PROVIDING MULTIDIMENSIONAL DECOMPOSITION IN OBJECT-ORIENTED ANALYSIS AND DESIGN

Constantinos Constantinides
School of Computer Science and Information Systems
Birkbeck, University of London
Malet Street, Bloomsbury
London WC1E 7HX, UK
cc@dcs.bbk.ac.uk

Therapon Skotiniotis
College of Computer and Information Science
Northeastern University
Cullinane Hall, 360 Huntington Avenue
Boston, Massachusetts 02115, USA
skotthe@ccs.neu.edu

Abstract

In this paper we argue that the explicit capture of crosscutting concerns in code should be the natural consequence of good and clean modularity in analysis and design, based on fine-grained (multidimensional) functional decomposition, and not the result of a corrective measure due to a tangled implementation. Aspect-Oriented Software Development (AOSD) is an emerging paradigm that builds on the foundations of established paradigms while providing additional principles to address their limitations in dealing with crosscutting concerns. The main theme of this paper is how to accommodate AOSD into existing software development processes by adapting or extending established object-oriented development techniques to an aspect-oriented context.

Key Words

Crosscutting concerns, aspect-oriented software development.

1. Background

The principle of *separation of concerns* [1] refers to the realization of system concepts into separate software units and it is a fundamental principle to software development. The associated benefits include better analysis and understanding of systems, readability of code, a high-level of design-level reuse, easy adaptability and good maintainability. Functional decomposition with stepwise refinement enables developers to break down a problem into manageable subproblems and address them relatively separately in order to provide designs that can be implemented in some programming language. To this end, different programming paradigms and languages provide mechanisms which support various kinds of functional decomposition. Despite the success of object-orientation in this effort, certain properties in object systems cannot be directly mapped from the problem domain to the solution space as a one-to-one mapping, and thus they cannot be localized in single modular units. This is because in object-orientation the requirements space is N-dimensional whereas the design and implementation space is one-dimensional as functional decomposition, through the modularization mechanisms provided by various languages, is performed along a single axis, namely the notion of a class. This *decomposition tyranny* [2] imposes two symptoms on software development: (1) code scattering, which refers to the fact that the implementation of certain concerns tends to cut across the decomposition hierarchy of the system, and (2) code tangling, which refers to the fact that a modular unit (class) may contain implementation elements (code) for various concerns. As a result, the benefits of the object-oriented paradigm cannot be fully utilized. Developers are, therefore, faced with the implications of crosscutting such as: (1) low cohesion of modular units resulting in a low level of comprehensibility of code, (2) strong coupling between modular units resulting in classes that are difficult to change and where changes in code are difficult to trace, (3) low level of reusability of code, (4) low level of system adaptability, and (5) programs that tend to be more error prone. As a result, the community has recognized that better linguistic separation mechanisms are needed. Aspect-Oriented Programming (AOP) [3] is a term adopted to describe an increasing number of technologies and approaches that support the explicit capture of crosscutting concerns (or aspects) whereby the implementation of functional components and crosscutting concerns is performed (relatively) separately, and their composition and coordination (referred to as weaving) is specified by a set of rules. Even though AOP is not bound to object systems, most of the current approaches involve extensions to current object-oriented languages that provide linguistic support for the explicit definition of crosscutting concerns, together with special compilers (weavers) that can combine them with components.

2. Problem and motivation

In order to produce a clean (tangled-free) implementation and achieve the maximum benefits of advanced separation of concerns, the analysis and design artifacts themselves (such as UML interaction diagrams and the class diagram) must in turn explicitly address crosscutting con-

This is a revised version of the paper that was presented at the IASTED SE 2004 conference. (Revision No. 2) cerns. We therefore need to provide the means to identify and model crosscutting concerns from the early stages of the software life cycle. Fine-grained (or multidimensional) decomposition implies that analysis and design artifacts are built with no dominance of components over crosscutting concerns. As a result, the explicit capture of crosscutting concerns in code would be the natural consequence of good and clean modularity and not the result of a corrective measure due to a tangled implementation. To this end, Aspect-Oriented Software Development (AOSD) has extended AOP to model and document crosscutting concerns properly, and make them traceable throughout software development. Naturally, this does not guarantee that multidimensional decomposition can never pose some level of crosscutting, especially as requirements might change during development and new concerns (including some that may be crosscutting) may show up during implementation. However, with the aid of an iterative software development process, developers should be able to control changing requirements and produce quality software.

The motivation behind this work is to show how AOSD can be utilized throughout the different phases of development, and how it can prove successful. In this paper we will present a case study to investigate how crosscutting requirements can be captured, modeled and handled as well as to what degree they remain visible and can propagate throughout development. We will also investigate the nature of mutually dependent crosscutting concerns. The main theme of this study is to investigate to what degree developers can adapt established analysis and design techniques for object-oriented systems to an aspect-oriented context, as well as what extensions might be required.

3. Case study: A conference room reservation system

In this section we will consider a case study of a conference room reservation system which would be deployable in facilities like a college campus or an industrial complex. Multiple clients access the system in order to make a reservation of one (or more) room in the facility. The system should maintain a book with all reservations made. Clients may also cancel existing reservations. Furthermore, clients may view reservations, or view the entire calendar. We can recognize this setting to be an implementation of the readers-writers concurrency protocol. We can, therefore, regard the operations of making and canceling reservations as write operations, while we can regard the operations of viewing reservations and viewing the calendar as read operations (we will refer to them as such in order to improve clarity where applicable). To maintain data integrity, we require that a writer client would operate with self- and mutual exclusion, whereas readers would operate with mutual exclusion only. For this case study we would like the system to give priority to write operations over read operations.

