Crosscutting Interfaces for Aspect-Oriented Modeling*

Christina Chavez[1], Alessandro Garcia[2], Uirá Kulesza[3], Cláudio Sant'Anna[3] and Carlos Lucena[3]

[1] Depto de Ciência da Computação, UFBA
Av. Adhemar de Barros, s/n
Salvador – Brasil
flach@dcc.ufba.br

[2] Computing Department, Lancaster University
South Drive, InfoLab 21, LA1 4WA, Lancaster – UK
garciaa@comp.lancs.ac.uk

[3] Depto de Ciência da Computação, PUC-Rio
R. Marquês de São Vicente, 225
Rio de Janeiro – Brasil
{uira,claudios,lucena}@inf.puc-rio.br

Abstract

Aspect-oriented software development promotes improved separation of concerns by introducing a new modular unit, called aspect, for the modularization of crosscutting concerns. As a new kind of modular unit, aspects should have explicit interfaces that describe the way they interact with the rest of the system and how they affect other modules. This interaction can be homogeneous or heterogeneous. In this paper, we present crosscutting interfaces as a conceptual tool for dealing with the complexity of heterogeneous aspects at the design level. Crosscutting interfaces have been incorporated by the aSideML modeling language in order to enhance aspect description at the design level. Moreover, we present a modeling notation for the description of architecture-level aspects that also supports the explicit representation of crosscutting interfaces. Finally, we present two large-scale case studies we have performed using this modeling language that support our arguments in favor of crosscutting interfaces.

Keywords: Aspect-oriented modeling, crosscutting interfaces, heterogeneous aspects, software design, software architecture, separation of concerns.

1. INTRODUCTION

There is an increasing level of complexity of software systems and the kinds of *concerns* they address, imposing new challenges to the mainstream software engineering paradigms. The object-oriented paradigm is not sufficient to modularize some common concerns found in most complex systems. They have been called *crosscutting concerns* because they naturally cut across the boundaries of other concerns [29, 40]. *Aspect-Oriented Software Development* (AOSD) [1, 15] is an emerging approach with the goal of improving the separation of crosscutting concerns throughout the software development lifecycle. AOSD considers acknowledged contributions to separation of concerns and modularity provided by previous technologies (mainly the object-oriented paradigm, but not constrained to it), while introducing a new modular unit, called *aspect*, for the modularization of crosscutting concerns. The expected benefits of AOSD are improved comprehensibility, ease of evolution and increased potential for reuse in the development of complex software systems.

However, an aspect itself may be the locus of further complexity. Aspects can be *homogeneous*, for example, by providing a logging behavior that affects all procedures in a certain interface; but they may also be *heterogeneous*, for example, by implementing the two sides of the subject-observer protocol that affects two different classes [4, 11, 14]. As a new kind of modular unit, aspects should have explicit interfaces [32]. The *aspect interface* should describe the way an aspect interacts with the rest of the system and how it affects other modules. Moreover, since aspects may possibly affect several classes heterogeneously, the aspect interface could be decomposed into two or more partial interfaces that aggregate the different ways an aspect affects the rest of the system. Explicit aspect interfaces should ensure proper modularity and promote predictability of composition, enhancing comprehension and reuse [4, 28, 33]. Finally, the notion of aspect interfaces should be supported from early stages of design, not only at the implementation stage.

Despite of the importance of the subject, very little work has addressed issues related to the definition of aspect interfaces. Kiczales and Mezini [28] have discussed the impact of aspect-oriented programming on modular reasoning. However, they were concerned with the influence of aspects on the specification of classical interfaces of components (classes) rather than the definition of aspect interfaces. Mezini and Ostermann [33] have explored this issue, but only in the context of a specific aspect-oriented programming language. Moreover, they do not elicit the main principles underlying the definition of aspect interfaces. Similar limitations are found in the definition of a behavioral interface specification language for AspectJ, called Pipa [43]. In addition, existing aspect-oriented modeling languages [9, 10, 39, 41] lack explicit support for aspect interfaces. In this context, there is an urgent need for understanding the common properties of aspectual interfaces at a higher level of abstraction and supporting them through modeling notations.

This paper introduces the notion of *crosscutting interfaces* as a conceptual tool for dealing with the complexity of heterogeneous aspects at the design level. A *crosscutting interface* is a set of structural or behavioral enhancements specified inside the aspect to affect homogeneously one or more modules at some specified join points [5]. In addition, we have incorporated crosscutting interfaces in a modeling language in order to enhance aspect description at the design level [4]. Our modeling notation also allows the description of architecture-level aspects that

similarly include the explicit representation of crosscutting interfaces. As a consequence, our approach provides, from an early stage of design, a systematic foundation for minimizing the complexity caused by the handling of heterogeneous aspects. Finally, we have also performed some case studies to support our arguments in favor of crosscutting interfaces using our modeling language.

The remainder of this paper is organized as follows. Section 2 presents some background and motivates the need for supporting crosscutting interfaces at different development stages. Section 3 presents the definition and properties associated with crosscutting interfaces. Section 4 introduces our notation for specifying aspects and crosscutting interfaces at the architectural and detailed design stages. Section 5 presents our evaluation of both the concept of crosscutting interfaces and our modeling notations in terms of usability and usefulness.

Section 6 presents some additional discussion. Section 7 summarizes our contributions in the light of related work. Section 8 presents concluding remarks.

2. AOSD: Basic Concepts and Modeling Notations

This section revisits the basic concepts associated with modularity and AOSD, and motivates the need for adequate support for clear aspect interfaces at the design level.

2.1. Basic Concepts

Separation of concerns is a well-established principle in software engineering that addresses the limitations of human cognition for dealing with software complexity. A *concern* is some part of the problem that we want to treat as a single conceptual unit [13]. *Modularity* is also a fundamental principle for managing software complexity [32, 34]. Complex software systems should be decomposed into a set of highly cohesive modules, each implementing well-defined *interfaces* and dealing with a single concern. An interface is a well-defined prescription of how the module, which realizes it, interacts with the rest of the system [32, 33]. The basic modules used in object-oriented software development (OOSD) are classes and objects.

However, the modules and the composition mechanisms provided by OOSD may not be sufficient for separating some concerns found in most complex systems. These concerns have been called

crosscutting concerns since they naturally cut across the modularity of other concerns. Aspect-oriented software development (AOSD) [1, 15] has been proposed as a technique for improving the separation of crosscutting concerns. AOSD addresses the modularization of these concerns by providing a new abstraction, called *aspect,* which makes it possible to separate and compose them to produce the overall system. Thus, an aspect-oriented (AO) system is composed of two kinds of modules: classes and aspects. The predominant definition for aspects is the one that comes from the AspectJ programming language [2] - aspects are implementation-level modules that specify and localize: (i) *refinements* and *redefinitions* of behaviors at well-established points localized in the other system's modules, (ii) *additions* of members (state elements and behaviors) to other system's modules, and (iii) modifications of type relationships with existing modules. In AspectJ's terminology, (i) is implemented by means of *pointcuts* and *advice*, and (ii) and (iii) are implemented by means of inter-type declarations [2].

2.2. ASPECT-ORIENTED MODELING AND A MOTIVATING EXAMPLE

Following the above implementation-level definition, aspects must also be supported at preliminary development phases, such as the architectural definition stage and the detailed design stage. Indeed, both homogeneous and heterogeneous aspects may show up early in software development, and aspect-oriented modeling is essential to support their specification. *Aspect-Oriented Modeling* (AOM) [41] is a critical part of AOSD that focuses on notation and techniques for specifying, visualizing and communicating aspect-oriented solutions along the path from requirements to implementation with *Aspect-Oriented Programming* (AOP). Different views of an aspect are useful for different tasks. In order to model a system and communicate its properties, a high-level view is suitable.

This subsection motivates the need for clear aspect interfaces at the design level by means of a simple well-known example: the design of the Observer pattern [17] using the AODM approach [39]. The design is abstracted from the pattern implementation in AspectJ [25]. Conceptually, the Observer design pattern emphasizes the use of two key participants: Observer and Subject. The Subject participant knows its observers and realizes an interface for attaching, detaching and notifying Observer objects. The Observer participant defines an updating interface for

objects that should be notified of changes in a subject. The design-level specification of the Observer aspect should clearly define these two participants as two modules that interact with each other via well-defined interfaces. The Observer aspect is a classical example of heterogeneous aspect.

