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

Alex Bustos
Computer Science Department
Pontificia Universidad Católica de Chile
Project Manager, Cursor Ltd.
Santiago, 897-0117
alex.bustos@cursor.cl

Yadran Eterovic
Computer Science Department
Pontificia Universidad Católica de Chile
Casilla 306, Santiago 22, Chile
yadran@ing.puc.cl

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 such designs, in particular, for fast, agile developments. In this paper, we describe a UML-based design notation to model the main concepts of aspects, their behavior, and their relationships with the base system. 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. The notation is being used by a company that works on short projects, with limited time for design activities.

KEYWORDS

Aspect Orientation, Software Design, UML.

1. Introduction

We focus on the software design process of a technology integrator company, which works with a major mobile communications operator, both 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 aspect oriented programming (AOP), which allows software developers to modularize crosscutting concerns. Developments in programming languages, such as JBoss [11], AspectJ [1, 16], and Spring AOP [13], and programming techniques, such as refactoring, have been catalysts for the adoption of AOP. While software code has improved, its design hasn't. As a first step, we need a notation to represent aspects at the design level.

There is no standard notation to represent aspects during design; several notations have been proposed [6, 8, 10, 11], but none has been broadly adopted. For us, an important consideration is the limited time devoted to analysis and design, which means that design representations must be right to the point and centered on the main issues, even if this means leaving out detail.

In this paper we propose a way to accomplish this by using UML. UML has been used to represent aspect oriented designs, but those approaches are not appropriate for our purposes. Some replicate a particular language [10], others use several new diagrams [6] or just cover a specific aspects' dimension [7], 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.

Our objective is that all members of the development team understand the aspects and the relationships between aspects and the base system. The design is mainly used during the construction of the system; it is not for maintenance or documentation purposes. In fact, the software life span is only a few months long due to the dynamics of the market. Also, the highly volatile nature of software requirements makes agile development [4] appropriate, which means that the company favors working software over extensive documentation.

The remainder of this paper is organized as follows. Section 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, Section 3 presents our UML-based notation for representing aspect oriented designs. Section 4 summarizes related work in the area and compares those to our work. Finally, Section 5 presents conclusions and describes ongoing work.

2. Design-Level Aspects

Our proposal offers an answer for the particular software development environment described in Section 1. To devise a notation that is expressive without being complex, we must decide what concepts of aspect

orientation should be considered when designing software. In our opinion we should be able to represent the following concepts:

- The **unit** that modularizes the crosscutting concern (i.e., the aspect).
- How this unit **interacts** with the rest of the system (i.e., join points and pointcuts).
- The **internal behavior** of the unit (i.e., advice).

This is consistent with the way in which object oriented designs are commonly represented using UML: we find classes and their relationships, interactions between class instances, and the internal behavior of these instances [9].

2.1 Aspects

An aspect is a unit of *modularity*, *encapsulation* and *abstraction*. It has many similarities with an object oriented class, but it also has some important differences; one difference is the capability of the aspect to implement *crosscutting concerns* in a modular way.

At the design level, we need to represent the unit (similar to a class), the fact that the unit is related to other system's components, and the unit's internal behavior.

2.2 Join Points and Pointcuts

Join points are points in the execution of the code. To make ideas concrete, we consider the following subset of AspectJ join points:

- Method call.
- Method execution.
- Attribute access or attribute update.
- Object initialization.
- Exception handler execution.

We made this selection based on AspectJ because in the industrial setting presented in this work, development is done in Java, AspectJ is a stable and popular AOP language, and AOP languages differ among themselves in terms of join points and pointcuts.

Pointcuts are used to select relevant join points; they act like filters, selecting only those join points that match the pointcut definition. Our notion considers the composition of pointcuts by means of boolean operators, such as `&&` (and), `||` (or) and `!` (not).

At the design level, we need to represent the fact that through the pointcut the aspect knows what join points are meaningful to it, or, in other words, where must the aspect advice the base system. This means that aspects, pointcuts, and classes should be represented in the same design diagram. We represent all the information in the same diagram because in our opinion it is easier to understand an aspect-class relationship if we can see it in one diagram.

2.3 Advice

An advice specifies what an aspect must do when a relevant join point (i.e., a join point specified in a related pointcut) is reached during execution.

An advice is similar to a traditional method; however, advices not only say what to do, they also say when to do it. Following AspectJ, we consider three temporal advice's modifiers: `after`, `before` and `around` (during) the execution of the join point.

Advices can have parameters, but these are *implicitly* passed to the advice; furthermore, no one calls the advice, it is implicitly called when the relevant join point is reached. This is important for behavior diagrams. For this same reason, there is no need of unique identifiers or visibility modifiers (such as `private` or `public`).

