscar Nierstrasz, Klaus Ostermann, J. Andrés Díaz Pace, Renaud Pawlak, Elke Pulvermüller, Bert Robben, Martin Robillard, Andreas Speck, Erlend Stav, Patrick Steyaert, Peri Tarr, Tom Tourwé, Eddy Truyen, Bart Vanhaute, John Zinky.

2 Safe Composition

Group members: Laurent Bussard, Lee Carver, Erik Ernst, Mathias Jung, Martin Robillard, and Andreas Speck

Issues: Semantically correct composition of aspects, concerns

Categories: Challenge problems, requirements statements

2.1 Problem Overview

Different aspects or concerns should only be composed when the result of composing them is semantically meaningful. Two classes of conflicts may occur among concerns that are to be composed which, if present during composition, may affect the correctness of the composed result:

- Semantic mismatch conflicts: Semantic mismatch conflicts are ones in which the composition of two or more concerns produces semantically incorrect programs. The mismatches occur when composed concerns contain behaviors that interact in subtle (often multiple) ways, and the interactions are not addressed correctly in the composed program.

  We further divide the class of semantic mismatch conflicts into *intentional* and *unintentional* conflicts. Intentional conflicts are ones that are introduced deliberately by a developer—i.e., the developer designs two or more concerns or aspects to conflict (e.g., to be mutually exclusive). Unintentional conflicts, on the other hand, are ones that occur inadvertently, often due to non-obvious semantic interactions among the concerns or aspects. While these kinds of semantic mismatch conflicts look the same, we distinguish them because addressing them requires different kinds of support from advanced separation of concerns mechanisms. Intentional mismatches require a means for allowing developers to assert that a set of concerns or aspects is intended to conflict. Unintentional mismatches, on the other hand, require compositor and/or run-time detection capabilities, and they must be resolved, either automatically or manually (e.g., with "glue" aspects or mediators), to produce a correct composed program that contains all of the (originally conflicting) aspects.
- Spurious conflicts: As their name suggests, spurious conflicts are ones that are identified as conflicts erroneously—i.e., the composed program would actually run correctly—due to the limitations of static or otherwise overly conservative type checking. Addressing spurious conflicts is difficult and may necessitate a choice between aspect reuse and static type checking, but it may be addressable with more powerful type analyses.

Generally, name resolution plays an important role in the emergence and handling of composition conflicts. The various composition technologies necessarily relax conventional name resolution rules. The inclusion of an aspect or concern in a composition relies on a tolerant form of name matching that precludes detection of over-defined or under-defined names. Existing tools, such as Hyper/J™ [22]and AspectJ [18], provide no mechanisms to prevent or require the inclusion of specific aspects or enhancements. This issue must be addressed by advanced separation of concerns approaches.

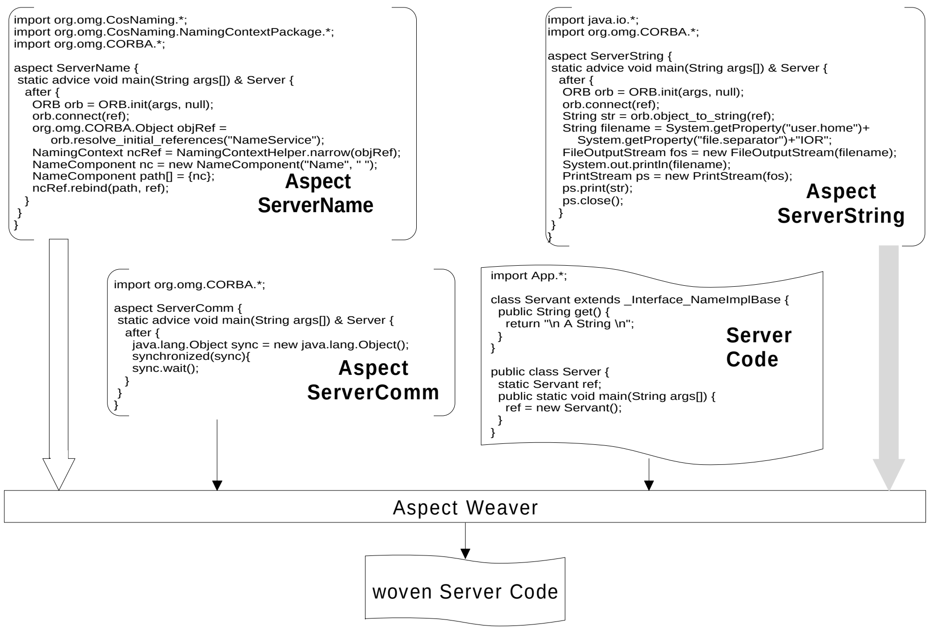
Advanced separation of concerns mechanisms must be able to identify these classes of conflicts among aspects that are to be composed and to address the conflicts, where possible.

Motivating papers: See [23] for intentional semantic mismatch conflicts; [8] for unintentional semantic mismatch conflicts; and [13] for spurious conflicts.

2.2 Challenge Problems and Requirements

Challenge Problem 1: Recognition, Representation and Enforcement of Intentional Semantic Mismatch Conflicts Among Concerns

To demonstrate intentional semantic mismatch conflicts, we present a challenge problem in which aspects are used to separate distribution-related code (implemented for CORBA) from problem-domain code. Separating these concerns reduces the complexity of the code and improves its reusability. A simplified version of the code is shown in Fig. 1; the full version can be found in [23].



```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

aspect ServerName {
 static advice void main(String args[]) & Server {
   after {
     ORB orb = ORB.init(args, null);
     orb.connect(ref);
     org.omg.CORBA.Object objRef =
          orb.resolve_initial_references("NameService");
     NamingContext ncRef = NamingContextHelper.narrow(objRef);
     NameComponent nc = new NameComponent("Name", " ");
     NameComponent path[] = {nc};
     ncRef.rebind(path, ref);
   }
 }
}
```

```
import java.io.*;
import org.omg.CORBA.*;

aspect ServerString {
 static advice void main(String args[]) & Server {
   after {
    ORB orb = ORB.init(args, null);
    orb.connect(ref);
    String str = orb.object_to_string(ref);
    String filename = System.getProperty("user.home")+
        System.getProperty("file.separator")+"IOR";
    FileOutputStream fos = new FileOutputStream(filename);
    System.out.println(filename);
    PrintStream ps = new PrintStream(fos);
    ps.print(str);
    ps.close();
   }
 }
}
```

```
import org.omg.CORBA.*;

aspect ServerComm {
 static advice void main(String args[]) & Server {
   after {
     java.lang.Object sync = new java.lang.Object();
     synchronized(sync){
     sync.wait();
   }
 }
}
```

```
import App.*;

class Servant extends _Interface_NameImplBase {
  public String get() {
    return "\n A String \n";
  }
}

public class Server {
  static Servant ref;
  public static void main(String args[]) {
    ref = new Servant();
  }
}
```

Fig. 1. CORBA server example: Use `ServerName` or `ServerString` but not both

As shown in Fig. 1, clients send requests to a server, which responds with a string. The implementation consists of four modules: `ServerCode` and the aspects `ServerName`, `ServerString`, and `ServerComm`. `ServerCode` implements the server, while the aspects encapsulate CORBA-related code.

To use a service, a client must obtain an initial reference to the server. This can be done either by contacting a name server, which provides the reference, or by accessing a disk file that contains the information, but the designer intended that *only one* of these methods would be chosen. The `ServerName` aspect encapsulates the solution based on contacting a name server, while the `ServerString` aspect realizes the file-based solution. Thus, only one of these two aspects should be composed with `ServerCode` to select an approach.

Clearly, it is not always possible for a compositor to identify circumstances like this one—where seemingly compatible aspects are, in fact, semantically incompatible. It must, therefore, be possible for a developer to specify this intentional semantic mismatch (mutual exclusion), and for a compositor to use this information to enforce the associated semantics.

Requirement 1 Advanced separation of concerns approaches must permit the static specification of *concern compatibility properties*, such as mutual exclusion. Compositor tools must understand and enforce such specifications. While automatic detection of all semantic conflicts is undecidable, the use of user-specified annotations is an example of a simple, tractable approach.

Sample solution: Hyper/J™ [22] could easily be extended to express mutual exclusion by putting the two aspects into the same dimension and specifying that only one coordinate can be selected from that dimension.

Challenge Problem 2: Recognition, Representation, and Enforcement of Unintentional Semantic Mismatch Conflicts Among Concerns

Unintentional semantic mismatch conflicts arise when a set of aspects or concerns interact with one another in a composed program in unanticipated ways to produce erroneous behaviors. When unintentional semantic mismatches occur, the desired course of action is to *detect* the conflict and then *resolve* it by adjusting the concerns or their composition. Unintentional semantic mismatch conflicts are common problems in software integration in general, and they are a large part of what makes integration a difficult problem. We believe that unintentional semantic mismatches will occur more commonly, and will become more problematic, as aspects and concerns are reused in different contexts, and as aspects or concerns written by different people (who may make different assumptions) are integrated.

To demonstrate unintentional semantic mismatch conflicts, we present a challenge problem involving a simplified banking application (the full version is presented in [8]). The core part of the application contains a bank `Account` class. Two aspects are also defined: `Event` and `Transaction`, add event handling and transactions on `Accounts`, respectively. The `Event` aspect transmits events on certain channels whenever a significant state transition occurs on an `Account`. Other objects may then listen on those channels to be notified about such transitions. The `Transaction` aspect supports standard transaction semantics via the operations `begin`, `commit`, and `rollback`.

When the `Event` and `Transaction` aspects are used together, they interact in such a way that a semantic conflict can arise if events are propagated out of uncommitted transactions (e.g., [26]). For example, consider what if a client performs a `deposit` operation on an `Account` object, and this operation is later terminated by a `rollback` (i.e., the enclosing transaction aborts). The `Event` aspect may already have broadcast the `deposit` event to unknown listeners. Thus, while the rollback will leave the `Account` object itself in a correct state, the event listeners will not be in a correct state—they have already seen, and possibly acted upon, the `deposit` event.