Aspect-Oriented Design Model (AODM) [39] is an UML extension that enhances the existing UML specification with aspect-oriented concepts that mimic the crosscutting characteristics of the AspectJ language. The AODM defines: (1) a special stereotype for standard UML classes (<<aspect>>) to capture the semantics of aspects, (2) a new stereotype for standard UML operations (<<pointcut>>) to capture the semantics of AspectJ's pointcuts, (3) a new stereotype for standard UML operations (named <<advice>>) that capture the semantic of AspectJ's advice, and (4) a new stereotype for UML collaboration templates (<<introduction>>) to describe inter-type declarations. Figure 1 illustrates how an aspect-oriented implementation for the Observer design pattern [25] is modeled using the AODM. In the example, the Observer design pattern is used to make a color label (playing the Observer role) change its color whenever a button (playing the Subject role) is clicked. One abstract aspect (SubjectObserverProtocol) implements the Observer pattern and a concrete aspect (SubjectObserverProtocolImpl) implements a particular instance of this pattern for the Button and ColorLabel classes.

The AODM provides a visual notation for AspectJ's programs, where boxes that represent aspects are polluted with very detailed, implementation-specific information that is only useful for AspectJ. Note that, although the collaboration templates stereotyped with <<introduction>> provide some means to modularize inter-type declarations in a *per-participant* basis, the specification of join points (pointcuts) and advice are not properly modularized. Therefore, it lacks adequate support for dealing with heterogeneous aspects.

In the AODM approach, all pointcuts and advice are top-level elements and should be described in the aspect's Operations compartment. As a consequence, the notation does not provide means to express that both the pointcut stateChanges and the advice advice_id01 are related to the Subject participant. Moreover, the aspect's local operations are supposed to be mixed with pointcuts and advice. This design reflects the poor separation of concerns inside AspectJ's aspects [4, 33] and leads to a poor separation of concerns inside AODM design-level aspects as well.
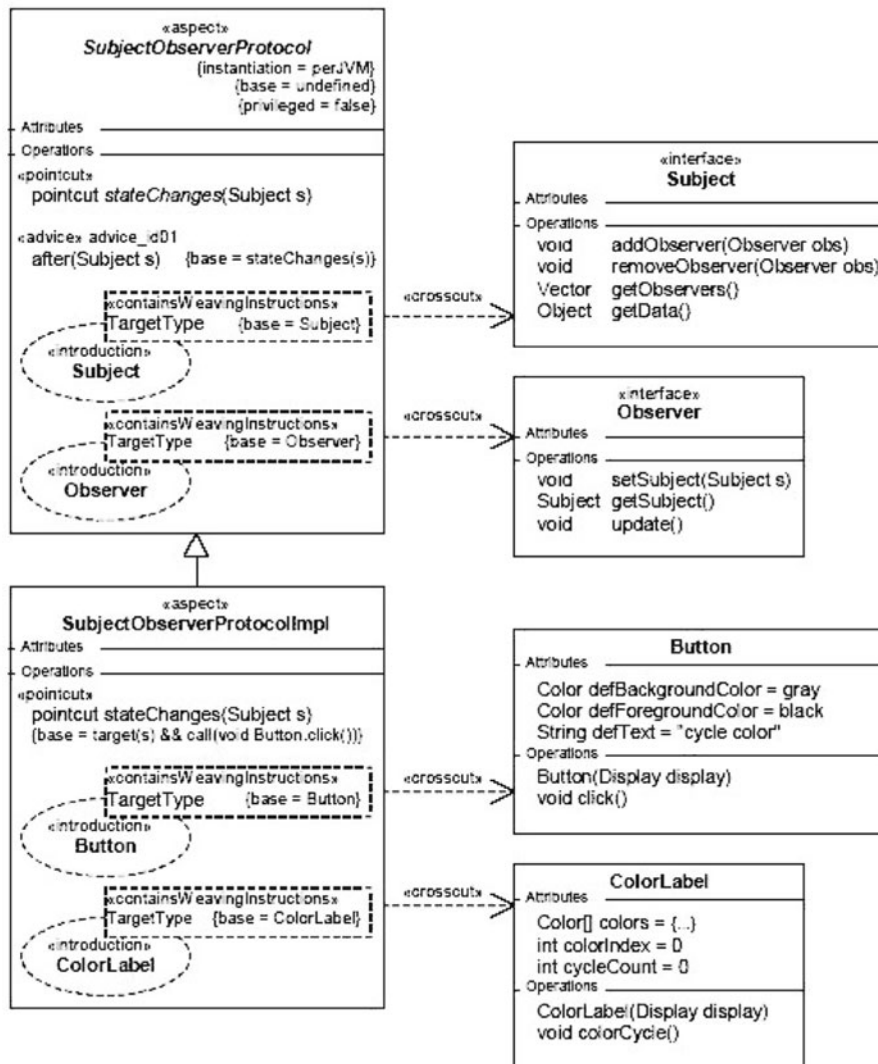
Figure 1. Using the AODM notation to describe the Observer aspect [39].

This design also leads to a poor scalability: as the complexity of an aspect increases, the aspect's interface may become bloated. Although advice and pointcuts are fundamental concepts of aspect-oriented languages, there is a recognized need for the definition of higher-level module concepts on top of them [4, 33], especially for dealing with heterogeneous aspects. Finally, the design does not provide a big picture for the solution that abstracts from programming-specific details and emphasizes high-level relationships between aspects and classes. From the diagram presented in Figure 1, it is very difficult to capture even basic information such as Button plays the Subject role whilst ColorLabel plays the Observer role.

In this context, explicit aspect interfaces are necessary for dealing with multi-abstraction aspects and

improving comprehensibility at the design level. The idea of crosscutting interfaces is essential for the modularization of the different ways that a heterogeneous aspect affects the different parts of a system.

3. CROSSCUTTING INTERFACES

The design of modular AO systems is fundamental for managing software complexity. According to our viewpoint, the concept of crosscutting interfaces is an important step in this direction.

This section provides a conceptual framework for crosscutting interfaces, which consists of a set of definitions (Section 3.1) and a set of fundamental properties (Sections 3.2 through 3.4). This framework was abstracted from both the aspect models of existing AO programming

languages [2, 27, 30, 33] and our extensive modeling of AO systems (Section 5).

According to the literature [32, 35], an interface is a well-defined prescription of how the module, which realizes it, interacts with the rest of the system. Our definition of crosscutting interfaces refines this traditional definition of module interfaces and adapts it to the AOSD context as described in the following subsections.

3.1. DEFINITIONS

Crosscutting interfaces are defined as named sets of *crosscutting features* that characterize the crosscutting behavior of aspects with respect to other modules [4]. We use the term *crosscutting feature* whenever we refer to any structural or behavioral enhancement specified inside the aspect to affect one or more modules at some specified join points [5]. The concept of interface presented here should not be tied up with the idea of object-oriented interfaces and their specific models in programming languages, such as Java; yet, the definition of interfaces as provided here should be related to the more general idea of modular interfaces and their properties [32].

Crosscutting interfaces modularize sets of related crosscutting features. Without proper means for separation and structuring, crosscutting features tend to be mixed with each other and with other aspect features, such as local attributes and methods. The natural consequences are reduced comprehensibility of aspects as well as difficulties for predicting their composition, especially for heterogeneous aspects. As the interfaces of conventional modules, crosscutting interfaces are sub-contracts between the aspectual module, which implements them, and each module affected by that aspectual module. However, the notion of crosscutting interfaces is different from the traditional notion of interfaces because each crosscutting interface also embodies the definition of how an aspect partially crosscuts other system modules. The *aspect interface* (or *aspectual interface*) therefore, consists of one or more crosscutting interfaces.

Crosscutting interfaces follow six fundamental properties, which are described below. Note that each property either holds or refines the definition of a classical property of module interfaces [32]. Crosscutting interfaces also entail new properties. They are classified into three categories according to their nature: (i) interface realization, (ii) interfaces and crosscutting, and (iii) interface specialization. In this context, the term *crosscutting* denotes a relationship between an aspect and one or more modules, so that the aspect may affect the modules structure and behavior at well-defined points. The term *normal interfaces* denote interfaces of conventional modules, other than aspects.

