



PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
ESCUELA DE INGENIERIA

# **MODELING ASPECTS WITH UML'S CLASS, SEQUENCE AND STATE DIAGRAMS IN AN INDUSTRIAL SETTING**

**ALEX D. BUSTOS GONZÁLEZ**

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering.

Advisor:

**YADRAN ETEROVIC SOLANO**

Santiago of Chile, March, 2008

© 2008, Alex D. Bustos G.



PONTIFICIA UNIVERSIDAD CATOLICA DE CHILE  
ESCUELA DE INGENIERIA

# **MODELING ASPECTS WITH UML'S CLASS, SEQUENCE AND STATE DIAGRAMS IN AN INDUSTRIAL SETTING**

**ALEX D. BUSTOS GONZÁLEZ**

Members of the Committee:

**YADRAN ETEROVIC S.**

**ROSA ALARCÓN C.**

**SERGIO OCHOA**

**MARIO DURÁN T.**

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the Degree of Master of Science in Engineering.

Santiago de Chile, March, 2008

*To my parents*

## **ACKNOWLEDGEMENTS**

This thesis represents not only my work at the keyboard, so I would like to extend my gratitude to the following who and which made it possible for me to end this thesis.

My parents, have been a constant source of support –emotional, moral and of course financial– during all my life, and this thesis would certainly not have existed without them.

I have been indebted in the preparation of this thesis to my advisor Yadrán Eterovic, whose patience and kindness, as well as his academic experience, have been invaluable to me.

A very special thanks to my fiancée Cynthia, who continues to encourage and challenge me to end this work.

To my friends Fernanda, Cony, Cata, Raúl and Cristian, to be with me all the time.

Finally, to Starbucks, for providing the best frappuccinos, a non so good internet connection and a comfortable place to share ideas and discuss about this thesis.

## INDEX

	Page.
DEDICATORY .....	ii
ACKNOWLEDGEMENTS.....	iii
IMAGES INDEX.....	v
RESUMEN .....	vii
ABSTRACT .....	viii
1 INTRODUCTION .....	9
2 DESIGN-LEVEL ASPECTS.....	12
2.1 Using an aspect to encapsulate a crosscutting concern.....	12
2.2 Aspects.....	14
2.3 Join points and pointcuts.....	15
2.4 Advices .....	15
2.5 A design-level notation for aspects.....	16
3 AN ASPECT MODELING NOTATION.....	18
3.1 Case study .....	19
3.2 Structural view .....	23
3.3 Dynamic view .....	27
3.3.1 Aspect-System Interaction .....	27
3.3.2 Aspect Internal Behavior.....	30
4 MODELING THE BOTTLENECK DETECTION ASPECT.....	32
4.1 Structural view .....	32
4.2 Dynamic view .....	35
5 RELATED WORK.....	37
6 CONCLUSIONS .....	44
BIBLIOGRAPHY.....	47

## IMAGES INDEX

	Page
Figure 1 - Two Figure class realization, Point and Line.....	12
Figure 2 – An object-oriented approach to resolve the screen update problem. ....	13
Figure 3 – AspectJ code that encapsulates the update crosscutting concern. ....	14
Figure 4 – Overview of the CDP system, which communicates with clients/users (top module), mobile operators’ systems (left module), and contents providers (right module). .....	19
Figure 5 – Case study system: Partial class diagram of the Router component of the CDP system .....	22
Figure 6 – Kerberos aspect’s structural view: First Level. This level shows a bird’s eye view of the system generated by the base system and the aspects. ....	23
Figure 7 – Kerberos aspect’s structural view: Second level. This level shows named associations for pointcuts.....	24
Figure 8 – Kerberos aspect’s structural view: Third level. This level shows pointcuts as association classes. ....	26
Figure 9 – Sequence diagram: execution pointcut.....	28
Figure 10 – Sequence diagram: this and target pointcuts.....	29
Figure 11 – Kerberos state diagram.....	30
Figure 12 – Monitor aspect first level structural view.....	33
Figure 13 – Monitor aspect second level structural view .....	34
Figure 14 – Monitor aspect third level structural view.....	35
Figure 15 – Monitor aspect sequence diagram .....	36
Figure 16 – Monitor aspect state diagram .....	36
Figure 17 – Stein et al.’s approach to specify the advice weaving order (taken from the work of Stein et al. (2002))......	38
Figure 18 – Logging aspect model proposed by Kandé et al. ....	39
Figure 19 – Logging aspect class diagram .....	40
Figure 20 – Logging aspect sequence diagram using our proposed notation.....	41

Figure 21 – Evermann’s AspectJ profile application example .....	42
Figure 22 – Evermann’s AspectJ profile application example using our proposed modeling notation .....	42

## RESUMEN

La programación orientada a aspectos permite a los desarrolladores de software modularizar las responsabilidades transversales (*crosscutting concerns*) al código. Mientras el énfasis de la investigación en esta área ha estado focalizado en la implementación de los programas, se ha argumentado que la aplicación de la orientación a aspectos a nivel del diseño también puede ser beneficiosa. En este caso, falta una notación conveniente —es decir, tanto simple como expresiva— para representar diseños orientados a aspectos, en particular, para la elaboración de diseños en procesos de desarrollo ágiles y cortos. En esta tesis proponemos una notación basada en UML para modelar aspectos, en particular la unidad que representa al aspecto, su relación con el sistema base y su comportamiento interno. La notación usa los diagramas de clases, secuencia y estado del UML, a los cuales se les agregó unos pocos nuevos elementos para permitir la especificación de pointcuts, su activación y el comportamiento interno de los aspectos; la especificación del pointcut es modelada hasta con tres niveles incrementales de detalle. La propuesta ha sido aplicada inicialmente con éxito en una compañía que trabaja en proyectos cortos, con un limitado tiempo para actividades de diseño, tiene resultados iniciales exitosos con el uso de esta notación: Hemos sido capaces de modelar los aspectos a nivel de diseño de software, estos modelos tienen el nivel apropiado de detalle considerando las características de los proyectos, y hemos mejorado la comunicación de las ideas de diseño en el grupo de desarrollo.

Palabras Claves: diseño orientado a aspectos, UML, diseño de software, orientación a aspectos, desarrollos ágiles.

## ABSTRACT

Aspect-oriented programming allows software developers to modularize crosscutting concerns. While the emphasis has been on program implementation, it has been argued that applying aspect orientation at the design level can also be beneficial. However, we lack a convenient —i.e., both simple and expressive— notation to represent aspect-oriented designs, in particular, for fast, agile developments. In this thesis, we propose an UML-based design notation to model aspects, in particular the modular unit representing the aspect, its interaction with the base system, and its internal behavior. The notation uses UML's class, sequence, and state diagrams, to which it adds few new elements to model pointcut specification, pointcut activation, and the aspects' internal behavior; pointcut specifications can be modeled at three levels of detail. A company that works on short projects, with limited time for design activities, is successfully using the notation: We have been able to model aspect at the software design level, these models have the appropriate level of detail considering the projects' characteristics, and we have improved the communication of design ideas among the development team.

Keywords: aspect-oriented design, UML, software design, aspect orientation, agile development.

## 1 INTRODUCTION

Aspect-oriented programming (AOP) allows software developers to modularize crosscutting concerns using a new type of programming module: the aspect, as Kiczales et al. (1997) defined. Developments in programming languages, such as JBoss (the JBoss community (2008)), AspectJ of the AspectJ Team (2007), and Spring Framework (2008), and programming techniques, such as refactoring (Hannemann, 2006), have been catalysts for the adoption of AOP. While the emphasis of AOP has been on program implementation, it has been argued that applying aspect orientation at the design level can also be beneficial according to Baniassad et al. (2006): it contributes savings in programming hours and overall project duration, by reducing or eliminating refactoring efforts; and we can even expect an improvement in software design quality.

We focus on the software design process of a Chilean technology integrator company, which develops systems for a major mobile communications operator located in Chile. The market forces the company to work on short projects (3-6 weeks), with limited time for analysis and design (just a few days), and still develop very reliable products (uptime of 24x7). Recently, the company introduced AOP. While software implementation —i.e., code— has improved (e.g., classes are more cohesive and less coupled to other classes), software design has not. As a first step, we need a notation to represent aspects at the design level. However, there is no standard notation for this; several notations have been proposed (Kiczales et al., 1997; Grassi & Sindico, 2006; Evermann, 2007; Hannemann, 2006; Kandé, Kienzle & Strohmeier, 2002), but none has been broadly adopted.

In this thesis we propose an UML-based notation to model aspects and their interactions with the rest of the system. UML has been used to represent aspect-oriented designs, but those approaches are not appropriate for our purposes. Some replicate a particular programming language (Stein, Hanenberg & Unland (2002)), others use several new diagrams like Grassi & Sindico (2006) or just cover a specific aspects' dimension as Evermann (2007) does, and they all are expensive in time to learn or introduce overhead. We use UML's class, sequence and state diagrams, to which we add few new elements, to

model the structure and behavior of aspects. However, we keep the notation simple by not attempting to represent every facility provided by aspect-oriented programming languages.

For us, there are two important facts to consider when devising an aspect-oriented design notation:

- Analysis and design activities are allotted very limited time; following the core ideas of agile development (Mellor, 2005; Manifesto for agile software development, 2007), companies in the mobile communication industry favor working software over extensive documentation; and
- Design representations are mainly used during the construction of the systems and not for maintenance or documentation purposes; in fact, in the particular company used as a case study in this thesis, the software life span is only a few months long due to the dynamics of the market, so documentation or maintenance are not critical issues.

This means that design representations must be right to the point and centered on the main issues, leaving out detail. Our objective is that all members of the development team understand the aspects and the relationships between aspects and the rest of the system. Therefore, we need a convenient —i.e., both simple and expressive— notation to represent such designs, in particular, for fast, agile developments.

In summary, our hypothesis is that it is possible to define an easy to learn, lightweight notation to capture the basic properties of aspects in a manner independent of a programming language.

The general objective of this thesis is to devise an UML-based design notation to model aspects. The specific objectives of the thesis are the following:

- To decide what aspect's properties should be considered when designing software.
- To decide which UML diagrams are most appropriate to represent those properties.

- To devise a notation for representing such aspects properties, to improve the communication of design ideas within development teams working with agile processes.
- To apply the notation to the design of aspects for an actual project.

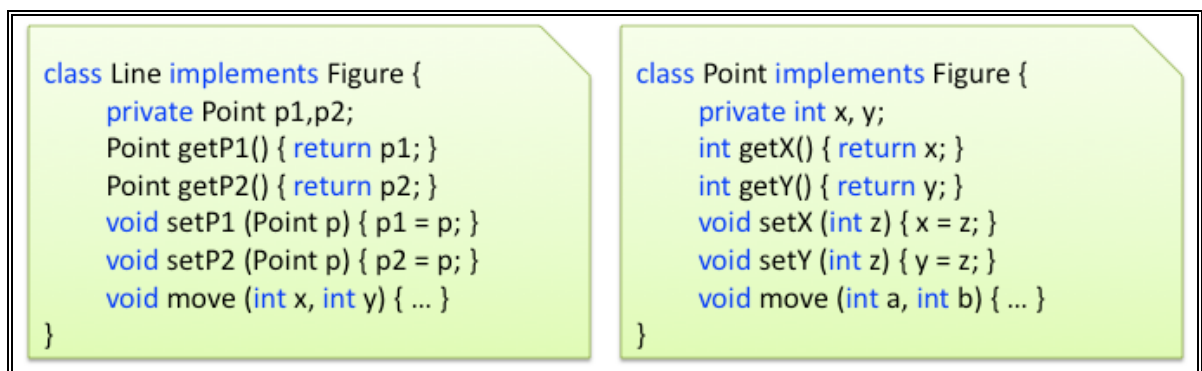
The remainder of this thesis is organized as follows. Chapter 2 identifies the elements that we consider relevant to be represented in an aspect-oriented design. Then, after describing a real case study used for the examples, Chapter 3 and 4 presents our UML-based notation for representing aspect-oriented designs. Chapter 5 summarizes related work in the area and compares those to our work. Finally, Chapter 6 presents conclusions and describes ongoing work.