This unintentional semantic mismatch conflict can produce incorrect behavior in the composed program, but it can be corrected using a standard technique: delaying the actual broadcasting of events until the transaction commits (or suppressing the broadcast of events if the transaction aborts). Clearly, however, detecting and correcting semantic conflict problems would, in the general case, require intervention by a developer.

Requirement 2 Advanced separation of concerns approaches must provide a means for identifying and resolving unintentional semantic mismatch conflicts. In contrast with intentional semantic mismatches, which can be identified statically, identification and resolution of unintentional semantic mismatches may occur statically, at composition-time, and/or at run-time, and they may occur manually (by a developer), semi-automatically, and/or automatically.

Sample solutions: One possible solution to this challenge problem would be to use an approach like composition filters [1][3], which could intercept all event transmissions and store the events until the commit of a transaction. Even with composition filters, however, it would not be straightforward to make the aspects work together seamlessly. Another possible solution, using an extended Hyper/J, would be to require the `Event` aspect to provide certain transaction-related services—or, conversely, to write such services separately and compose them into the `Event` aspect—thus enabling the `Transaction` aspect to integrate the `Event` aspect into the transactional environment.

Detection of Spurious Conflicts Among Concerns: The previous two challenge problems addressed the detection and resolution of actual conflicts among composed aspects. The opposite situation may also arise—namely, where a composition mechanism reports a conflict, even when one does not exist.

To illustrate this situation, we consider composition mechanisms that are based on textual transformations from aspect languages to other languages (like Java™), and that then rely on compilers for the target language to compile the translated, composed code. In such cases, the generated code may appear to the target language compiler to be unsafe, even though it is type safe, because the static type analyzer in the target language is not sufficiently powerful to capture concepts expressed in the aspect languages.

An example of this is given in Section 4 of [13]; we describe a simpler case of the problem here. Consider the definition of a simple node/edge-based graph class:

```
class Node { void attach(Edge e); }
class Edge { boolean incident(Node); }
```

Thus, `Nodes` and `Edges` are used together as a unit. We may also define a general method that operates on graphs:

```
method m(Node n, Edge e) {... e.attach(n); ...}
```

Next, assume that we define two different enhancements of this kind of graph: one that adds colors to nodes to permit cycle detection, and another that adds printing. A given system may wish to use both colorable and printable graph instances, but it might not want to equip all graphs with both facilities, due to the extra performance and memory overhead.

Since the method `m` depends only on the basic graph abstraction, not on coloring or printing, it should be reusable on all kinds of graphs—printable, colorable, both, or neither. Thus, we would like to include multiple aspects in different instances of a graph, and we would like to access all kinds of graphs polymorphically (m is applicable to all kinds of graphs, with or without the aspects). In standard object-oriented type systems, like those in C++ and Java™, it is not possible to achieve all of these goals in a type-safe manner, because there are only two implementation approaches:

1. Define separate, unrelated `Node`, `Edge`, `ColoredNode`, `ColoredEdge`, `PrintableNode`, and `PrintableEdge` classes. In this case, the `Node` class' `attach` method explicitly takes `Edge` parameters, while the `attach` methods in `ColoredNode` and `PrintableNode` take `ColoredEdge` and `PrintableEdge` parameters, respectively. In this case, the solution is completely type-safe, but it has lost the reusability property of object-oriented systems: the `attach` methods, which were independent of the printable and colored aspects, are copied into all the node classes, as is the implementation of method `m`—it is impossible to write a single method that works on all kinds of graphs.
2. Define `ColoredNode` and `PrintableNode` as subclasses of `Node`, and `ColoredEdge` and `PrintableEdge` as subclasses of `Edge`. In this case, reusability is maintained, but at the cost of type safety: the attach methods of all Node classes expect an argument of type `Edge`, not `ColoredEdge` or `PrintableEdge`. Thus, it is possible to attach a `PrintableEdge` to a `ColoredNode`, etc.

In short, a spurious conflict arises if a composed program contains more than one aspect or concern composed with a "base" class, and if the program tries to visit different instances of the class—which may have different combinations of aspects or concerns attached to them—polymorphically. The problem is that the different families of classes will either be unrelated in the type hierarchy (the first case), or they will be indistinguishable (the second case). This means that the polymorphic code will be rejected by the compiler (a spurious conflict), or it will not be type safe.

A solution to this problem is to use more expressive type analysis, such as that described in [13]. Static analysis is performed at the highest possible level of abstraction—the "base code" and the aspects themselves—not on composed, tangled code (or other artifact, like design) in an underlying lower-level representation, as might be produced by a compositor tool.

Requirement 3 Implementations of advanced separation of concerns approaches should not lead to spurious errors, such as type errors in generated code when there are none in reality. Depending on the goals of particular implementations, type checking may be done statically (at compile- or translation-time), at composition time, and/or at run-time.

3 Generic, Reusable, and "Jumping" Aspects and Concerns

Group members: Lodewijk Bergmans, Krzysztof Czarnecki, Lutz Dominick, Erlend Stav, Bart Vanhaute, and Kris de Volder

Issues: The need for highly generic, reusable aspects and concerns as first-class reusable components, aspect and concern configuration and instantiation, syntactic *and* semantic join point specification, and context-sensitive join point selection ("jumping aspects").

Categories: Challenge problems, requirements statements

3.1 Problem Overview

A key goal of advanced separation of concerns is to achieve better modularity and better reuse by increasing the kinds of concerns and aspects that can be separated and encapsulated. While better modularity has certainly been a result of much work in this field, the ability to reuse aspects and other kinds of concerns in different contexts still falls short of the goal. Achieving these goals requires:

– The ability to model aspects and concerns as first-class, reusable components. A key issue in achieving reusability is ensuring that concerns are not coupled—that is, they do not encode knowledge of, or direct dependencies on, each other (or on any "base" code).
– Support for both syntactic *and* semantic join points. "The method `C.setFoo()`" is an example of a syntactic join point that is specific to a particular class and method. "All methods that modify object state" is an example of a semantic join point that can be applied to any class. The use of semantic join points may improve the set of contexts in which various kinds of concerns can be reused without modification. It also reduces coupling among concerns.
– The ability to use both context-insensitive join points and context-sensitive join points. Context-sensitive join points are ones that determine at run-time, rather than solely at composition time, whether or not to dispatch to particular aspects or concerns that are composed at that join point, based on the value of some run-time state information[1]. When this happens, the set of aspects or concerns that are composed before, after, or around a given join point may change from execution to execution—i.e., the concerns or aspects appear to "jump" from one join point to another. Note that context sensitivity can be either *static* or *dynamic*; when they are purely static, approaches like code duplication may be sufficient to address them, but when they are dynamic, they require dynamic checks to be performed. Context-sensitive join points may increase the circumstances under which a given aspect or concern can be reused.
– The ability to model multiple decompositions of software systems, to reflect the perspectives of different users. Of particular importance is the ability to represent decompositions based on user-oriented features vs. reuse-oriented components, aspects, and other concerns.
– A mechanism that addresses the inheritance anomaly problem [19][20]. This problem arises in languages that support advanced separation of concerns in much the same way as it does in concurrent object-oriented languages, and it has the corresponding adverse effect on reusability and encapsulation of aspects and other concerns as components. In particular, if a set of aspects or concerns is composed

[1] The conditions for executing a given concern may actually be specified *declaratively,* based (for example) on dynamic structures such as call graphs and data flow, which can be also described statically.

with a class, *C*, then all of *C*'s subclasses should, at minimum, be composed with the same aspects or concerns. Furthermore, the introduction of new methods in *C*, or the overriding of *C*'s methods by a subclass, may require new aspect- or concern-level definitions. These new definitions may end up being invasive—that is, they may affect the existing ones, which is undesirable and which may cause unexpected errors.

Motivating papers: See [12] for a discussion of issues in defining generic, reusable aspects, [7] for a full description of the jumping aspects problem, and [19] [20] for an analysis of the inheritance anomaly in concurrent object-oriented languages.

3.2 Challenge Problems and Requirements

Challenge Problem 3: Highly Reusable, Configurable Aspect and Concern Components

To demonstrate some of the key issues in, and requirements on, making aspects and concerns first-class, reusable, configurable components, we use running example that starts with a simple stack in C++ (taken from [11]):

```
Template<class Element_, int S_SIZE>
class Stack {
   public:
      // export element type and empty and maximum top
      // value
      typedef Element_ Element;
      enum { EMPTY   = -1,
             MAX_TOP = S_SIZE-1};

      Stack() : top(EMPTY) {}
      void push(Element *element) {
         elements [++top] = element;
      }
      Element *pop() {
         return elements [top--];
      }
   protected:
      int top;
   private:
      Element *elements [S_SIZE];
};
```

To use this stack in a multithreaded environment, concurrent access to stacks must be synchronized. In particular, three synchronization locks are needed: one to provide mutually and self-exclusive access to the push() and pop() methods, and the other two to implement synchronization guards for threads that are trying to push an element onto a full stack, or to pop an element from an empty stack. The push() and pop() methods must be made to use these locks appropriately.

A key idea of aspect-oriented programming [18], composition filters [1], subject-oriented programming [16], and other work in the advanced separation of concerns area is that manually mixing concerns, like synchronization code, into the source implementation of pop() and push() of the original stack is a bad idea—it results in tangled code, which is hard to understand and to maintain, as shown in [11] (page 280). In addition, the invasive modification produces just a synchronized stack, although having both a synchronized and unsynchronized stack might have been useful.

Requirement 4 Advanced separation of concerns mechanisms must permit developers to implement synchronization code (as well as any other “cross-cutting” or other kind of concerns) separately from any code to which it may apply.