3.2. INTERFACE REALIZATION

Property #1 - Multiplicity. Each homogeneous aspect implements only one crosscutting interface. However, similarly to the realization of normal interfaces by conventional modules, each aspectual module can also realize more than one crosscutting interface. One aspect can define more than one crosscutting interface because it may affect heterogeneously different categories of modules in the system.

Property #2 – Distinct Realizations. One or more components can realize the same crosscutting interface. An interface, whether aspectual or not, is a contract between two modules. As a consequence, it is a reusable abstraction and can be realized in distinct manners by different aspects. Each aspectual module may provide different implementations to the crosscutting interfaces it realizes.

3.3. INTERFACES AND CROSSCUTTING

Property #3 - Quantification. A given crosscutting interface can affect one or more modules. Each crosscutting interface affects these modules homogeneously. The modules can be either aspectual ones or conventional ones. The interface prescribes the same changes to the structures and/or behaviors of the target modules. In fact, this property of crosscutting interfaces satisfies the quantification property of AOP [16] and is not satisfied by traditional notions of module interfaces.

Property #4 – Contractual Crosscutting. A crosscutting interface can directly affect (crosscut) normal interfaces in addition to the internals of a module itself. Crosscutting interfaces change the definition of normal interfaces by adding new elements or refining their existing elements. As a consequence, aspectual interfaces may modify the contract prescribed by each affected normal interface. This property is not found in conventional definitions of module interfaces. The crosscutting characteristic of aspectual interfaces means that a given aspect interface can crosscut more than one normal interface. The next property is an important special case of this property.

Property #5 – Chain of Crosscutting Interfaces. A crosscutting interface can affect other crosscutting interfaces, not only normal interfaces. In this way, an arbitrary crosscutting interface X may crosscut a crosscutting interface Y, which in turn can crosscut an interface Z, and so on, producing a chain of crosscutting interfaces. This is a particularly important property for aspect-oriented design because heterogeneous aspects are often interactive and overlapping in real complex systems (Section 5).

3.4. INTERFACE SPECIALIZATION

This property is related to a special kind of relationship between interfaces: specialization.

Property #6 - Extensibility. A crosscutting interface can be extended by other crosscutting interfaces. The extending interface may add, refine or redefine some crosscutting features defined in the parent crosscutting interface. This property is in line with the notion of specialization of aspectual modules. The specialization of crosscutting interfaces is a property required in several examples of heterogeneous and homogeneous aspects, such as code mobility [19, 23], learning [20, 22], and design patterns [18, 24].

4. NOTATION

Crosscutting interfaces can be regarded as a conceptual tool for dealing with the inner complexity of aspects at the design level. They can be supported by an aspect-oriented modeling language that provides notation for specifying aspects at the design level. Moreover, the description of architecture-level aspects may also benefit from the support for explicit representation of crosscutting interfaces. In this section, we introduce our notation for specifying aspects and crosscutting interfaces at the architectural and detailed design stages. Our notation follows the principles underlying the conceptual framework for crosscutting interfaces (Section 3).

4.1. CROSSCUTTING INTERFACES AND DESIGN-LEVEL ASPECTS

The aSideML [4] is an aspect-oriented modeling language that provides notation, semantics and rules for specifying aspects and crosscutting at the design level. In particular, the language supports the explicit definition of one or more explicit *crosscutting interfaces* that organize the aspect's join point description and crosscutting behavior.

The aSideML language enables the designer to build models that focus on keys concepts, mechanisms and properties of AO systems, in which aspects and crosscutting are explicitly treated as first-class citizens. These models help in dealing with the complexity of aspect-oriented systems, by providing essential views of structure and behavior that emphasize the role of crosscutting elements and their relationships to other elements. Some of these models present a detailed design view of AO systems that may also serve as preliminary blueprints to be evolved towards the implementation models of AO programming languages and tools.

The aSideML model elements can be structural or behavioral. The main structural model elements are aspects, crosscutting interfaces, crosscutting features, base elements (elements that aspects are supposed to enhance) and the relationships between them. Aspects are defined as *parameterized* elements with one or more explicit *crosscutting interfaces* to organize join point description and aspect crosscutting behavior. Aspects abstract over the identity of the elements they will eventually crosscut, by declaring template parameters to hold actual names of classes and methods. A new kind of relationship, the *crosscutting relationship*, subsumes a relationship between an aspect and a base element; it also performs a binding that defines the base elements and operations that replace the aspect's template parameters. The behavioral model elements and the detailed semantics of aSideML aspects are presented in [4].

Figure 2 presents the design of the Observer pattern using the aSideML notation. The aspect is drawn as a dashed rectangle, with a diamond symbol containing the aspect name. Crosscutting interfaces are declared inside aspects and are drawn as solid-outline rectangles with inner compartments separated by horizontal lines. Crosscutting features are listed in different compartments, depending on the kind of enhancement they support.
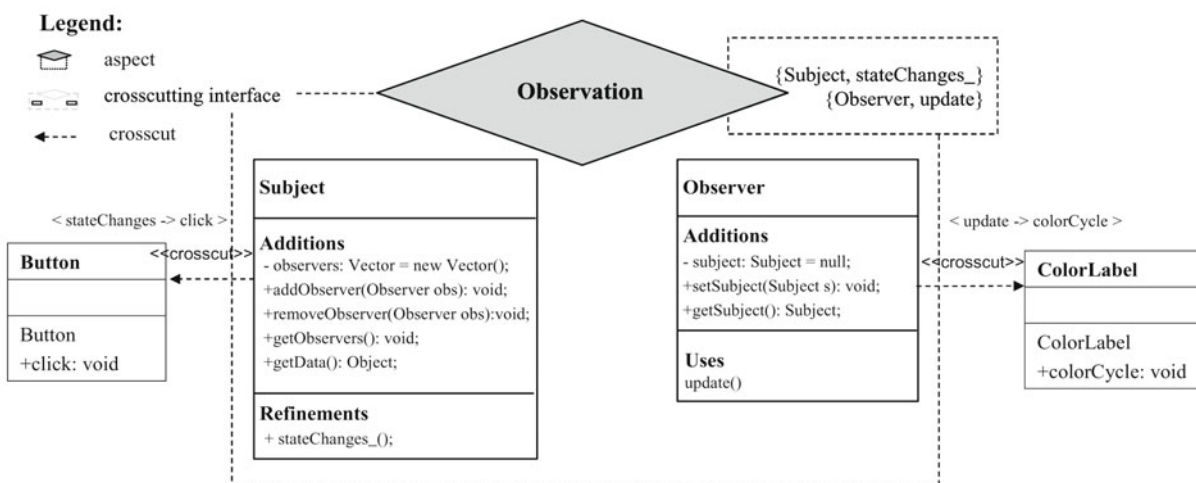

Figure 2. Design-level aspects and crosscutting interfaces in aSideML.

The *Additions* compartment lists data and operations to be introduced in classes. The *Refinements* compartment lists crosscutting operations to be combined before, after or before/after class operations and the *Redefinitions* compartment lists crosscutting operations that override class operations. In these two compartments, each operation name (op) is adorned with the _ symbol, with three permitted combinations: _op, op_ and _op_. These adornments indicate that the crosscutting operation provides behavior to be combined before, after or before/after the base operation behavior. Finally, an optional compartment may be supplied to define placeholders for required operations (*Uses*). The aspect is presented with a small dashed rectangle superimposed on the upper right-hand corner of the rectangle for the aspect. This rectangle is a template parameter box that contains lists of formal parameters, one list for each aspect's crosscutting interface. The first parameter of each list is the name of the corresponding crosscutting interface.

The Observation aspect presented in Figure 2 has two crosscutting interfaces, one for each pattern participant. Observer is a crosscutting interface that modularizes crosscutting features that enhance arbitrary objects so that these elements become observers. Observer declares three additions – the attribute subject and two public operations, setSubject(Subject s) and getSubject( ) – and one requirement – update( ). The Subject crosscutting interface modularizes crosscutting features that enhance objects so that these elements become subjects. Subject declares five additions and one refinement – stateChange_( ) – that denotes behavior to be executed after the affected base behavior.

