Journal of Universal Computer Science, vol. 17, no. 5 (2011), 639-669
submitted: 8/2/10, accepted: 4/11/10, appeared: 1/3/11 © *J.UCS*

Modeling Quality Attributes with Aspect-Oriented Architectural Templates [1]

Mónica Pinto, Lidia Fuentes
(University of Málaga, Spain
{pinto, fuentes}@lcc.uma.es)

Abstract: The quality attributes of a software system are, to a large extent, determined by the decisions taken early in the development process. Best practices in software engineering recommend the identification of important quality attributes during the requirements elicitation process, and the specification of software architectures to satisfy these requirements. Over the years the software engineering community has studied the relationship between quality attributes and the use of particular architectural styles and patterns. In this paper we study the relationship between quality attributes and Aspect-Oriented Software Architectures - which apply the principles of Aspect-Oriented Software Development (AOSD) at the architectural level. AOSD focuses on identifying, modeling and composing crosscutting concerns - i.e. concerns that are tangled and/or scattered with other concerns of the application. In this paper we propose to use AO-ADL, an aspect-oriented architectural description language, to specify quality attributes by means of parameterizable, and thus reusable, architectural patterns. We particularly focus on quality attributes that: (1) have major implications on software functionality, requiring the incorporation of explicit functionality at the architectural level; (2) are complex enough as to be modeled by a set of related concerns and the compositions among them, and (3) crosscut domain specific functionality and are related to more than one component in the architecture. We illustrate our approach for usability, a critical quality attribute that satisfies the previous constraints and that requires special attention at the requirements and the architecture design stages.
Key Words: Aspect-Oriented Software Architecture, AO-ADL, Quality Attributes, Reuse, Usability
Category: D.2, D.2.11, D.2.13

1 Introduction

The critical quality attributes (QAs) of a software system must be well understood and articulated early in the development of a system, so that the architect can design a software architecture that satisfies them [Bachmann et al., 2005]. The list of documented QAs is very large. Some of them have strong functional implications and so can be easily modeled by software components (like security, usability, error handling, etc.). Others, like efficiency, price, portability, etc., could be mapped to architectural or implementation decisions, and not directly to functional components. In this paper we will focus only on those QAs with strong functional implications, which we will call functional quality attributes (FQAs).

[1] This research was funded by the Spanish Project TIN2008-01942 and by the regional Project FamWare P09-TIC-5231.

Modeling FQAs is not a straightforward task for several reasons. On the one hand, they are usually very complex, being composed by many concerns. For example, security is an FQA composed of authentication, access control, encryption and non-repudiation concerns, among others. Furthermore, FQA concerns also have dependencies and interactions among them. For example, the confidentiality concern will depend on the authentication, encryption and access control concerns. Moreover, these FQAs also have dependency relationships with other FQAs, since some concerns are shared and required by different FQAs. Security is a typical example, as its concerns are required to satisfy other FQAs such as, for example, usability, adaptability or context awareness.

On the other hand, modeling FQAs completely independently of the base applications is not always possible. Firstly, some FQAs require specific services from the application core components, meaning that the FQA concerns depend greatly on the base application. An example of this is the feedback concern of the usability FQA, which requires introducing a new panel into the graphical interface of the base application in order to show the feedback information. Secondly, FQA concerns normally crosscut several core application components. This means that the concern is scattered and/or tangled with the base functionality of such components. For example, the encryption concern of the security FQA is required by the components that both send and receive encrypted data, and thus it is scattered among them, and tangled with their base functionality. Finally, these FQAs are normally recurrent, being required by several applications. Therefore, final applications need ready-to-use (re)usable architectural patterns documenting and modeling the complex dependency relationships of FQAs. Moreover, these architectural patterns should be parameterizable in order to be instantiated to different applications.

Among the various existing approaches, some simply opt to model FQAs as non functional concerns, but this does not provide enough information about what kind of architectural artifacts (i.e. functional components) must be incorporated into the architecture in order to satisfy a given quality attribute [Cysneiros et al., 2005]. Other proposals [Welie, 2007] [Folmer and Bosch, 2004] [Juristo et al., 2007] [Juristo et al., 2003] suggest a complementary approach, in which FQAs are incorporated into the architecture as functional concerns. However, these approaches do not overcome all the problems presented above. Although they identify the dependencies between the concerns of a particular FQA, these approaches normally specify these concerns using tabular views or by textual descriptions of intricate scenarios [Geebelen and et. al., 2008]. Some of them focus on specific FQAs (e.g. usability in [Juristo et al., 2007]), rather than paying attention to the dependencies identified across different FQAs. Moreover, the solutions provided suffer from the scattered and tangled code problem, since they do not separate the FQA crosscutting concerns that are required by several base components.

In this paper we propose to use an architectural description language (ADL) to document and formally specify all the different kinds of dependencies between concerns belonging to the same or to different FQAs. We follow an aspect-oriented approach in order to separate crosscutting concerns of FQAs into separate modules (i.e. *aspects* in aspect-oriented terminology). The language proposed to model FQAs is AO-ADL [Pinto and Fuentes, 2007], an aspect-oriented architectural description language (ADL) that composes (*weaves* in aspect-oriented terminology) crosscutting concerns inside an *aspectual connector*, but separately from base components. Other aspect-oriented approaches have already been used to model FQAs [Noda and Kishi, 1999] [Moreira et al., 2002]. They usually model a FQA simply by adding a single *aspectual component* to the architecture.

However, modeling a FQA using aspect-orientation is more complicated than just adding an *aspect* to our architecture. Not all the FQA concerns are crosscutting, so only those which are should be modeled as aspectual components (i.e. as aspects), and the others should be modeled as base components. Also, some levels of parametrization are required in order to specify the dependencies between the FQA concerns and the application components, and also those dependencies between the components modeling concerns that belong to different FQAs. In this sense, we have defined a process for specifying aspect-oriented and parameterizable architectural patterns, which will be instantiated in this paper for the usability quality attribute. AO-ADL allows the parametrization of architectural patterns by means of connector templates, encapsulating the weaving information between base and aspectual components. These architectural patterns can be stored in a repository, ready for (re)use. The AO-ADL Tool Suite provides support for all this functionality. The benefit of this proposal is that it provides a language support flexible enough to separate and inject (re)usable crosscutting concerns modeling FQAs in a non intrusive way at architectural level.

After this introduction, we outline the motivations of our approach in Section 2. Then, the AO modeling process is presented in Section 3 using the usability FQA, and the support provided by the AO-ADL language to implement the process is described in Section 4. Then, Section 5 discusses the AO-ADL Tool Suite, which supports the definition and instantiation of AO-ADL architectural templates. Finally, in Section 6 we discuss the main benefits and shortcomings of our approach and in Section 7 we present our conclusions.

2 Motivating case study: usability

In this section we describe the motivation for our approach, discussing the issues raised in the introduction in more detail. We have chosen the usability attribute because: (1) it is a complex attribute that can be decomposed into several concerns, with many dependency relationships; (2) it has an important impact on the base functionality of the system; (3) it is usually tangled or scattered with other system functionalities; (4) it depends on several other FQAs; and (5) it is a recurrent FQA often present in many software systems. In this section we will show how all these complex interdependency relations are poorly documented in current approaches, how scattered the FQAs concerns are among base functional components, and we will describe why it is necessary to define reusable and parameterizable architectural patterns ready-to-use in different applications.

2.1 The Usability Attribute

Usability is an essential software attribute, and as such, guidelines have been published both in the ISO standards [ISO 9241-11, 1994] [ISO 9126-1, 2000], and by different individuals and organizations. However, most usability related concerns are usually discovered very late in the software development process, for instance, during testing and deployment [Folmer, 2005]. Recently, the implications for the usability of the application core have been explicitly highlighted from a software engineering perspective, with several approaches recognizing the importance of addressing usability at the requirements [Juristo et al., 2007] and architectural stages [John et al., 2004] [Harrison and Avgeriou, 2007].

Specifically in [Folmer, 2005], a framework has been defined for extracting architectural information related to usability. As illustrated in Figure 1, the basic idea of this framework is that architectural *Usability Patterns* are defined in the solution domain in order to satisfy a list of identified *Usability Properties*. These usability properties help to satisfy the *Usability Attributes* identified in the problem domain. For instance, the framework defines that the *Progress Indication*, *Alerts*, *Status indication* and *History Logging* usability patterns collaborate to satisfy the *Provide Feedback* usability property. Furthermore, this property, together with *Consistency*, *Guidance* and *Explicit User Control*, helps to achieve the *Learnability* attribute identified in the problem domain.