At the design level, we need to represent what happens in the aspect when an advice executes; and when does the advice execute with respect to the join points, either before, after, or around.

3. Aspect Modeling Notation

Our goal for software design is to achieve quality software [9]. We devised an aspect oriented design notation which supports the communication among the development team's members. We don't try to cover all aspect orientation topics or make a rigorous translation from AspectJ to UML. In fact, according to Harel [3], it is not a promising approach to try to assign a precise meaning (in terms of a programming language) to a UML-based notation.

Our approach considers two views of the relationships between an aspect and the base system: a static view and a dynamic view. The static view shows which elements in the base system are relevant to the aspect. We represent the **unit** that modularizes the crosscutting concern, and the associations between this unit and elements of the base system, using a class diagram.

The aspect nature is almost completely dynamic; we cover this issue using the sequence and state diagrams. We use sequence diagram to represent the **interaction** between the aspect and the base system. We use state diagrams 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.

In the rest of the section, we describe first a case study, and then each proposed diagram with more detail.

3.1 Case Study

To illustrate our notation, we apply it to the design of bottleneck detection for a real system currently under development for a major mobile communications operator in Chile.

The Content Delivery Platform (CDP) system controls every transaction (download, service subscription, etc.) of

images, themes, ringtones, backtones, applications, etc. The CDP interacts with multiple subsystems, such as SMSC (Short Message Service Center), MMSC (Multimedia Messaging Service Center), SAC (Service Authorization Center, which determines if a client can use or acquire a content/service), Billing, and ReCenT (a centralized statistics platform). The CDP also interacts with external systems.

Clients can request content in many ways: a web page, 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. Therefore, we use aspects to extract crosscutting concerns; in particular, we identified *bottleneck detection* as our first crosscutting concern.

CDP has two components: `Router` and `Content Manager`. The `Router` is responsible for finding the correct route for every requirement received by the system; it analyzes the requirement and determines which subsystem should process it: either the `Content Manager`, for company proprietary content, or an external platform.

The `Router` has more than 130 classes and works using a plug-in approach: incoming requirements are classified by `Finder`, which assigns them to the appropriate `Plugin`, which can delegate part of its work to a `PluginModule`.

Figure 1 shows the system's class diagram; we have omitted classes unrelated to the aspects. Inside `Plugin` and `PluginModules`, many calls to internal and external subsystems are performed. To solve the bottleneck detection problem, we monitor the method calls and store the elapsed time and whether or not the method finished normally. Controlling the elapsed time or monitoring the result of the calls are not part of the responsibilities of the classes.

3.2 Static View

We represent the static view of an aspect using a class diagram, showing the relationships between the aspect 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 2; and then to add detail to this diagram according to the needs of the design, as shown in Figures 3 and 4. We assume that a class diagram for the base system already exists, but probably with few details. We use the stereotype `«aspect»` to differentiate aspects from standard classes.

The first level diagram allows the development team to show/notice quickly which classes are affected by the aspect. This level is ideal for brief meetings and sketches. In Figure 2, we establish the relationships between the `Monitor` aspect (corresponding to the modularization of the bottleneck detection crosscutting concern) and the `SDPUtils`, `ExternalProviders`, `PluginModule` and `Plugin` 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 3 we see that the `Monitor` aspect is associated to two pointcuts: `Benchmark`, which matches join points in two external classes, `ExternalProviders` and `SDPUtils`; and `Logger`, which matches 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. Of course, the utility of this addition depends on the good choice of pointcut names.

Eventually, the level of detail of the base system design will allow the development team to add more detail to pointcut associations. Figure 4 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 classes names to reference them in pointcut declarations.

In the third level diagrams, we change named associations to association classes. In the association class we include the pointcut declaration using AspectJ syntax. E.g. consider the `logPluginException` pointcut declaration:

```
execution (GenericPluginResponse
cl.cursor.router.plugin.Plugin.*(..))
```

This declaration means that the pointcut is interested in calls to any method of `Plugin` subclasses, having at least one parameter and returning an instance of `Generic-PluginResponse`. If we want, we can add constraints regarding who calls the methods or the target objects.

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 loosing the capability of expressing the aspects' concepts mentioned in Section 2.