The Observation aspect is connected to the elements it affects (Button and ColorLabel) by means of two *crosscutting relationships* (shown as a dashed arrow with the tail on the crosscutting element and the arrowhead on the base element, and the keyword <<crosscut>>). The crosscutting information is displayed as a comma-separated list of template parameter matches. The crosscutting relationships connect the Observation aspect to Button

(binding stateChange to click) and ColorLabel (binding update to colorCycle). Observation enhances Button by means of the Subject crosscutting interface; the structure of instances of Button includes new attributes and operations listed in the Additions compartment and their behavior is enhanced at the defined join point (click). Observation enhances ColorLabel by means of the Observer crosscutting interface.

4.2. CROSSCUTTING INTERFACES AND ARCHITECTURAL-LEVEL ASPECTS

The very nature of the detailed design notation for crosscutting interfaces does not provide a big picture of the AO system. Hence this section presents a model for specifying and communicating AO software architectures, depicting a high-level view of the AO design and respective crosscutting interfaces. An AO software architecture provides components for aspectizing crosscutting concerns at an early stage of design. Our architectural model provides notation and semantics that enable architects of AO software to build models that focuses on the key concepts and properties of AO systems at the architectural level. The main goal is to prevent the architect from dealing with detailed design issues that are not relevant at the architectural level.

Figure 3 illustrates the notation elements of our architectural model, based on the example of the Observer design pattern, as in the previous sections. In our modeling approach, the architecture designers should concentrate on two main issues. First, they work on the specification of the central components of the AO system. The architect has modeling support to distinguish between normal components and aspectual components. *Aspectual components* (or architectural aspects) are aspects [19, 21] at the architectural level. Architectural aspects are UML components [42] represented as diamonds. Each aspectual component is related to more than one architectural component, representing its crosscutting nature. Note that the architectural view of an aspect suppresses all information about its inner elements.
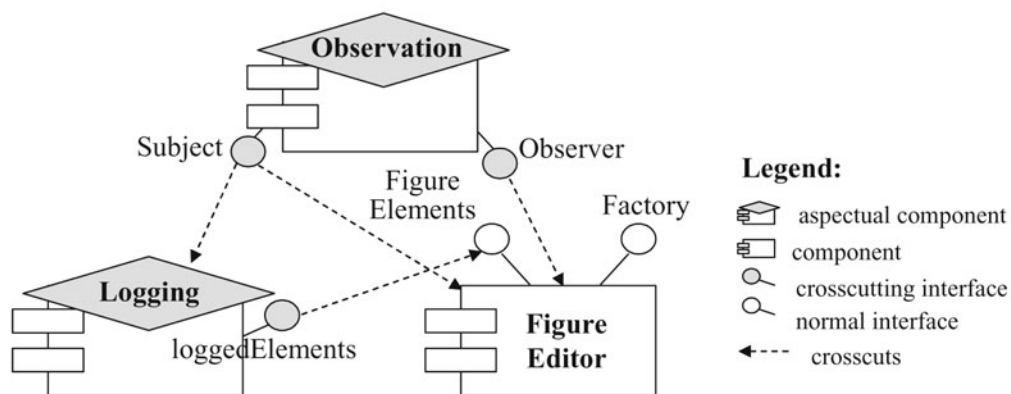

Figure 3. Crosscutting Interfaces and Architecture-Level Aspects.

Second, software architects define the interfaces of the architectural components in a higher-level fashion. Figure 3 illustrates some architectural components and interfaces. Each interface is displayed as a small circle with the interface name placed next to the circle. The interfaces are attached to the architectural components, and are categorized in two groups: *normal interfaces* and *crosscutting interfaces*. Normal interfaces (or regular interfaces) are colored in white and crosscutting interfaces in gray. Each architectural component has one or more interfaces (Property #1), and different components can realize the same interface (Property #2).

Crosscutting interfaces in the architectural model specify which architectural components an aspectual component affects; they do not declare how the components are affected. A crosscutting interface is different from a normal interface. The latter only provides services to other components. Crosscutting interfaces specify when an architectural aspect affects other architectural components. An aspectual component conforms to a set of crosscutting interfaces. The aspect interface crosscuts either internal elements of architectural components or other interfaces. The first case means that the architectural aspect directly affects the internal structure or dynamic behavior of the target component (Property #3). The second case means that the aspect affects the contract defined by other interfaces (Properties #4 and #5). The specialization of crosscutting interfaces (Property #6) is supported only in the detailed design notations (see Section 5.1 for an illustrative example).

The purpose of crosscutting interfaces here is to modularize parts of a concern which usually crosscut other concerns in traditional kinds of architectural decomposition, such as object-orientation. For example, Figure 3 shows the Subject interface in the Observation component that modularizes the event observation mechanism and the reference to observers, which are issues that usually crosscut the other concerns [18, 25].

5. CASE STUDIES

The applicability of the concept of crosscutting interfaces and the usefulness and usability of our modeling approach (Section 4) have been evaluated in the context of several case studies [4, 12, 18, 19, 21, 22] that encompassed different characteristics, diverse domains, and different degrees of complexity. The selected domains include the GoF design patterns [4, 18], multi-agent systems [19, 21, 22], web-based information systems [12, 37, 38], and a Telecom example [4]. Due to space limitation, from all these systems, we have selected three particular case studies to be presented in this paper. This selection was based on the heterogeneity of the crosscutting concerns and their

respective domains. For further details about the other case studies, the reader should refer to [12].

The first case study example is the Observer pattern, which was presented in the previous sections. It is a canonical example in the sense that it is frequently used by several modeling approaches [39, 41] to illustrate their features. Moreover, it represents crosscutting concerns relative to the GoF patterns [18, 25], which are recurring design solutions used in every kind of application. The second case study (Section 5.1) is a web-based information system that entails classical crosscutting concerns, such as persistence and distribution. The third case study (Section 5.2) is a multi-agent system that has been chosen for a number of different reasons: (i) it involves both domain-specific and application-dependent concerns; (ii) it is not focused only on traditional crosscutting concerns (such as logging and tracing); and (iii) it addresses concerns that have not been deeply investigated by the AOSD community.

5.1. HEALTH WATCHER

This section presents the architectural and detailed design modeling of HealthWatcher (HW), a web-based information system (WIS) developed by the Software Productivity research group from the Federal University of Pernambuco [37, 38]. The HW system was implemented in AspectJ [2].

We present an overview of the modeling of the HW system and discuss how the concept of crosscutting interfaces and our notations were useful to cope with the complexity of modeling heterogeneous aspects in the HW system. For further details, the reader should refer to [12]. The main purpose of the HW system is to allow citizens to register complaints to the health public system.

Figure 4 presents the architectural model of the HW system. There are three functional components: GUI (graphical user interface), Business, and Data. They are structured by following the Layer architectural pattern [3]. The HW architecture model also specifies three aspectual components to capture typical crosscutting concerns encountered in web-based information systems: (i) Distribution – this aspectual component is responsible for making distributed the Business component services; (ii) Persistence – models the implementation of several crosscutting issues (transaction management, data update, repository configuration) related to the data management in web-based information systems; (iii) Concurrency – specifies mechanisms to apply different concurrency strategies to the functional components.