Sample solutions: A simple solution to this problem in a standard object-oriented language might be to use inheritance to separate the synchronization aspect. In this case, we could derive a subclass of class `Stack`, which defines the necessary locks and refines the push() and pop() methods from the superclass by wrapping them into the necessary synchronization code (also from [11]):

```
template<class Element_, int S_SIZE>
class Sync_Stack : public Stack<Element_,S_SIZE> {
   public:
      typedef Stack<Element_,S_SIZE> UnsyncStack;
      // get the element type and empty and maximum top value
      typedef typename UnsyncStack::Element Element;
      enum { EMPTY   = UnsyncStack::EMPTY,
             MAX_TOP = UnsyncStack::MAX_TOP };
      ...
      Sync_Stack() : UnsyncStack(),
                     push_wait(lock),
                     pop_wait(lock) { }
      void push(Element *element) {
         ACE_Guard<ACE_Thread_Mutex> monitor(lock);
         while (top == MAX_TOP) push_wait.wait();
         UnsyncStack::push(element);
         ...
      }
      Element *pop() {
         Element *return_val;
         ACE_Guard<ACE_Thread_Mutex> monitor(lock);
         while (top == EMPTY) pop_wait.wait();
         return_val = UnsyncStack::pop();
         ...
         return return_val;
      }
      ...
}
```

This kind of solution effectively separates synchronization from the core stack. The major limitation of this solution is that existing clients, which use the `Stack` class, must be modified to instantiate the new class, `Sync_Stack`, instead of `Stack`. The standard object-oriented solution to this problem is to use factories in the client code, but this implies that the client code anticipated such a change and used factories to start with. This kind of “anticipatory generality” is responsible for much of the complexity in object-oriented code and frameworks. Unfortunately, the predictions are often wrong and much of the complexity turns out to be unnecessary. A solution to this problem is the use of a composition mechanism, which, unlike inheritance, facilitates the extension of classes without invalidating client code. This permits non-invasive changes without pre-planning [16][22][25].

Requirement 5 Approaches to advanced separation of concerns should provide composition mechanisms that permit the addition of new aspects or concerns to existing code *non-invasively*, without invalidating existing clients, and without pre-planning, to whatever extent possible.

At some later date, a developer discovers the need for a FIFO (first-in, first-out) queue to implement a communication buffer in a multi-threaded application. Clearly, the synchronization aspect developed for the stack is exactly the same as the one needed for the new FIFO queue. This suggests the need for a generic, reusable synchronization aspect.

Sample solution: One way to convert `Sync_Stack` into a reusable aspect is to model it as a parameterized type:

```
template<class Queue>
class Synchronizer : public Queue
   {...}
```

This synchronization aspect can be instantiated for either a stack or a queue:

```
Synchronizer<Stack>
Synchronizer<FIFO_Queue>
```

This solution, like `Sync_Stack`, suffers from the problem that it requires an invasive modification to client code to use it. A solution to Requirement 5 would also address this problem.

Requirement 6 Advanced separation of concerns mechanisms must provide a means of representing generic concerns, which can be specialized for use in different contexts. Parametric types are one example of a means of specifying and specializing concerns.

A significant limitation of the Synchronizer aspect above is that is relies on a consistent set of method names—for example, it assumes the presence of methods push() and pop(), which is wraps to achieve synchronization. In practice, however, a developer might prefer to call the corresponding methods "enqueue()" and "dequeue()". Further, the Synchronizer aspect should also be applicable to other methods that a synchronized class might define. For example, a queue class might define a method size(), which should be mutually exclusive with operations that modify the queue. This suggests that to produce truly generic, reusable aspects and concerns, developers must be able to define *semantic categories* for join points, such as "read-only methods," "adding methods," and "removing methods," rather than simply syntactic join points, such as "the push() method" and "the pop() method." In the synchronization example, we might choose to define the following semantic categories:

ADD category: ADD is self exclusive, and the buffer is not FULL
READ category: no ADD or REMOVE is busy
REMOVE category: REMOVE is self exclusive, and the buffer is not EMPTY
…

Requirement 7 Advanced separation of concerns mechanisms must provide a means of defining semantic join points, as well as syntactic join points.

Challenge Problem 4: The Inheritance Anomaly
The concept of general, reusable aspects, as presented in the previous challenge problem, suffers from the same "inheritance anomaly" problem that was identified in the domain of concurrent object-oriented programming languages [20]. The inheritance anomaly in that domain occurs when developers attempt to write subclasses of classes that contain synchronization code. When these subclasses override individual methods defined in their superclasses, they generally must redefine (or just copy the implementation of) all of the synchronization code contained within the superclass' method.

In the context of advanced separation of concerns, the inheritance anomaly problem manifests itself in much the same way as in concurrent languages: subclasses should inherit any aspects or concerns that are composed with their superclasses.[2] Moreover, the introduction of new methods, or the overriding of superclass methods, may require additional aspect or concern definitions. These new definitions should be non-invasive—that is, they should not affect existing aspect or concern definitions.

The general solution to the inheritance anomaly in concurrent languages requires developers to define semantic categories on which the synchronization requirements can be expressed. Mappings between object states and the semantic categories are then defined. The object states, the semantic categories, and the mapping between them are kept separate. A similar approach is one way to address the inheritance anomaly in advanced separation of concerns mechanisms.

To illustrate the inheritance anomaly and one solution approach in the context of advanced separation of concerns, we return to the stack/queue example from the previous challenge problem, in which a generic producer/consumer synchronization aspect was defined. As described earlier, the synchronization constraints are specified with respect to categories of method calls, and thus, they pertain to many possible concrete representations (stacks, queues, lists, etc.). The set of semantic categories described were:

ADD category: no READ or REMOVE is busy, and the buffer is not FULL
READ category: no ADD or REMOVE is busy
REMOVE category: no ADD or READ is busy, and the buffer is not EMPTY
…

For each concrete representation, like a stack or queue, a mapping must be defined between these categories and the concrete methods of the representation, and between the categories and concrete object state. For example, for the stack, we might define:

push(Element) → ADD
peek() → READ
pop() → REMOVE
FULL == (top == size+1)

For a queue, the mapping might be:

enqueue(Element) → ADD

[2] In fact, synchronization is simply one example of an aspect that might be composed with a class whose subclasses should have the same synchronization aspect composed with them.

dequeue() → REMOVE
FULL == (head == tail)
EMPTY == (tail+1 % size == head)

These mappings could then be used as join point specifications for a compositor, which would compose the synchronization code appropriately with the stack and queue.

Other proposals for addressing the inheritance anomaly in concurrent languages also exist and could likely be adapted for use in the context of advanced separation of concerns.

Requirement 8 All advanced separation of concerns approaches for object-oriented languages (and other languages that include behavioral polymorphism mechanisms) must address the inheritance anomaly in some way. Further, any concerns that are composed with a class should also be composed with its subclasses. It should be possible to adapt "inherited" concerns for any subclass without affecting the superclass, just as it is possible to adapt superclass methods in a subclass without affecting the superclass.

Challenge Problem 5: "Jumping Aspects" in Model-View-Controller Change Notification

The previous two challenge problems addressed some significant issues in providing support for highly generic, reusable, configurable aspects and other concerns, including the need to specify semantic join points. This problem highlights another major issue: the need to specify *dynamic* join points.

To illustrate this issue, we use the Smalltalk Model-View-Controller pattern. In Smalltalk applications, it is common practice to separate the *representation* of an object from its *presentation* using MVC. In MVC, a View must be notified of any change in its Model. To accomplish this, the model must invoke the method `changed` (i.e., `self changed`) from every method that changes its state. This results in the propagation of an update event to all views of the model. Change notification, in the form of `self changed` calls, represents an aspect that can be composed into ListModel methods that change the model state.

Two methods from a Model for a `List` class are shown below: `add`, which adds a single element to a list, and `addAll`, which adds multiple elements at the same time:

```
ListModel>>add: anElement
   myElements at: currentIndex put: anElement.
   currentIndex := currentIndex + 1.
   self changed.  "***aspect code for change
                    notification"

ListModel>>addAll: aCollection
   aCollection do: [ :each| self add: each].
```

This naïve implementation results in poor performance, however, because change updates occur each time `addAll` uses `add` to insert a new element into the list. The best solution is for `addAll` to invoke `self changed` itself, and for `add` to call `self changed` *only* if it is invoked by an external `ListModel` client, and *not* if it is invoked by `addAll` or other `ListModel` methods. Thus, `add` performs different actions—it includes or excludes the change notification aspect—depending on from where it is called:

***For calls that occur directly to add from outside ListModel*:**

```
ListModel>>add: anElement
   myElements at: currentIndex put: anElement.
   currentIndex := currentIndex + 1.
   self changed.   "***aspect code is included"
```

For calls to add that occur from addAll, the aspect code "jumps" to the calling context

```
ListModel>>addAll: aCollection
   aCollection do: [ :each| self add: each].
   self changed.   "***aspect code is included"

ListModel>>add: anElement
   myElements at: currentIndex put: anElement.
   currentIndex := currentIndex + 1.
   "***no self changed when called through addAll"
```

Synchronization code that obtains and releases locks upon entering and exiting methods, as in Challenge Problem 3, is also an example of jumping aspect code, because once a lock has been obtained, nested calls made from within the method that obtains the lock need not execute the code to obtain the lock, because they obtain it from the method that calls it.

These examples suggests the following requirement on advanced separation of concerns mechanisms:

Requirement 9 Static join points, such as method names, cannot themselves always specify the location at which aspects or concerns are to be composed. For context-sensitive compositions ("jumping aspects"), it is necessary to specify join points *and* the (possibly dynamic) context in which the composition should occur, and to provide any associated declarative, compositor, and run-time support needed to enable compositions to be affected by context-sensitive conditions. Such mechanisms must handle aspects that jump either *within* a class or *across* multiple classes.