Although considerable work has been done on modeling both usability concerns and dependency relationships among different usability concerns (or properties), much more work needs to be done on the definition of ready-to-use architectural patterns. Firstly, the current format for presenting architectural solutions for FQAs does not help developers understand the peculiarities of each concern. Normally, the patterns descriptions are not very detailed, and they are often too abstract for the software architect who will have to apply them. In general these patterns are poorly documented, specifically they use a tabular view with brief, textual descriptions of the architectural implications. Furthermore, although many pages provide an informal description of FQAs, only small and non complete designs are provided. As a result, the applicability of such approaches is

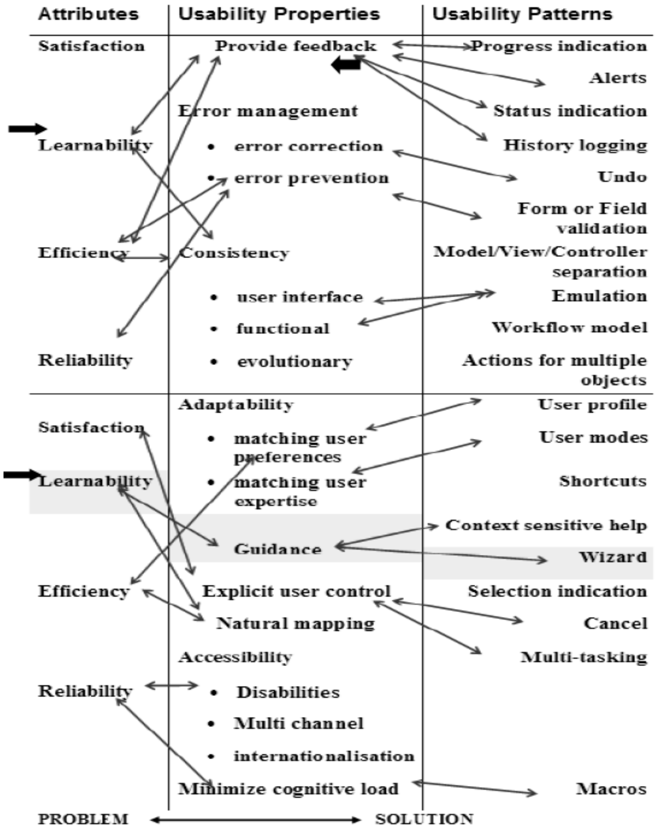


Figure 1: Usability Framework defined in and taken from [Folmer, 2005].

limited, at architectural level. Architectural description languages may be used to specify these architectural patterns more formally and with the advantage of being reusable at the architectural level. Secondly, it is easy to identify that some usability properties affect more than one usability pattern. For example, the *Provide Feedback* property affects both *Status indication* and *History Logging*. This means that the *Provide Feedback* property cuts across and is tangled with both *Status indication* and *History Logging*. So, some kind of "crosscutting" is present in many usability properties. Aspect-Oriented Software Development (AOSD), and specifically the Aspect-Oriented Software Architecture (AOSA) may help to alleviate this problem, because it aims to remove crosscutting and tangling by means of aspects (or *architectural aspects* at architectural level). Finally, this framework has been specifically designed for usability, so all the concerns related to usability were considered. Nevertheless, it should be noted that the authors were unaware that many of the concerns related to usability (like *Adaptability* for example) are also part of other quality attributes and as a result, the relationships between these concerns have not been considered in this work.

In general, the constraints that a FQA imposes on the core application and the impossibility of defining completely reusable architectural solutions for some concerns are not always appropriately highlighted. Furthermore, how the architectural solutions are presented does not help users to understand the peculiarities of each concern. The details about different alternatives to satisfy a particular concern, about the dependencies with other concerns and with the core application are not well-documented from the early stages.

2.2 Complexity of FQAs and dependencies among them

The goal of this section is to justify the necessity of specifying parameterizable architectural patterns, in order to model only once such concerns that participate in several FQAs, as well as to specify the relationships between different FQAs. In order to illustrate this, in Figure 2 we have represented several FQAs, the set of related concerns for each FQA, and the interactions between different FQAs. The circles in dark grey are examples of FQAs and the circles in white are concerns identified for each attribute. The dashed lines represent dependencies and/or interactions between FQAs. This graph is not complete and has been constructed based on information provided mainly in [Juristo et al., 2003] and [Barbacci and et. al., 1995].

In Figure 2 it can be seen that most concerns, for example the concerns comprising the usability FQA (alert, progress indicator, undo, authentication, encryption, etc.) have a functional nature. So, these concerns must be added to the architecture before it is possible to reason about, in this case, the usability attribute. Also, for each FQA, we can distinguish between those functional concerns that are exclusive to this FQA, and others that are shared by several FQAs. For instance, the authentication concern (Security FQA) is required to achieve the contextual help concern (Usability FQA). But in practice, these shared concerns are usually modeled from scratch inside a framework modeling a particular FQA. For instance, fault-tolerance is modeled as a concern of the usability attribute in [Juristo et al., 2007], of the security attribute in [Geebelen and et. al., 2008], and of the context awareness attribute in [Geebelen and et. al., 2008]. So, if we add the usability, security and context aware attributes to the same application, the concerns related with fault-tolerance would be triplicated, which is not acceptable from the architectural point of view.

In conclusion then, there must be a clear separation of the concerns belonging to several FQAs and those that are exclusive to one particular FQA. Those concerns needed to satisfy several FQAs must be modeled separately and only

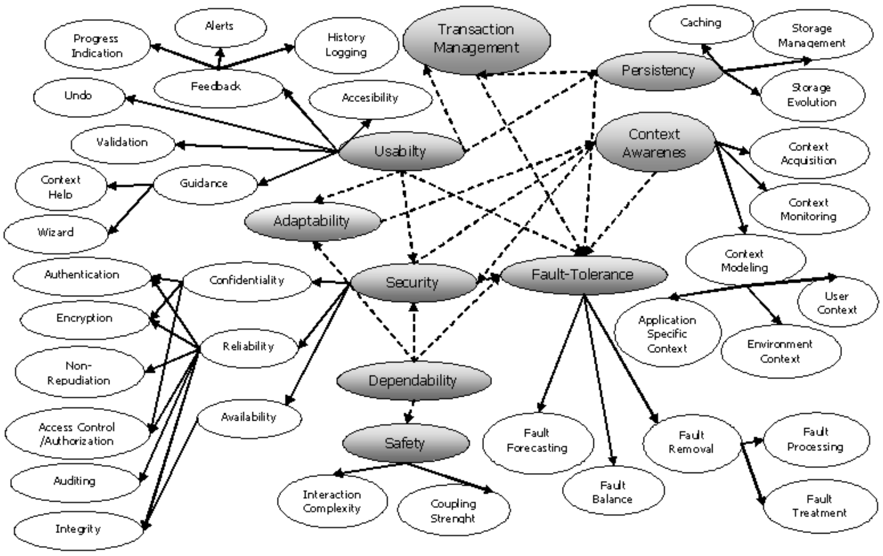


Figure 2: Concern and Dependencies Quality Attribute Graph.

once. It is also important that the decomposition of concerns for each FQA should be independent of the application, and therefore it will be possible to define a repository of (re)usable solutions (reusable architectural pattern) for each FQA. Also, the relationships between different FQAs normally do not depend on the final application, so they can also be modeled as pre-fabricated solutions (reusable architectural patterns). But there are situations in which these architectural patterns need to be parameterized. We have identified at least two situations: (1) applications may demand different levels of quality (i.e. only a subset of concerns of a FQA should be incorporated into the application); (2) some components modeling a concern, require data that must be provided by the application. In the first case, it would be better to provide solutions by means of a family of FQAs, adaptable to the necessities of final applications. Therefore, it should be possible to generate specific FQA configurations from a family of architectural patterns. One possible solution to instantiate a configuration of a family of FQAS is the parameterizations of architectural patterns, by means of optional concerns. The second situation that needs parameterizations is that components of the detailed model of an FQA concern depend on the characteristics of each particular application, – i.e. the designs and/or implementations of the services provided/required by that component cannot be directly reused without particularizing them for each core application. For instance, the data formatting and the data consistency concerns of the usability FQA will have to be provided by each particular application because they will depend on the type of data being manipulated by each application. This can be indicated by using parameters in the definition of those concerns. The parameters are then instantiated with particular interfaces/components when the FQA solution is used in a particular context. In this paper we focus on the latter situation.

2.3 Dependencies of FQAs and base applications

In this section we discuss a component-based modeling versus an aspect-oriented modeling of FQAs. We distinguish between *core components* and *quality-related components*. The former are the components that model the core functionality of the system, without any consideration of quality attributes. The latter are the components that model the additional functionality that needs to be incorporated for satisfying the system quality attributes. For instance, in a shopping cart application, the ShoppingCart, ClientCredit and ProductStock components would be the core components. If usability is required, and a Feedbacker component is incorporated to achieve it, this would be a quality-related component.

As will be shown with the examples below, in a component-based (CB) software architecture, the core and the quality-related components are highly coupled. For instance, if the feedback property needs to be included in a system, the core components are responsible for sending the information to be fed back and for the views showing this feedback information, which are sent by the model to the user. However, this involves modifying the interfaces and behaviors of core components when a new functionality related to the quality attributes needs to be incorporated or updated.

In Figure 3 we have modeled in UML 2.0 the feedback architectural pattern defined in [Juristo et al., 2003]. Applying this pattern to the shopping cart application, the ShoppingCart, ProductStock and Credit components are the *active processes* that generate the information that needs to be fed back to the user. The feedback functionality in the user view is represented by the FeedbackInfo interface, according also to the pattern in [Juristo et al., 2003]. Finally, the Feedbacker component may require updating not only the user view but also other components in the model (represented by System components, once again according to the pattern in [Juristo et al., 2003]). Observing this pattern we detect that feedback is clearly a crosscutting concern:

1. It is tangled with all the active processes in the application model, which need to be aware of the events that need to be fed back to the users.
   (a) All the active processes require the FeedbackEvents interface, including, for instance, the alert() event.
   (b) The ShoppingCart component may generate the alerts: alert("Adding new product") or alert("Deleting product"). The ProductStock component may generate the alerts: alert("Checking stock") or alert("Not enough stock").