The use of crosscutting interfaces was particularly useful in this system modeling because each of these three architectural aspects define more than one crosscutting

interface. For example, the Distribution aspectual component defines three crosscutting interfaces in order to enable the distributed communication between the GUI and Business functional components. The Server interface models the crosscutting behavior to become distributed the Business component services. On the other hand, the RemoteAccess interface models the addition of behavior in the GUI component that makes possible the remote access of the Business component. Finally, the SerializableParticipant interface defines which elements in the Business component need to be prepared to be transmitted over the network. Thus, each crosscutting interface defined for an aspectual component in the HealthWatcher architecture modularizes different behaviors to be included in its functional components in order to implement a specific crosscutting concern.
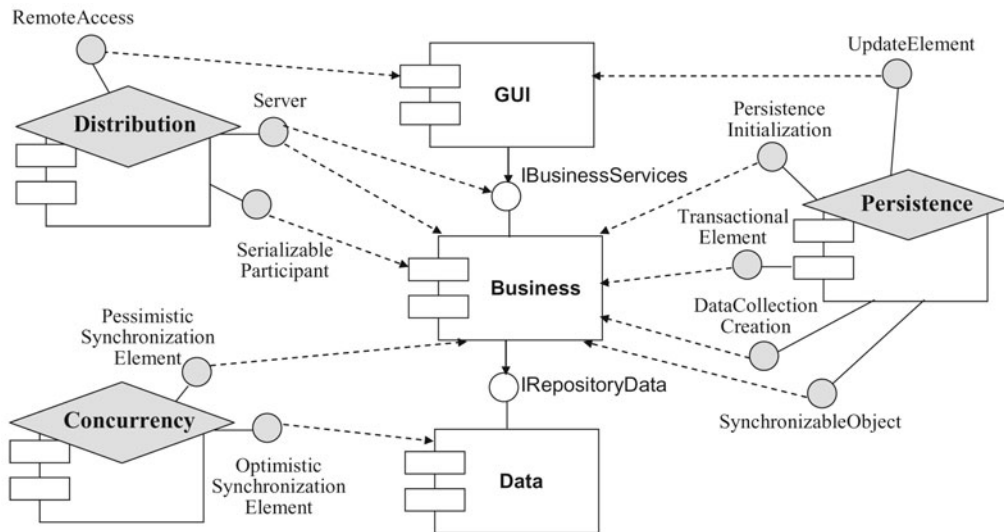

Figure 4. The architectural model of the HW system.

Figure 5 shows the detailed design of the HealthWatcher system. Due to space limitation, Figure 5 only presents the detailed design of the GUI, Business and Distribution components. These components are detailed to present the different classes, interfaces and aspects responsible to implement their respective functionality. Also, the different relationships between their internal elements are specified to define clearly how they are connected. Finally, the crosscutting interfaces of the Distribution aspectual component are refined to specify the way in which it affects the different elements of the GUI and Business components.

As mentioned before, the GUI component provides a web user interface for the HealthWatcher system. Thus, it is implemented as a set of Java Servlet classes. The Business and Data components are refined by using the Persistent Data Collection (PDC) [31] design pattern. This pattern is used to offer a better separation between the code related to the user interface, business rules and data access. Following the guidelines of the PDC pattern, a facade class (HWFacade), was specified to expose the services of the Business component. Each servlet can access the services of the Business component by invoking the methods of the HWFacade class.

The HWFacade class aggregates several business collection classes (such as, ComplaintRegister, EmployeeRegister), which implements the business logic related to a specific and significant business class (such as, Complaint and Employee). Business classes are used to store relevant information manipulated by the system. Figure 5 omits the classes and interfaces that refine the Data co mponent. For further details, please refer to [12].

The notion of crosscutting interfaces was also important to support a clear documentation of the detailed design of the HW system. The detailed design of the Distribution aspectual component (Figure 5) defines two distinct hierarchies of aspects. These hierarchies model the distribution concern from the different perspectives of server and client elements. The first hierarchy contains the ServerSide abstract aspect and the DistributedServer aspect. These aspects are modeled to realize the crosscutting behavior of making distributed the Business component.

The ServerSide aspect aggregates the Server interface. This crosscutting interface offers the _facadeMainExecution_ redefinition which is responsible to init a specific class of the Business component as a

distributed object. The ServerSide aspect also defines a set of abstract methods which are realized by the DistributedServer subaspect. This latter aspect is also responsible for specifying how the crosscutting interfaces Server and SerializableParticipant must affect classes of the Business component. For example, the SerializableParticipant crosscutting interface affects the Complaint and Employee classes in order to make serializable those elements. On the other hand, the Server crosscutting interface affects the HWFacade class to allow that class to implement the IRemoteFacade interface and to initialize its distributed service during the execution of the main() method.

The ClientSide abstract aspect and the HWClient aspect model the crosscutting behavior responsible for allowing the Servlet classes to access remotely the distributed services of the HWFacade class. The ClientSide aspect defines: (i) the getRemoteFacade() method responsible to model the behavior of looking up a distributed facade object; and (ii) the abstract _facadeLocalCalls_ redefinition which models the redirection of local calls to the HWFacade class to its equivalent remote calls. The HWClient subaspect specializes the RemoteAccess crosscutting interface by realizing the _facadeLocalCalls_ redefinition to intercept the local method invocations from the servlets classes to the HWFacade class.
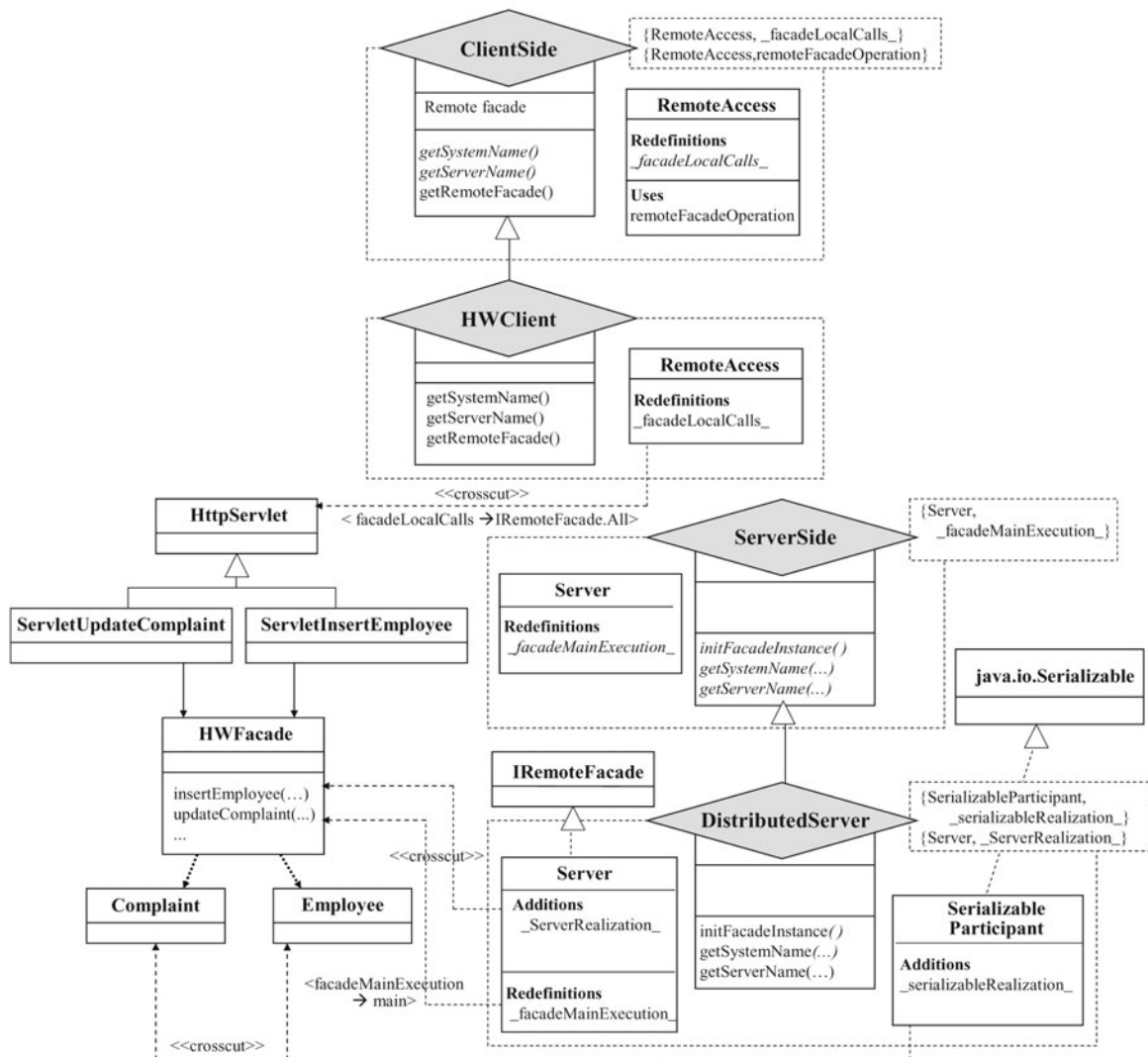

Figure 5. Detailed design of the HealthWatcher system.