4 Modularization and Evolution Using Advanced Separation of Concerns

Group members: Jean Paul Arcangeli, Isabel Brito, Robert Filman, Eric Hilsdale, Donal Lafferty, Bert Robben

Issues: Non-invasive addition of new concerns or aspects to an existing software system; ease of evolution in the presence of advanced separation of concerns; the need for dynamic join points; the need for context-sensitive join points (the "jumping aspects" problem—see also Section 3)

Categories: Challenge problems, requirements statements

4.1 Problem Overview

A fundamental principle of software engineering is that systems that can be separated into (relatively) independent parts are simpler to develop, and easier to maintain and evolve. Traditionally, this separation has been about the primary functionality of systems—related functionality would be grouped into objects or components, with well-defined interfaces between elements. The key hypothesis of this work is that other separations are possible; that technologies can be created that allow separate expression and development of non-functional concerns while still yielding efficient, working systems. Several characteristics of any particular approach to advanced separation of concerns affect that approach's ability to provide these additional modularization and evolution benefits:

– How well the approach separates the expression of different facilities.
– Whether or not specifying a new concern can be done non-invasively—i.e., the new concern does not require any existing code to be changed ("core" functionality or other concerns or aspects).
– The kinds of join points the approach supports for composition, including context-sensitive join points (as described in Section 3), dynamic join points, and class- vs. instance-based join points.
– How well the approach can limit the impact of an evolutionary change to a given concern or aspect on the rest of the system, and how the change is effected (e.g., stop the system, recompile it, and rerun it vs. on-line evolution).
– How readily the approach facilitates the representation of some key kinds of aspects, such as synchronization.

Motivating papers: The challenge problems reported in this section were motivated by [14] and [24].

4.2 Challenge Problems and Requirements

This set of challenge problems is based on variations of an example involving mobile agents that visit servers. These agents wander the web, attempting to accomplish some task. In pursuit of their goals, they need to execute programs on servers. Sometimes these agents must share information with one another. They do so using "coordination loci," through which they can communicate results and task assignments.

Fig. 2 presents a high-level architecture for such a mobile agent system. In this architecture, servers respond to two kinds of messages:

3. `enter(program)`→`boolean`: This method used is by an agent to dock at a new remote server. It returns true or false, depending on whether the remote server accepts the agent.
4. `query(string)`→`string`: This method is used to obtain information from the server with which an agent is docked. Servers only accept queries from agents that are docked with them. Note that some of the strings that are passed to, or returned from, a query method may encode object references, as is the case with IORs in CORBA. (In the absence of this facility, one could define a "name server" that converts strings to object references.)

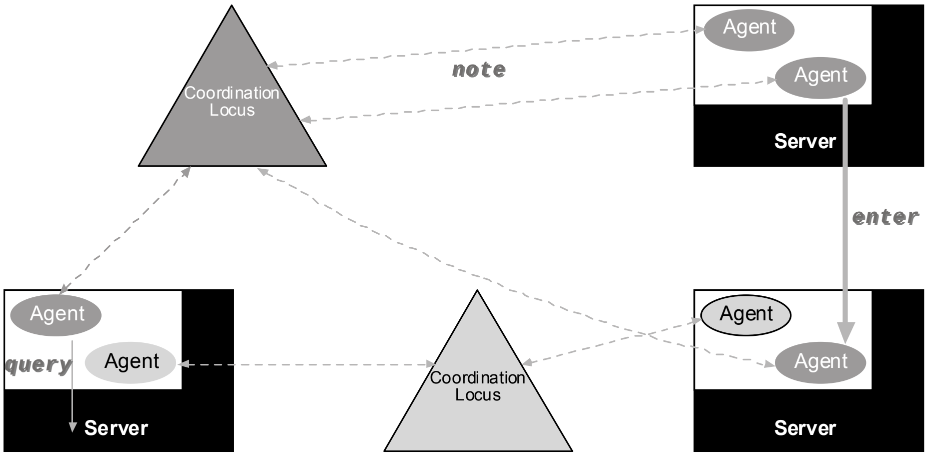


Fig. 2. High-level architecture for the mobile agent system.

Agents are subclasses of coordination loci. Both support the `note` method, which is used to communicate arbitrary information. Agents can `enter` servers, `query` servers, and send messages to, and receive messages from, coordination loci via the `note` method.

The following series of challenge problems work from the assumption that there is already an existing implementation of this mobile agent system. Since we are interested in software systems evolution, each challenge problem requires either the addition of some new capability to, or the modification of an existing capability in, the system.

Challenge Problem 6: Security in Mobile Agent Systems

To introduce security into the mobile agent system, it is necessary to incorporate the concept of *agent identity* into the system, and to use it as needed to ensure that agents only have access to the servers and information to which they are permitted. Agent identity means that each agent must have an identifier (its "identity"), which allows it to be distinguished from all other agents in the system. (Think of this as the identity of the user that spawned the original request.) The agent identifier is used to enforce several security policies:

– *"Spawned agent" equivalence*: When a given agent executes the `enter` method on another server, it spawns a new agent that runs on that server. The spawned agent should be treated the same way as the original—that is, it should have the same identity as the spawning agent. Servers may then choose to accept or reject an `enter` request based on the identity of the calling agent, as well as on any other factors they deem appropriate.
– *Restricted queries*: Servers should only process `query` requests from `enter`ed agents. Further, the set of queries that any given agent can run may be restricted. Based on the identity of a given agent, a server should be able to reject the query (e.g., by raising an exception or returning an error code that indicates the requesting agent does not have permission to issue that query), filter either the

contents of the query to make it acceptable or the results of the query before returning them, or simply service the query request as-is.

The addition of the new identity concern should be done non-invasively—that is, its definition should not require any modifications to the existing system (Requirement 5).

The ability to restrict queries illustrates the "jumping aspects" problem (Section 3). Therefore, it also illustrates the need for context-sensitive join points, where the agent's identity is the required context information (Requirement 9). It also, however, implies the following additional requirements:

Requirement 10 Compositors must be able to examine *method parameter values* and *return values* as part of the information they use to determine whether or not to execute particular concerns or aspects at a given context-sensitive, dynamic join point.

Requirement 11 Advanced separation of concerns mechanisms must provide the ability for one concern to guard the execution of code at any given join point (e.g., a method) that belongs to other concerns, including "base" code.

Challenge Problem 7: Concern Dynamics

Security policies may change over time. Consider what must be done to change the behavior of a concern. There is a wide spectrum of approaches to replacing concerns or aspects, ranging from completely static mechanisms to completely dynamic ones, and each point on the spectrum has its own advantages and disadvantages. For example:

– Completely static: An example of a completely static approach is one that entails terminating the execution of the composed mobile agent system or some components of it, recompiling and/or recomposing the affected parts, and restarting the program or subcomponents. A key advantage to this approach is that it is statically type checkable and it imposes little run-time overhead—the cost of composition is primarily a compile-time one, which may be particularly advantageous to performance-critical applications. A major disadvantage, in some contexts, is that it necessitates the termination and restarting of the running software.
– Completely dynamic: A completely dynamic mechanism might include a means of replacing a concern in-place, while the composed program is running, as in the case of composition filters [1] and reflective and interpretive approaches. This type of approach is considerably more flexible and also provides the richest set of context-sensitive and dynamic join points, but it may result in deferring the identification of some kinds of errors to run-time, where they cost more to detect, and it may impose more run-time overhead (for the additional error-checking and to enable dynamic changes to a running executable).

Requirement 12 Many points on the static↔dynamic spectrum of approaches to concern replacement exist, each representing different tradeoffs. Ideally, any given approach to advanced separation of concerns should address multiple points on the spectrum, if possible, or it should clearly be specialized for a target domain in which a single point is clearly the best one.

Challenge Problem 8: Synchronization in Mobile Agent Systems
So far, the mobile agent system has assumed that only single messages are sent to coordination loci at any given time. Clearly, a given agent may have to send a series of "notes" to a coordination locus, and to do so without interference from any other agents. This challenge problem includes two variants of this problem:

– Locking: If an agent needs to send a series of messages to a single coordination locus without interference, standard locking of a coordination locus permits agents to have exclusive access to the locus while they send a series of "notes." Locking represents a new concern in the system.
– *Transactions*: While locks permit exclusive access, they generally do not provide properties such as atomicity or serializability—properties associated with standard transactions. Further, locks do not provide a good means of permitting agents to send a series of messages to multiple coordination loci. Thus, standard transactions over coordination loci should be supported as an additional concern. Note that, unlike locks, transactions affect the existing mobile agent system in some fairly deep ways. For example, the mobile agent system will have to be enabled for transaction abort, rollback, and logging, which affects most of the methods in the "base" system. Further, transactions may interact with other aspects or concerns as well, necessitating changes to them to permit the transactions to preserve the ACID (atomicity, consistency, isolation [serializability], and durability [persistence]) properties.

Requirement 13 Some kinds of concerns, like transactions, interact with others. An advanced separation of concerns mechanism must permit the composition of concerns with other concerns, and must provide a non-invasive means for concern-to-concern interactions, as well as concern-to-"base" interactions.

Challenge Problem 9: Dynamic Join Points in Mobile Agent Systems
To this point, all of the aspects or concerns that have been added to the mobile agent system were intended to be composed at the *class* level. This challenge problem highlights the need for composition at the *method* and *instance* level as well. In particular, it illustrates the need for *dynamic join points*.

The first part of this challenge problem operates at the *method* level. It involves the introduction of a debugging mechanism to the mobile agent system, which sends a `note` to a specific coordination locus upon every query.

Requirement 14 Advanced separation of concerns approaches must provide a means of composing new behaviors into some set of functions in an existing code base. Ideally, it should be possible to select the set of functions using a fairly rich specification notation (e.g., regular expressions).