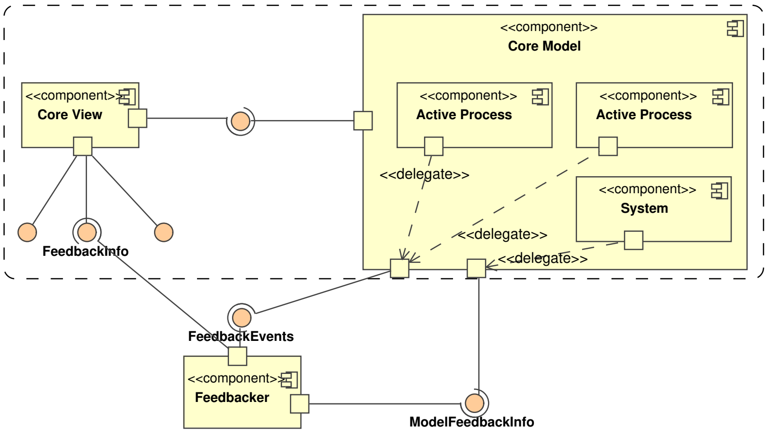


Figure 3: UML 2.0 adaptation of the Feedback pattern described in [Juristo et al., 2003].

(c) This dependency between the core and the quality components would occur even if the architectural pattern is later designed and implemented using monitor components [Buschmann et al., 1996], or using architectural styles such as the blackboard style that allows asynchronous messaging [Buschmann et al., 1996].

2. It is tangled with the core components in the application view. Feedback-specific view interfaces (see the FeedBackInfo interface in Figure 3) need to be defined to provide the feedback information to the user.

3. It may be tangled with other components in the application model that, without being active processes generating feedback information, need to be aware of the feedback information generated in other parts of the system (see ModelFeedBackInfo in Figure 3).

Modeling feedback is even more complicated if we consider that according to the usability framework in Figure 1, this property must be decomposed into several functionalities, such as progress indication, alerts, status indication and history logging. Moreover, these components would have different behaviors, monitoring different kinds of feedback-related events that may occur in the model and they would show different kinds of information to the user. This means that the model is even more complicated, and the Feedbacker component should be in fact modeled as a composite component that includes the ProgressIndication, Alerts, StatusIndication and HistoryLogging sub-components, able to monitor different sets

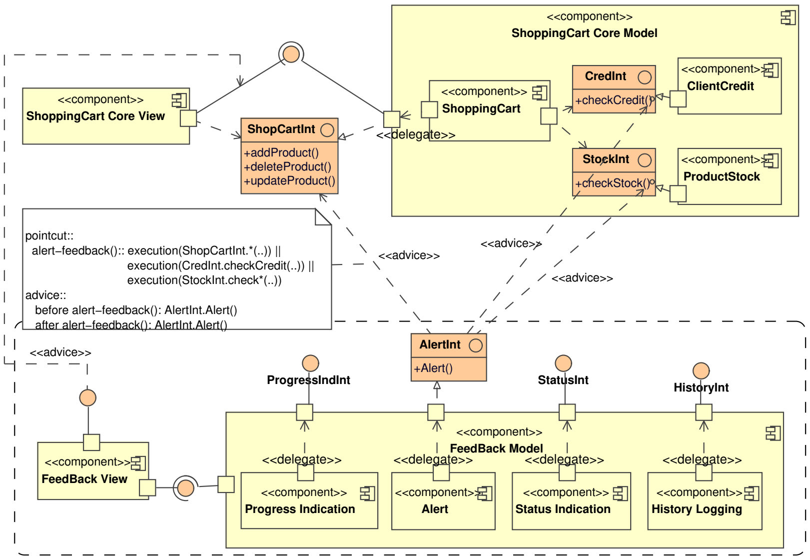


Figure 4: Use of the AO Feedback Property in a ShoppingCart Application.

of events for each variant of the feedback property. This approach would complicate even more the design of the active processes defined in [Juristo et al., 2003].

Aspect-oriented (AO) software architectures avoid this shortcoming by modeling the core components of the application and the quality-related components completely independently of each other. This means that the required interface named FeedbackEvents that appears in the active processes of Figure 3 is no longer required in the AO version of the modeling (see top right of Figure 4), where the core components of the system make no explicit reference to the feedback property, neither in the view nor on the model. Instead, the focus is now put on the sub-components modeling the core application and in the interactions among them.

In Figure 4 we have modeled the relationship between the feedback components and the core components as follows:

1. In the shopping cart application we model the interaction between the view and the model (interface ShopCartInt), the interaction between the ShoppingCart and the ClientCredit component (CredInt interface) and the interaction between the ShoppingCart and the ProductStock component (interface StockInt). In the AOSD terminology these are examples of *join points* – i.e. points of a model or programming language that can be intercepted by aspects. At the architectural level, examples of join points are the interaction between two components through a particular interface, the reception of a

message/event in a particular component, the modification of a component state attribute, etc. The only constraint is that following a black-box component model [Buschmann et al., 1996], at the architectural level, only the join points explicitly exposed by the public interfaces of components can be part of the join point model.

2. All the sub-components in the Feedback Model composite component model aspects. Their provided interfaces (the ProgressIndInt, AlertInt, StatusInt and HistoryInt interfaces) model the aspect *advice*. In the AOSD terminology, the aspect advice is the behavior that is injected by the aspect into the core model. In Figure 4, we have used a UML dependency relationship with the <<advice>> stereotype to indicate this special kind of relationship between the quality-related components and the core components. Notice that there are different AO extensions to UML following different approaches to model the AO concepts [Chitchyan and et. al., 2005]. Here, our only purpose is to explain the motivation of our approach. In section 4 we describe how to represent these concepts in our AO-ADL language.

3. The anchor note linked to these dependency relationships specifies a *pointcut*. In the AOSD terminology, a *pointcut specification* captures one or more join points that need(s) to be advised by an aspect. In our example, the join points affected by the Alert component are: (1) the reception of any message defined in the ShopCartInt interface; (2) the reception of the checkCredit() message in the CheckInt interface, and (3) the reception of any message beginning with check in the CreditInt. We have used the notation defined by AspectJ [Laddad, 2003], one of the most popular AOP languages.

4. The advice specifications in the anchor note indicate that the alert() message in the *AlertInt* interface will be injected *before* and *after* the join points specified in the pointcut specification. In the AOSD terminology, the *type of advice* specifies the place in which the aspect advice is injected into the core behavior. There are different types of advice, though the most common are *before*, *after* and *around* the core behavior. In our example, this means that the alert() advice in the Alert aspect will be executed before the addProduct(), delProduct() or updateProduct() messages are received by the ShoppingCart component. The same advice will also be executed after the messages are processed by the ShoppingCart component. Moreover, it will also be executed before and after the checkCredit() message is received by the ClientCredit component and before and after the ProductStock component receives any message with a name beginning with "check".

In summary, the core components in the AO version do not need an explicit reference to feedback because aspects are able to intercept the traditional interactions or communications between components (for instance, the interaction between the view and the model using the ShopCartInt), and more importantly, the components are completely unaware of these actions.

Therefore, one of the primary aims of this work is to review the architectural patterns identified in [Folmer, 2005], providing aspect-oriented versions of such patterns using the AO-ADL aspect-oriented architectural description language.

3 AO Modeling Process

Figure 5 shows an AO Modeling Process that defines the main tasks to model an AO architectural solution (previous versions at [Pinto and Fuentes, 2008a] [Pinto and Fuentes, 2008b]). This process was defined independently of a particular aspect-oriented modeling language. As we already stated in the introduction, most aspect-oriented solutions model FQAs focusing only on their crosscutting nature. So, these solutions normally propose to model a FQA by a more or less complex aspectual component. In this process we explicitly define some tasks for documenting important information and decisions that need to be considered to identify aspect-oriented architectural solutions. This process takes into account all the types of dependencies identified in Section 2 and in particular, it enforces the (re)use of previously defined aspect-oriented architectural patterns modeling FQAs. So, the main contribution of this process is to serve as a guide to model FQAs using any aspect-oriented architectural approach. In this section we will illustrate the different tasks of this process in modeling the usability FQA. For simplicity we will only model a subset of concerns comprising the usability FQA defined in previous sections.

The input to our process is the usability taxonomy shown in Figure 1. This is an iterative process where the specification of a reusable model for each relevant concern of the usability FQA is the central activity of the process. In order to illustrate the activities of *Model the concern* task, we will describe the contextual help concern. Table 1 and Figure 6 show the architectural implications, the tangled/scattered behavior, the dependencies with other usability concerns, the dependencies with other FQAs and the dependencies with the core application for this concern.