5.2. EXPERT COMMITTEE

This section presents the modeling of a multi-agent system (MAS), named Expert Committee (EC) [19, 21], using the notation for crosscutting interfaces presented in Section 4. First, we present the architectural model for the software agents in the EC system, and then the detailed design of some architectural components. Figure 6 introduces the model of the AO agent architecture that

encompasses the components for aspectizing common crosscutting concerns in MASs, such as learning and collaboration. Each crosscutting agent property is modularized by an individual architectural aspect [19]. For simplicity, some additional normal and aspectual components are omitted.

The AO architecture is composed of two main normal components. The Environment component represents the agent location and the system services, such as naming service, registration, communication, and so forth. The Kernel component encapsulates the basic services provided by the agent for its clients; these services are non-crosscutting. As a result, this component realizes the Services normal interface to make those basic agent functions available to the external entities. This component is also responsible for modularizing the knowledge elements, such as actions, plans, goals, and beliefs. The KnowledgeUpdating interface is used to alter and evolve the internal agent knowledge.

There are also aspectual components that separate the crosscutting agent-related concerns from each other and from the Kernel component. Most of the aspectual components crosscut multiple agent components in different ways, capturing their crosscutting characteristic. As a result, they realize more than one crosscutting interface and also affect other architectural aspects. For example, the MessageSending interface crosscuts the Kernel component and the Collaboration architectural aspects.

Figure 7 shows a partial representation of the AO design of the EC system using the aSideML modeling language. Due to space limitation, we illustrate only the detailed design of the Kernel, Learning and Collaboration components; parameters and attribute types were also omitted. Note that these architectural components are refined as a set of classes and aspects with additional design information. The figures present some of these classes and aspects; others essentially follow the same pattern. A complete description of the design elements can be found at [19, 21].
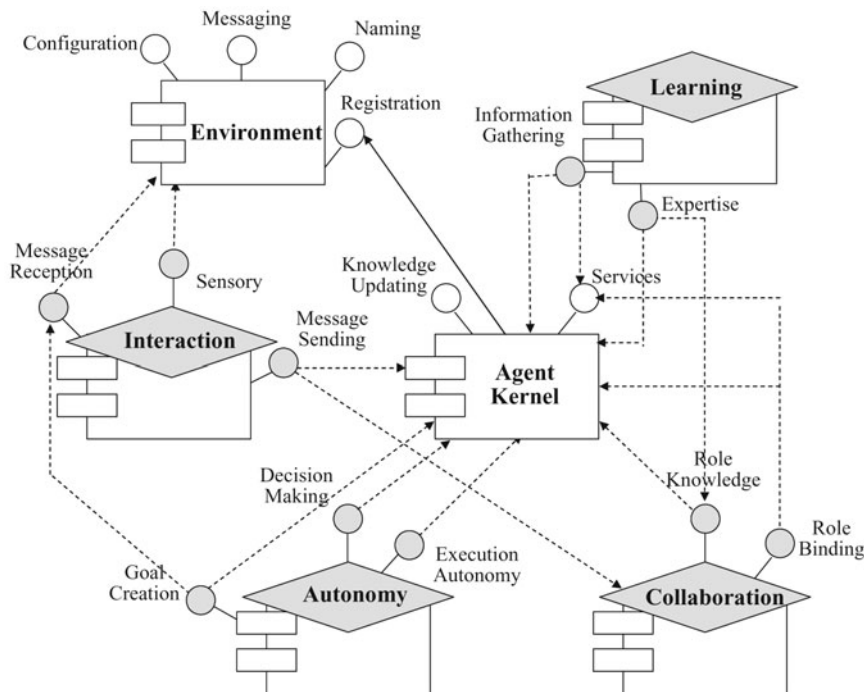

Figure 6. The Aspect-Oriented Architecture of EC Agents.

The Kernel component is refined as a set of classes, which represent the agent itself, and knowledge elements (e.g. plans). The hierarchy, derived from the Agent class, contains the methods that implement the agent actions and agent's basic services (i.e. the intrinsic interface Services presented in Figure 6). The Learning and Collaboration architectural aspects are decomposed in terms of abstract aspects, concrete aspects, and auxiliary classes (omitted for simplicity). Each crosscutting interface is refined as a set of additions, refinements, and uses

definitions, which are all realized by the attached aspect. Learning aspects are heterogeneous aspects that encapsulate the entire implementation of the learning concern. Their heterogeneity is mastered by the Expertise interface and the InformationGathering interface. These crosscutting interfaces are defined by the Learning abstract aspect and specialized by the ReviewerLearning aspect (Figure 7). The specialization of crosscutting interfaces was useful to model the EC system since it has a number of both abstract and concrete aspects.
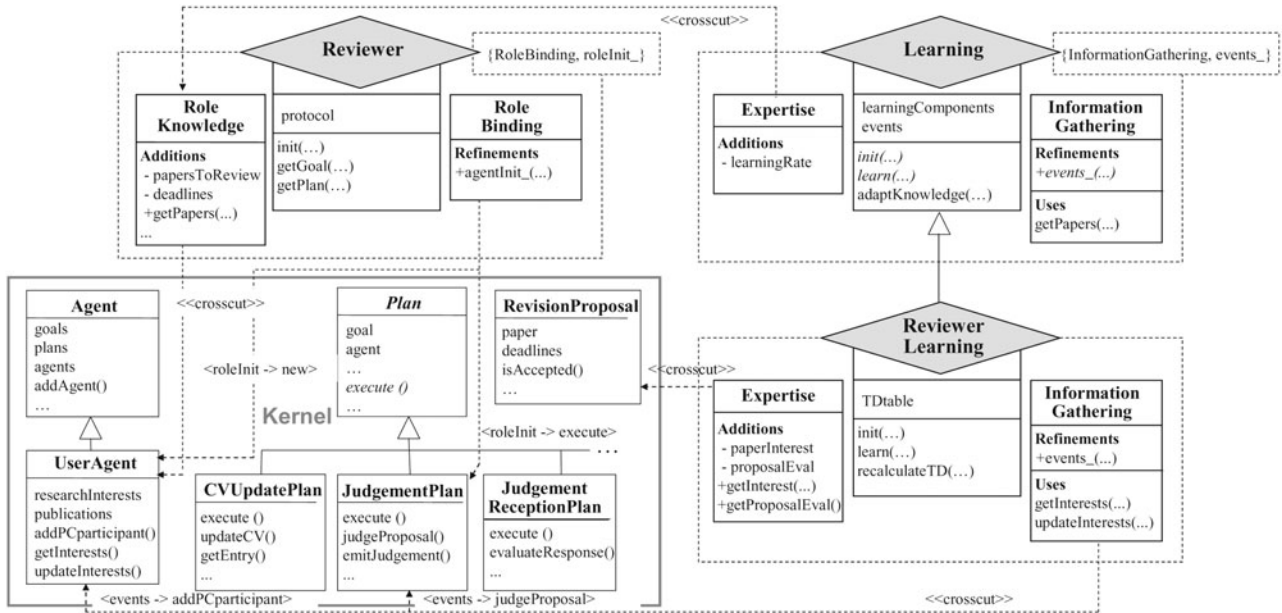
Figure 7. Refining the Architectural Aspects and Crosscutting Interfaces.

The Collaboration component aggregates collaboration protocols and roles played by the agents during their collaborative activities. Each role is represented by a design aspect and, as a consequence, the Collaboration component is realized by a set of inner role aspects. It is composed by four inner aspects, each one for a specific agent role: Author, Reviewer, PCMember, and Chair. Figure 7 illustrates the Reviewer aspect. Each inner aspect implements the RoleBinding interface and the RoleKnowledge interface. The first interface determines the events in which a given role is bound to the agent; the events_ refinement specifies the binding behavior. The second interface defines a set of additions which comprise the role-specific knowledge introduced to the agent playing that role.