The second part of this challenge problem involves the introduction of the same debugging code on a *specific* agent, and on all of its descendants (its spawned agents). In this case, the intent is to compose the debugging behavior into a set of *instances*. We specifically exclude approaches in which the debugging concern performs a lookup on each agent that runs a query to see if it is supposed to run the debugging code, because doing so imposes a number of localization/failure problems on the distributed mobile agent application.

Note that the selection of instances with which to compose behaviors is an example of the need for *dynamic join point selection* (as distinguished from context-sensitive join points).

Requirement 15 Advanced separation of concerns approaches must provide dynamic join points as well as static. Dynamic join points must also provide the ability to compose new behaviors into some set of instances. This mechanism should *not* require an aspect or concern to keep track of all instances, or of all instances that should have the behavior.

5 Exception Handling and Timing Constraints

Group members: Andrew Black, Maja D'Hondt, Wolfgang De Meuter, Mik Kersten, Oscar Nierstrasz, J. Andrés Díaz Pace

Issues: The need for context-sensitive join points in exception handling and systems with timing constraints.

Categories: Challenge problems, requirements statements

5.1 Problem Overview

The need for context-sensitive join points (described in Section 3) arises in many areas in software engineering, including some non-obvious ones like exception handling and systems with various kinds of timing constraints (including real-time). These issues are critical. The reusability of a given component depends, in part, on the encapsulation of exception and failure processing as coherent concerns, because how a component handles exceptional conditions and failures often varies from one usage context to another (e.g., in one context, it may be appropriate to throw exceptions upon detecting an erroneous condition, while in another, it may be necessary to take corrective actions, and the corrective actions may also be usage- and/or context-specific). The ability of components to satisfy timing constraints is also affected by the ability to separate the usage context from time-critical concerns.

Motivating papers: The "jumping aspects" problem is identified in [7]. The particular need for advanced separation of concerns to address issues in satisfying timing constraints was originally described in [6], but its classification as a "jumping aspect" problem was established during the workshop.

5.2 Challenge Problems and Requirements

Challenge Problem 10: Context-sensitive exception and failure handling
The need for context-sensitive exception handling is described in [7] as a manifestation of the "jumping aspects" problem. Consider a component that can be used either as a stand-alone application or by a larger application. Thus, this component may be used in different contexts with different requirements. Among these differences are:

– Error condition definition: Different applications may define what constitutes an "error" differently. For example, one application may consider it an erroneous condition if a lookup on a hash table fails to find an element with the given key; another application may be coded to assume that not all queries will find a matching element, and that if no matching element is found, a "null" reference will be returned. The ability to specify error conditions separately from any component that may recognize those conditions as erroneous is critical to being able to reuse components in multiple contexts.
– Error handling: Once an erroneous condition is detected, different applications (or, more generally, different usage contexts) must be able to specify how to respond to that condition.

Requirement 16 Any approach to advanced separation of concerns must permit the encapsulation of exception detection and handling as separate concerns. At minimum, such mechanisms must provide the ability to encapsulate as separate concerns both the condition that denotes the exceptional event, and the response to such a condition. Since both the definition of "exceptional event" and the response to it can vary between different usage contexts, it is necessary for a solution to this requirement to address Requirement 9 (i.e., solutions must provide dynamic and context-sensitive join points) and Requirement 11 (solutions must permit "guards" around join points).

Sample solution: We can hypothesize an aspect-oriented exception handling mechanism where one can declaratively state that "if method *x* fails in concern *A*, then execute method *y* in concern *B*." The advantage of such an approach is the separation of the failure concern, consisting of what could go wrong and how to address it, from other functionality. A language that realizes this failure concern should be able to express what might fail and where (e.g., corresponding to *throws* statements in Java™), as well as where and how to resolve the conflict (e.g., corresponding to *try-catch* statements). Notice that the context-sensitivity here arises from the requirement that the component be able to run both on its own and as part of other applications. If it runs on its own, it should handle exceptional conditions itself, whereas if it runs as part of another application, the exceptions should propagate to the application to be handled there. This results in the need for context-sensitive, dynamic join points.

Challenge Problem 11: Time in Information Flow

The domain of this challenge problem and the next is real-rate systems (described in [6]), where data is moved from one location to another at defined rates. Example systems in this domain are audio conferencing and streaming video. A general characteristic of real-rate systems is that they usually deliver data (such as audio or video frames) from a source (usually a server or file system) to a sink (e.g., a display or a sound generator). A key requirement on such systems is that the data must arrive periodically, with constrained latency and jitter. Such systems are difficult to design and construct, which suggests the need for generalization of these issues into an information flow framework. *InfoPipes* [6] is such a framework, which enables the building of real-rate systems from pre-defined components such as buffers, pipes, filters and meters. Properties such as latency, jitter and data-rate of the resulting pipeline are calculated from the corresponding properties of the components. In the development of the InfoPipes framework, the components naturally mapped onto objects: there are objects representing bounded and unbounded buffers, straight pipes, sources, and sinks, where each of them has zero or more inputs and outputs. A properly connected pipeline consists of a set of connected components; each component's inputs are connected to an output of another component.

An important concern in information flow is time: the flow of data through the pipeline is subject to hard timing constraints. Consider, for example, the simple pipeline illustrated in Fig. 3: the server can provide audio frames at a certain rate, via a network with variable rate, either directly to a sound generator or via a decompressor. Both the decompressor and the sound generator must be supplied with audio frames at a determined rate. Suppose that the network's rate decreases considerably, causing a violation of the decompressor's and the sound generator's required rates. If the audio frames must flow via the decompressor, then the decompressor must handle this violation (e.g., by resending the last audio frames). If, on the other hand, the frames go directly to the sound generator, the sound generator must handle the violation. Therefore, the timing failure concern "jumps" in and out of the sound generator's code that handles the receipt of audio frames, depending on whether the caller of this code is the network or the decompressor—i.e., it is context-sensitive.

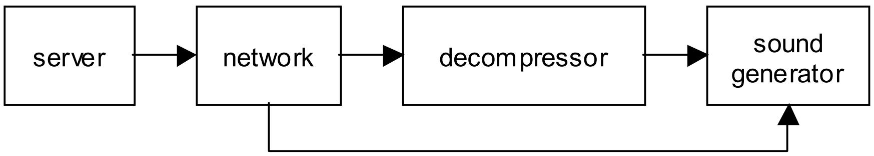


Fig. 3. An InfoPipe pipeline for a sound-streaming application.

This challenge problem clearly demonstrates Requirement 9. It also imposes the following requirement:

Requirement 17 Advanced separation of concerns approaches must separate the specification of concerns from the specification of when, and to what, those concerns apply. That is, it must be possible to separate the specification of a concern from the specification of the contexts in which it is applicable. It should be possible, for example, to write a generic mechanism that handles violations of timing constraints, and to specify separately the join points at which that mechanism should be composed, or the circumstances under which it should be composed. For the same reason that object-oriented languages incorporate dispatch mechanisms based on an object's run-time type instead of forcing developers to implement "typecase" structures (i.e., extensibility via polymorphism), it is critical that developers not be required to implement "concern case" structures.

6 Multiple Views

Group members: Thomas Kühne, Cristina Lopes, Mira Menzini, Renaud Pawlak, Eddy Truyen

Issues: The need for multiple levels of abstraction and stepwise refinement [29] in advanced separation of concerns; the need for dynamic, context-sensitive join points; the need for fine-grained access control.

Categories: Challenge problems, requirements statements, problem generalization

6.1 Problem Overview

Many changes to a software system can be made without knowledge of the full details of the system's implementation. It is desirable to permit developers to work at the highest level of abstraction possible and to facilitate navigation between different levels of abstraction as needed, since doing so reduces the complexity of development tasks and promotes comprehension [29].

The interactions among different components also appear different at different levels of abstraction, just as the components themselves do. Thus, the ability to model and navigate among different levels of abstraction effectively depends on the ability to describe both components and their interactions at different levels of abstraction.

Access control provides yet another view of a system. A given object, *A*, sees a view of another object, *B*, depending on the class of *B and* on the access privileges of *A*'s owner (e.g., the user who created *A*). It must be possible to protect parts of objects against unauthorized access.

Motivating papers: The need for multiple levels of abstractions is taken from [4]. The dynamic wrappers solution is adapted from [28]. The access control issue is raised in [24], and also illustrates the need for context-sensitive join points (the "jumping aspects" problem, described Section 3).

6.2 Challenge Problems and Requirements

Challenge Problem 12: Multiple Levels of Abstraction in Software Systems
The need for multiple levels of abstraction in software is illustrated by a simple piece of software that writes data to a file. At the highest level of abstraction, at which all implementation details are omitted, the system has three abstractions: a `Writer`, which needs to write `Data` to a `File`, as illustrated in Fig. 4.

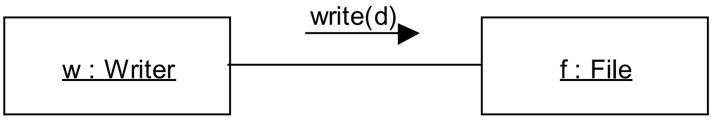


Fig. 4. Highest-level abstraction of file writer software

Notice that the `write` method takes a single parameter—the data to be written. Moving down one level of abstraction, however, it becomes clear that a `File_Manager` actually controls access to files, as shown in Fig. 5.

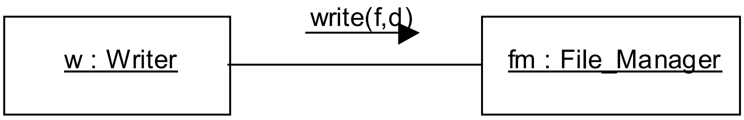


Fig. 5. Design-level abstraction: Access through a file manager

Although the basic interaction between components is the same as that shown in Fig. 4, two significant changes are apparent:

- The server has a different identity. It is now a file manager, rather than a file object.
- The write function now takes two parameters instead of just one: a file identifier, and the data.