Architectural implications. In this task we will identify the functionality that a concern must provide, and the corresponding decomposition into components. Specifically, we separate the functionality of the contextual help concern into three main components, according to a classical 3-tier architectural style. The Contextual Help View component models the graphical information with which an application needs to be extended in order to incorporate this concern. The Help Content component models the help information that will be displayed for a specific application. Finally, the component Contextual Help Aspect is in charge

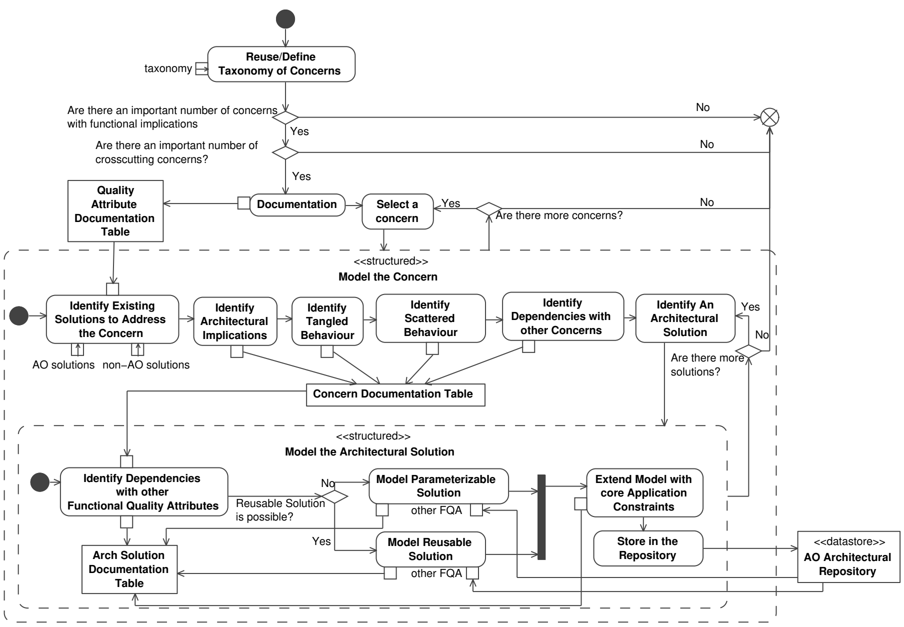
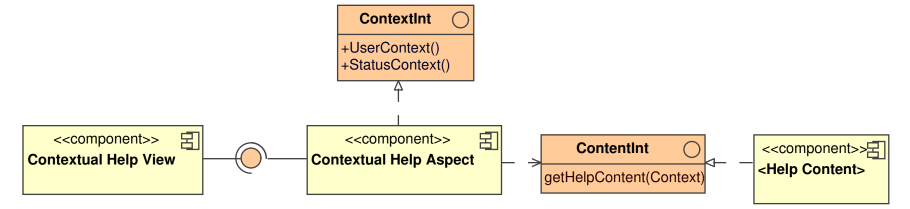


Figure 5: Activity Diagram of the AO Modeling Process

of intercepting the operations of the application that need help. In addition, since the help information displayed depends on the context, it will also have to intercept operations of other FQA concerns related to the context (e.g. user role obtained by the security FQA). In the next step we will decide if the relationships between the Contextual Help Aspect component and both the core application and other FQAs will be aspectual or non aspectual. All these components form the Contextual Help Model of Figure 6.

Tangled/Scattered Behavior. In this step we have to identify if the functionality of the Contextual Help Model composite component crosscuts the core application, as well as components modeling other usability concerns and other FQAs (see Figure 6). In order to inject the usability framework into the application in a non intrusive way, we will define aspectual relationships between the usability framework concerns and the rest of the application components. Firstly, any operation/service of the core application for which contextual help has to be provided will be intercepted by the aspectual sub-component Contextual Help Aspect of the Contextual Help Model component. This relationship is stereotyped with <<advice>> denoting that the advice StatusContext will be executed for

Table 1: Relevant Information for the Contextual Help Model



Architectural implications	- 3-layer architecture to model contextual help: Data/Model (Help Content), View (Contextual Help View) and Controller (Contextual Help Aspect). - The context must be defined in terms of user information and application status. - Help content provided based on user/status context.
Tangled/scattered behaviour	- This sub-concern crosscuts all the components in the core application, as well as other usability sub-concerns and other FQAs.
Dependencies with other usability concerns	Help information needs to be provided for the new functionalities incorporated by the rest of Usability concerns.
Dependencies with other FQAs	Dependencies with the Security FQA: - Users need to be enrolled in the application. - Information about the role played by the user needs to be available to achieve this concern. Dependencies with the functionalities incorporated by any other FQA.
Dependencies with the core application	- The Contextual Help Aspect will advice any service provided/required by the core application susceptible to need help information. - The core view needs to be augmented with the contextual help view.
Alternative architectural solutions	- Reference to a particular security framework versus a generic representation of security by parameterization and later instantiation with different security framework candidates. - Advice-like dependency between contextual help and security versus interaction-like dependency.

every operation requiring contextual help. Also, the graphical interface of the core application must be extended with the functionality encapsulated by the Contextual Help View sub-component of the Contextual Help Model component. In aspect-orientation this can be done in a non intrusive way by using the relationship stereotyped as <<introduction>>, able to 'introduce' new behavior to a component without modifying it. Below, we will comment on the 'crosscutting' relationships between this concern and other concerns from the same or from different FQAs.

Dependencies with other usability concerns. Now we have to specify the connections (dependencies) between all the components of the usability FQA. These connections could be aspectual or non aspectual. Basically, three kinds of dependencies between concerns have been identified at the architectural level: (1) a concern requires a service from other concern(s); (2) a concern provides a service to other concern(s), or (3) a concern is composed with other concern(s) in an aspectual manner (<<advice>> relationship). In our example, only aspectual dependencies of type (3) were identified (e.g. between the contextual help

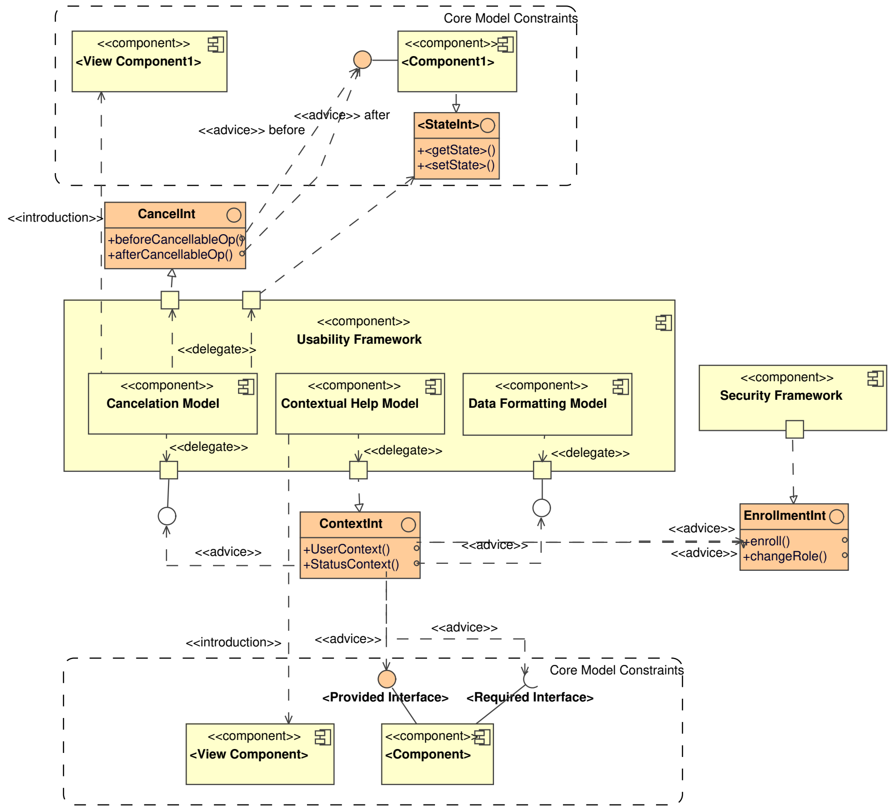


Figure 6: An example of the dependencies identified for Usability

concern and the rest of the usability concerns). The contextual help concern crosscuts those concerns for which help information needs to be provided. As an example, in Figure 6 we show that the contextual help concern also intercepts operations of the cancelation and the data formatting concerns.

Dependencies with other FQAs. In the motivation section we already identified dependencies between different FQAs as an important task. Coming back to our example, in Figure 6 we only modeled the dependency between the contextual help concern and the security FQA, for reasons of simplicity. Specifically, the user needs to be enrolled in the system and information about the role played by the user is required in order to contextualize the help provided. So, the role of the user (EnrollmentInt interface) will be used by the contextual help component to show the help information according to the user expertise. At this point we decided to compose the security framework with an <<advice>> relationship, in order to capture the user role without the knowledge of the security framework concerns.

Dependencies with the core application. The composition of an FQA model and the architectural model of the core functionality of an application may require that this core architecture satisfies certain constraints (e.g. that it exposes their components' state). As previously mentioned, these constraints should also be modeled and documented when modeling the FQA. At this point we should identify which part of the architectural solution for the usability FQA can be added (i.e. reused, Reusable Solution) to the application as it is, or if some kind of parameterizations are needed (Parameterizable Solution). We will focus on those cases needing parametrization. We use <> to indicate that a name of a component/interface/operation is a parameter. In our example, in the Contextual Help Model component, the Help Content will be provided by the core application, which needs to provide the file (or any data store) with the appropriate help information, by means of a parameter of this component. In the lower part of Figure 6, the <View component> is defined as a parameter of the Contextual Help view sub-component that will extend it. Similarly, application components requiring help information will be defined as parameters of the Contextual Help Model component. Then, these parameters will be instantiated to application-specific components, interfaces and operations for each application to which usability is added. More complex dependencies with the core application can also be modeled. As an example, upper part of Figure 6 also shows the dependencies of the cancelation concern with the core application. Concretely, canceling an action is not possible unless components affected by the cancelation expose their state. If as a best practice, all components specify a 'setState' and 'getState' operation, then it is possible to reuse the cancelation concern as it is. But, if we are not sure of this, then it is possible to define an interface specifying the components state as a parameter (<StateInt>). Finally, the complete FQA specification needs to be stored in an AO repository of (re)usable architectural solutions. This is an important contribution that considerably increases the possibilities of reusing the FQA models in different contexts, since all the dependencies either with the core application or with other FQA concerns are considered. Moreover, models are available to be directly (re)used by parametrization and instantiation for different core applications.