5.3. ANALYSIS

This section analyses the results of the application of our modeling approach in terms of its usability and usefulness to master complex situations involving heterogeneous aspects. First, through the application of our approach, we were able to easily specify aspectual modules with multiple crosscutting interfaces both at the architectural and detailed design levels. The notation is even suitable to support the modeling of heterogeneous aspects with three or more crosscutting interfaces, such as the Distribution aspect (Section 5.1) and the Interaction aspect (Section 5.2). It is unlikely that we would have straightforwardly addressed this issue with other modeling approaches, such as AODM [39] and Theme/UML [10], either because they do not directly support crosscutting interfaces as modeling elements or because they do not enforce the concept of crosscutting interfaces and the relevance of aspect interfaces.

Also, we observed that our design language was effective to cope with intriguing crosscutting relationships (Section 3.3). We can see from Figures 6 and 7 that the presence of contractual crosscutting (Property #4) and chains of crosscutting interfaces (Property #5) are recurring in complex AO systems. With explicit support for crosscutting interfaces, it was possible to express which exact part (interface) of a component, whether aspectual or not, a given aspect is acting over. It is particularly interesting in the case of chains of crosscutting interfaces because it is easier to understand the final result of the weaving process; it minimizes the need for looking at the code to understand the inter-module composition. This may be otherwise difficult to determine based on other existing modeling notations.

Figure 7 also shows that our notation is effective to represent the refinement of elements in different compartments of a crosscutting interface (Property #6). Note that the InformationGathering interface, in the concrete aspect ReviewerLearning, specializes not only the declaration of refinements, but also the specification of the elements in the *Uses* compartment. A similar extension is made for the Expertise interface. Hence, it also makes evident how the proposed notation scales to support different kinds of extensions (Property #2). Finally, the design of the InformationGathering interface illustrates how different implementations can be provided for certain

compartments in the crosscutting interfaces, such as the definition of the *events_* abstract pointcut.

6. Discussion and Lessons Learned

This section provides further discussion of issues and lessons we have learned in the evaluation of our approach.

Mastering the Internal Complexity of Aspects. Heterogeneous aspects are very complex to be represented in a single rectangle, since they aggregate numerous disparate members, such as additions, refinements, redefinitions, and internal methods and attributes. Our notation, with its support for representing crosscutting interfaces separately from the internal structure of aspects, helped to organize these members in distinct inner rectangles, enhancing the design comprehension. In addition, instead of providing a single aspect interface, decomposing the aspect interface into two or more partial interfaces that aggregate and provide boundaries for related sets of join points and crosscutting behavior further enhances understandability and promotes predictability of composition.

Design Guidance. The explicit modeling of crosscutting interfaces helps the software architects and designers in achieving good design decisions. The definition of crosscutting interfaces allows the software engineers reasoning about the aspect design in terms of separate, well-structured design elements. In addition, when there is an aspect realizing two interfaces with no coupling between them, it potentially means that this aspect should be decomposed in two loosely-coupled aspects. Otherwise, the designer will come up with a non-cohesive aspectual module.

Language-Independent Approach. Several existing modeling languages are strongly tied up to AspectJ constructs, such as the AODM approach [39]. As a result, design models look like snapshots of the AspectJ code. Our notation is language agnostic because it encompasses a set of generic operators, namely additions, refinements, and redefinitions. These operators are commonly found in several programming languages. Redefinitions, for example, can be implemented as *around advice* without *proceed* in AspectJ [2], and by using *override* integration in Hyper/J [27].

Traceability. We found that the aSideML language provides traceability by explicitly linking elements of the architectural model (Section 4.2) to their corresponding elements in the detailed design model (Section 4.1). Our proposal allows the software developers to traceably refine architectural interfaces into design interfaces and vice-versa; crosscutting interfaces are supported in both

models. Our detailed design notations are also straightforwardly transformed to specific aspect models of well-known programming languages, such as Caesar [33], AspectJ [2], and Hyper/J [27]. With respect to Caesar, for example, our design models are more directly mapped to code because this language has explicit support for aspect interfaces [33]. Considering AspectJ, although it does not support aspectual interfaces, all the other modeling elements have a 1-to-1 mapping to their counterparts in the AspectJ code, as discussed in Section 2.1. For further details about the traceability between aSideML and specific aspect models, including Hyper/J, the reader should refer to [4, 8].

Scalability. Our architectural notation is scalable in several senses. First, it supports the description of the main structure and relationships of more than twenty aspects and normal components in a single sheet of paper. It also copes with the complexity of modeling multiple crosscutting interfaces. Finally, the notation also supports the expression of aspects affecting each other both at the internal structure and at the interface-level.

Maturity. In our experience, the support for crosscutting interfaces is a natural step to make AO modeling languages more mature and modular. The aSideML modeling language, with its support for crosscutting interfaces, has reached this maturity and, more than that, has been applied into a number of case studies. A number of adaptations into the aSideML language have been carried out in response to the flaws and inconsistencies detected in our experiments. In addition, the conceptual framework presented in Section 3 and our modeling notations are defined on the basis of a systematic extension to the UML metamodel [4, 6] and a consistent theory of aspects for AOSD [5]. In order to enable the use of the aSideML language in the modeling of other aspect-oriented systems, we are implementing a tool based on the Eclipse platform [36] that supports the modeling of aspects and crosscutting interfaces as well as the structural code generation to specific aspect-oriented languages.

7. Related Work

The idea of crosscutting interfaces has been originally defined in [4, 7]. Other researchers have already proposed similar abstractions. However, their work focuses on discussing those abstractions only at the implementation level.

Lieberherr et al [30] proposed an AOP model, in which aspects are captured by aspectual components. The functionality captured by an aspectual component is written in terms of its own class graph, called participant graph (PG), referring to abstract join points when needed.

The participants forming the PG play the role of crosscutting interfaces.

Also in the implementation level, Caesar [33] has been proposed. It defines an AOP model based on the notion of Aspectual Collaboration Interfaces (ACI). ACI is an interface that provides support for: (a) expressing an aspect as a set of collaborating abstractions, comprising the modular structure of the world as seen by the aspect, and (b) structuring the interaction between two parts of an aspect: aspect implementation, and aspect binding into a particular code base. Then, ACI can be regarded as crosscutting interfaces.

For the design level, Composition Patterns [9] is the most referenced AOM approach. Interestingly, it has its roots on Subject-Oriented Design, a design counterpart for Subject-Oriented Programming [26]. Clarke's Composition Patterns are based on the Subject-Oriented Design Model [9]. Therefore, Composition Patterns specify crosscutting concerns in a subject-oriented manner that is inappropriate for the design of AO programs in AspectJ in several ways. To overcome these limitations, Clarke's research on Composition Patterns approach has evolved to Theme/UML [10], with the goal of providing a "generic AOSD design language". In [10], the authors provide a mapping from Composition Patterns (or Crosscutting Themes in Theme/UML) to the programming elements of AspectJ. Composition patterns are UML templates for design subjects that expect classes and operations as *template parameters*. *Pattern classes* are the placeholders to be replaced by real class elements. Although pattern classes provide some sort of separation of concerns inside the "theme", the notation does not enforce the concept of crosscutting interfaces and the relevance of aspect interfaces.

Stein's AODM [39], on the other hand, presents a design model that complies with the semantics of AspectJ. He proposes a set of extensions that supplements the UML with means for the design of aspect-oriented programs with AspectJ exclusively. The use of collaboration templates to modularize inter-type declarations provides very limited support for crosscutting interfaces. Our approach is language independent and provides full support for crosscutting interfaces, as previously discussed.

Pinto et al [35] have proposed DAOP-ADL, an architecture description language used to describe software architectures composed of components and aspects as first-order elements. The specification of a component in DAOP-ADL is composed of two interfaces: (i) a provided interface – which describes the component services; and (ii) a required interface – which specifies the output messages and events that a component is able to produce. The aspect specification in DAOP-ADL contains: (i) an evaluated interface – which defines the messages that the aspect is able to intercept; (ii) a required interface – which specifies the output messages required to the aspect provides its service; and finally (iii) a target events interface – responsible to describe the events which the aspect can capture. The composition between components and aspects in DAOP-ADL is supported by a set of aspect evaluation rules. They define when and how the aspect behavior is executed. Thus, DAOP-ADL somewhat makes explicit the interfaces of an aspect by defining its evaluated and target events interfaces. The aspect evaluation rules are responsible to realize those interfaces to specific components. However, opposed to the aSideML language, the use of DAOP-ADL has been restricted to a specific platform proposed by its authors. In addition, it does not fully support all the important properties for crosscutting interfaces presented in Section 3.