3.1 Requirements analysis

Requirements constitute system descriptions, or system capabilities or features that systems must have as well as constraints that systems must satisfy in order to be accepted by stakeholders. In the literature, requirements are widely placed in two groups, namely functional and non-functional. Functional requirements refer to the functionality or services that the system is expected to provide, describing interactions between the system and its environment, while not focusing on implementation details. Functional requirements are observable during software execution [4, 5]. On the other hand, the term non-functional requirements (NFRs) is a collective term adopted to describe aspects of the system (quality attributes and constraints) that are not directly related to the functional behavior of the system [6]. NFRs normally relate to the system as a whole (or to parts of it) rather than to individual system features. As such NFRs seem to be major candidates for crosscutting concerns. Requirements elicitation is a process that provides a specification that stakeholders can understand, and requirements analysis provides a model that developers can unambiguously interpret [6] (expressed in formal or semi-formal notation). In [4] the author provides a separation of terms by adopting the term "user requirements" to refer to high-level abstract descriptions, and "system requirements", that precisely describes system services and constraints. Scenarios (and use cases) tend to bridge this gap [6]. A scenario describes a series of interactions between an actor and the system, and a use case describes a class of scenarios. Use cases are generally used to capture functional requirements [7]. In scenario-based elicitation, developers elicit requirements by observing and interviewing users. A use case diagram for the current case study is shown in Figure 1, and the descriptions of use cases[1] Make Reservation and View Reservations are shown in Figures 2, and 3 respectively.

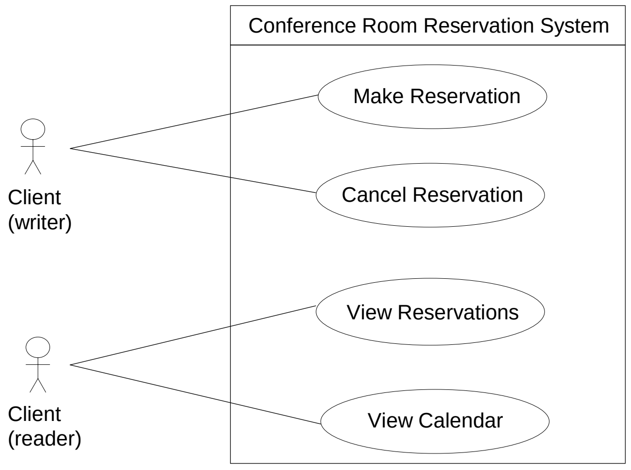

Figure 1. Use case diagram

[1] Only success scenarios are considered in this example.

Use case Cancel Reservation (not shown here) is similar to Make Reservation and use case View Calendar will not be implemented in this example. In order to capture the specification of non-functional requirements, we have adopted a variant of the template proposed in [8]. The specifications for synchronization and scheduling illustrate that both requirements pose a crosscutting nature over all use cases (Figure 4), indicated by the 'where' row.

Use Case UC1	Make Reservation
Primary Actor	Client (writer)
Stakeholders and interests	Staff who want to make reservations.
Preconditions	None
Postconditions	A room was reserved on a time slot.
Main success scenario	A client reserves a room for a given room number and time slot. The system responds with a confirmation and a reservation number. Optionally a client may make subsequent reservations.

Figure 2. Use case Make Reservation

Use Case UC3	View Reservations
Primary Actor	Client (reader)
Stakeholders and interests	Staff who want to view reservations.
Preconditions	None
Postconditions	None.
Main success scenario	A client requests to view all reservations. The system responds with a list of reservations.

Figure 3. Use case View Reservations

Name	Synchronization	Scheduling
Description	Specifies the eligibility of an entity for execution.	Specifies what must be done next among eligible entities.
Decomposition	None	None
Where	Actors: All Use Cases: All	Actors: All Use Cases: All

Figure 4. Specifications for synchronization and scheduling

3.2 Analysis

During analysis, it is useful to investigate and define the behavior of the software as a black box, that is to provide a description of what services the system should provide. Use cases describe how external actors interact with the software system. During this interaction, an actor generates events that initiate operations upon the system. A system sequence diagram (SSD) is a picture that shows, for a particular scenario of a use case, the events that external actors generate and the temporal order of interaction with the system [5]. An SSD treats a system as a black box, placing emphasis on events that cross the system boundary from actors to the system. The set of all required system operations within a SSD is determined by identifying the system events. The SSDs for Make Reservation and View Reservations are shown in Figures 5, and 6 respectively.