Of course, there is no single architectural solution to model FQA concerns. Alternative solutions may also be considered when modeling the same concern. As an example, for the contextual help concern an alternative solution will be: instead of defining an advice-like dependency between contextual help and security, an interaction-like dependency can be defined. This means that the usability

Table 2: AO-ADL support

Architectural implications	AO-ADL is an AO architecture description language to model the component&connector view of software architectures. The main architectural building blocks are components, connectors and configurations. - Using *composite components* different levels of abstractions can be modeled. - Using *aspectual connectors* the aspectual interactions between base and aspectual components can be modeled. - *Configurations* model particular architectural solutions.
Tangled/scattered behaviour	- The semantic of AO-ADL connectors is extended to model aspect-oriented information. The <<advice>> relationship is modeled using the aspectual role and the aspectual binding extensions of connectors. - The <<introduction>> relationship is modeled using composite components.
Dependencies with other concerns of the same FQA	Connectors are used to model these dependencies. - Traditional connectors are used to model *interaction-like* dependencies. - Aspectual connectors are used to model *advice-like* dependencies. *Composite components* are used to model high-level representations of other concerns of the same FQAs and of other FQAs.
Dependencies with other FQAs	Ditto for 'Dependencies with other concerns of the same FQA'. Additionally, the models of other FQAs may be parameterized in order to postpone the decision of which particular FQA framework to instantiate.
Dependencies with the core application	Ditto for 'Dependencies with other concerns of the same FQA'. Additionally, in this case, the elements (operations, interfaces, components) modeling the core constraints need to be parameterized.
Alternative architectural solutions	The different alternatives to model an FQA can be stored in the AO-ADL architectural templates repository ready to be instantiated and (re)used.

concern would require the security service by directly interacting with it.

4 The AO-ADL Approach

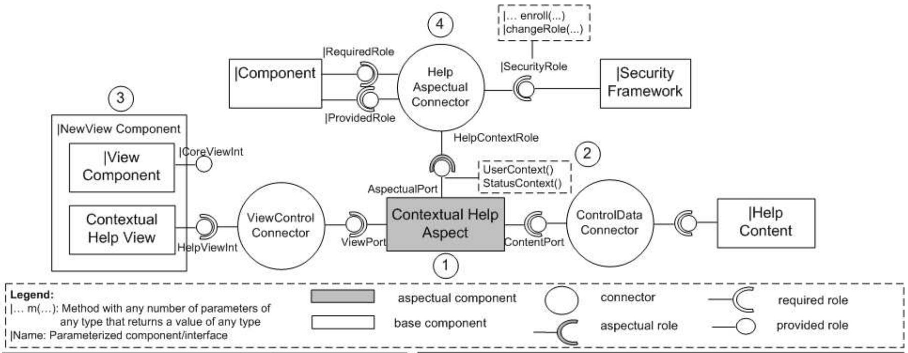
In order to support the AO software process described in the previous section, we need to use an architectural modeling approach able to manage all the issues discussed in Table 1 (see first column). In this section we discuss how the AO-ADL [Pinto and Fuentes, 2007] language satisfies these requirements (see Table 2).

Architectural Implications with AO-ADL. AO-ADL is an aspect-oriented architecture description language that allows the specification of the component&connector view of software architectures. In AO-ADL both crosscutting and non-crosscutting concerns are modeled by components. Thus, similarly to other ADLs the main elements of AO-ADL are interfaces, components, connectors and configurations. In Figure 7 we partially show the AO-ADL specification of the contextual help model shown in Table 1. We can observe that components are specified by means of their ports (ports are like interfaces), which can be provided or required (label 1 in the XML code). Each port is linked to a particular interface (label 2 in the XML code). Components can be single or composite ones. Composite components allow creating models at different levels of abstraction.

They are also defined in terms of their provided and required interfaces, where each of their ports is attached to the internal subcomponents (label 3 in the XML code). Connectors (label 4 in the XML code) encapsulate the communication and interaction between components. The interfaces of connectors that bind components' ports are called roles (more details of these are provided below). Finally, configurations allow the specification of a sub-architecture comprising the set of component and connector instances and the attachments among them. The syntaxis of configurations is similar to the configuration section of composite components (label 3 in the XML code), so the configuration of the usability framework in AO-ADL is not shown here.

Tangled/Scattered Behavior with AO-ADL. There are at least two main issues to be considered in an aspect-oriented approach: (1) how the tangled/scattered behavior is modeled, and (2) how the aspectual relationships are modeled. Regarding the first issue, since in AO-ADL the same building block is used to model base and aspectual behavior, the distinction as to whether a component is acting as a base or aspectual component is performed depending on the particular 'role' the component is playing in a particular composition. 'Base components' are those connected to either a *provided role* or to a *required role* of a connector, as in traditional ADLs [Medvidovic and Taylor, 2000]. For instance, the |Component and |Security Framework components in Figure 7 are base components connected to the provided/required roles of the Help Aspectual Connector connector (lines 4.2 to 4.4 in the XML code). Components are considered to be 'aspectual' when they are connected to an *aspectual role* of a connector – i.e. the semantics of connectors is extended with a new kind of role, the aspectual role. An example of an aspectual component is the Contextual Help Aspect component connected to the aspectual role HelpContextRole of the Help Aspectual Connector connector (lines 4.6).

Regarding the second issue, AO-ADL connectors include an *aspectual bindings* section used to model the *advice* relationship of section 3. These bindings specify both the places where the aspectual behavior will be injected into the base components – i.e. the pointcut specifications, and the aspectual behavior that will be injected – i.e. the advice specification. For instance, the specification of the Help Aspectual Connector connector in Figure 7 defines two aspectual bindings, the SecurityBinding (line 4.9) and the CoreApplicationBinding (line 4.19). An aspectual binding specifies the set of join points that will be intercepted (<pointcut–specification>) and the when the aspectual behavior (the advice) will be injected (binding order). In AO-ADL, it is possible to use wildcards like '*' to specify names of components, interfaces, etc. that are part of a pointcut definition. The type of advices that are possible to use in AO-ADL are similar to those of AspectJ, like before, after, etc. (see line 4.13). We will explain the meaning of




```
<component name="ContextualHelpAspect">
   <provided_interface
      uri="//interface[@name='ContextInt']"
      portName="AspectualPort"/>
   <required_interface uri="..."
      portName="ContentPort"/>
   <required_interface uri="..."
      portName="ViewPort"/>
</component>
```
1

```
<interface name="ContextInt">
     <operation name="UserContext"/>
     <operation name="StatusContext"/>
</interface>
```
2

```
<compositeComponent name="|NewView Component">
  <provided_interface type="MSG"
                     portName="HelpViewInt">
     <attachment component="ContextualHelpView"/>
  </provided_interface>
  <provided_interface type="MSG"
                     portName="|CoreViewInt">
     <attachment component="ViewComponent"/>
  </provided_interface>
  <configuration>
     <component instance_name="ContextualHelpView"/>
     <component instance_name="|ViewComponent"/>
  </configuration>
</compositeComponent>
```
3

```
<connector name="HelpAspectualConnector">
 4.2 <provided_role roleName="|ProvidedRole"/>
 4.3 <required_role roleName="|RequiredRole"/>
 4.4 <required_role roleName="|SecurityRole"
        role_specification="methods_of[//interface[@name='SecurityInt']]"/>
 4.6 <aspectual_role roleName="HelpContextRole"
        role_specification="//interface[@name='ContextInt']" />
     <aspectualBindings>
 4.9   <aspectual_binding name="SecurityBinding">
 4.10     <pointcut_specification>
            <pointcut>//required_role[@name='|SecurityRole'] and //operation[@name='*']</pointcut>
          </pointcut_specification>
 4.13     <binding order="first" operator="after">
 4.14       <aspectual_component aspectual_role_name="HelpContextRole">
 4.15         <advice label="UserContext"/>
            </aspectual_component>
          </binding>
        </aspectual_binding>
 4.19   <aspectual_binding name="CoreApplicationBinding">
 4.20     <pointcut_specification>
            <pointcut>//provided_role[@name='|ProvidedRole'] or
                //required_role[@name='|RequiredRoleole']</pointcut>
          </pointcut_specification>
 4.24     <binding operator="before">
 4.25       <aspectual_component aspectual_role_name="HelpContextRole">
 4.26         <advice label="StatusContext"/>
            </aspectual_component>
          </binding>
        </aspectual_binding>
     </aspectualBindings>
</connector>
```
4

Figure 7: Architectural template in AO-ADL for the Contextual Help Model

these two bindings in the context of the example described above.