8. FINAL REMARKS

In this paper, we presented crosscutting interfaces as an important conceptual tool for taming the complexity of heterogeneous aspects at the design level. First, we presented a conceptual framework for crosscutting interfaces at the design level that includes a set of definitions and fundamental properties (Section 3). A subset of these properties has been already reified by some well-known aspect-oriented programming languages (Section 7). We also proposed a set of notations in the aSideML language (Section 4) that conforms to and implements our conceptual framework for crosscutting interfaces. Our language uniformly supports aspect interfaces at both the architectural stage and the detailed design stage.

It is important to highlight that, especially in a young research area such as AOSD, other researchers may identify further properties for crosscutting interfaces or may intend to refine the properties and definitions presented here. However, these properties constitute a first important survey and may be regarded as a first approach towards the identification of fundamental properties of design aspects, their interfaces and relationships. Besides, they have emerged from practical modeling demands while developing real case studies.

REFERENCES

[1] *Aspect-Oriented Software Development.* http://aosd.net

[2] AspectJ Team. *The AspectJ Programming Guide.* http://eclipse.org/aspectj/.

[3] Buschmann, F. et al. Pattern-Oriented Software Architecture: A System of Patterns. 1996: Wiley and Sons.

[4] Chavez, C. A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, PUC-Rio, Rio de Janeiro, Brazil, April 2004.

[5] Chavez, C., Lucena, C. A Theory of Aspects for Aspect-Oriented Development. *Proc. of the Brazilian Symposium on Software Engineering (SBES'2003)*, Manaus, Brazil, October 2003, pp. 130-145.

[6] Chavez, C., Lucena, C. A Metamodel for Aspect-Oriented Modeling. In: *Workshop on Aspect-oriented Modeling with the UML, 1st Intl Conf. on Aspect-Oriented Software Development*, Netherlands, 2002.

[7] Chavez, C.; Lucena, C. Design Support for Aspect-oriented Software Development. In: *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, Tampa, USA, October 2001.

[8] Chavez, C.; Garcia, A.; Lucena,, C. Some Insights on the Use of AspectJ and Hyper/J. In: *Work. on Aspect-Oriented Programming and Separation of Concerns*, Lancaster, UK, 2001.

[9] Clarke, S., Walker, R. J. Composition Patterns: An Approach to Designing Reusable Aspects. In: *Proc. of the 23rd International Conference on Software Engineering (ICSE)*, Toronto, Canada, May 2001.

[10] Clarke, S., Walker, R. J. Generic Aspect-Oriented Design with Theme/UML. In: Aspect-Oriented Software Development, Addison-Wesley, pp. 425-458, 2005.

[11] Colyer, A., Clement, A. Large-scale AOSD for middleware. *Proc. of the 3rd Intl Conf. on Aspect-Oriented Software Development (AOSD'2004)*, March 2004, Lancaster, UK, pp. 56-65.

[12] Crosscutting Interfaces for Aspect-Oriented Modeling. http://www.teccomm.les.inf.puc-rio.br/SoCAgents/CI/index.htm.

[13] Dijkstra, E. A Discipline of Programming. Prentice-Hall, 1976.

[14] Elrad, T. et al. Discussing aspects of AOP. *Communication of the ACM*, 44(10), October 2001, pp. 33-38.

[15] Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[16] Filman, R. What Is Aspect-Oriented Programming, Revisited. In: *Workshop on Advanced Separation of Concerns, 15th ECOOP*, Budapest, June 2001.

[17] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.

[18] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., Staa, A. Modularizing Design Patterns with Aspects: A Quantitative Study. In: *Proc. of the 4th Intl Conf. on Aspect-Oriented Software Development (AOSD'2005)*, Chicago, USA, March 2005, pp. 3-14.

[19] Garcia, A., Kulesza, U., Lucena,, C. Aspectizing Multi-Agent Systems: From Architecture to Implementation. *Software Engineering for Multi-Agent Systems III.* Springer-Verlag, LNCS 3390, December 2004, pp. 121-143.

[20] Garcia, A. et al. The Learning Aspect Pattern. *Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004)*, Monticello, USA, September 2004.

[21] Garcia, A. From Objects to Agents: An Aspect-Oriented Approach. Doctoral Thesis, PUC-Rio, Rio de Janeiro, Brazil, April 2004.

[22] Garcia, A., Lucena, C., Cowan D. Agents in Object-Oriented Software Engineering. In: *Software: Practice & Experience*, Elsevier, 34 (5), April 2004, pp. 489-521.

[23] Garcia, A., Sant'Anna, C., Chavez, C., Lucena, C., Staa, A. Separation of Concerns in Multi-Agent Systems: An Empirical Study. *In: Software Engineering for Multi-Agent Systems II*, Springer, LNCS 2940, Jan 2004, pp. 49-72.

[24] Garcia, A., Silva, V., Chavez, C., Lucena, C. Engineering Multi-Agent Systems with Aspects and Patterns. *J. Braz. Computer Society*, 1(8), July 2002, pp. 57-72.

[25] Hannemann, J., Kiczales, G. Design Pattern Implementation in Java and AspectJ, In: *Proc. of OOPSLA'02*, November 2002, pp. 161-173.

[26] Harrison, W., Ossher, H. Subject-Oriented Programming (A Critique of Pure Objects). In: *Proc. of OOPSLA'93*, 1993, pp. 411-428.

[27] Hyper/J Web Page, http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm, 2001.

[28] Kiczales, G., Mezini, M. Aspect-Oriented Programming and Modular Reasoning. In: *Proc. of the ICSE'05,* New York, NY, USA, 2005. ACM Press, pp. 49-58.

[29] Kiczales, G. et al. Aspect-Oriented Programming. In: *European Conf. on Object-Oriented Programming (ECOOP)*, LNCS 1241, Springer, Finland, 1997, pp. 220-242.

[30] Lieberherr, K., Lorenz, D., Mezini, M. Programming with Aspectual Components. Tech. Report NU-CCS-

99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[31] Massoni, T., et al. PDC: Persistent Data Collections Pattern. In *Proc. of the SugarLoafPLoP'2001*, Brazil, October 2001.

[32] Meyer, B. Object-Oriented Software Construction. 2nd Edition. Prentice Hall, 1997.

[33] Mezini, M., Ostermann, K. Conquering Aspects with Caesar. In: *Proc. of the 2nd Intl Conf. on Aspect-Oriented Software Development (AOSD'2003)*, Boston, USA, March 2003, pp. 90-99.

[34] Parnas, D. On the Criteria to Be Used in Decomposing Systems into Modules. In: *Communications of the ACM*, 15 (12), December 1972, pp. 1053-1058.

[35] Pinto, M. et al. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In: *Generative Programming and Component Engineering (GPCE 2003)*, pp. 118-137.

[36] Shavor, S. et al. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.

[37] Soares, S. An Aspect-Oriented Implementation Method. Doct. Thesis, Federal University of Pernambuco, Recife, Brazil, October 2004.

[38] Soares, S., Laureano, E., Borba, P. Implementing Distribution and Persistence Aspects with AspectJ. In: *Proc of OOPSLA'02*, 2002, pp. 174-190.

[39] Stein, D. An Aspect-Oriented Design Model Based on AspectJ and UML, Master Thesis, University of Essen, January, 2002.

[40] Tarr, P. et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: *Proc. of the ICSE'99*, May 1999, pp. 107-119.

[41] The 5th Aspect-Oriented Modeling Workshop. In Conjunction with UML 2004. October 11-15, 2004 Lisbon, Portugal.http://www.cs.iit.edu/~oaldawud/AOM/.

[42] Unified Modeling Language (UML) Specification: Infrastructure Version 2.0, December 2003. www.omg.org/uml/.

[43] Zhao, J.,Rinard, M. Pipa: A Behavioral Interface Specification Language for AspectJ. In: *6th Intl Conf. on Fundamental Approaches to Software Engineering, (FASE 2003)*, Warsaw, LNCS 2621, pp. 150-165.