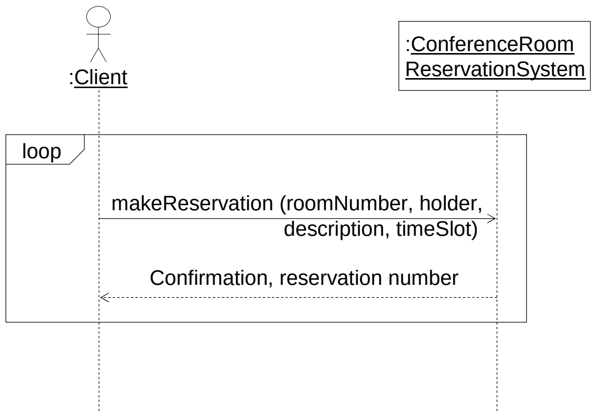
:Client
:ConferenceRoom ReservationSystem
loop
makeReservation (roomNumber, holder, description, timeSlot)
Confirmation, reservation number

Figure 5. System sequence diagram for Make Reservation

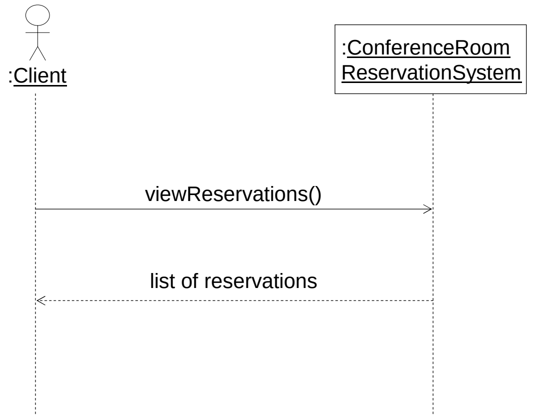

Figure 6. System sequence diagram for View Reservations

Object-oriented analysis investigates the problem domain and its requirements, by placing emphasis on finding and describing the concepts of the problem domain where the dimension of decomposition is by domain entities. A domain model is a static structure diagram that illustrates (real-world) concepts in the problem domain. In [5] the author discusses two mechanisms by which developers can identify conceptual classes: (1) linguistic analysis (noun-phrase identification) based on the scenarios of the use cases, and (2) a candidate conceptual class category list. From a list of candidate conceptual classes, we can identify synchronization and scheduling (as abstract noun concepts) and their respective policies (as rules) that are inherent in both use cases, even though they are not visible to the actor as they do not constitute functional requirements.

While development shifts from requirements to analysis, it is perhaps vital to distinguish between requirements and concerns. Requirements refer to the views of the stakeholders, whereas concerns are informally defined as particular matters of interest in a software system, and they are introduced or occur throughout the software life cycle. As such they refer to the views of developers only. Stakeholders may have no interest in them, as they merely want their requirements implemented. Building a domain model implies the identification of initial system concerns from a set of requirements (use cases). In the domain model (Figure 7), both synchronization and scheduling are expressed as conceptual classes which define policies associated with conceptual class `ReservationsBook`.

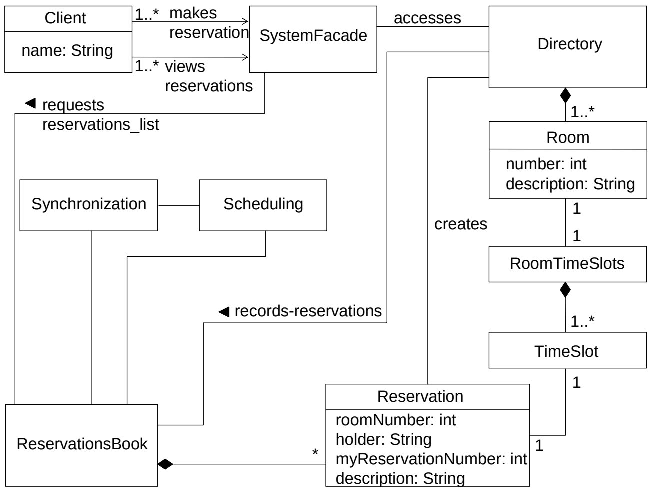

Figure 7. Domain model

The set of system operations in an SSD constitutes a subset of the overall behavior of the system, as the entire set of system operations (across all use cases) defines the complete public interface of the system. In order to investigate the system behavior with respect to a system operation, we can build an operation contract. Operation contracts emphasize what task(s) an operation must perform. We would generally choose to build operation contracts for those system operations that pose complex behavior (especially those that include complex preconditions and postconditions). In this example, we will create operation contracts for the operations `makeReservation` and `viewReservations`. In both operation contracts, the system must initially determine the eligibility of a client for using the service (addressed by the synchronization aspect) before establishing what should be the race condition between eligible clients (addressed by the scheduling aspect). An obvious precondition for `makeReservation` is that the room to be reserved must be available on the requested time slot. Once an instance of `Reservation` is created, it must be attached (written) to `ReservationsBook`. The request to attach `Reservation` to `ReservationsBook` must wait if any client (writer or reader) is already active. The request will be given eligibility to proceed in order to resolve race conditions if there are no waiting writers. Otherwise, if there are any writers who are currently waiting, then the request must wait. The set of postconditions refers to the creation and initialization of an instance of class `Reservation` and the formation of an association with an instance of `ReservationsBook`. If there are waiting clients, the system notifies them by giving priority to waiting writers over waiting readers before decrementing the count of active writers.

Operation	makeReservation (roomNumber: int, holder: String, description: String, timeSlot: TimeSlot)
Cross-references	UC1: Make Reservation

Pre-conditions	// create instance of Reservation
	Room is available on requested time slot.
	// place Reservation in ReservationsBook
	// synchronization
	if (no active writers)
	    and (no active readers) {
	        if (waiting writers)
	          then {
	              (increment waiting writers);
	              wait; }
	          else (increment active writers);
	else { // some client is active
	    (increment waiting writers); wait; }
	// scheduling
	if (waiting writers) **then** {
	    (increment waiting writers); wait;}

Post-conditions	A Reservation instance r was created.
	Attributes of r were initialized.
	r is associated with ReservationsBook.
	// scheduling
	if (waiting writers) **then** notify them
	else if (waiting readers) **then** notify them;
	// synchronization
	(decrement active writers);

Figure 8. Operation contract for makeReservation

The set of preconditions for `viewReservations` imposes the following constraints: If there are any active or waiting writers, then the request must wait. Otherwise, the system will check to see if there are any readers already waiting in which case it will notify them before allowing the request to proceed. There are no scheduling constraints imposed before viewing reservations, as multiple readers may concurrently access the system. The set of postconditions would check to see if there are any waiting writers and no active readers in which case it will give priority to (waiting) writers by notifying them, before decrementing the count of active readers.