Dependencies with other concerns of the same FQAs. In AO-ADL, these dependencies are modeled in connectors. The connector will be an aspectual connector (with aspectual roles and aspectual bindings sections) when the dependency is identified as an *advice-like* dependency. Otherwise, a traditional connector will be used. When needed, composite components will be used to model high-level representations of part of the models. In Figure 6 the Contextual Help Model has a dependency with the Cancelation Model and the Data Formatting Model components. Instead of defining separate bindings between these components and the Help Aspectual Connector we define one single binding, in which the component requiring help as well as the corresponding provided and required roles where the component must be attached are defined as parameters. So, later, the |Component, |ProvidedRole and |RequiredRole parameters will be instantiated with the Cancelation Model and the Data Formatting Model components affected by the contextual help concern. We use | before the names of roles, components, etc. to indicate that this is a parameter. This is only a syntactic sugar used to simplify the description of the templates. In the next section we describe how AO-ADL templates are implemented in our toolkit.

Dependencies with other FQAs. These dependencies are modeled in the same way as the dependencies with other concerns of the same FQAs. In our example, the Security Framework component provides information to the Contextual Help Aspect component, through the aspectual binding SecurityBinding (line 4.9). This binding specification indicates that the UserContext advice (lines 4.14–4.15) is injected 'after' (after operator in line 4.13) the interception of 'any operation of the required role' SecurityRole (pointcut specification in line 4.10). In other words, user information needed to provide contextual help is intercepted by this aspectual component 'after' the user is enrolled in the system or his/her role changes. Note that in this case, to postpone the decision of which particular Security FQA framework to instantiate, we define the |Security Framework component and the operations associated to the SecurityRole role as parameters. This kind of parametrization could be applied to any other FQA.

Dependencies with the core application. Once again, these dependencies are modeled similarly to the dependencies with other concerns of the same FQA. In our example, as other concerns of the usability FQA, components of the core application also require contextual help. Since the architectural solution already proposed for other usability concerns is parameterizable, it can be reused for all the components of the core application. Thanks to the parametrization therefore we have defined a generic and (re)usable architectural solution for the contextual help concern, which can be (re)used for any kind of architectural components. Any component (denoted by |Component) attached to the |ProvidedRole and |RequiredRole roles of the Help Aspectual Connector connector will be composed by the aspectual binding CoreApplicationBinding (lines 4.19–4.26). This aspectual binding specifies how the StatusContext advice intercepts the join points in the core application, or in any other components, in order to provide the appropriate contextual help. The other aspectual relationship (the *introduction*) is modeled in AO-ADL creating a new composite component that encapsulates as internal components the two components participating in the relationship (see the |NewView Component, label 3, in Figure 7), which encapsulates the ContextualHelpView component and the |View Component component in the core application.

Summarizing, the most important contributions of AO-ADL, which make it suitable to model complex FQAs, are aspectual connectors and architectural templates. As shown in the examples, architectural templates model a reusable sub-architecture (or 'configuration' in the ADL terminology) for a particular quality attribute or complex concern. Architectural templates allow a high level of parametrization since interfaces, components, connectors and even the attachments among them can be parameters of a given sub-architecture. Template parameters will be later instantiated for a particular application domain, by the AO-ADL toolkit, allowing the complete reuse of the aspectual bindings in different contexts. Moreover, different alternative templates for the same FQA can be modeled and made available through the repositories of the AO-ADL toolkit. The AO-ADL toolkit is detailed in the next section.

5 The AO-ADL Tool Suite

The AO-ADL language is supported by the AO-ADL Tool Suite, implemented as an Eclipse plug-in [2]. As shown in Figure 8, the toolkit is organized in several modules that communicate by means of different repositories.

Firstly, the toolkit provides support to specify software architectures in AO-ADL using the Component&Connector Editor and the Architecture Configuration Editor modules. The main responsibility of the Component&Connector Editor is the definition of (re)usable components and connectors, which are stored in the Component&Connector Repository. These architectural building blocks can be imported through the Architecture Configuration Editor during the description of a software architecture configuration.

Secondly, the same editors can also be used for the definition of (re)usable architectural templates, which are stored in the Architecture Template Repository. Any element of an AO-ADL configuration (interfaces, components and connectors) may be defined as a 'parameter' to be later instantiated. As shown in Figure 9 (label 1), an architectural element is defined as a parameter by only marking a box named parameter beside each parameterizable element. The toolkit uses JET code to define and instantiate the parameters of AO-ADL templates, and embeds

[2] It can be downloaded and installed from http://caosd.lcc.uma.es/AO-ADLUpdate

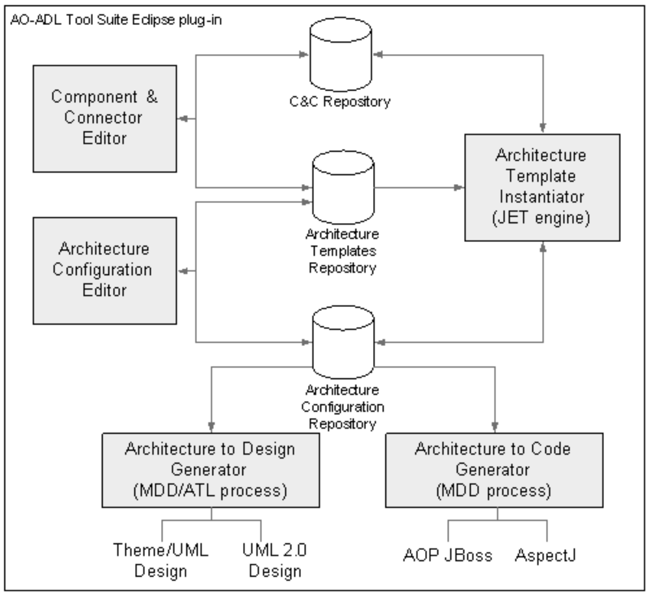


Figure 8: AO-ADL Toolkit

it in the XML definition of the element being defined (label 2). As an example, at the bottom of Figure 9 (label 2), we can observe the JET code automatically generated for the parameters of the connector template of Figure 7.

Thirdly, the Architecture Template Instantiator module semi-automatically instantiates the architectural templates previously stored in the Architecture Template Repository. Basically, this module selects the appropriate configuration template from the repository and instantiates it substituting the formal parameters with the domain specific parameters. The AO-ADL toolkit provides a wizard to guide this instantiation process.

An additional contribution of the AO-ADL toolkit is the use of MDD (Model-Driven Development) [Beydeda et al., 2005] technology to propagate the decisions taken at the architectural level to lower levels of the software life cycle. Concretely, the Architecture to Design Generator module provides support for the automatic transformation of AO-ADL software architectures to UML 2.0, either plain UML 2.0 or aspect-oriented extensions of UML 2.0 as is the case of Theme/UML. These transformations are defined using ATL (Atlas Transformation Language) [Jouault and Kurtev, 2005], which is an Eclipse plug-in for the definition of model-to-model transformation rules. Moreover, the Architecture to Code Generator module also uses MDD technologies to automatically generate a code skeleton in two aspect-oriented languages, AspectJ [Laddad, 2003] and AOP JBoss [JBosss AOP, 2010].

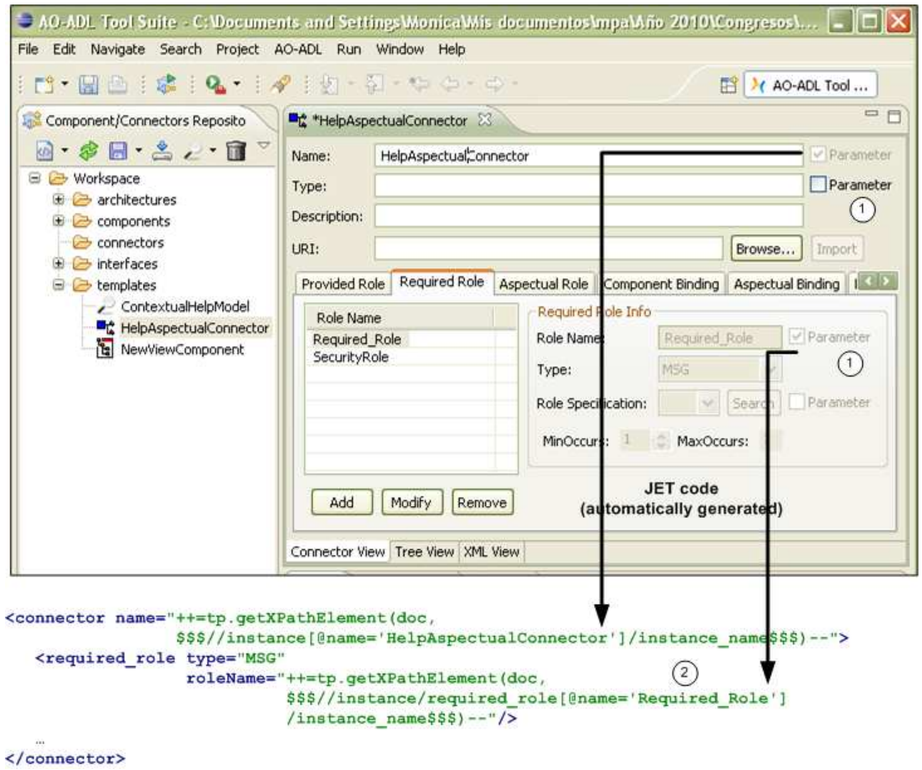


Figure 9: Snapshot of the AO-ADL Tool Suite.