Proceeding down still another level of abstraction to add implementation details reveals that file managers are actually distributed components—that is, clients must access them remotely, as depicted in Fig. 6.

Notice that the direct interaction between clients and file managers has been replaced by an interaction between object request brokers (ORBs) and the necessary client/ORB and server/ORB interactions. Further, the name of the function called by the Writer changed from write to request and now includes three parameters instead of two—the first parameter indicates which service is required.

Many additional details might appear in lower levels of abstraction—for example, the communication between ORBs might require encryption.

Note that the interactions among components may potentially be scattered across different subsystems. This characteristic clearly identifies this problem as one that requires advanced separation of concerns.

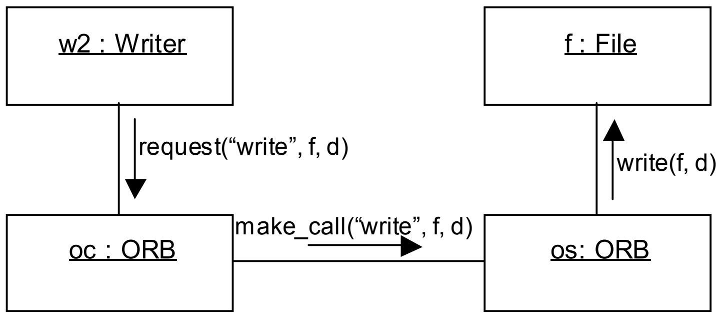


Fig. 6. Implementation-level abstraction: Distributed file management in CORBA

This challenge problem demonstrates an important requirement on advanced separation of concerns approaches:

Requirement 18 Advanced separation of concerns mechanisms must provide a means of providing multiple related views of a system. Multiple views permit the representation of different levels of detail for both the components that are modeled *and* their interactions. Such a mechanism must also provide a means of relating the views to one another, to permit navigation among them. Approaches must provide both *modularity* and *flexibility*. Modularity is required to permit the representation of the incremental changes that comprise a stepwise refinement. These incremental changes must be represented as a coherent module (either logical or physical). Flexibility is needed to permit selection—possibly dynamically—among a set of possible implementations, and thus, it also illustrates the need for a range of static to dynamic composition (see Requirement 12).

Sample solutions: The "stratified architecture" approach, described in [4], directly addresses the problem of how to support multiple levels of abstraction. As depicted in Fig. 7, different levels of abstraction are defined, where Fig. 4 through Fig. 6 show the contents that might appear at particular levels. The relationships among different levels of abstraction are also defined using refinement patterns.

A different possible solution, called *dynamic wrappers*, is presented in [28]. This approach particularly attempts to address the need for flexibility by permitting the building of systems as dynamic compositions of different *interaction refinements* (a means of modifying component interactions by intercepting and modifying messages flowing between the components). It permits the design of a system using a component-oriented methodology that decouples the type of a component (its specification or interface) from its implementation, so that the two can vary independently [17]. Interaction refinements are defined by wrapping the component type, using the Decorator pattern [15]. A "variation point" (a context-sensitive, dynamic join point—see Section 3) manages the wrappers and decides, based on an externally specified composition policy, which wrappers (and thus, which interaction refinements) to invoke, and in which order to invoke them. Interactions between component types are reified into a basic interaction flow and a context flow. The

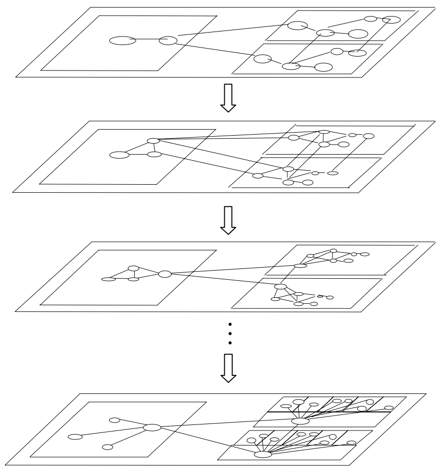


Fig. 7. A stratified architecture.

context flow carries the composition policy that was specified for that interaction. Dynamic wrappers are illustrated in Fig. 8.

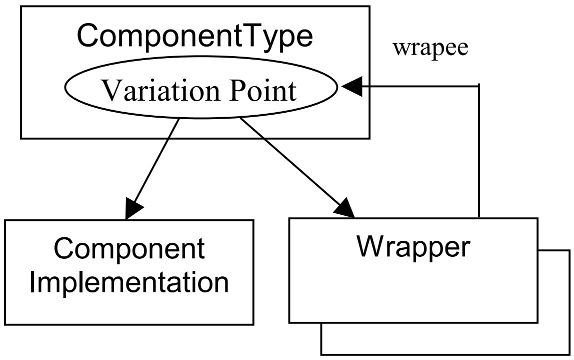


Fig. 8. The dynamic wrappers approach.

The dynamic wrappers approach permits dynamic selection among implementations for a given data type. For example, it allows clients to choose implementations that either do or do not encrypt their data as it is sent to the file manager. Here, encryption and decryption of data can be seen yet another interaction refinement that is integrated selectively—and dynamically—whenever necessary.

The main disadvantage to the dynamic wrappers approach is that it does not satisfy the requirement for modularity well. The implementation of an interaction refinement involving several wrappers simultaneously could not be modularized well using this approach.

Challenge Problem 13: Access Control
This challenge problem demonstrates the need for access control in advanced separation of concerns. The problem involves the introduction of authorization features into an existing television broadcast planning system [24]. Both short- and long-term planning of broadcast programming are business-critical information; thus, access to this information should be restricted to a limited subset of employees. Each product (e.g., a movie, series, or program) is assigned a status, and each employee is assigned a role in the broadcast planning process. Users are either granted or denied access to particular product information (which they access through various tools, such as a GUI or a report generator) based on their particular role, on the product's status, and on the tool that is used to access the information.

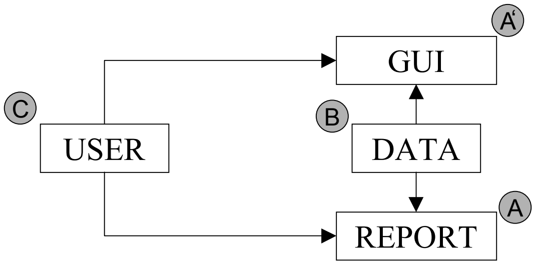


Fig. 9. Accessing authorized data in the television broadcast planning system.

Fig. 9 summarizes this scenario. As it suggests, access privileges to the data (*B*) are determined based on both the tool that attempts to access the data (*A*) *and* on the role of the user who is running the tool (*C*). Note that some portions of the tool may be inaccessible to certain kinds of renderings—for example, a particular user may, in principle, be able to view a piece of data with a GUI, but, for security reasons, that data may not be accessible using the report generator, which might send it to a printer where it would be vulnerable to unauthorized access.

Note that this problem also illustrates the need for context-sensitive join points (Section 210), where the context, in this case, include the access rights of the user and of the tool that are attempting to access the data, as well as the data itself and its particular status; it therefore also imposes Requirement 9. In addition, it suggests the following requirement:

Requirement 19 Advanced separation of concerns mechanisms must provide a means for a single component (object, etc.) to provide different interfaces for different clients or categories of clients, and it must enforce correct usage of such interfaces. Correct usage might be determined statically, dynamically, or both.

A suitable solution to Requirement 9 should also address Requirement 19.

Sample solutions[3]**:** The stratified architecture approach could be used to address this problem. With this approach, developers first implement the broadcast planning system without authorization, and then annotate those relationships that involve authorization. Next, developers must define the refinement patterns that govern the transformation of these annotated relationships and their components into a different set of relationships and possibly additional and/or altered components.

An AspectJ [18] solution might entail the use of the "cflow" construct, which indicates the identity of the initiating caller (the user, in this case). Given this information, it would be possible to check the access rights of the user. Note that this approach actually uses context-sensitive join points to address this challenge problem.

The dynamic wrappers approach, as depicted in Fig. 10, might define a component type for each of the entities depicted in Fig. 9. A component type specifies the service interfaces and dependencies that components of this type have. None of the implementations of these component types contains any authorization code. Instead, authorization is implemented using a set of wrappers. Wrappers for the GUI type and the Report type each implement a customized authorization strategy. Note that propagation of user identification and access rights does not occur automatically; instead, it must be implemented by a third wrapper, for the User type, which adds user identification and privileges to the context flow of the reified interaction.

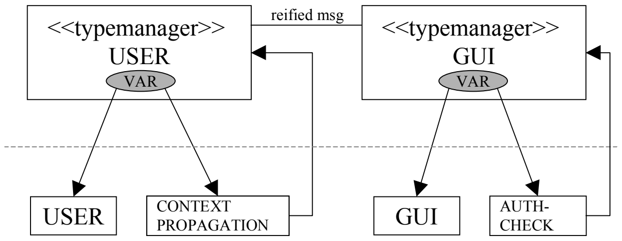


Fig. 10. Using dynamic wrappers to solve the authorzation challenge problem.

7 Context-Dependent Specialization of Existing Behavior

Group members: Johan Brichau, Pascal Costanza, Gregor Kiczales, Elke Pulvermueller, Patrick Steyaert, and Tom Tourwe

Issues: The need for context-sensitive and dynamic concerns and join points; the object identity problem.

Categories: Challenge problems, requirements statements

[3] A word of caution is in order here. The original problem, as defined in [24], is considerably more complex than the simplified version given here. Thus, some of the sample solutions may not scale up to the original problem.

7.1 Problem Overview

In some cases, a given component's behavior may have to be specialized based on run-time context information. In particular, at some join point, a component may need to incorporate (i.e., be composed with) one or more aspects or concerns; those aspects or concerns may need to be executed, executed differently, or not executed at all, depending on the dynamic context in which the component is used. This is a manifestation of the "jumping aspects" problem (see also Section 3).