## 2 DESIGN-LEVEL ASPECTS

In this chapter, we first introduce by means of a simple programming example the concepts of crosscutting concern and aspect. We then explain and analyze aspects' properties at the programming level —the aspect as a programming module, join points and pointcuts, and advices— and determine what parts of these properties should be represented at the design level. We base our decisions in part on the characteristics of the software development environment described in Chapter 1, and in part on the way in which object-oriented designs are commonly represented using UML

### 2.1 Using an aspect to encapsulate a crosscutting concern.

Consider the following code at Figure 1 that covers the update screen problem on a simple geometric design program that only considers the concepts of line and point.



```

class Line implements Figure {
    private Point p1,p2;
    Point getP1() { return p1; }
    Point getP2() { return p2; }
    void setP1 (Point p) { p1 = p; }
    void setP2 (Point p) { p2 = p; }
    void move (int x, int y) { ... }
}

class Point implements Figure {
    private int x, y;
    int getX() { return x; }
    int getY() { return y; }
    void setX (int z) { x = z; }
    void setY (int z) { y = z; }
    void move (int a, int b) { ... }
}

```

Figure 1 - Two Figure class realization, Point and Line

Figure 1 shows two concrete classes, both of which implement a `Figure` interface that only declares the `move` method. These classes represent the geometric concepts of line (`Line`) and point (`Point`); they exhibit high cohesion and low coupling. The system requires an update of the screen when the position of a figure changes, i.e., we need to notify the display controller when a setter method of any class

implementing the `Figure` interface is called. An object-oriented approach requires that we include, in each setter, some code to notify the change in the position of the figure to an observer class or directly to the display controller. This notification code is known as a crosscutting concern because it appears in several classes. Including crosscutting concern code increases the coupling and reduces the cohesion of classes `Line` and `Point`: both have to know about the `Display` class and are responsible for calling a specific method of this class, as shown in Figure 2.

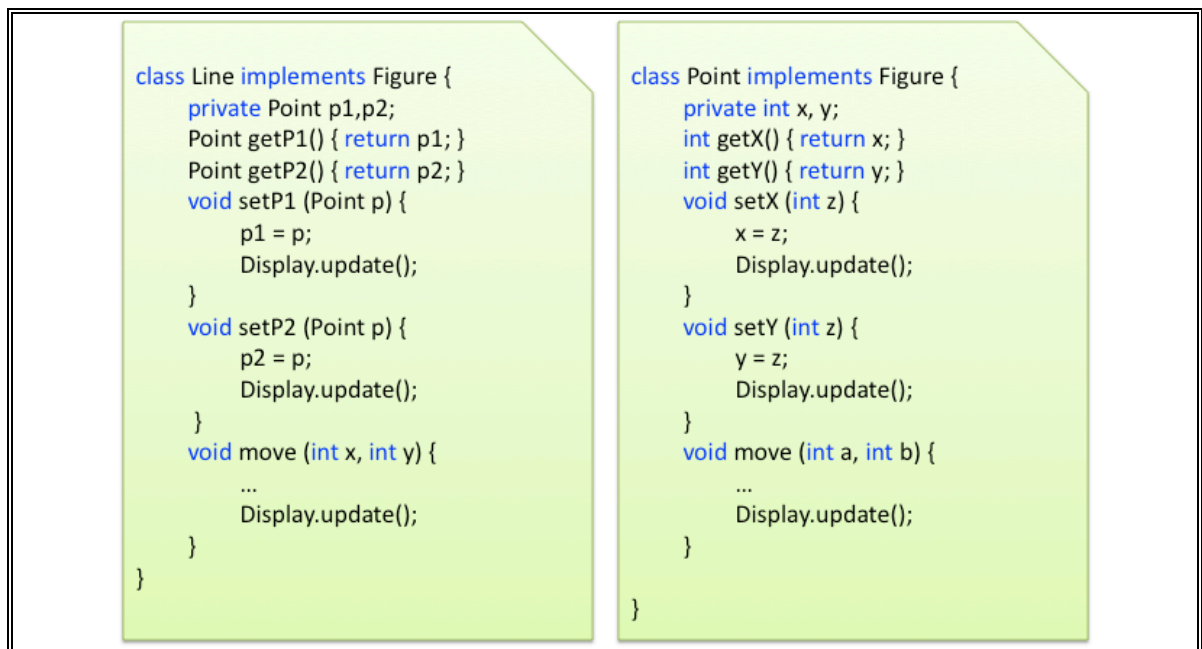


Figure 2 – An object-oriented approach to resolve the screen update problem.

An aspect-oriented approach, on the other hand, encapsulates the crosscutting concern into an aspect. Figure 3 shows an aspect implementation using AspectJ syntax; this aspect interacts with classes `Line` and `Point` in Figure 1, which now do not need to be changed. How this aspect works and which aspect's elements should be considered during system design are questions covered in the following sections.

```

aspect UpdaterMonitor {
    pointcut move :
        call (* Line.setP1(*) || call (* Line.setP2(*)
        || call (* Point.setX(*) || call (* Point.setY(*) );

    after() : move() {
        Display.update();
    }
}

```

Figure 3 – AspectJ code that encapsulates the update crosscutting concern.

## 2.2 Aspects

An aspect is a type of programming module, or unit, that allows the programmer to encapsulate the code corresponding to a crosscutting concern, thus syntactically separating this code from the base code. Semantically, however, the aspect code must be weaved at run time to the base code, thus restoring the original behavior of the system. To support the syntactic separation and at the same time the semantic weaving, an aspect, as a programming module, must include the code itself, i.e., how to act, and also a specification of where and when to act. Figure 3 shows a unit of modularity (i.e., the aspect) that encapsulates the crosscutting concern.

At the design level, we need to represent the module, similarly (although not identically) to a standard object-oriented class in a UML's class diagram, and the relationships between this module and the relevant classes in the base system. These relationships, however, are not either client-server association or generalizations — two standard object-oriented class relationships, for which UML offers special notations. Conceptually, classes in the base system are not aware of the aspects; an aspect is related to a class through a pointcut relationship, as explained below.

### 2.3 Join points and pointcuts

Any aspect-oriented programming approach considers a join point model. This model is a definition of the set of run time events —i.e., join points— visible to the aspects during program execution. In the problem described in Section 2.1, we need to indicate to the aspect which code statements produces a variation of the figure's position in order to refresh the screen. For example, a joint point model could include the following join points: method calls, method executions, attribute accesses, attribute updates, object initializations, and exception handler executions.

The way to specify which join points are relevant to a particular aspect is through a pointcut. A pointcut —a specification of join points— acts like a join point filter: occurring join points are compared to the specification, and only those that match the specification are selected and passed on to the aspect. A pointcut can also specify the composition of simpler pointcuts by means of boolean operators, such as `and`, `or`, and `not`. Figure 3 shows the use of the pointcut; we define the `move` pointcut, which filters four join points that produce a change in the position of a figure.

At the design level, we need to represent the fact that an aspect specifies its relevant join points through pointcuts. Therefore, aspects, pointcuts, and base system's classes should all be represented in the same design diagram, to make it simpler to understand the aspect-class relationships.

### 2.4 Advices

An advice is an ordered sequence of program statements included in an aspect that are executed when a particular join point is reached during the execution of the base system. An advice is similar to a standard object-oriented method. However, there are three important differences:

- The advice is implicitly called when the relevant join point is reached; No statement in the base system's program explicitly calls the advice. Therefore

there is no need of unique identifiers or visibility modifiers (such as private or public).

- Advices can have parameters, but these are implicitly passed from the base system to the advice through the pointcut.
- The specification of an advice includes a temporal modifier that specifies when should the advice execute with respect to the execution of the join point. In particular, following AspectJ, we consider three temporal modifiers: *before*, *after*, and *around* (during) the execution of the join point.

Considering the aspect's example shown in Section 2.1, we specified an advice that executes when a figure movement is performed. The advice's code calls the `update` method of `Display` class, triggering an update of the screen.

These differences are important when we consider the use of standard UML notation to represent aspects at the design level. For example, we cannot use the traditional lines that represent method calls in a sequence diagram to represent implicit advice calls. At the design level, we need to represent implicit advice calls and what happens to an aspect when its advices execute.

## 2.5 A design-level notation for aspects

Our proposed notation, described in Chapters 3 and 4, can be used to represent aspects during software design. The notation captures the semantics of the concepts described in the previous sections, to a level of detail appropriate for the particular software development environment described in Chapter 1:

- The aspect, as a software module similar to an object-oriented class.
- The aspect's pointcuts, which describe the relationship between the aspect and the classes of the base system; this relationship is different from a standard object-oriented client-server association or a generalization.
- The pointcuts' behavior, which describe the interactions between the aspect and the class instances of the base system; these interactions have some

similarities, but also important differences, with standard object-oriented method calls.

- The aspect's advices, which describe the internal behavior of the aspect similarly to the methods of an object-oriented class.

The notation is based on UML's class, sequence, and state diagrams, and is consistent with the way in which object-oriented designs are commonly represented in this language: you have classes and their relationships, interactions between class instances, and the internal behavior of these instances. In Chapter 5, we compare our notation to existing notations.

### 3 AN ASPECT MODELING NOTATION

Our aim is to devise an UML-based notation to model aspects during software design. We chose UML, because it is a broadly adopted and a de facto standard software design notation that offers different software views

In principle, we do not try to cover all aspect orientation concepts (e.g., as described by Kiczales et al. (1997)) or make a rigorous translation from some aspect-oriented programming language (e.g., AspectJ) to UML. In fact if we tried to do that translation, the result would not be precise because UML is not; and also, according to Harel & Rumpe (2004), it is not a promising approach to try to assign a precise meaning (in terms of a programming language) to a UML-based notation. Also, the time constraints forced by the industrial setting on the company's projects, imply that these can only spend very limited time on analysis and design. Therefore, we look for a notation to mainly support communication among the development team's members during analysis/design meetings without imposing work overhead.

Our approach considers two views of the relationships between an aspect and the base system: a structural view and a dynamic view: the structural view shows which elements in the base system are relevant to the aspect. We use UML's class diagram to represent the aspects, i.e., the unit that modularizes the crosscutting concern, the base system classes, and the associations between them.

Aspects are dynamic entities: they have a run-time effect on the execution of the base system. Our approach deals with this issue using the sequence and state diagrams. We use UML's sequence diagrams to represent the interaction between the aspect and the base system. State diagrams are used to describe the internal behavior of the aspect; in particular, we show how aspects act when relevant pointcuts in the base system are reached during execution.

To illustrate our notation we will apply it to the design of two aspects for a real system currently under development for a major mobile communications operator in Chile. In the rest of this chapter, we first describe the system and then each proposed diagram as they apply to one of the system's aspects.