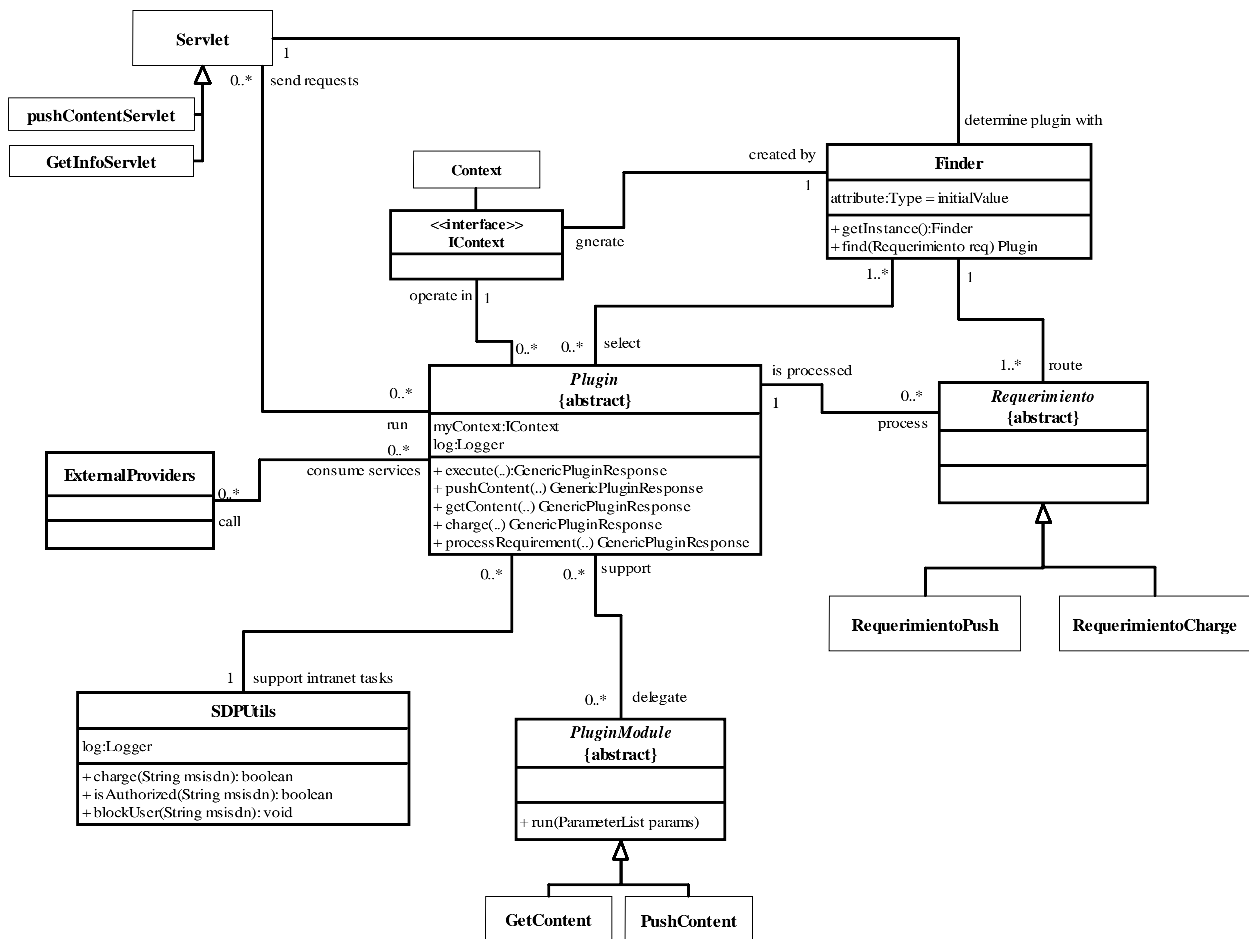

Figure 1 – Case study system: Class diagram of Router component of content delivery platform (CDP)