3.3 Design

Object-oriented design provides a conceptual solution by defining software objects and how they collaborate in order to fulfill the requirements. The next step during development would be to create an interaction diagram for each operation contract. Interaction diagrams illustrate how objects interact via messages. The communication diagrams for the operation contracts `makeReservation` and `viewReservations` (shown in Figures 10, and 11 respectively) illustrate the integration of functional and non-functional requirements as both synchronization and scheduling are modeled as objects with before and after behavior which corresponds to the precondition and post-condition semantics of the operation contracts.

Operation	viewReservations()
Cross-references	UC3: View Reservations

Pre-conditions	// synchronization
	if (no active writers) **and**
	(no waiting writers) **then** {
	if (waiting readers) **then**
	(notify waiting readers);
	(increment active readers); }
	else { (increment waiting readers);
	wait;}

Post-conditions	// scheduling
	if (waiting writers) **and**
	(no active readers) **then**
	(notify waiting writers);
	// synchronization
	(decrement active readers);

Figure 9. Operation contract for viewReservations

The order in which synchronization and scheduling constraints are evaluated brings up the notion of mutual dependency of crosscutting concerns. In general, it is highly unlikely that crosscutting concerns are completely independent (orthogonal) of each other. The issue of non-orthogonality is important, as it addresses the correct semantics of the underlying protocol that the system must implement. The interaction (conflict) between synchronization and scheduling in this example falls under two types identified in [9]: (1) conditional execution, where the applicability of one aspect is dependent on the other, and (2) aspect ordering, as both aspects influence the same joinpoints (operation calls).

Once the interaction diagrams have been completed it is possible to identify the specification for the software classes and interfaces and illustrate them in a class diagram (Figure 12). A class diagram depends upon the domain model and the interaction diagrams. Furthermore, a design should ideally be language (implementation) independent.

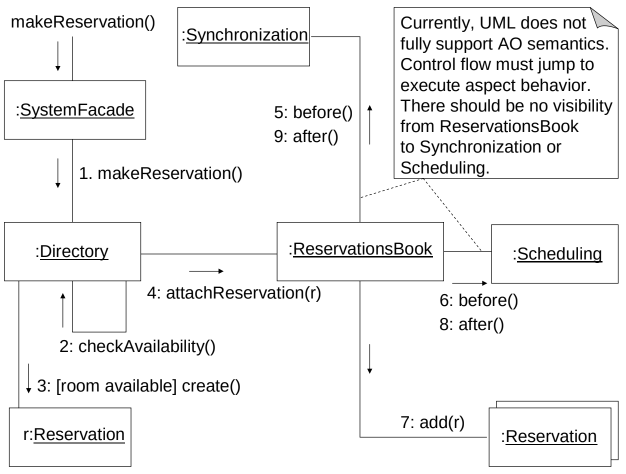

Figure 10. Partial communication diagram for makeReservation

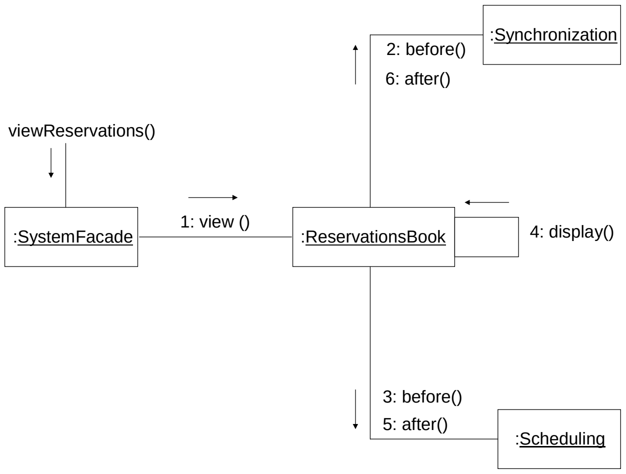

Figure 11. Communication diagram for viewReservations

3.4 Implementation

We used AspectJ (Version 1.0.6) to implement the system. AspectJ [10] is an aspect-oriented extension to Java, providing constructs to express crosscutting concerns (some familiarity is assumed). The definitions of classes `Directory` and `ReservationsBook` are shown in code Listings 1, and 2 respectively. An abstract aspect `Mutex` (not shown here), which is a superaspect to both synchronization and scheduling, defines the explicit pointcuts that capture calls to methods upon which crosscutting behavior is to be introduced as advice. The superaspect also defines the static variables that synchronization and scheduling definitions use in order to exchange information. These two aspects are non-orthogonal, since data is shared between them and also their order of execution has to preserve the readers-writers protocol. The definitions of aspects synchronization and scheduling are shown in Listings 3, and 4 respectively.