### 3.1 Case study

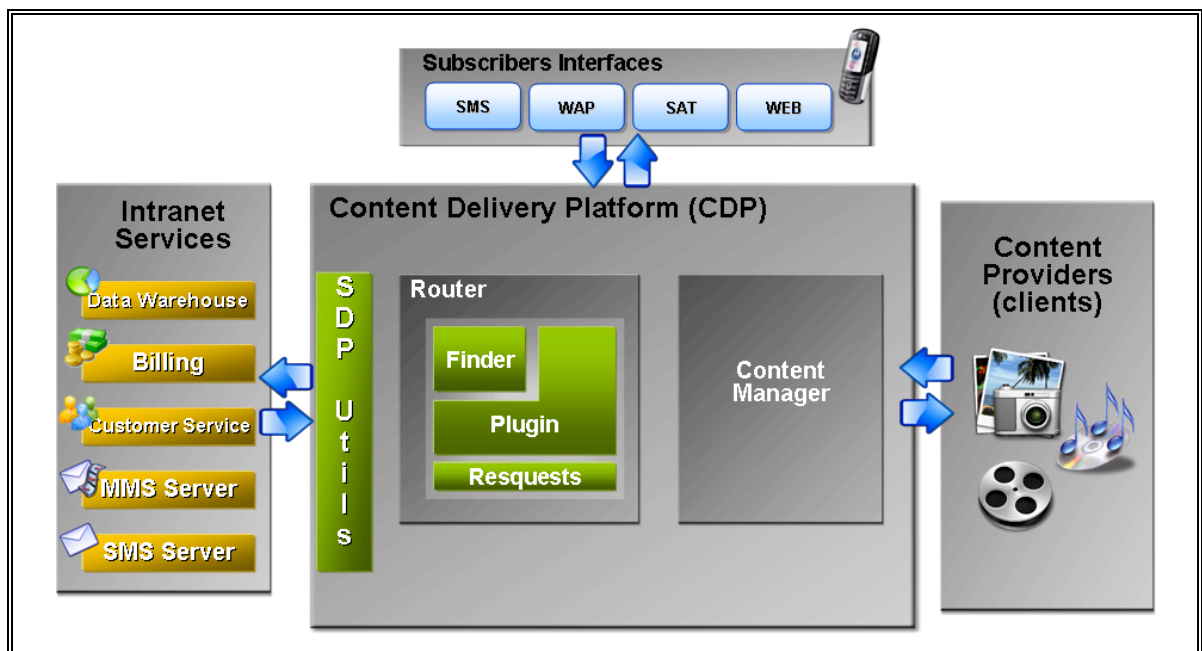


Figure 4 – Overview of the CDP system, which communicates with clients/users (top module), mobile operators' systems (left module), and contents providers (right module).

The Content Delivery Platform (CDP) system controls every transaction (download, service subscription, etc.) of images, themes, ringtones, backtones, applications, etc. The CDP system interacts with multiple subsystems, such as SMSC (Short Message Service Center), MMSC (Multimedia Messaging Service Center), SAC (Service Authorization Center, which determines whether client can use or acquire a

content/service), Billing, and ReCenT (a centralized statistics platform). The CDP system also interacts with external systems. Figure 4 presents an overview of the CDP system and its interactions with other systems.

The CDP system has two main components: `Router` and `ContentManager`. The `Router` is responsible for finding the correct route for every request received by the system; it analyzes the request and determines which subsystem should process it: either the `ContentManager`, for company proprietary content, or an external platform. The `Router` has more than 130 classes and works using a plug-in approach: incoming requests are classified by its `Finder` component, which assigns them to the appropriate `Plugin`, which, in turn, can delegate part of its work to a `PluginModule`. Figure 5 shows a partial class diagram for the `Router`; we have only included classes relevant for this presentation.

For the purpose of this thesis we have identified two crosscutting concerns in the CDP system: client authentication and bottleneck detection. We deal with client authentication in this chapter, and with bottleneck detection in Chapter 4.

The CDP system interacts with several external content providers —its clients. Not all clients have the same access privileges. For example, one client could only dispatch free contents (advertisements), while others could also use the system's billing interface to sell contents. Therefore, the need exists to control these accesses through client authentication. Initially, the CDP system performed client authentication by checking all clients' calls to the CDP system and performing the privilege checks; if the request was denied, the CDP would notify the client. This approach added the client authentication concern to all the classes that interact with the clients. For example, charge tasks (`ChargePlugin`) or the delivery of contents (`PushContentPlugin`). But client authentication should not be part of the core concerns of any of these classes, therefore we encapsulated this crosscutting concern into its own module: the client authentication aspect. In addition, and

probably more importantly, the extensibility of the system poses an interesting challenge, because we must be able to guarantee that all future improvements to the CDP system will check these privileges too.

We next describe our aspect-modeling notation as it applies to the client authentication aspect.

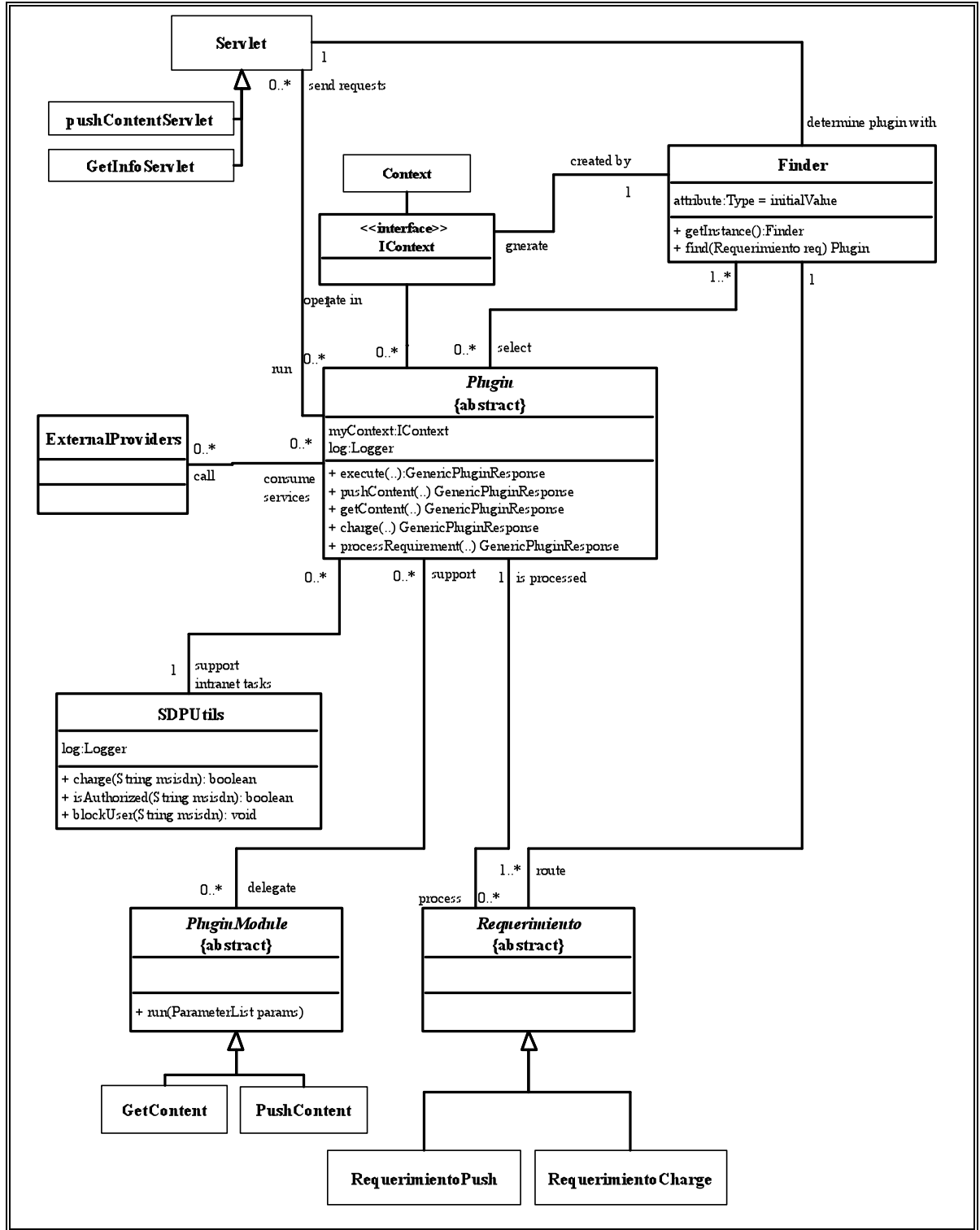


Figure 5 – Case study system: Partial class diagram of the Router component of the CDP system

### 3.2 Structural view

We represent the structural view of an aspect-oriented system using a class diagram, showing the aspects and the relationships between the aspects and the rest of the system. From a methodological point of view, we propose to start with a first level diagram that only shows basic information about the relationships between the aspect and the system's classes, as shown in Figure 6; and then to add detail to this diagram according to the needs of the design, as shown in Figure 7 and Figure 8. We assume that a class diagram for the base system already exists similar to the diagram shown in Figure 5, but probably with fewer details. We use the stereotype «aspect» to differentiate aspects from standard classes.

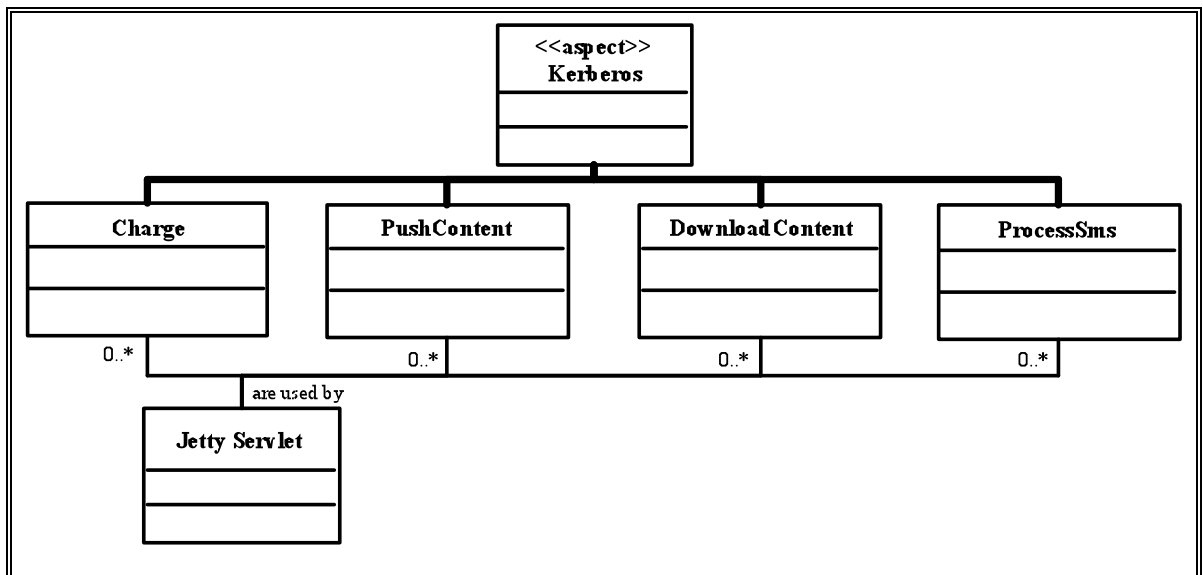


Figure 6 – Kerberos aspect's structural view: First Level. This level shows a bird's eye view of the system generated by the base system and the aspects.

The first level diagram allows the development team to quickly show/notice which classes are affected by the aspect. This level is ideal for brief meetings and sketches. For example, in Figure 6 we establish the relationship between the Kerberos aspect (corresponding to the modularization of the authentication crosscutting

concern) and the Charge, PushContent, DownloadContent and ProcessSms classes.

As the design of the base system evolves, the development team needs to add details to the associations between the aspect and the base system's classes. The second level diagram adds pointcuts. For example, in Figure 7 we see that the Kerberos aspect is associated to one pointcut, AuthenticatorGuard, which matches join points in four classes: Charge, PushContent, DownloadContent and ProcessSms. Notice the addition of the stereotype «pointcut» and a name to the associations to show the intended semantics, i.e., authenticator guard. Of course, the utility of this addition depends on the good choice of pointcut names.