6 Discussion and Related Work

We evaluate our proposal from two points of view. Firstly, we compare our approach with other solutions that also focus on modeling FQAs from early stages of the development. Secondly, we discuss some lessons learned after adding usability to the software architecture of several applications specified in AO-ADL.

6.1 Comparison with Existing Solutions

As we already mentioned in the introduction, our approach is only applicable to those QAs that have important functional implications. For some QAs such as the ones discussed in this paper (usability, security, etc.), which appear in Figure 2, these functional implications are evident. For others, such as cost or standards support, it is clear that there are no functional implications. Finally, there are some QAs [Barbacci and et. al., 1995] that may have some functional implications, but some kind of architectural decisions must also be taken in order to satisfy the QA completely. Examples of these QAs are safety, performance, or dependability, where the functional implications are not very evident and therefore not considered in our approach. For instance, even though there are some functional implications associated to the performance attribute, such as resource allocation, queueing and scheduling, most of the performance concerns are non-functional ones and thus not suitable to be modeled following our approach. For example, a more powerful processor, or any other improvement in the hardware would undoubtedly improve the performance, but would not have any effect on the application architecture.

The importance of modeling the functional part of QAs at the architectural level has been identified in both the AO and the non-AO communities. Examples in the AO community are the studies of crosscutting concerns in [Tanter et al., 2006] [Geebelen and et. al., 2008], which include security, persistence, context awareness and mobility. An example in the non-AO community is [Juristo et al., 2003] [Juristo et al., 2007] where all the functional implications of the usability attribute have been studied and documented. They propose a usability framework that relates abstract usability attributes, such as satisfaction and learnability, with specific functionalities (e.g. wizards, alerts, etc.) and architectural patterns in the system. Further examples are the reports generated from QA workshops [Barbacci and et. al., 1995], where a taxonomy of concerns and factors that influence the satisfaction of those concerns have been defined for security, safety, dependability and performance. These works about FQA taxonomies are a good starting point to formally specify FQAs at the architectural level.

Although it is crucial to start from an accurate and complete taxonomy of concerns modeling FQAs, sometimes this is not enough. Often the frameworks that model FQAs fail to specify at the architectural level all the concerns and dependency relationships identified in the taxonomy. This happens especially in the aspect-oriented approaches. Indeed, one of the contributions of our approach is the definition of an AO process that highlights the necessity of carefully considering the AO modeling of FQAs, specifying both crosscutting and non crosscutting concerns that comprise a given FQA. For example, we have found that experts on persistence have identified concerns such as caching and schema evolution. But these concerns are not usually modeled in AO approaches. Basically, the problem with current AO approaches is that they focus on modeling crosscutting concerns of an FQA, leaving aside those concerns that are not crosscutting, but are required for modeling complete solutions for the FQAs. In our approach we provide complete architectural solutions modeling both the crosscutting and non-crosscutting functionalities required to satisfy a particular FQA. Similarly to our approach, the work in [Landuyt et al., 2009] defines an AO method that tries to improve the modularization and composition of crosscutting concerns at the early stages of the development. Concretely, this method is defined at the requirements and architecture levels focused especially on defining more stable pointcut specifications.

As described throughout the paper, another problem of existing solutions is that they sometimes fail to define completely (re)usable solutions. This is basically because it is not always trivial how the different alternatives for a given FQA must be specified. Firstly, although the software architect may define a complete solution modeling all the concerns of an FQA, then only a subset of those concerns may be required in a particular context. In our approach this is solved using AO-ADL to define different patterns to model the same FQA. Also, with our approach it is possible to instantiate only a subset of the concerns that comprise an architectural pattern modeling an FQA. So, it is possible to define complete architectural patterns for a given FQA, encompassing many concerns following different alternatives, and to instantiate only part of this pattern according to the application requirements. Secondly, some solutions strongly depend on contextual information that must be provided by the core application or by the environment. When this information is not available, that particular solution can not be considered. Finally, some concerns are shared by several FQAs and they should not be repeated in several FQA frameworks. However, after analyzing existing FQA frameworks we have realized that those concerns shared by several FQA are specified repeatedly in each framework.

An example of the above situation is the authorization concern specified in [Geebelen and et. al., 2008]. The description of this concern states that *'there are multiple authorization scenarios and thus the reference monitor can take the decision based on the user secret, on more general information (time or location for example) or on application-specific information'.* Although several alternatives for specifying authorization are described here, the reference architecture that is finally provided in [Geebelen and et. al., 2008] only includes an Authorization, Session Handler and ExceptionHandler components. So, the different alternatives for authorization were encapsulated in only one component (i.e. Authorization). Thus, it can be deduced that: (1) it will be difficult to define a completely reusable architectural pattern coping with authorization under certain circumstances; (2) in the alternative for controlling access based on contextual information such as location, a dependency with the context awareness FQA has been identified. So, instead of defining the location concern as part of the security FQA, at least part of the context aware FQA (dealing with location) must be included here; and (3) for control access based on application-specific information, the relationship with the core application is not clear. In our approach all these problems can be faced by using parameterizable templates, defining for instance, the application dependant information needed to authorize the users as a parameter. Since it is possible to define different sub-architectures for a given FQA, in our proposal it is also possible to define different alternatives to control access, and all these alternatives will be available through the AO-ADL repository.

But not only the AO-ADL language can be used to support our process for modeling FQAs. Other modeling languages can also be used, but they must support: (1) the separate modeling of crosscutting concerns (preferably using an aspect-oriented approach) and, (2) the parametrization of software architectures. UML 2.0 provides support to define templates, so it fulfills the second requirement but not the first one. Several AO extensions of UML exist, one of the most representatives ones being Theme/UML [Clarke and Baniassad, 2006]. Concretely, Theme/UML uses UML templates in the specification of aspects, so it can be used as an alternative to AO-ADL. But, Theme/UML was not designed to model software architectures, being particularly well suited for the detailed design phase. Therefore, we have incorporated Theme/UML into our proposal, to refine the initial architecture specified in AO-ADL with more specific details. The AO-ADL Tool Suite defines an MDD transformation to automatically generate a Theme/UML representation of AO-ADL software architectures. Finally, several AO architecture description languages exist nowadays [Pérez and et. al., 2003] [Garcia and et. al., 2006] [Navasa et al., 2005] [Pessemier et al., 2006]. The main inconvenience is that most of them do not provide enough tool support to define templates and to create the repository of solutions required by our approach.

6.2 Adding Usability to Existing Applications

We have applied the AO architectural patterns defined for usability to the software architecture of a Toll Gate System (TG) [Pinto and et. al., 2008], a Health-Watcher Application (HW) [Pinto et al., 2007] and an Auction System (AS) [Chitchyan and et.al., 2006]. For these applications, an AO software architecture was already available in AO-ADL, but the usability FQA was not explicitly part of these architectures. We have focused on analyzing if the core architecture: (1) already provided some functionality related to usability or not; (2) satisfied all the constraints imposed by the usability architectural patterns, and (3) already modeled some crosscutting concerns related to usability (Table 3). Also, some lessons learned after using the AO-ADL language on supporting our process are discussed in this section.

In the first column of Table 3 we can observe that the TG and the HW already included some functionality related to usability. Concretely, the software architecture of the TG provided some level of user feedback, including two aspectual components in charge of intercepting events of interest and showing them to the user in their corresponding displays. It also includes an error handling aspectual component. For the HW [Pinto et al., 2007] several security concerns

Table 3: Incorporating usability to existing software architectures

	Provides Usability	Satisfy All Constraints	Attributes From Scratch
TG	Feedback, Error Handling	No (Cancelation)	Usability, Security, Fault Tolerance, Persistence
HW	Consistency, Data Formatting, Error Handling	Yes	Usability, Security, Fault Tolerance, Persistence
AS	No	Yes	Security, Fault Tolerance, Persistence

(authentication and enrollment), one error handling concern and several usability concerns (consistency and data formatting) were identified and modeled from scratch. The requirements of the AS however did not include any reference to usability, and thus, any usability related functionality was present in the software architecture defined in [Chitchyan and et.al., 2006].

A software architect that was not involved in the specification of any of these case studies was in charge of adding the architectural patterns for usability defined in this paper. The AS case study does not include any concern related to usability, and thus the pattern was added without needing to modify the core functionality of the system. But, in the other case studies, in which some concerns related to usability were already defined, some level of modification in the core architecture was required. These modifications consisted of removing the components modeling the usability related concerns from the core functionality. But, this only will happen if we are refactoring a software architecture for adding new FQAs. In other cases, following the process proposed in this paper, we would have already identified during the requirement phase that some concerns of the core application also helped to satisfy an FQA, and consequently they could be imported from the repository of FQA architectures of the AO-ADL toolkit. For instance, the data formatting concern identified in the requirements of the health watcher would be an instantiation of the data validation model in Figure 6.

The information in the second column of the table is the result of the incorporation of the usability concerns that impose some constraints on the core application. Here, some limitations where found when we tried to add the cancelation concern to the TG. The problem was that in the architecture we were refactoring, the components implementing the principal actions of the TG system, do not expose their state, so the cancelation concern of the usability pattern was not able to cancel these actions. Fortunately, this constraint on the core application was well-documented both in the process tables and in the proposed architectural solution so we were able to clearly identify that in order to reuse this pattern the original software architecture has to be extended by adding methods exposing the components' state.