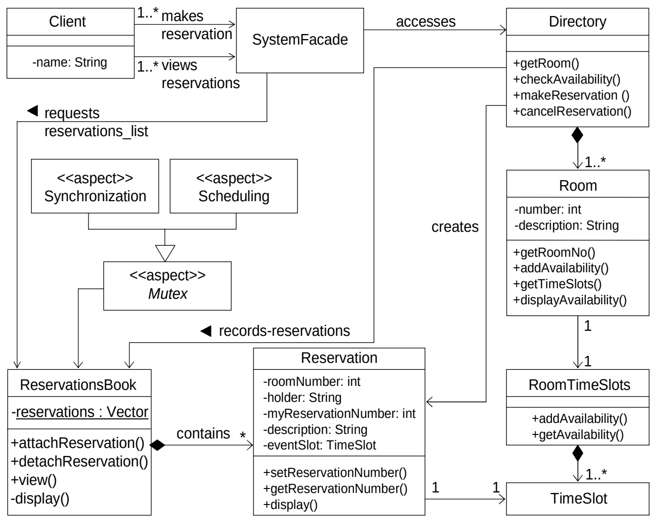

Figure 12. Class diagram

```
public class Directory {
  private ReservationsBook book;
  private LinkedList allRooms = new LinkedList();
  Directory(ReservationsBook book) {
    this.book = book;
  }
  public Room getRoom(int no){...}
  public boolean checkAvailability(LinkedList
  available, TimeSlot event){...}
  public int makeReservation (int roomNumber,
  String holder, String description, TimeSlot
  eventsTime) {
    Iterator roomIT = allRooms.iterator();
    Room rm = getRoom(roomNumber);
    if (rm != null){
      if (checkAvailability(rm.getTimeSlots(),
      eventsTime)){
        int rNum = book.attachReservation(new Res-
        ervation (roomNumber, holder, descrip-
        tion,eventsTime));
      System.out.println("Reservation created.
      Your reservation Num is :" + rNum);
        return rNum;
      }else {
        //Signal an Error “not Available”
        return -1;}
    }else { //Error “no such room”
      return -1;}}
  public void cancelReservation (int resNum) {
book.detachReservation(resNum);}}
```

Listing 1. Definition of class Directory

Since both synchronization and scheduling apply to exactly the same joinpoints, AspectJ allows for ordering of aspects through the usage of `dominates` in their definition. Using AspectJ’s `around` advice we are able to have full control as to when the actual method is to be called through the `proceed()` call. This way we can dictate both which client is allowed and when a client will be allowed to continue and actually execute the code in `ReservationsBook`. Code placed before the calls to `proceed()` corresponds to the set of preconditions that has to be enforced by the specific aspect on the specific type of operation. Code that is placed after the call to `proceed()` refers to the set of postconditions that the aspect has to verify for the specific type of operation. Pointcuts are named according to the type of operations that we would like to advice (i.e. write and read) capturing the same terminology as the one used during design. Refactoring of the advice could provide an even more elegant solution. However, we cannot replace the whole advice by calls to methods inside an aspect as AspectJ does not support `proceed()` calls outside of advice blocks.

```
public class ReservationsBook {
  private static Vector reservations;
  private static int reservationsNumber = 0;
  public static int getNextReservationNumber(){
    //gen fresh number and return }
  public int attachReservation (Reservation r) {
    int rNum = Reservations-
    Book.getNextReservationNumber();
    r.setReservationNumber(rNum);
    reservations.add(r);
    this.display();
    return rNum;}
  public void detachReservation (int resNum) {
    Iterator reservationsIT = reserva-
    tions.iterator();
    while(reservationsIT.hasNext()){
    Reservation res = (Reservation) reservation-
    sIT.next();
    int tempResNum = res.getReservationNumber();
    if (tempResNum == resNum){
      if(reservations.removeElement(res)){
      System.out.println("Reservation No: "+
      resNum+" has been removed");
      break;
      }else {
        //signal an error and break
      }
      }else if (!reservationsIT.hasNext()){
        System.out.println("[ERROR]”);
        }else
          // Do nothing
    this.view();}
  public void view () {
     this.display();
  }
  private void display () {
    //call display() on each vector element
 }}
```

Listing 2. Definition of class ReservationsBook

4. Discussion

We believe that AOSD can help achieve development-time qualities [11] such as adaptability, composability and reusability which provide business value, as well as run-time qualities such as correctness and reliability. In the subsequent subsections we will discuss the above issues together with certain shortcomings of AOSD.