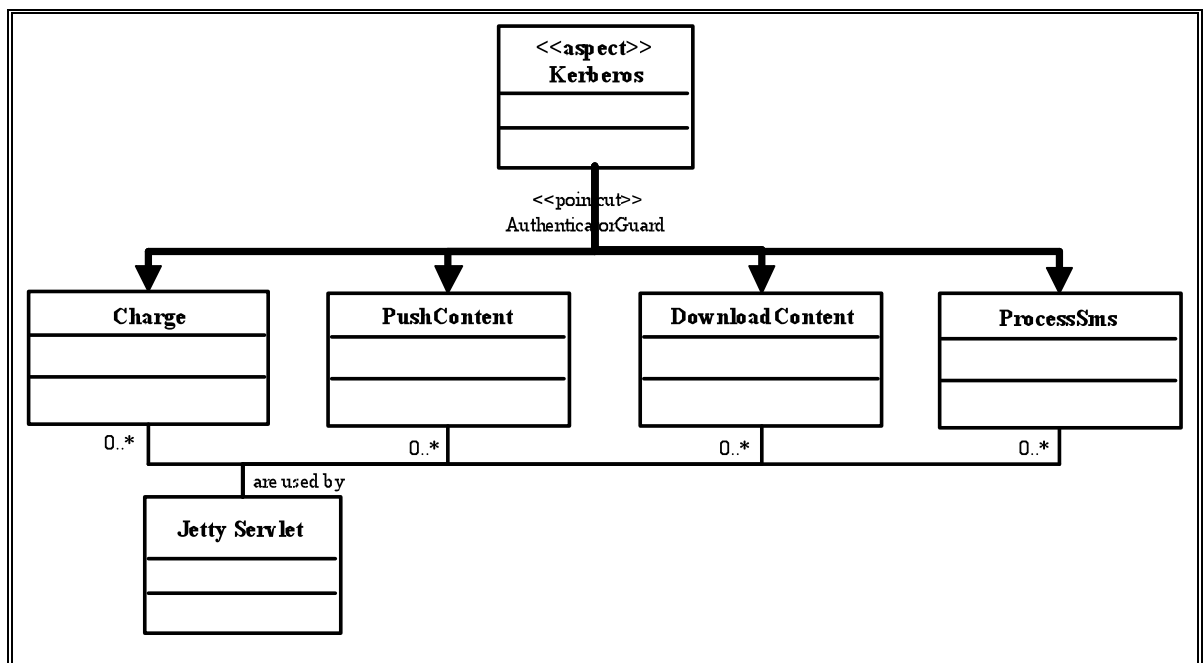


Figure 7 – Kerberos aspect's structural view: Second level. This level shows named associations for pointcuts.

Eventually, the level of detail of the base system design will allow the development team to add more detail to pointcut associations. Figure 8 shows a third level diagram, introducing the AspectJ syntax to declare the pointcuts. The use of this level of detail assumes an advanced design, because it is necessary to define methods' signatures, attributes and class names to reference them in pointcut declarations.

In the third level diagrams, we change named associations to association classes. In an association class we include the pointcut declaration using AspectJ syntax. For example, consider the following part of the `AuthenticatorGuard` pointcut declaration:

```
execution (* HttpServlet+.doGetLogic(
HttpServletRequest, HttpServletResponse))
```

This declaration means that the pointcut filters the following join points: calls to methods `doGetLogic` of the `HttpServlet` subclasses, having `HttpServletRequest` and `HttpServletResponse` parameters and returning an object of any type. The pointcut definition in Figure 8 is slightly different because it includes composition through the `or` operator.

If we wanted, we could add constraints regarding who calls the methods or which are the target objects. For example, in the pointcut definition in Figure 8 we can use the `target` and `this` keywords to specify restrictions on the caller and called objects, respectively; in this case, both objects must be instances of the `HttpServlet` class.

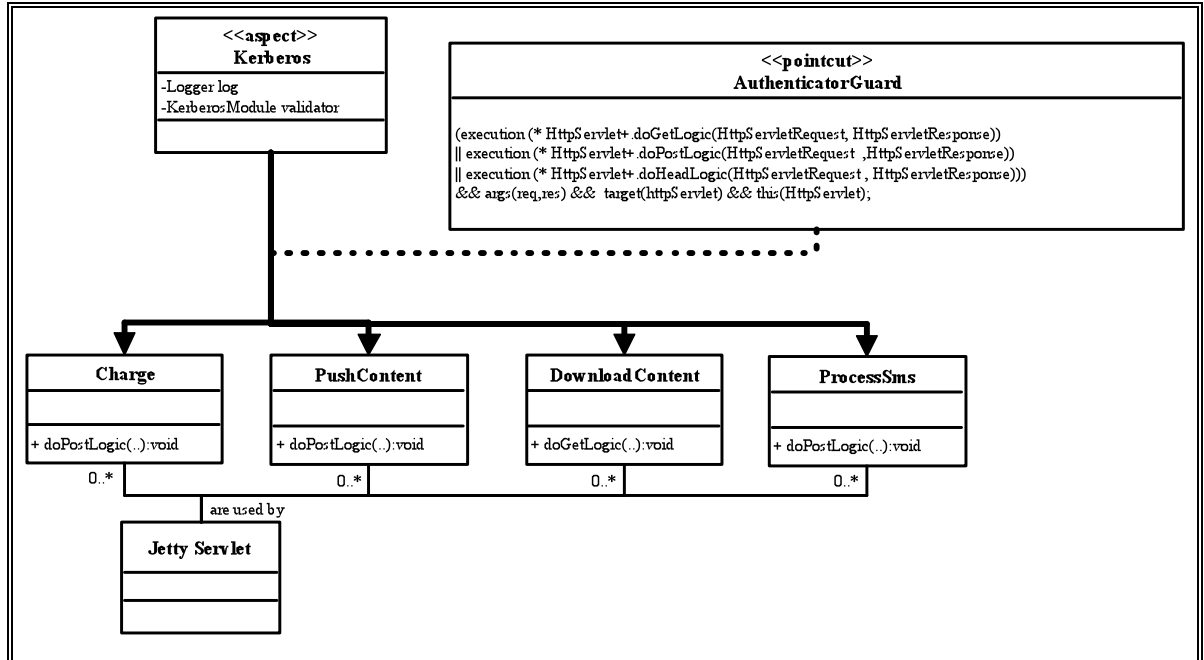


Figure 8 – Kerberos aspect’s structural view: Third level. This level shows pointcuts as association classes.

In general, there is a tradeoff between a notation’s simplicity and its semantics representation capability. We attempt to strike a reasonable balance by avoiding complex constructions or too many new diagrams, but without losing the capability of expressing the aspects’ concepts mentioned in Chapter 2.

It can be argued that the tight coupling between the proposed modeling notation and a particular AOP language is a disadvantage. We do not intend to offer a generic AOP modeling notation. As Evermann (2007) points out, the conceptual differences between aspect-oriented languages (e.g., AspectC++, AspectJ or Aspect#) are substantial. Therefore, we only model those concepts that are common to AOP, but we use AspectJ syntax because it probably is the most popular AOP language. This allows us to cover some of these particular concepts, such as pointcut definition, in early stages of the software development.

### 3.3 Dynamic view

To represent the behavior of aspects we use UML's sequence and state diagrams. The sequence diagram describes the interactions between the aspect and the base system. The state diagram shows the internal changes in the aspect.

#### 3.3.1 Aspect-System Interaction

During the execution of the code, some join point could be reached, activating a pointcut. We represent a pointcut activation in a sequence diagram by a dotted line-arrows stereotyped with the keyword *pointcut*, going from the join point at the base system object to the corresponding aspect, as shown in Figure 9 and Figure 10. The dotted line stresses the implicit nature of the call, and prevents confusing this call with standard calls; the keyword differentiates this arrow from a return message. Above the dotted line and next to the keyword *pointcut*, we can optionally write the pointcut's name (e.g., `AuthenticatorGuard` according to Figure 8) or definition, as we do in Figure 9. We next show three commonly used pointcuts: `execution`, `this`, and `target`.

An `execution` pointcut becomes active just before a called method begins executing. The syntax is the following:

```
execution(method-signature)
```

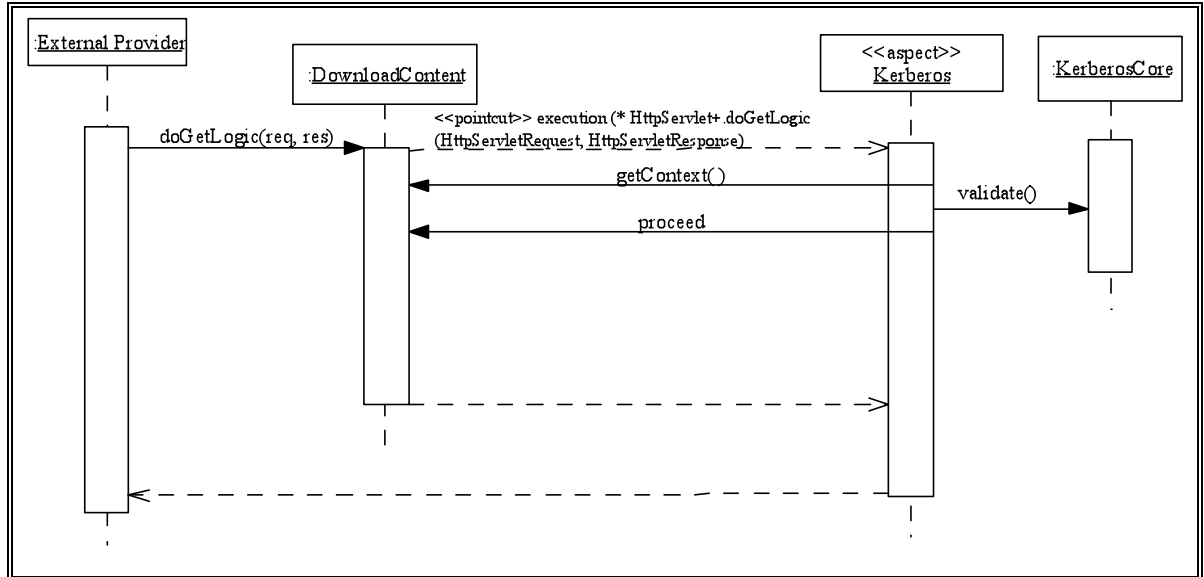


Figure 9 – Sequence diagram: execution pointcut.

Figure 9 shows message `doGetLogic(req, res)` sent by the `ExternalProvider` object to the `DownloadContent` object. This call triggers an implicit *call* to the `Kerberos` aspect due to the match with the execution of the `AuthenticatorGuard` pointcut. During the advice execution, the aspect calls the method `getContext` of the `DownloadContent` object to retrieve auxiliary objects (i.e., loggers or database client) and method `validate` of the `KerberosCore` class to verify if the method call is allowed or not. If the call is allowed, the aspect sends a `proceed` signal to the `DownloadContent` object to continue the normal flow of the `doGetLogic` method. Once the method finishes, the control goes back to the aspect, which returns the response of the `DownloadContent` object to the `ExternalProvider`.

The execution pointcut is commonly used in combination with other pointcuts. For example, we can filter join points according to the object that activates the pointcut using the `this` pointcut:

```
this (Type-pattern|Identifier)
```

With the `this` pointcut, we can add a constraint to specify which objects trigger the matched pointcut, indicating that only calls to method `doGetLogic` made by an instance of the `HttpServlet` class (or of its subclasses) will match the pointcut.