Finally, the third column shows that not only the usability attribute was identified in these software architectures. Other FQAs such as security, fault tolerance and persistence were also identified and could benefit from the software process defined in this paper.

Regarding the use of AO-ADL to support our approach, we have been able to specify (re)usable and parameterizable architectural patterns for most usability concerns, although two main limitations have also been identified. The first one is that more mechanisms to model variability in AO-ADL are needed. Thus, although we can use the parametrization of concerns in AO-ADL to model a family of FQAs, AO-ADL was not explicitly defined to model variable software architectures, as in software product lines approaches. In order to solve this limitation we are currently working on combining AO-ADL with VML [Sánchez et al., 2009], a variability modeling language that provides support to instantiate a configuration of a family of products.

The second limitation that have been identified is that a new kind of advice, which is not included yet in AO-ADL, is needed in order to model complex crosscutting behaviors at the architectural level. Concretely, an advice like the cflow advice of AspectJ [Laddad, 2003] is needed in order to model the cancelation concern of the usability FQA. Basically, the model of the cancelation concern includes an aspectual component that will intercept the cancelable actions and will restore the state of the affected components when that action is canceled. However, since an action in one component will normally imply new interactions with other components, canceling that action will also imply canceling the actions generated by those interactions. In this sense, cflow is an advice that allows capturing join points that occur in the control flow of another join point. That is, the aspectual component will intercept the interaction between any two components in the system, but only when they occur inside an action that is considered as cancelable.

7 Conclusions

Modeling an FQA is not a straightforward task, since it can be decomposed into several concerns, with many dependency relationships. One of these dependencies is that part of those concerns can be crosscutting with the core or with other FQAs concerns. In order to mitigate the defects caused by crosscutting concerns, this paper proposes using aspect-orientation as a complementary approach to traditional architectural approaches. An aspect-oriented approach can help to achieve a better modularization by decoupling the core and the quality-related components. We have illustrated the complexity of modeling an FQA, and have defined a process to guide software architects in the use of AOSD to model FQAs that have both a crosscutting nature and important functional implications. This process makes explicit all kinds of dependencies between the concerns modeling an FQA, other FQAs and the core application, proposing architectural solutions to each of them, normally based on AO and parametrization. So, this process and the generated documentation promotes the understanding of FQAs and increases their reusability in different contexts by providing parameterizable architectural solutions. We have illustrated how the AO-ADL Tool Suite was used for this purpose, though other AO architectural approaches satisfying the requirements imposed by our process may also be used.

References

[Bachmann et al., 2005] Bachmann, F., Bass, L., Klein, M., and Shelton, C. (2005). Designing software architectures to achieve quality attribute requirements. *IEE Proceedings*, 152(4):153–165.

[Barbacci and et. al., 1995] Barbacci, M. and et. al. (1995). Quality attributes. Technical Report CMU/SEI-95-TR-021.

[Beydeda et al., 2005] Beydeda, S., Book, M., and Gruhn, V., editors (2005). *Model-Driven Software Development*. Springer.

[Buschmann et al., 1996] Buschmann, F., Meunir, R., Rohnert, H., and Sommerlad, P. (1996). *Pattern-Oriented Software Architecture:A System of Patterns*. John Wiley & Sons.

[Chitchyan and et. al., 2005] Chitchyan, R. and et. al. (2005). Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design. Technical Report AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, Lancaster University.

[Chitchyan and et.al., 2006] Chitchyan, R. and et.al. (2006). Mapping and refinement of requirements level aspects. AOSD-Europe NoE Public Documents (AOSD-Europe-ULANC-24).

[Clarke and Baniassad, 2006] Clarke, S. and Baniassad, E. (2006). *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley.

[Cysneiros et al., 2005] Cysneiros, L. M., Werneck, V. M., and Kushniruk, A. (2005). Reusable knowledge for satisficing usability requirements. In *RE'05*.

[Folmer, 2005] Folmer, E. (2005). *Software Architecture Analysis of Usability*. PhD thesis, University of Groningen,.

[Folmer and Bosch, 2004] Folmer, E. and Bosch, J. (2004). Architecting for usability; a survey. *Journal of Systems and Software*, 70(1):61–78.

[Garcia and et. al., 2006] Garcia, A. and et. al. (2006). On the modular representation of architectural aspects. In *3rd. European Workshop on Software Architecture (EWSA'06)*.

[Geebelen and et. al., 2008] Geebelen, K. and et. al. (2008). Design of frameworks for aspects addressing 2 additional key concerns. Technical Report AOSD-Europe D117, AOSD-Europe-KUL-14.

[Harrison and Avgeriou, 2007] Harrison, N. B. and Avgeriou, P. (2007). Leveraging architecture patterns to satisfy quality attributes. In *ECSA*, number 4758 in Lecture Notes in Computer Science, pages 263–270. Springer-Verlag.

[ISO 9126-1, 2000] ISO 9126-1 (2000). Software engineering - product quality - part 1: Quality model.

[ISO 9241-11, 1994] ISO 9241-11 (1994). Ergonomic requirements for office work with visual display terminals (vdts) – part 11: Guidance on usability.

[JBosss AOP, 2010] JBosss AOP (2010). Framework for organizing cross cutting concerns. Last visited: March 2010.

[John et al., 2004] John, B. E., Bass, L., Sanchez, M. I., and Adams, R. J. (2004). Bringing usability concerns to the design of software architecture. In *Proceedings of the 9th IFIP Working Conference on Engineering for Human-Computer Interaction and the 11 th International Workshop on Design, Specification and Verification of Interactive Systems*, pages 11–13.

[Jouault and Kurtev, 2005] Jouault, F. and Kurtev, I. (2005). Transforming Models with ATL. In Bruel, J.-M., editor, *Satellite Events at the MoDELS Conference*, volume 3844 of *LNCS*, pages 128–138, Montego Bay (Jamaica).

[Juristo et al., 2003] Juristo, N., Lopez, M., Moreno, A. M., and Sanchez, M.-I. (2003). Improving software usability through architectural patterns. In *ICSE Workshop on SE-HCI*, pages 12–19.

[Juristo et al., 2007] Juristo, N., Moreno, A. M., and Sanchez, M.-I. (2007). Guidelines for eliciting usability functionalities. *IEEE Transactions on Software Engineering*, 33(11):744–757.

[Laddad, 2003] Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications.

[Landuyt et al., 2009] Landuyt, D. V., de beeck, S. O., Truyen, E., and Joosen, W. (2009). Domain-driven discovery of stable abstractions for pointcut interfaces. In *Proc. of AOSD 2009*, pages 75–86.

[Medvidovic and Taylor, 2000] Medvidovic, N. and Taylor, R. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transaction on Software Engineering*, 26(1):70 – 93.

[Moreira et al., 2002] Moreira, A., Jo ao Araújo, J., and Brito, I. (2002). Crosscutting quality attributes for requirements engineering. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 167–174, New York, NY, USA. ACM.

[Navasa et al., 2005] Navasa, A., Pérez, M. A., and Murillo, J. (2005). Aspect modelling at architecture design. In *2nd European Workshop EWSA'05*, pages 41–58.

[Noda and Kishi, 1999] Noda, N. and Kishi, T. (1999). On aspect-oriented design: An approach to designing quality attributes. *Asia-Pacific Software Engineering Conference.*

[Pérez and et. al., 2003] Pérez, J. and et. al. (2003). PRISMA:towards quality, aspect-oriented and dynamic software architectures. In *3rd IEEE Intl Conf. on Quality Software.*

[Pessemier et al., 2006] Pessemier, N., Seinturier, L., Coupaye, T., and Duchien, L. (2006). A model for developing component-based and aspect-oriented systems. In *SC'06*, pages 259–274.

[Pinto and et. al., 2008] Pinto, M. and et. al. (2008). Report on case study results. Technical Report AOSD-Europe D118, AOSD-Europe-Siemens-11.

[Pinto and Fuentes, 2007] Pinto, M. and Fuentes, L. (2007). AO-ADL: An ADL for describing aspect-oriented architectures. In *Early Aspect Workshop at AOSD 2007.*

[Pinto and Fuentes, 2008a] Pinto, M. and Fuentes, L. (2008a). Aspect-oriented modeling of quality attributes. In Meersman, R., Tari, Z., and Herrero, P., editors, *Proc. of IWSSA 2008*, volume 5333 of *LNCS*, pages 334–343. Springer-Verlag.

[Pinto and Fuentes, 2008b] Pinto, M. and Fuentes, L. (2008b). Towards a software process for aspect-oriented modeling of quality attributes. In *Proc. of ECSA 2008*, volume 5292, pages 334–337. Springer-Verlag.

[Pinto et al., 2007] Pinto, M., Gámez, N., and Fuentes, L. (2007). Towards the architectural definition of the health watcher system with AO-ADL. In *Early Aspect Workshop at ICSE.*

[Sánchez et al., 2009] Sánchez, P., Loughran, N., Fuentes, L., and Garcia, A. (2009). Engineering languages for specifying product-derivation processes in software product lines. pages 188–207.

[Tanter et al., 2006] Tanter, E., Gybels, K., Denker, M., and Bergel, A. (2006). Context-aware aspects. In *SC'06*, pages 227–242. Springer-Verlag.

[Welie, 2007] Welie, M. V. (2007). The amsterdam collection of patterns in user interface design.