A common manifestation of this problem occurs in the presence of delegation (the Decorator pattern [15]). Depending on the usage context of a wrapped object, it may be desirable to treat it either as having a wrapper or not. This problem has been referred to as the "object identity" problem.

Motivating papers: The need for context-sensitive and dynamic concerns and join points is taken from the "jumping aspects" problem [7]; the object identity problem was identified in [10].

7.2 Challenge Problems and Requirements

This set of challenge problems is based on variations of a running example involving a simple figure editor application (taken from [7]). The editor permits the drawing of lines and points, where a line is defined by two points. The application supports multiple simultaneous depictions of a figure; each figure element (lines and points) can have a different color in each depiction. A sample UML diagram for such a figure editor is shown in .

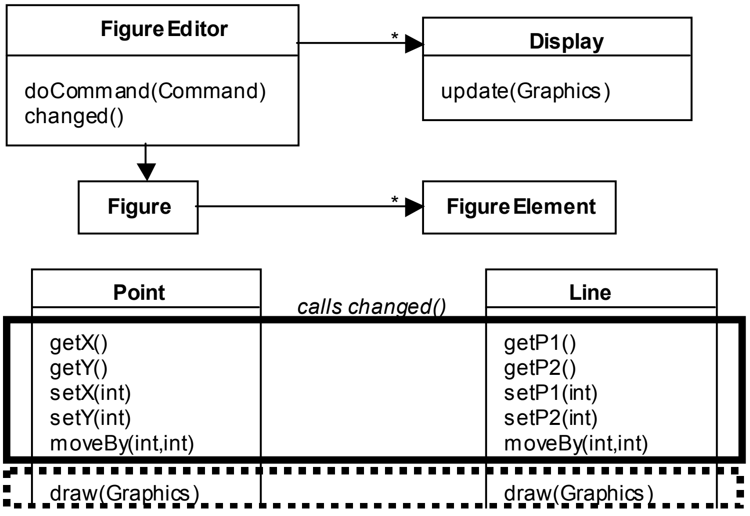


Fig. 11. UML diagram for figure editor application.

Challenge Problem 14: Context Sensitivity to Change Notification
The figure editor's user interface is decoupled from the core functionality of the application using the Model-View-Controller (MVC) pattern [15]. Thus, changes in the state of figure elements (position, color, etc.) should trigger an update in the user interface. To achieve this update, each method that modifies the state of a figure element must send a `changed()` message to the `FigureEditor`, which ultimately invokes the user interface's update mechanism.

The invocation of the update mechanism is clearly a cross-cutting feature—many methods modify figure elements and must send the same `changed()` messages—and, therefore, it should be encapsulated as a separate concern. Notice, however, that the movement of a line is handled differently from the movement of a point: the line's `moveBy()` method will call the `moveBy()` method of its points, which then result in change notification upon the movement of each point in the line. This is undesirable, as described in detail in Challenge Problem 5—the change notification should occur only once, after the line has been moved in its entirety.

This requirement to defer change notification represents the context dependence of this aspect: all `moveBy()` methods represent potential join points for this aspect, but the `changed()` method should not be called (i.e., the notification aspect should not be executed) if the `moveBy()` method was called from another `moveBy()` method (another join point of the same aspect). Thus, this problem illustrates Requirement 9.

```
aspect DeferMoves {

  pointcut deferTo():
      !withinall(FigureElement) & movecalls();

  pointcut movecalls():
      calls(Line, void moveBy(int, int))    |
      calls(Line, void setP1(Point))        |
      calls(Line, void setP2(Point))        |
      calls(Point, void moveBy(int, int))   |
      calls(Point, void setX(int))          |
      calls(Point, void setY(int));

  static after(): deferTo() {
    System.out.println("update!");
    fe.changed();
  }
}
```

Fig. 13. AspectJ solution to Challenge Problem 14.

Sample solution: A solution using AspectJ [18] is given in . This figure defines an aspect, `DeferMoves`, which invokes the change notification mechanism by weaving an **advice** after all calls to "movement" methods, but only when the calls to such methods are not made from within the `FigureElement` class.

Although this solution is almost sufficient to address Challenge Problem 14. it is not entirely correct, because it excludes calls to "movement" methods that are made from within the `FigureElement` class itself. A correct solution would weave an **advice** after all "movement" methods when the call to the method is not made from within any other "movement" method; for example:

```
pointcut deferTo():
   !withinall(movements()) & movecalls();
```

In the current version of AspectJ, however, the **withinall** construct cannot take entire **pointcuts** as arguments.

Challenge Problem 15: Context Sensitivity in Changing Colors
The actual drawing of figure elements on the screen is done by calling a `draw()` method on each figure element. The `draw()` method takes as a parameter a `Graphics` context object, which represents the display on which to draw the figure element.

The context sensitivity issue arises in this problem because each figure element can be depicted in a different color on each display. The algorithm for determining the color is the same for every `draw()` method, and it depends on the calling context (i.e., on the display from which the `draw()` method was invoked, directly or indirectly). Thus, setting the drawing color is clearly a cross-cutting concern, as the `draw()` methods that this capability affects are scattered across the class definitions for all of the figure elements. Again, this is a context-dependent concern, since the actual setting of the color depends on the calling context of the join point method.

Sample solution:
Fig. 15 shows an AspectJ solution to the changing colors problem. The `DrawingCanvas` aspect encapsulates the canvas (the display on which the drawing is depicted) and makes it available to the entire flow of control, starting at the invocation of the `update()` method on the canvas itself. Because the invocations of the `draw()` methods occur in that flow of control, the canvas is made available to the execution code of the `draw()` method (i.e., it can access the relevant context information). The `PerCanvasColor` aspect actually introduces the coloring code into the `draw()` method. This aspect wraps code around the `draw()` methods that retrieves the canvas (which is supplied through the `getDrawingCanvas()` method of the `DrawingCanvas` aspect) and sets the appropriate color.

Note that while Challenge Problems 14 and 15 both illustrate the need for context-sensitive join points, they use the context information differently. In Challenge Problem 14, context information is needed to determine whether or not the change-notification aspect should be executed at a given join point. In this case, the compositor tool uses the context information. In Challenge Problem 15, the aspect itself needs the context information instead, to determine the painting color.

Requirement 20 Context sensitivity may affect a compositor tool or the definition of an aspect or a concern. Advanced separation of concerns approaches must provide a means of permitting context information to be used by any entity that requires it. This includes the ability to identify new context information as the need arises and to ensure that it is made available where needed, all encapsulated within a concern that can be added to existing code non-invasively.

```
aspect DrawingCanvas of eachcflow(callToUpdate(Canvas)) {
    Canvas canvas;

    pointcut callToUpdate(Canvas c):
       instanceof(c) & receptions(void update(Graphics));

    before(Canvas c): callToUpdate(c)
      { canvas = c; }

    static Canvas getDrawingCanvas()
      { return (DrawingCanvas.aspectOf()).canvas; }
}

aspect PerCanvasColor of eachobject(instanceof(FigureElement)) {
    static Hashtable table = new Hashtable();
    Color getColor(Canvas canvas)
       { return (Color)table.get(canvas); }
    static void setColor(FigureElement fe, Canvas canvas,
                         Color color)
       { (PerCanvasColor.aspectOf(fe)).table.put(canvas, color); }

    pointcut drawOperations(Graphics g):
        instanceof(FigureElement) & receptions(void draw(g));

    around (Graphics g) returns void: drawOperations(g) {
      Canvas canvas = DrawingCanvas.getDrawingCanvas();
        Color  color  = getColor(canvas);
        if ( color == null ) {
            thisJoinPoint.runNext(g);
        }
        else {
            Color oldColor = g.getColor();
            g.setColor(color);
            thisJoinPoint.runNext(g);
            g.setColor(oldColor);
        }
    }
}
```

Fig. 15. AspectJ solution to the color-changing problem.

8 Distributed Systems and Middleware

Group members: Mehmet Aksit, Federico Bergenti, Constantinos Constantinides, Pertti Kellomaki, Blay Mireille, Tommi Mikkonen, Klaus Ostermann, John Zinky

Issues: Quality of service concerns in distributed middleware; collection of data about components in distributed middleware (e.g., for monitoring and control of a distributed system)

Categories: Requirements statements

8.1 Problem Overview

Distributed systems pose many challenges for advanced separation of concerns. They generally consist of software components and middleware, where the middleware provides services for connecting and integrating components in distributed systems. Quality of service (QoS) is a key concern in such systems; it includes such critical issues as performance and reliability, which impact many (if not all) of the components and middleware in a distributed system.

Motivating papers: Some issues in achieving advanced separation of concerns for distributed systems appear in [2][5][9].

8.2 Challenge Problems and Requirements

Installation of QoS-Aware Middleware
Installing a distributed system in any given environment may require considerable effort to specialize the system for the new context, since different customers of such software typically have different QoS requirements that necessitate different configurations to achieve. This is particularly true for QoS-aware middleware, where multiple general-purpose and QoS-specific components must be deployed and instantiated to obtain the desired QoS support. This means that, given a distributed application and middleware context, the application, middleware, and QoS components must be installed, instantiated, and tuned—a tedious and error-prone task.

The requirements on QoS-aware middleware suggest a number of research challenges for the area of advanced separation of concerns. For example:

– How should QoS concerns be expressed?
– How should configurations be expressed? What kinds of concern specification and specialization formalisms should be used to keep specialization information "untangled" from the rest of a distributed system?
– What kinds of protocols must be defined between a configuration site and the distributed system so that the desired components can be created, installed, and tuned? How can this protocol express the required changes to the actual configuration after the system is installed?

Requirement 21 It must be possible to express QoS concerns *non-invasively*, and to specialize them for use in particular (distributed) contexts.