Similar to `this`, the `target` pointcut references the destination object of the pointcut:

```
target (Type-pattern|identifier)
```

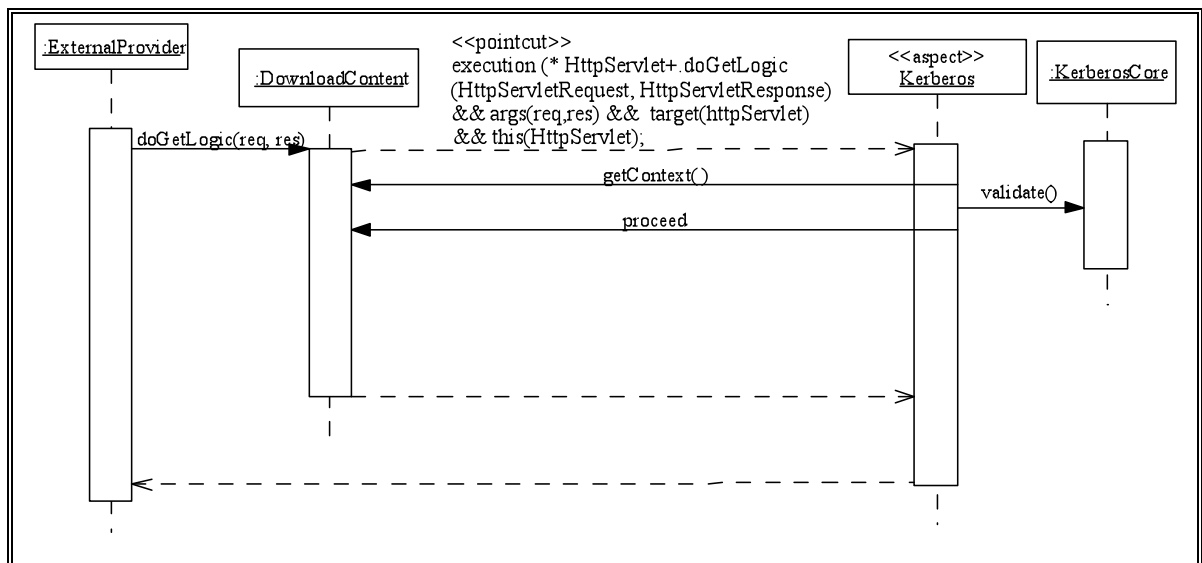


Figure 10 – Sequence diagram: `this` and `target` pointcuts.

Figure 10 shows the combined use of the `target`, `this`, and `execution` pointcuts. The result is that we select all executions of the `doGetLogic` method of the `HttpServlet` class (object `DownloadContent` is an instance of a subclass of `HttpServlet`) with `req` and `res` parameters, coming from a `HttpServlet` class (or subclass) instance.

### 3.3.2 Aspect Internal Behavior

We use state diagrams to represent the internal behavior of aspects. A state diagram shows what an aspect does when a pointcut is reached and when does it do it. For example, Figure 11 shows the behavior of the `Kerberos` aspect. Initially, the aspect is in the observing state, waiting for a pointcut to be reached; when the pointcut is reached, a transition is triggered. We use a dotted-line arrow to represent this transition and to show that the triggering event is the matching of a pointcut. Above the dotted line, we can write additional information about the transition, using the following syntax:

`<pointcut_reached> / <when_advice_occur>`

The triggering event's name is the name of the pointcut reached (in Figure 8, the `AuthenticatorGuard` pointcut). The second element indicates when does the advice occur with respect to the base system; its possible values are `before`, `after`, and `around`.

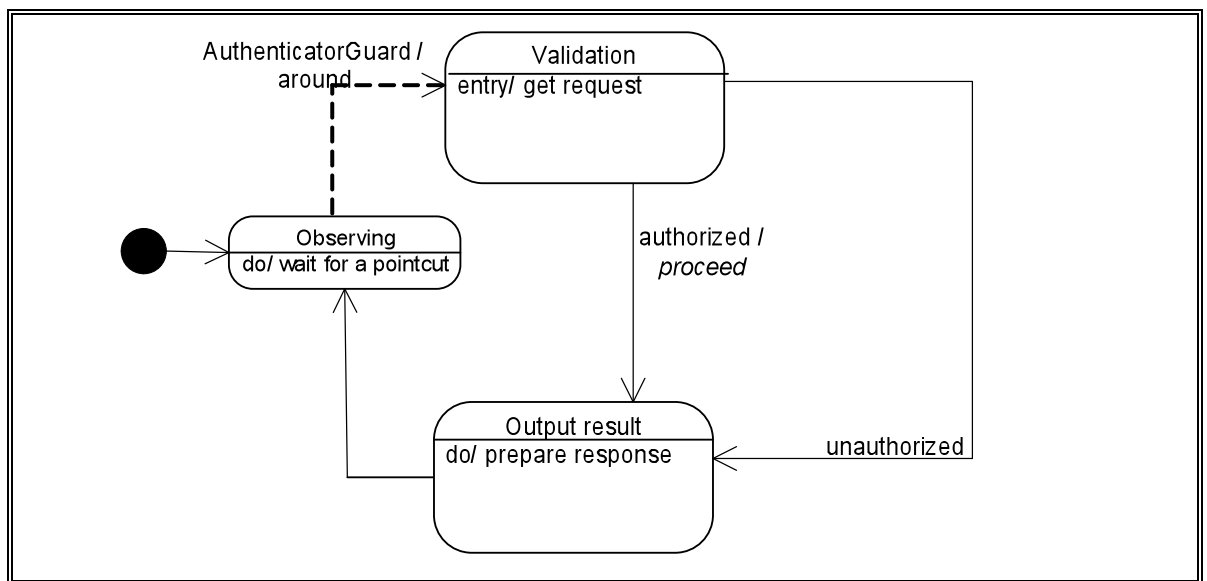


Figure 11 – Kerberos state diagram

The rest of the diagram is a standard state diagram. In the example, we arrive at the `Validation` state. After the initialization of the aspect, its retrieves the request from the caller and validates it. If the call is refused, a simple transition to the `Output result` state is performed. On the other hand, if the call is authorized, then a transition to the `Output result` state is triggered and, during this transition, the activity `proceed` is executed; this allows the base system's normal flow to continue. In both cases, once the response is ready, an unconditional transition from the `Output result` state to the initial `Observing` state is performed.

## 4 MODELING THE BOTTLENECK DETECTION ASPECT

We now present the aspect-oriented modeling for a second crosscutting concern: bottleneck detection. As shown at the top of Figure 4, clients can request content in many ways (interfaces): a web page (WEB), SMS, USSD, SAT (integrated mobile phone menus), WAP (Wireless Application Protocol). Due to the variety of internal and external subsystems involved in the content delivery process, it is critical to identify bottlenecks as soon as they appear in the system.

The traditional approach has been for every subsystem to time itself, but this introduces a crosscutting concern in the system, reducing the cohesion of the classes and increasing their coupling, both undesirable effects of the solution. Therefore, we use aspects to extract crosscutting concerns; in particular, we identified bottleneck detection as a crosscutting concern. In addition, this solution does not guarantee that every new system plugin performs the transaction time measure.

Even if we modularize the time measurement logic, we still have the problem of the crosscutting concern. If we try to modularize the measurement logic using classic object oriented techniques, we must include in every plugin additional lines of code to take care of this new concern and, most importantly, this concern is not part of the core concerns of the class or of the plugin.

To solve the bottleneck detection problem, the aspect monitors the method calls, stores the elapsed time and whether or not the method finished normally. Controlling the elapsed time or monitoring the results of the calls is not part of the responsibilities of the classes.

### 4.1 Structural view

At the first level of the structural view, it is possible to quickly identify which classes are relevant to the `Monitor` aspect. Figure 12 shows the classes relevant to the aspect: `ExternalProviders`, `Plugin`, `SDPUtills` and `PluginModule`.

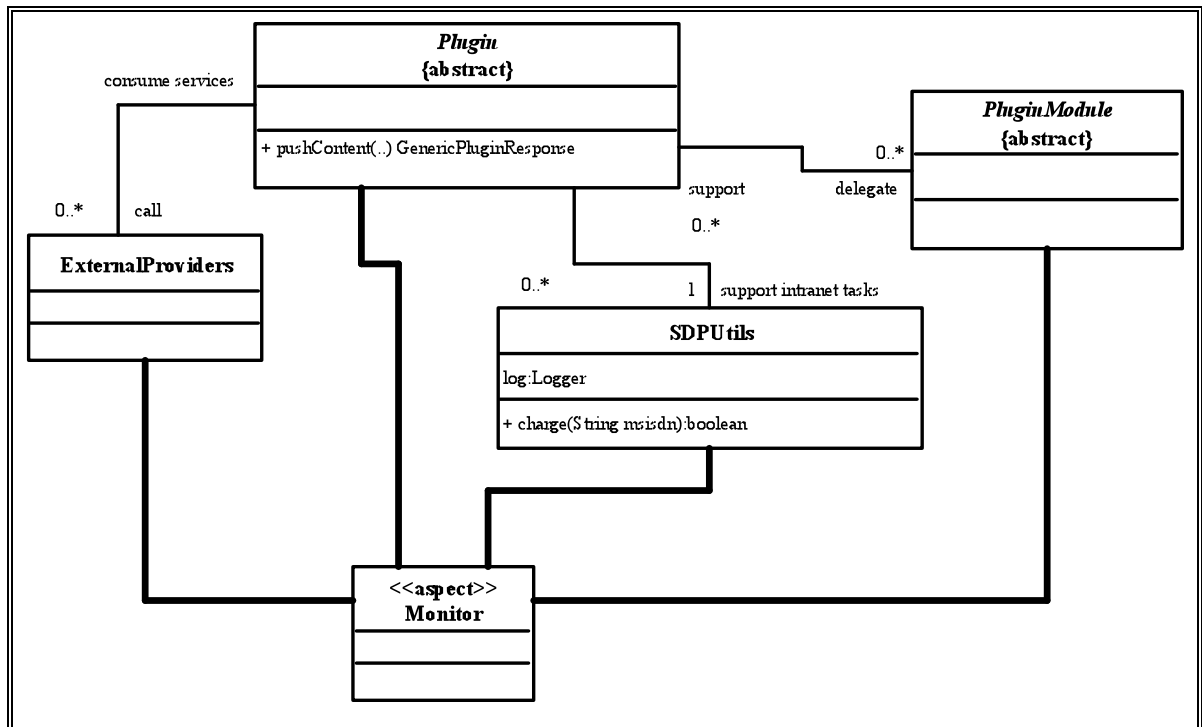


Figure 12 – Monitor aspect first level structural view

As the design evolves, the diagrams evolve too, and we move to the second level of detail: we specify the nature of the relationship between the aspect and the base classes, as shown in Figure 13. The figure shows that the `Monitor` aspect is associated to two pointcuts: `Benchmark`, which filters join points in two external classes, `ExternalProviders` and `SDPUUtils`; and `LogPluginExceptions`, which filters join points in the `Plugin` and `PluginModule` classes. Notice the addition of the stereotype `<<pointcut>>` and a name to the associations to show the intended semantics, i.e., benchmarking and logging.