4.1 Reusability and adaptability

As software products need to satisfy both technical and non-technical criteria, developers find it essential to combine theory and experience in order to reuse proven designs. The importance of reuse lies on the fact that it can speed up the development process, cut down costs, increase productivity and improve the quality of software. Design-level reuse is viewed as the attempt to share certain aspects of an approach across various projects.

```
aspect Synchronization extends Mutex
                                 dominates Scheduling{
pointcut writeOps(Reservation r):
  (makingAReservation(r)) ||
  (makingACancellation(r));
pointcut readOps(): tryingToView();
void around():
  readOps(){
    if (Mutex.waitingWriters.isEmpty()
       && Mutex.activeW == 0){
      Mutex.activeR += 1;
      proceed();
      Mutex.activeR -= 1;
    } else {
      Integer temp = new Integer(1);
      Mutex.waitingReaders.add(temp);
      try {
      Mutex.waitingReaders.wait();
      Integer dummy =
  (Integer) Mutex.waitingReaders.firstElement();
      waitingReaders.remove(dummy);
      Mutex.activeR += 1;
      proceed();
      Mutex.activeR -= 1;
      }catch (Exception ex){
       ex.printStackTrace();
      }}}
int around(Reservation r):
  makingAReservation(r){
   int temp = -1;
   if (Mutex.activeW == 0
       && Mutex.activeR ==0 ){
     if (Mutex.waitingWriters.isEmpty()){
       Mutex.activeW += 1;
       temp = proceed(r);
       Mutex.activeW -=1;
       return temp;
     } else {
       Mutex.waitingWriters.add(r);
       try{
        Mutex.waitingWriters.wait();
        Reservation res = (Reservation)
        Mutex.waitingWriters.firstElement();
        Mutex.waitingWriters.remove(res);
        Mutex.activeW += 1;
        temp = proceed(res);
        Mutex.activeW -= 1;
       } catch (Exception ex){
          ex.printStackTrace();}
       return temp;
       }
     } else {
       Mutex.waitingWriters.add(r);
       try{
         Mutex.waitingWriters.wait();
         Reservation res = (Reservation)
         Mutex.waitingWriters.firstElement();
         Mutex.waitingWriters.remove(res);
         Mutex.activeW += 1;
         temp = proceed(res);
         Mutex.activeW -= 1;
       } catch (Exception ex){
         ex.printStackTrace();}
       return temp;
     }}}
```

Listing 3. Definition of synchronization aspect

Object and component-based systems offer a wide spectrum of techniques to reuse designs on different levels, ranging from frameworks and patterns that constitute approaches on how to best program “in-the-large” to libraries and programming languages that constitute approaches on how to best program “in-the-small” [12]. On the higher level of the reuse spectrum in the context of AOSD, aspect-oriented frameworks have been discussed in the literature [13], and object-oriented design patterns have been migrated to aspect systems [14]. Furthermore, new design patterns and idioms have been demonstrated [15].

On the lower level of the reuse spectrum, the importance of the mechanism of inheritance and the contribution of Object-Oriented Programming (OOP) towards reusability has been extensively discussed in the literature.

```
aspect Scheduling extends Mutex{
  boolean nextWriter = false;
  pointcut writeOps(Reservation r):
    (makingAReservation(r)) ||
    (makingACancellation(r));
  pointcut readOps(): tryingToView();
  int around(Reservation r):
   makingAReservation(r){
     int temp = -1;
     if (Mutex.waitingWriters.isEmpty()){
      temp = proceed(r);
       if (!Mutex.waitingWriters.isEmpty()){
         Mutex.waitingWriters.notify();
       }else if (!Mutex.waitingReaders.isEmpty())
         Mutex.waitingReaders.notify();
         return temp;
       } else {
         try {
         Mutex.waitingWriters.wait();
         temp = proceed(r);
         if (!Mutex.waitingWriters.isEmpty()){
            Mutex.waitingWriters.notify();
         }else if
               (!Mutex.waitingReaders.isEmpty())
            Mutex.waitingReaders.notify();
         }catch (Exception ex){
            ex.printStackTrace();
         }
       return temp;
   }
  }
  void around():
   readOps(){
    proceed();
     if (!Mutex.waitingWriters.isEmpty() &&
         Mutex.activeR == 0)
      Mutex.waitingWriters.notify();
  }
}
```

Listing 4. Definition of scheduling aspect

Issues of reuse also arise in aspect-oriented languages. Even though the adoption of AOP results in a good separation of concerns, restricting aspect definitions to match class and method names of system core concerns leads to a strong binding between aspects and system core concerns. In such cases aspect definitions are not reusable, but they are restricted to be only applicable in one specific application context. Aspect-Oriented Programming has resulted in the provision of a higher level of functional code reuse than the one supported by OOP, as classes are not tangled and they tend to maintain a high degree of cohesion. However, this is not always the case with aspect definitions as adaptability of non-orthogonal aspects highly increases their dependencies. In [16] it was argued that aspect definitions in AspectJ may pose various visibility types based on the constructs the language has introduced in conjunction to all types of visibility relationships already known in OOP. More specifically, visibility over components can be posed in AspectJ by pointcuts, introductions and advice. It should be noted that even though we use AspectJ as an example language in our discussion about visibility and coupling issues between aspects and system core concerns, it is important to stress that these issues are not exclusive to the AspectJ language, but they can be found in other general-purpose aspect-oriented languages, at least the ones whose design dimensions tend to be along the lines of AspectJ.

4.2 AOSD and the "SMART" requirements model

Aspect-Oriented Software Development can help capture the "SMART" [11] characteristics (specific, measurable, attainable, realizable, traceable) of crosscutting non-functional requirements. (1) Specific: Separation of concerns allows for crosscutting concerns to be stated unambiguously and reasoned in isolation, (2) Measurable: The clean mapping from the requirements space to the solution space allows for verification of concerns, (3) Attainable: Linguistic constructs allow for an explicit capture of crosscutting concerns, (4) Realizable: Concerns can be captured through a number of available technologies, and (5) Traceable: Identifying crosscutting concerns from early stages helps in traceability from requirements to implementation. Concerns can be managed in (relative) isolation though the entire life cycle. A requirements specification is traceable if each requirement can be traced throughout development to its corresponding system function (and vice versa). Traceability also includes the ability to track dependencies among requirements, system functions and design artifacts (classes, objects, etc.). Traceability is critical for developing tests. When developing tests, traceability enables assessing the coverage of a test case, i.e. to identify which requirements are tested and which are not. Furthermore, traceability assists in software correctness, as a clear distinction between requirements and code is attainted providing a more convincing argument that the end product has addressed the requirements of stakeholders.