Requirement 22 It must be possible to express *software configurations* in terms of specializations on existing software, *non-invasively*.

Monitoring of QoS-Aware Components
To ensure that QoS requirements are being satisfied, it is necessary to collect various kinds of information from the components—e.g., timing statistics. The kinds of information that are needed are context-specific and depend on the particular QoS attribute being enforced. For example, in a transaction system, a QoS control system

may have to enforce requirements on the number of transactions per second that must be able to run. It must, therefore, gather information about the amount of data being operated on by transactions, the number of concurrent transactions executing on the transaction server, the network contention, and the priorities of transaction requests. On the other hand, a QoS control system that is enforcing a different QoS attribute, like availability, may require different kinds of information, such as up-time, last read and write times, number of users, etc.

Requirement 23 It must be possible to attach (*non-invasively*) QoS monitors on middleware components. Further, these monitors may be context-specific, which suggests that any solution to this requirement must also address Requirement 9.

Requirement 24 The process to collect data for QoS monitors from across the network must satisfy several requirements:

- The collection process must not create significant overhead.
- The collection process must operate within stated time constraints.
- Suitable time-stamp generation techniques must be adopted to overcome the lack of a global time.
- The effect of distance between the information sources (components) and the monitoring points must be taken into account in processing the data.

An example of this requirement is an on-line banking application, where the bank must offer mortgages within a given period of time. This process is realized through the interactions among multiple services that may reside in different locations. A QoS monitor must, therefore, gather information from different locations to permit the monitoring and control of mortgage offers.

Adaptation of Middleware

In a dynamically changing application context, and in case of multiple, distributed and sometimes conflicting executions, it may not be possible to fix the implementation of a service for a given QoS specification. In this case, the service may try to provide best effort to achieve the specified QoS specification.

For example, a transaction system may provide a fixed set of services such as begin, end and abort transaction. There are, however, many possible implementations of a transaction service, and each implementation may perform differently from others in different contexts. Similarly, it may be desired to select the implementation of a transport protocol among various alternatives based on the characteristics of the data to be transferred.

This observation implies Requirement 9, and it also imposes the following new requirement:

Requirement 25 It must be possible to dynamically adapt (*non-invasively*) the implementation of a middleware system so that the system provides the best effort within the context of agreed-upon QoS constraints.

9 Conclusions and Future Work

This workshop identified fifteen challenge problems and twenty-five requirements on advanced separation of concerns approaches. These problems and requirements are in need of further exploration and validation, and they are only a small subset of those that must ultimately be produced. Nonetheless, they represent a significant first step in defining boundaries of the emerging area of advanced separation of concerns, and in defining constraints on solution approaches.

It is our hope that researchers and technology providers will take these challenge problems and requirements in several directions:

– *Improvement of, and new directions for, existing approaches*: These challenge problems and requirements should be used to identify limitations in, and growth areas for, existing approaches to advanced separation of concerns. They may also help solution-providers to identify tradeoffs that they had to make in defining their approaches and to consider the overall impact of those tradeoffs.
– *Generation of new approaches*: While a number of advanced separation of concerns approaches already exist (e.g., compositors, logic meta-programming, etc.), these problems and requirements may help to suggest new approaches and technologies that help developers to use advanced separation of concerns more effectively throughout the software lifecycle.
– *Comparative analysis of approaches*: To date, there has been no consistent, comprehensive means of comparing different approaches and technologies. We hope that these problems and requirements provide a way for researchers and developers to classify their own approaches, and to compare and contrast them with other approaches. Towards this goal, we strongly urge technology providers to produce solutions to the challenge problems using their technologies, and to make those solutions available to the community.
– *Evolution of these problems and issues, and identification of new ones*: As noted earlier, a single, two-day workshop is not sufficient time to produce a complete set of requirements, issues, or challenge problems, or even to produce a representative set. The results reported in this document are preliminary. Further work will be required to identify incompleteness, inconsistencies, and other errors.

Above all else, it is our hope that these results will be discussed, revised and expanded by the advanced separation of concerns community, and that the community will identify new problems and issues in the future. We look forward to the individual research efforts, and to the successors of this inspiring workshop, that will so.

Acknowledgments

First and foremost, we would like to thank all of the workshop participants for their hard work before, during, and after the workshop. They are responsible for the stunning success of this workshop, and it is their efforts that made this report possible.

A few people merit special mention. We are grateful to the following participants for serving as scribes, contact persons, and coordinators for their groups after the workshop: Johan Brichau, Lee Carver, Constantinos Constantinides, Maja D'Hondt, Kris de Volder, Erik Ernst, Robert Filman, and Thomas Kühne. Several of these people also took primary responsibility for summarizing their groups' results, which greatly facilitated the writing of this report.

We would like to give special thanks to Mehmet Aksit and Oscar Nierstrasz for sharing their notes and recollections from the workshop, and to Harold Ossher for his very helpful comments on drafts of this paper.

Finally, we would like to thank Xerox PARC and IBM Research for their generous financial support for this workshop.

References

[1] Mehmet Aksit, Lodewijk Bergmans, and S. Vural. "An Object-Oriented Language-Database Integration Model: The Composition Filters Approach." Proceedings of ECOOP'92, Lecture Notes in Computer Science #615, 1992.

[2] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans and Akinori Yonezawa, "Abstracting Object Interactions Using Composition Filters", In Object Based Distributed Programming, R. Guerraoui, M. Riveill and O. Nierstasz (Eds.), Lecture Notes in Computer Science #791, 1993.

[3] Mehmet Aksit and Bedir Tekinerdogan. "Aspect-Oriented Programming using Composition Filters." In S. Demeyer and J. Bosch (Eds), *Object-Oriented Technology, ECOOP'98 Workshop Reader*. Springer-Verlag, 1998.

[4] Colin Atkinson and Thomas Kühne. "Separation of Concerns through Stratified Architectures." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[5] Lodewijk Bergmans, Aart van Halteren, Luis Ferreira Pires, Marten J. van Sinderen and Mehmet Aksit, "A QoS-Control Architecture for Object Middleware", Proceedings of IDMS'2000, Enschede, The Netherlands, in Lecture Notes in Computer Science #1905, Springer Verlag, 2000.

[6] Andrew P. Black and Jonathan Walpole. "Aspects of Information Flow" Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[7] Johan Brichau, Wolfgang De Meuter, and Kris de Volder. "Jumping Aspects." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[8] Laurent Bussard. "Towards a Pragmatic Composition Model of CORBA Services Based on AspectJ." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[9] Constantinos Constantinides, Atef Bader, and Tzilla Elrad. "Separation of Concerns in the Design of Concurrent Systems." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[10] Pascal Costanza. "Object Identity." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[11] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Reading, MA, June 2000.

[12] Krzysztof Czarnecki and Ulrich W. Eisenecker. "Separating the Configuration Aspect to Support Architecture Evolution." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[13] Erik Ernst. "Separation of Concerns and Then What?" Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[14] Robert Filman. "Applying Aspect-Oriented Programming to Intelligent Synthesis." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software." Addison-Wesley, 1994.

[16] William Harrison and Harold Ossher. "Subject-Oriented Programming (A Critique of Pure Objects)." In *Proceedings of OOPSLA'93*, September, 1993, pages 411-428.

[17] Barbara Liskov, Alan Snyder, Russell Atkinson, and J. Craig Schaffert. "Abstraction Mechanisms in CLU." Communications of the ACM, vol. 20, no. 8, August 1977.

[18] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. "Aspect-Oriented Programming." In *Lecture Notes in Computer Science (LNCS 1241)*. Springer-Verlag, June 1997.

[19] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. "Synchronization Constraints with Inheritance: What is Not Possible—So What Is?" Internal Report, Tokyo University, 1990.

[20] Satoshi Matsuoka and Akinori Yonezawa. "Analysis of Inheritance Anomaly in Concurrent Object-Oriented Languages." In *Research Directions in Concurrent Object-Oriented Programming*, G. Agha, P. Wegener, and A. Yonezawa (Editors), MIT Press, April 1993, pages 107-150.

[21] Harold Ossher and Peri Tarr. "Multi-Dimensional Separation of Concerns in Hyperspace." Workshop on Aspect-Oriented Programming at ECOOP'99 (position paper), Lisbon, Portugal, June 1999. (In *Lecture Notes in Computer Science (LNCS 1743)*. Springer-Verlag, 1999.)

[22] Harold Ossher and Peri Tarr. "Multi-Dimensional Separation of Concerns and the Hyperspace Approach." In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000. (To appear.)

[23] E. Pulvermüller, H. Klaeren, and A. Speck. "Aspects in Distributed Environments." In *Proceedings of the International Symposium on Generative and Component-Based Software Engineering (GCSE'99)*, Erfurt, Germany, September 1999.

[24] Bert Robben and Patrick Steyaert. "Aspects on TV." Workshop on Aspects and Dimensions of Concern at ECOOP'2000 (position paper), Cannes, France, June 2000.

[25] Subject-oriented programming and design patterns. Draft, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, see http://www.research.ibm.com/sop/sopcpats.htm.

[26] Stanley M. Sutton, Jr. “APPL/A: A Prototype Language for Software-Process Programming.” PhD thesis, University of Colorado at Boulder, Boulder, Colorado, August 1990.

[27] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. “*N* Degrees of Separation: Multi-Dimensional Separation of Concerns.” In *Proceedings of the International Conference on Software Engineering (ICSE’99)*, May 1999.

[28] Eddy Truyen, Bo N. Jørgensen, Wouter Joosen, Pierre Verbaeten, “Aspects for Run-Time Component Integration.” Workshop on Aspects and Dimensions of Concern at ECOOP’2000 (position paper), Cannes, France, June 2000.

[29] Nicholas Wirth. “Program Development by Stepwise Refinement.” Communications of the ACM, vol. 14, no. 4, April 1971, pp. 221-227.