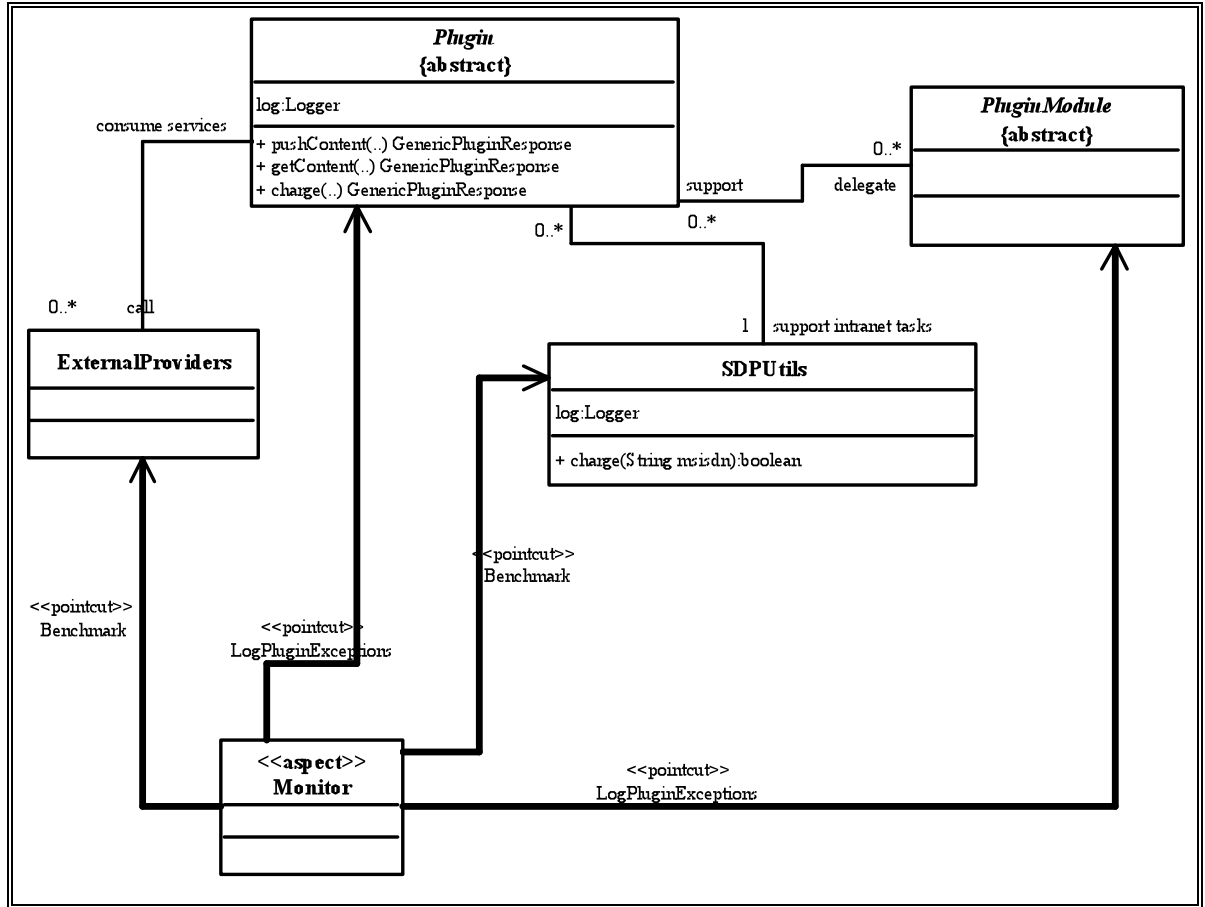


Figure 13 – Monitor aspect second level structural view

In Figure 14 we add the pointcut definition because the signatures of the methods in the base system are clearer. Pointcut `Benchmark` monitors two classes of the base system through the composition (use of operator `||`) of two pointcuts; the pointcut `LogPluginMethods` observe the `Plugin` and `PluginModule` classes.

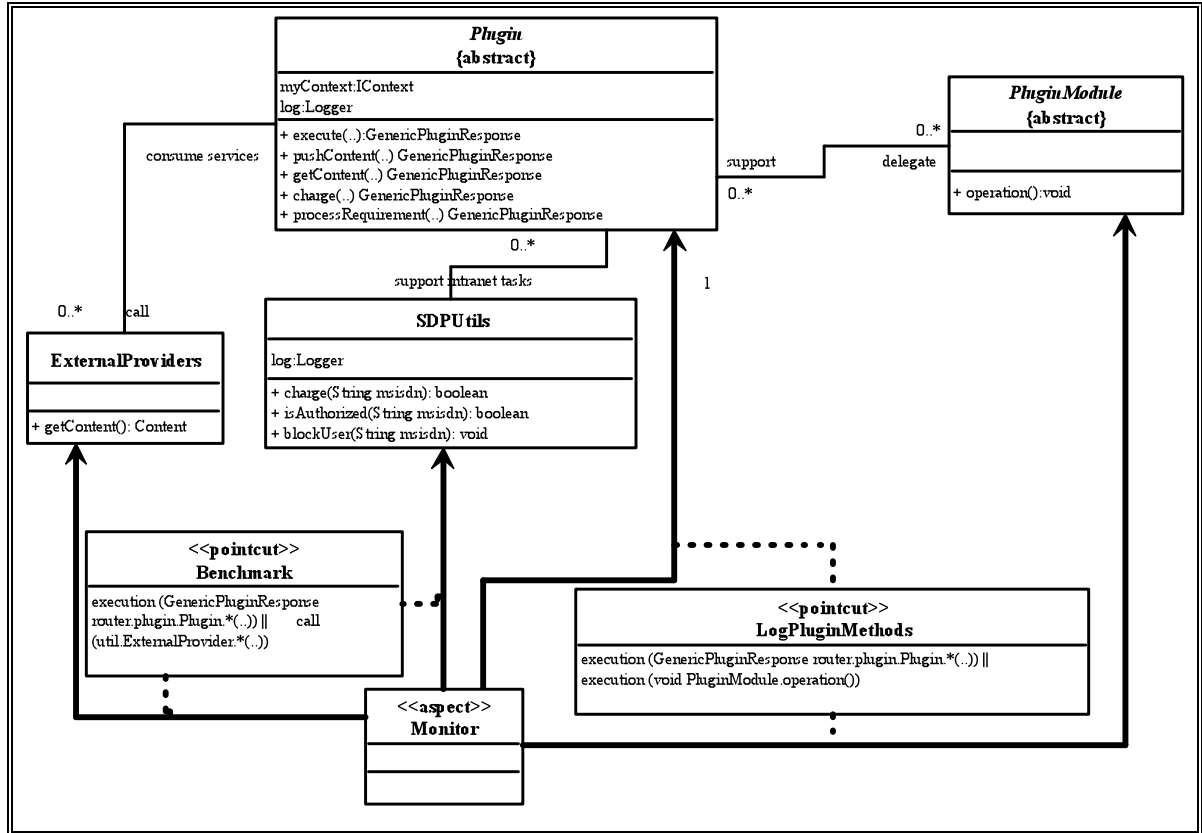


Figure 14 – Monitor aspect third level structural view

## 4.2 Dynamic view

Figure 15 shows message `pushContent(IReq req)` sent by `PushContentServlet` to `GenericPlugin`. This call triggers an implicit call to the `PluginLogger` aspect due to the matching with the execution `Plugin.*(..)` pointcut. The aspect calls the method `getContext` of `GenericPlugin` to recover context data, in particular, the `Logger` object. Then, the aspect allows the normal flow of the `pushContent` method by sending a `proceed` signal. Once the method finishes its execution, the control goes back to the aspect, which performs a recording through the `Recorder` class. Finally, the aspect returns to the `PushContentServlet` object.

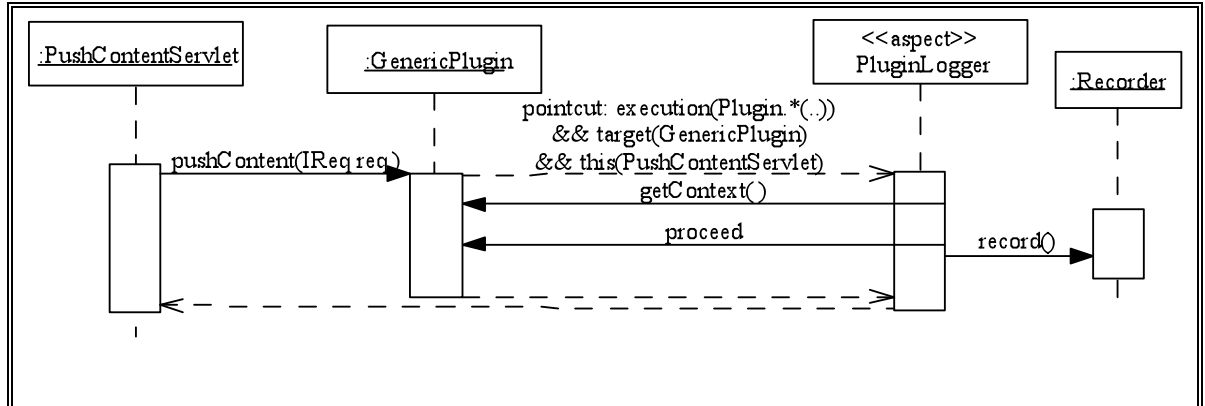


Figure 15 – Monitor aspect sequence diagram

The state diagram of the aspect is displayed in Figure 16. In the example, we arrive at the `PluginCall` state. After the initialization of the aspect, we start the timer and execute the plug-in method (through the `proceed` activity during the unconditional transition to `ResponseProcess` state, as seen in Figure 15). Once the original method finishes, the aspect stops the timer, records the elapsed time and exceptions, and, finally, unconditionally transitions to the initial `Observing` state.

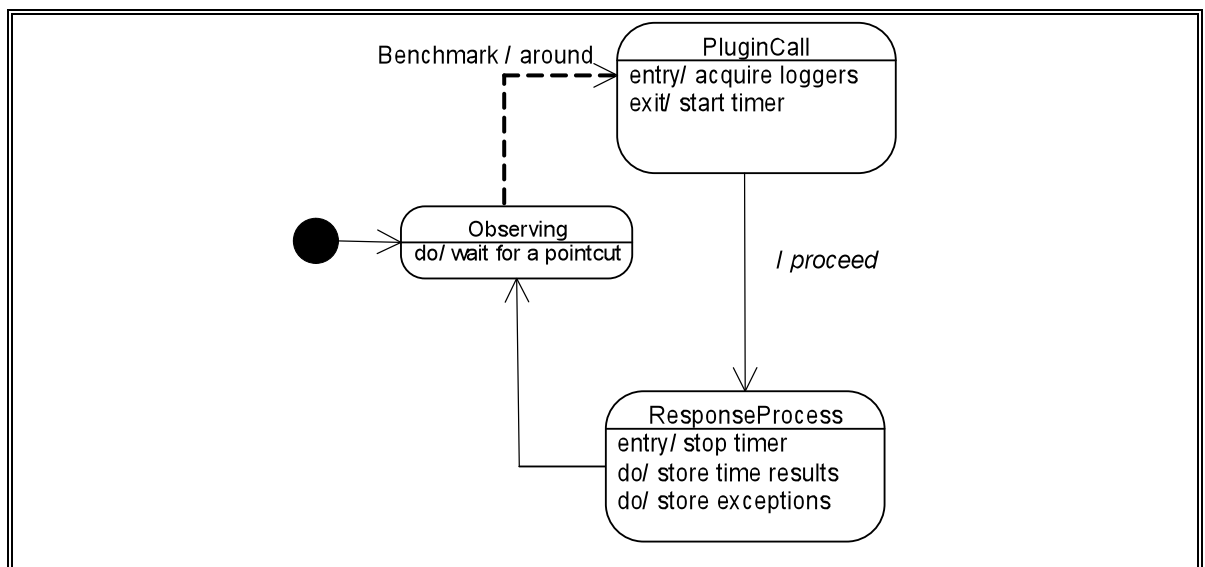


Figure 16 – Monitor aspect state diagram

## 5 RELATED WORK

There has been an increasing interest in identifying aspects during the early stages of software development, in particular, during the requirements and design phases. E.g., Baniassad et al. (2006) argue the need to capture and compose architectural aspects, but they do not propose a concrete notation.

We now review UML-based approaches to represent aspects at the design level. Some use UML's extension mechanisms, e.g., stereotypes; others add new diagrams. But none deal with the internal behavior of aspects.

Stein et al. (2002) offer a complete translation of AspectJ into UML. In our opinion, this is not a design notation, but a graphical programming language, difficult to use by nonprogrammer designers.

The correct use of Stein's notation requires precise knowledge of how the aspect will be implemented. In our opinion, this it is not the best approach for initial designs or during development meetings. One option is to not use the entire design notation proposed, thus reducing complexity; however, we also disagree with how Stein's approach represents some aspect-oriented issues. Consider Figure 17 taken from Stein et al. (2002), showing how to specify the advice weaving order. We consider more appropriate to use state diagrams to represent the internal aspect behavior, as shown in Figure 11 and Figure 16.

Conceptually, we question if it is necessary an AspectJ translation into UML. At least, considering the design detail level required in our industrial setting, there is not such need. For example, how could you apply an AspectJ translation if it is not clear how the crosscutting concern will be implemented?