4.3 Validation and verification

Validation is a process whereby we ensure that the software meets the expectations of stakeholders [4]. Users validate system description by reviewing scenarios. Requirements validation involves checking if the specification is correct, complete, consistent, unambiguous and realistic [6]. Requirements verification is checking that the software meets the specification. The specification is verifiable if, once the system is built, a repeatable test can be designed to demonstrate that the system fulfils the requirements. With AOSD, there is a clean and distinct mapping of design entities to implementation components. As a result, verification of requirements becomes straightforward.

4.4 Obliviousness and quantification

Even though the AOSD community does not yet completely agree what exactly constitutes aspect-oriented software (as opposed to the object community, which has provided solid definitions for OOP), one of the dominant views suggests that AOP can be characterized by two essential properties: obliviousness and quantification [17]. The first allows the system core to be clean of any representation or even any reference to potential aspectual properties. The second provides expressions that direct the insertions of desired aspectual properties code into all the right places at the right time.

Obliviousness brings up the notion of visibility between components and aspects. Crosscutting concerns are materialized as first-class abstractions during design, and modeled in a similar fashion in classes while illustrating a visibility relationship to all components they cut across. Visibility implies dependency and consequently coupling between components (aspects and system core concerns). In general, coupling can be viewed as restricting the scope of application of the components to their direct environment. In particular, strong coupling decreases component reusability. In [5] the author defines visibility as the ability of one object to see or have a reference to another object. There are four ways that visibility can be achieved from object A to object B: (1) attribute visibility (2) parameter visibility, (3) locally declared visibility, and (4) global visibility. In this case study, the development of components has been carried out independently of the existence of aspects. The converse, however, is not true as crosscutting concerns depend on the components whose semantics and performance they affect. In general we can say that with AOSD, design level reuse is increased on the high-level spectrum. However, reuse on the low-level spectrum (implementation of crosscutting concerns), through the deployment of current AOSD technologies, still remains low. As AOP language research continues, issues of reuse of aspect definitions are being explored through language mechanisms to support obliviousness and quantification.

4.5 AOSD and iterative development

To accommodate the fact that stakeholders usually have changing requirements, an iterative development process such as the one defined by the Unified Software Development Process (UP) [18] embraces change by being organized into a series of short fixed-length mini-projects called iterations, where each iteration represents a complete development cycle. Each iteration involves tackling new requirements (by choosing a small subset of the requirements, or perhaps revisiting previously addressed requirements for improvement) and it includes its own requirements engineering (elicitation and analysis), analysis, design, implementation and testing activities. As a result, the system grows incrementally over time, iteration by iteration. Initial iterations will deal with high-risk and high-value requirements. The outcome of each iteration is a tested, integrated, and executable system. This leads to rapid feedback, and provides an opportunity to modify or adapt understanding of the requirements or design. The system may not be eligible for production deployment until after several iterations.

Aspect-Oriented Software Development would align well with iterative software engineering processes, as crosscutting concerns can be identified and deployed at any stage of software development. In a linear process (such as the waterfall model), the identification of a crosscutting concern will require reengineering or refactoring of code which in turn would require the propagation of change throughout all existing design and analysis artifacts. The adoption of an iterative process, on the other hand, will allow incorporation of the new concern at the next iteration cycle.

4.6 Comprehensibility

Crosscutting tends to reduce the comprehensibility of modules. Comprehension becomes a problem in large programs, particularly those that are developed by a team or over a series of iterations or releases. To this effect, AOSD (and AOP) can constitute a divide-and-conquer strategy as modules can be managed in (relative) isolation.

A disadvantage of AOSD/AOP is that business logic is not well localized, as advice from several aspect definitions may affect the same joinpoints as in this case study. With AspectJ, composition rules are scattered in aspect definitions. Furthermore, if a set of aspects *{a[i]}* affect a component *c*, there is no trace from *c* to *{a[i]}*. The problem of non-locality of business logic is also apparent in cases where there is an interaction (interference) between aspects. As a result, the complete behavior of a module can only be understood with respect to its corresponding set of introductions and advice increasing the complexity of the module in terms of reasoning about its behavior.

Aspect interference together with unanticipated evolution may cause a component to pose unexpected behavior, particularly once an aspect is introduced independently and it is unaware of future aspects. However, no aspect introduction should be allowed to break the semantics of the system. Extensions to the system materialized as aspects become hard to reason about since a global understanding of the application and its semantics is required before design and deployment of the new aspect definition.

Due to the relative immaturity of AOSD as a paradigm (including the tools that are currently available) there are certain limitations to the benefits which AOSD aims to provide. Specifically the low level exposure of details in pointcuts renders certain aspects unusable in any system other than the one they were originally designed for. This form of coupling decreases reuse of aspects across systems. Furthermore the usage of `dominates` to denote the ordering of advice at joinpoints where more than one aspect applies is limiting. In situations where multiple aspects apply to the same joinpoint and many different combinations of executions of these aspects are to be maintained, the `dominates` approach becomes too low level and inflexible to express such complex relationships. It is easy to see how crosscutting concerns could easily blend in with an iterative approach to software development. However, this is not always the case. As more new crosscutting concerns are introduced in each iteration, one has to first deal with the interaction between a newly introduced concern with the current class hierarchy, and in the cases of non-orthogonality of crosscutting concerns one must deal with all cases of mutual dependencies. The difficulty arises when one no longer has to reason only about inheritance and type conformance of new calls, but when one also has to reason about the program’s control flow since it is altered and used by crosscutting concerns. This issue becomes complicated and error prone even with the assistance of some of the currently available tools that are provided in order to aid AOSD developers.

5. Conclusion

Aspect-Oriented Software Development calls for a more systematic treatment of concerns from the initial stages of development. The goal of AOSD is to provide approaches for the identification, separation, representation and composition of crosscutting concerns. We need to model and document crosscutting concerns properly and make them traceable.

In this paper we presented a case study to investigate the identification and nature of crosscutting concerns throughout the development process, while trying to adapt established analysis and design techniques for object-oriented systems to an aspect-oriented context. We have argued that developers should make the transition from functional to fine-grained decomposition. This will result in a multidimensional decomposition, where each concern is viewed equally and posing as a decomposition axis throughout the entire development process. This implies that analysis and design artifacts should pose no dominance of components over aspects. As a result, the explicit capture of crosscutting concerns in code should be the natural consequence of good and clean modularity and not the result of a corrective measure due to a tangled implementation.

Crosscutting concerns tend to appear during requirements analysis, and they may take different forms during development. During the requirements analysis stage, needs and expectations of stakeholders will be captured in use cases. Even though non-functional crosscutting requirements like synchronization and scheduling were not explicitly captured in the narrative description of use cases, they were described in a requirements specification template, identifying the use cases they cut across. During analysis, crosscutting concerns were modeled as conceptual classes. A more detailed description of synchronization and scheduling initially appeared in the form of preconditions and postconditions in the operation contracts. During design, crosscutting concerns were modeled as first-class abstractions in interaction diagrams. First-class realization of aspects can aid in the reusability and adaptability of software. Furthermore, multidimensional decomposition and the explicit capture of crosscutting concerns throughout all phases of development can also enable developers to trace crosscutting concerns from requirements to code artifacts.

Crosscutting of concerns seems to exist from the early stages of a development process and it is a natural effect of any complex domain. The ability to: clearly understand, explicitly address and model crosscutting behavior is the essential ingredient in order to provide for a better, cleaner design leading to a higher level of reuse and system adaptability. At the same time, modeling of crosscutting concerns is limited by the lack of standard UML support. Furthermore, implementation of crosscutting concerns is confined within the provisions of current technologies (languages).

References

[1] D. L. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM, 15*(12), 1972, 1053-1058.

[2] P. Tarr, H. Ossher, W. Harrison, S. M. Sutton Jr., N degrees of separation: Multi-dimensional separation of concerns, *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, USA, May 16-22, 1999, pp. 107-119.

[3] G. Kiczales, J. Lamping , A. Mendhekar, C. Maeda, C. V. Lopes, J-M. Loingtier, and J. Irving, Aspect-oriented programming, *Proceedings of the 11th European Conference on Object-Oriented Programming*, Jyväskylä, Finland, June 9-13, 1997, pp. 220-242.

[4] I. Sommerville, *Software engineering; Sixth edition*, (Essex: Addison-Wesley, 2001).

[5] C. Larman, *Applying UML and patterns; An introduction to object-oriented analysis and design and the Unified Process; Second edition,* (Upper Saddle River, NJ: Prentice Hall Inc., 2002).

[6] B. Bruegge, and A. H. Dutoit, *Object-oriented software engineering; Conquering complex and changing systems* (Prentice Hall, 2000).

[7] I. Jacobson, *Object-oriented software engineering: A use case driven approach* (Addison-Wesley, 1992).

[8] L. Chung, B. Nixon, E. Yu and J. Mylopoulos, *Non-functional requirements in software engineering* (Boston: Kluwer Academic Publishers, 2000).

[9] J. Hannemann, R. Chitchyan, and A. Rashid, ECOOP 2003 analysis of aspect-oriented software workshop report, *ECOOP 2003 Workshop Reader* (to be published by Springer-Verlag).

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, An overview of AspectJ, *Proceedings of the 15th European Conference on Object-Oriented Programming*, Budapest, Hungary, June 18-22, 2001, pp. 327-355.

[11] R. Malan, and D. Bredemeyer, Defining non-functional requirements, Bredemeyer Consulting white paper, 2001. URL: `www.bredemeyer.com`

[12] C. Szyperski, *Component software; Beyond object-oriented programming; Second edition* (Addison-Wesley, 2002).

[13] C. A. Constantinides, T. Elrad and M. Fayad, Extending the object model to provide explicit support for crosscutting concerns, *Journal of Software Practice and Experience, 32*(7), 2002, 703-734.

[14] J. Hannemann and G. Kiczales, Design pattern implementation in Java and AspectJ, *Proceedings of the 17th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications,* Seattle, WA, USA, November 4-8, 2002, pp. 161-173.

[15] S. Hanenberg and P. Costanza, Connecting aspects in AspectJ: Strategies vs. patterns, *Proceedings of AOSD 2002 Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Enschede, The Netherlands, April 23, 2002.

[16] Y. Hassoun and C. A. Constantinides, Considerations on component visibility and code reusability in AspectJ, *Proceedings of 3rd Workshop on Aspect-Oriented Software Development*, Essen, Germany, March 4-5, 2003.

[17] R. E. Filman, and D. P. Friedman, Aspect-oriented programming is quantification and obliviousness, *Proceedings of the OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Minneapolis, MN, USA, October 16, 2000.

[18] I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process* (Addison-Wesley, 1999).