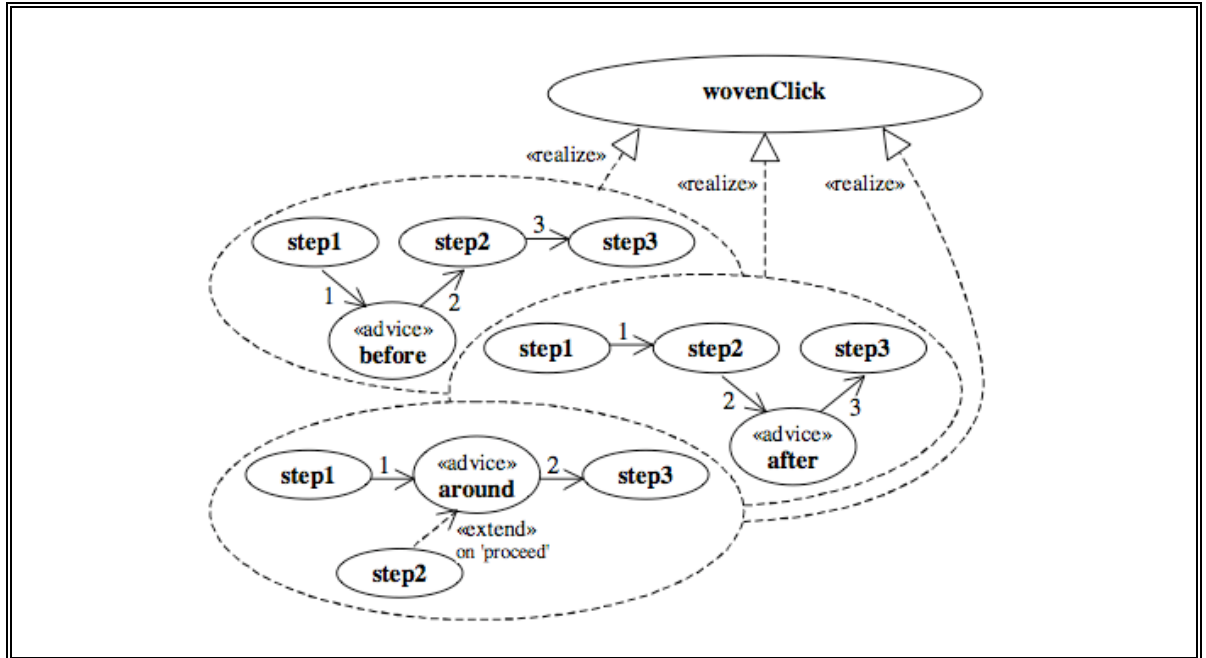


Figure 17 – Stein et al.'s approach to specify the advice weaving order (taken from the work of Stein et al. (2002)).

Suzuki & Yamamoto (1999) describe the design of *introductions*, i.e., structural modifications of system classes, through a new UML meta-class; however, in our opinion, it is not clear that introductions are a desirable feature of aspects. Also, they focus on the interchangeability of aspect models between development tools. But in projects where the use of CASE tools is not common (such as the project used as a case study for this thesis), this issue is not relevant. Finally and maybe more important, they do not explain how to represent pointcuts or the internal behavior of the aspect. It remains unclear how this notation could be used for modeling propose because the authors do not offers examples of the resulting model.

Grassi & Sindico (2006) proposes adding an advice diagram to the design, composed by two sub-diagrams: a pointcut diagram (PD), and a behavioral diagram. The PD specifies the occurrence of a particular set of events in the execution of a program. Each event is specified by a join point diagram (JPD), so the PD is a composition of JPDs. The

behavioral dimension is covered by an Interaction Overview Diagram, and the static dimension, by an Inter-Type declaration diagram.

If we consider the behavior diagram (BD) for the security concern example taken from the work of Grassi & Sindico (2006), it is relative clear when the pointcut occurs («Pointcut Occurrence» box), but it remains unclear what the aspect does when the pointcut is reached.

This approach introduces several new diagrams, and the resulting designs are difficult to understand, because the relationship between the new and the standard diagrams is not direct. More important, Grassi et al. present the total separation of aspects from the base system as the only view in aspect oriented modeling. However, as the weaving process required in all aspect-oriented systems demonstrates, aspects are intrinsically tied to the base system, and therefore the representation of the interactions between the aspect and the system is critical to understand aspect behavior.

Kandé et al. (2002) propose a UML collaboration stereotype to represent introductions and the interactions with the rest of the system using connection points. Figure 18 shows the model for the logging crosscutting concern; it consists of four types of elements; the aspect itself, the associated connection points, normal UML classes, and the binding relationships.

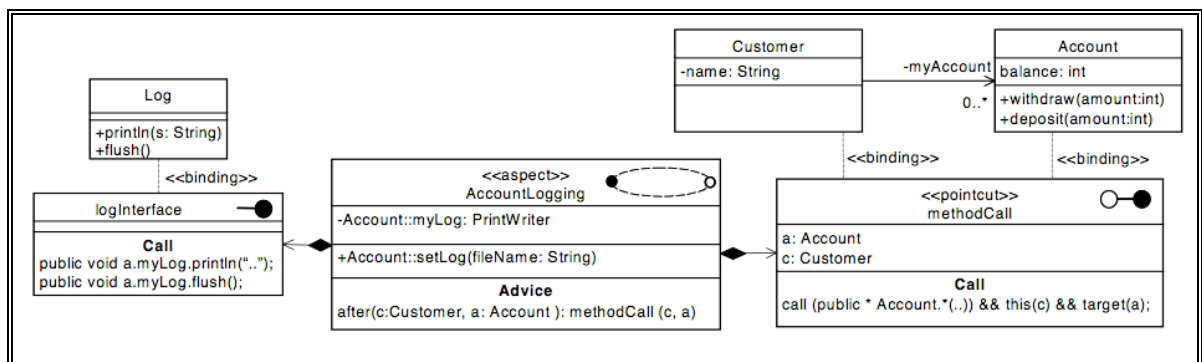


Figure 18 – Logging aspect model proposed by Kandé et al.

This approach merges the dynamic and static dimensions of the aspect in the same diagram, generating a confusing model. Furthermore, it is not clear how to represent other pointcuts, like `execution`, with the notation.

The use of sequence diagram to specify the relation between the base system and the aspect is in our opinion clearer than the connection points presented by Kandé. The class diagram generated by our notation is shown in Figure 19. The relationship of the `AccountLogin` aspect with the base system is covered using a sequence diagram as shown in Figure 20.

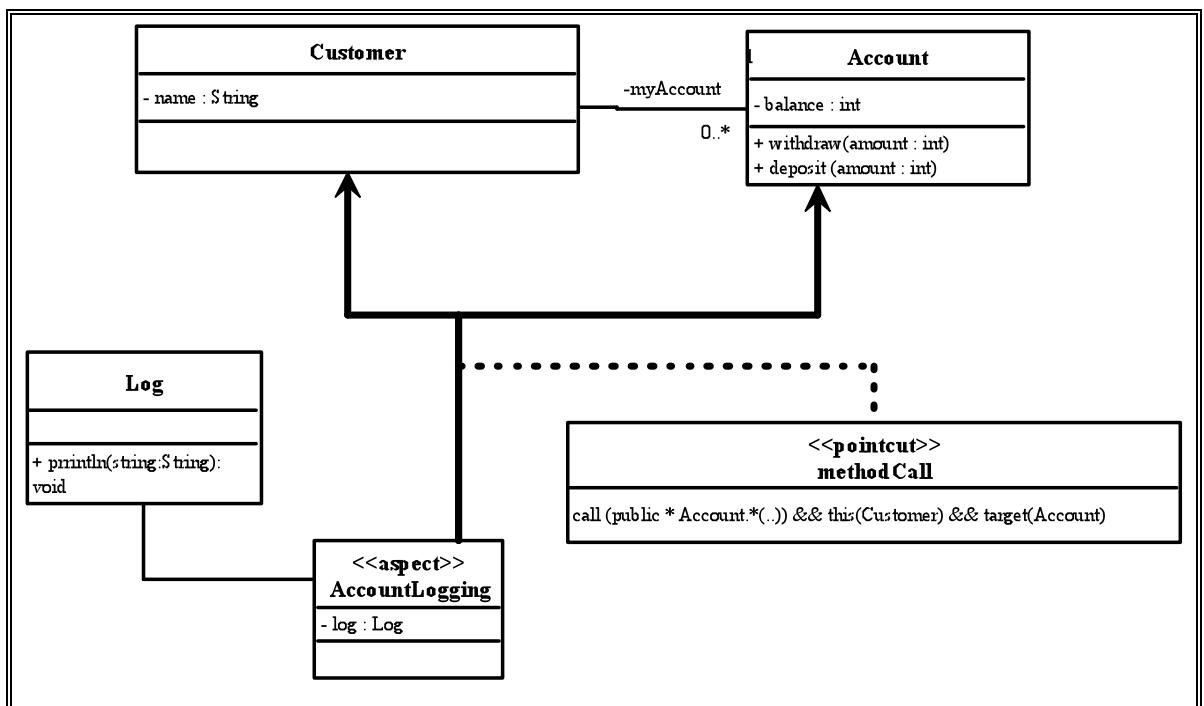


Figure 19 – Logging aspect class diagram

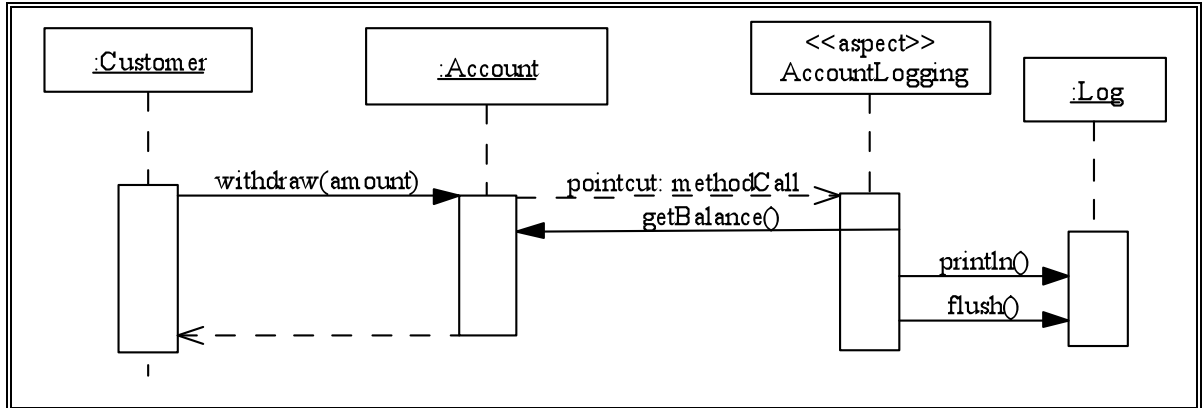


Figure 20 – Logging aspect sequence diagram using our proposed notation

The approach presented by Evermann (2007) shows a meta-level specification and profile for AspectJ in UML. This approach uses XMI (XML Metadata Interchange) to allow the use of CASE tools and to support easy code generation.

From our point of view, an important weakness of Evermann’s approach is the lack of a dynamic view to specify what the advice (i.e., aspect) does when the corresponding pointcut is reached. Our approach covers this issue using two diagrams: a sequence diagram to cover the aspect-system interaction, and a state diagram to specify the internal behavior of the aspect. Figure 21 shows the application of the profile in an example presented by Evermann (2007): we can see that the profile uses a package to identify the crosscutting concern that is stereotyped «CrossCuttingConcern». Inside the crosscutting concern package is the meta-class associated with the aspect using the stereotype «Aspect».

In Figure 22 we replicate part of the same example using our notation. The model shows the aspect `SampleAspect` monitoring all the calls to the `makeLine` method of the class `Figure`. To accomplish that we define `makeLine` pointcut, notice that we cannot replicate what the aspect does when the pointcut is reached, because Evermann’s example does not specify it.

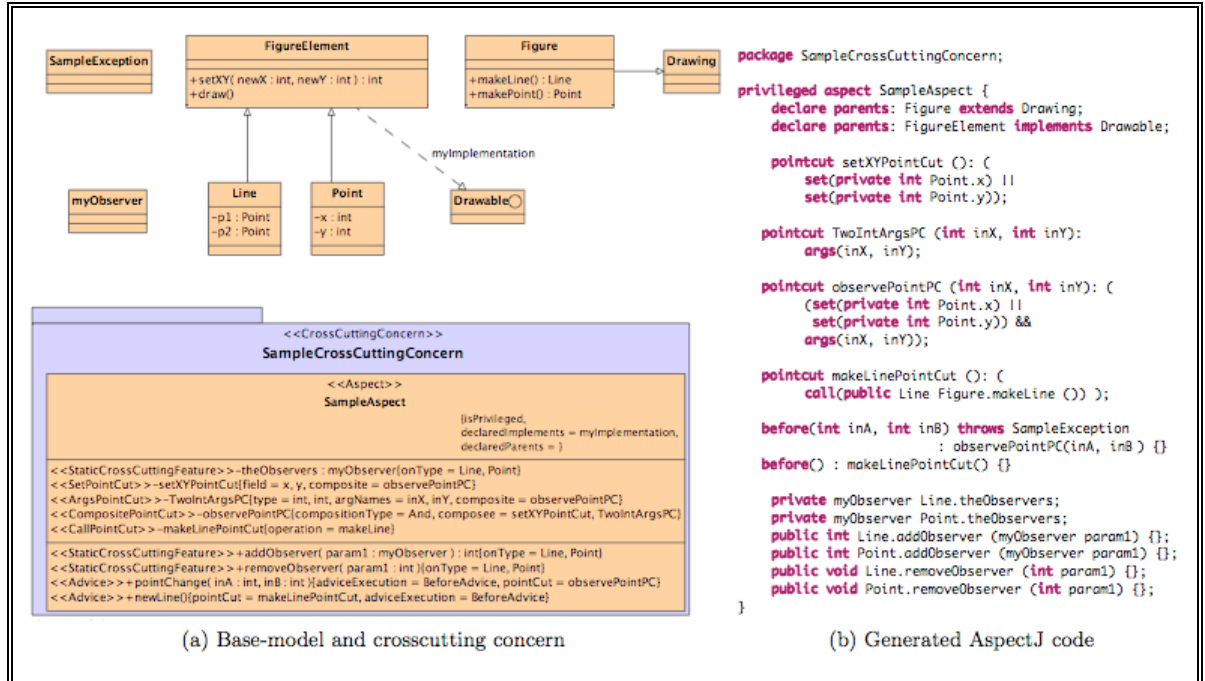


Figure 21 – Evermann's AspectJ profile application example

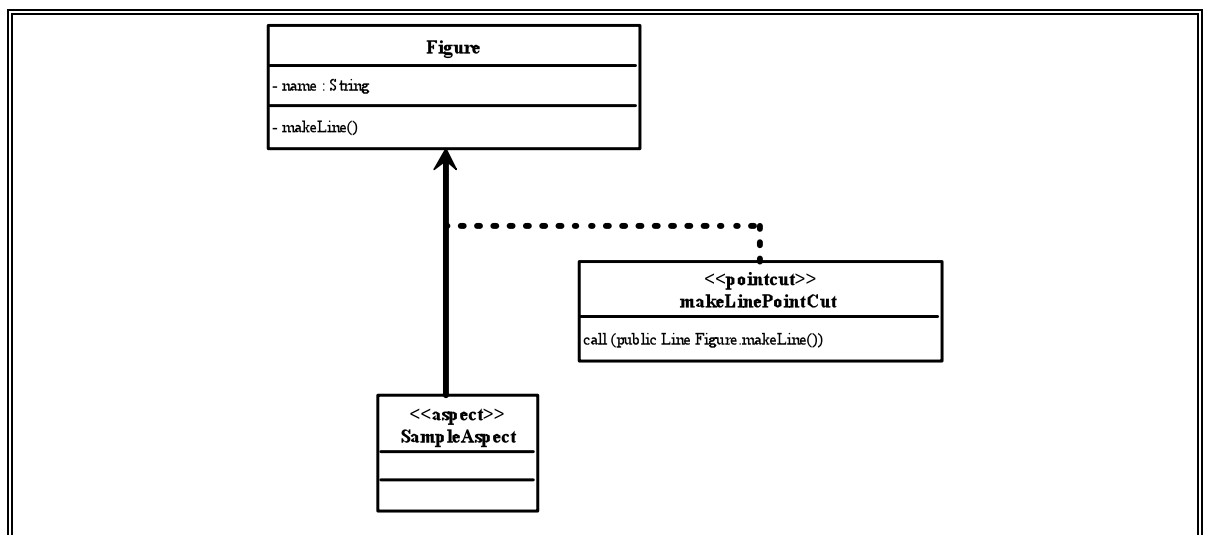


Figure 22 – Evermann's AspectJ profile application example using our proposed modeling notation

Another weakness of the profile described by Evermann (2007) is the lack of a pattern based textual specification of pointcuts, implying that each aspect-oriented feature that refers to a specific model element must be explicitly specified by the modeler. In contrast, our approach uses the AspectJ language pattern notation to select join points for pointcuts. For example, consider the `setXYPointCut` pointcut. The notation of Evermann's approach uses a stereotyped attribute:

```
«SetPoinCut» -setXYPointCut { field=x , y, composite
= observePointPC }
```

On the other hand, our notation is the following:

```
set(private int Point.x) || set(private int Point.y)
```

The use of patterns in the models also allows the use of expressions without full knowledge of the specific join points or type selected by a pattern. This is especially useful during initial designs where the detail level may not be complete.

## 6 CONCLUSIONS

We have described a UML-based notation to represent the following concepts of aspect-oriented designs: the aspect, as the unit that modularizes a crosscutting concern; how this unit interacts with the rest of the system; and the internal behavior of the unit.

Our notation adds few new elements to standard UML, and represents two complementary views of an aspect-oriented design: a structural view, and a dynamic view. For the structural view, we use UML's class diagram. For the dynamic view, we use the sequence diagram—to show the interactions between the aspect and the base system—and the state diagram—to show what the aspect does when a pointcut is reached.

We made these decisions based on the characteristics of an actual industrial setting—brief projects and agile methods—which forces a straightforward design approach. We needed an easy to use, reasonably accurate notation, that combined the aspect and the base system diagrams (rather than having several different new diagrams). The proposed notation allows us to reason about the weaving process between the aspect and the base system, increasing our understanding of the whole system.

We have also proposed to develop the structural views in three, increasingly more detailed stages. Our experience shows that the details of the designs are as varied as the nature of the system being designed, and that they depend on the software development stage.

Our intention is to help the communication of ideas and alternatives about what we are doing, only important issues that we want to run past the developers, or sections of the design that we want to visualize before their implementation begins.

To validate our approach, we are currently using the notation during project management meetings and for communication among developers. While the notation is not as expressive or rigorous as others, initial results show that it can indeed help to achieve good designs, specially when considering time restrictions; for example:

- Some development team's members did not know aspect-oriented concepts. In these cases our notation was used not just for aspect modeling, but also to introduce the aspect-oriented paradigm to the development team. This objective has been accomplished. For example, during the design of a module in the `ContentManager` component (the component without aspects), one developer correctly suggested using an aspect to modularize a particular task.
- To check if the notation could explain what an aspect does, we used it to represent the authentication aspect during a development meeting. We noticed that the developers understood the aspect and its behavior, because they could correctly explain it to us.

Preliminary results observed at the company show that the use of aspect-oriented programming is improving the quality of the resulting applications. There is, however, an important difference between using the technique directly at the implementation level, by refactoring the crosscutting concern code, and using the technique at the design level first, and then writing the code. The second approach allows the development team to discuss ideas and evaluate alternative designs before implementation, and, therefore, produces better implementations. But this is only possible if an appropriate aspect-oriented design notation, in which designs can be quickly sketched, communicated and understood, is available.

We conjecture that other software development teams, working under similar conditions (short projects, with limited time to perform analysis and design), can benefit from using the proposed notation. Our notation should facilitate the adoption of aspect-oriented techniques in general, for example, by groups that are considering applying aspect-oriented programming, and by groups that are doing aspect-oriented programming and wish they could apply the concepts at earlier stages in the development process.

Finally, this is one more work along a line that stresses the importance of modeling aspects with UML. One could hope that a future version of UML would include a specific notation to represent the corresponding semantics.

We currently are considering the addition of new semantic elements to our notation. In particular, we are studying how to represent in UML the exception handling pointcut and inheritance among aspects. Another relevant area is to determine whether or not other UML diagrams are useful to complement our current aspect modeling capabilities. We are considering the composite structure and activity diagrams.

By using the composite structure diagram we are trying to completely separate aspects from the base system. We are studying if the concepts of *required* and *provided* interfaces apply to the particular relationship between aspects and the base system.

By using activity diagrams we could represent the activation of concurrent advices, either of the same aspect or of different aspects. These diagrams could also be used to represent the concurrent execution of the aspect and the base system.

## BIBLIOGRAPHY

- AspectJ Team. (n.d.). *The AspectJ Programming Guide*. Retrieved December 01, 2007 from <http://www.eclipse.org/aspectj/doc/released/progguide/>
- Baniassad, E., Clements, P.C., Araujo, J., Moreira A., Rashid, A. & Tekinerdogan, B. (2006). Discovering early aspects, *IEEE Software*, 23 (1), 61-70.
- Basch, M. & Sanchez, A. (2003, March) *Incorporating Aspects into the UML*. Aspect-Oriented Modeling Workshop on AOSD'03, Boston, Massachusetts, USA.
- Deubler, M. & Krüger, I. (2005). Modeling Crosscutting Services with UML Sequence Diagrams. In Briand L. C. & Williams C. (ed.), *MoDELS'05*, (pp. 522-536). Montego bay, Jamaica: Springer.
- Evermann, J. (2007). A Meta-Level Specification and Profile for AspectJ in UML. In Proc 10<sup>th</sup> *Int. Aspect-Oriented Modeling Workshop on AOSD'07*, (pp. 21-27), Vancouver, British Columbia, Canada: ACM.
- Grassi, V. & Sindico, A. (2006, October). *UML modeling of static and dynamic aspects*. In Aspect-Oriented Modeling Workshop on MoDELS'06, Genova, Italy.
- Hannemann, Jan. (March, 2006). *Aspect-Oriented Refactoring: Classification and Challenges*. In Linking Aspect Technology and Evolution (LATE'06) Workshop on AOSD'06, Bonn, Germany.
- Harel D. & Rumpe B. (2004). Meaningful Modeling: What's the semantic of "semantics", *IEEE Computer*, 37 (10), 64-72.

Kandé, M., Kienzle, J. & Strohmeier, A. (2002, April). *From AOP to UML—A bottom-up approach*. In Aspect-Oriented Modeling with UML Workshop on AOSD'2002, Enschede, The Netherlands.

Kiczales G., Lamping J., Mendhekar A., Maeda C., Videira C., Loingtier J., Irwin J. (1997). Aspect-Oriented Programming. In Proc. 11<sup>th</sup> *European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241 (pp. 220-242). Jyväskylä, Finland: Springer-Verlag.

Larman, C. (2005). *Applying UML and Patterns—An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd ed.). Prentice-Hall.

Manifesto for Agile Software Development. (n.d.). Retrieved September 23, 2007 from <http://www.agilemanifesto.org/>

Mellor, S., (2005) Adapting Agile Approaches to Your Project Needs. *IEEE Software*, 22 (3), 17-20.

Spring Framework. (n.d.). *Aspect Oriented Programming*. Retrieved January 10, 2008 from <http://static.springframework.org/spring/docs/2.5.x/reference/aop.html>

Stein, D., Hanenberg S. & Unland, R. (2002). An UML-based aspect-oriented design notation for AspectJ. In Proc. 1<sup>st</sup> intl. conference on *Aspect-oriented software Development*, (pp. 106-112). Enschede, The Netherlands: ACM Press.

Suzuki, J. & Yamamoto, Y. (1999, June). *Extending UML with Aspects: Aspect support in the design phase*. In Aspect-Oriented Programming Workshop on ECOOP'99, Lisbon, Portugal.

The JBoss Community. (n.d.). *JBoss Aspect-Oriented Programming*. Retrieved January 04, 2008 from <http://labs.jboss.com/portal/jbossaop>