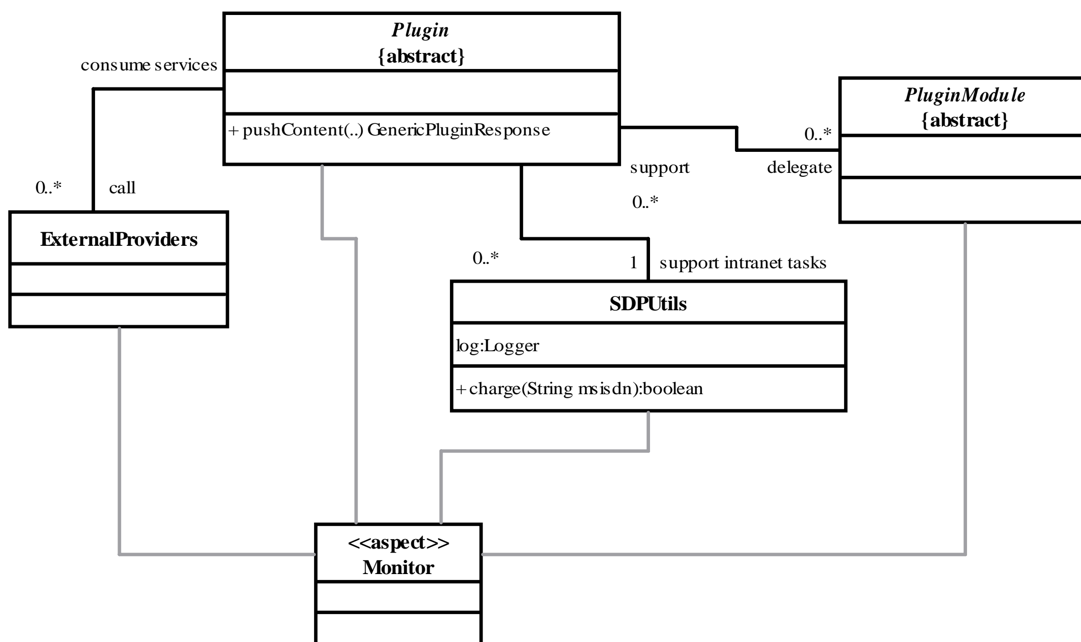

Figure 2 – Monitor aspect's static view: First Level. This level shows a bird's eye of the system generated by the base system and the aspects.

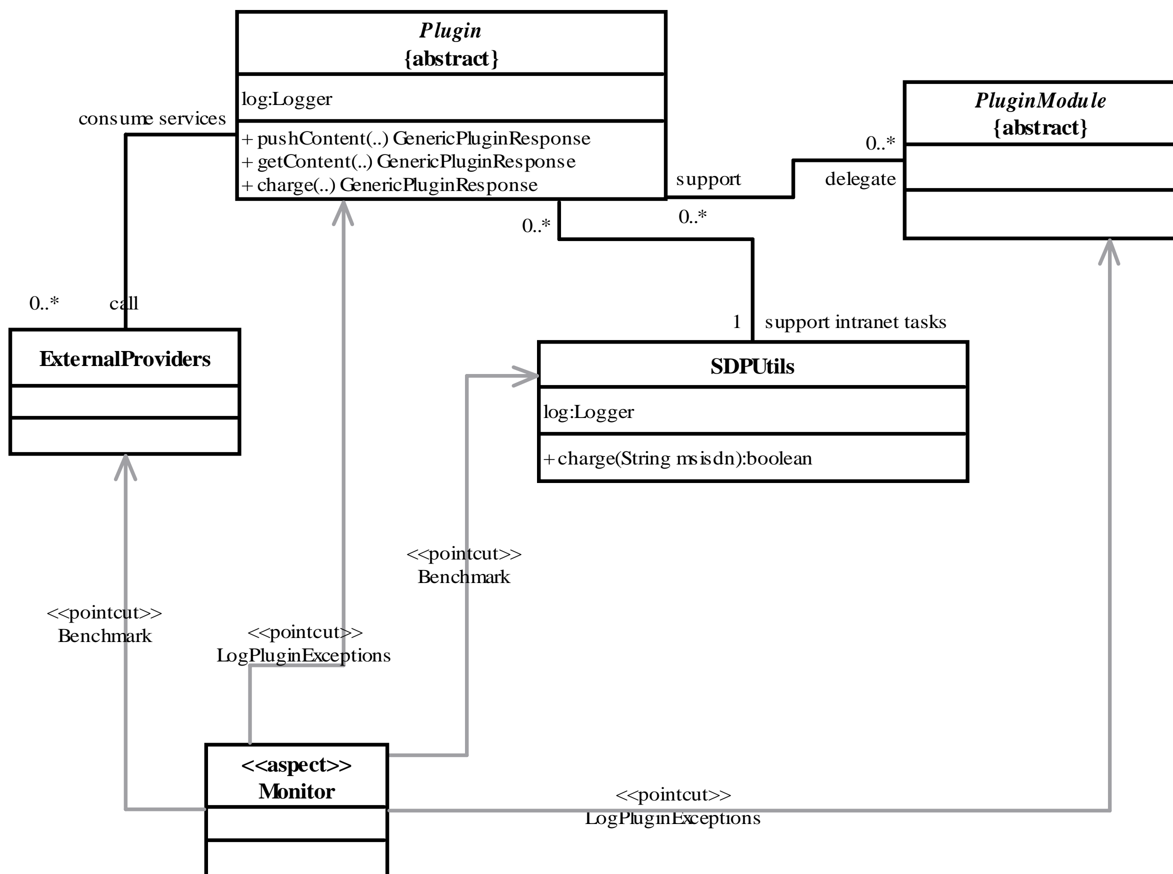

Figure 3 – Monitor aspect's static view: Second level. This level shows named associations for pointcuts.

3.3 Dynamics View

To represent the dynamic 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.

a) Aspect-System Interaction

We represent a pointcut activation in a sequence diagram by a dotted line going from the matching point at the base system class to the corresponding aspect, as shown in Figures 5 and 6. The dotted line stresses the implicit nature of the call, and avoids confusing this call with standard sequence diagram's calls. Above the dotted line, we optionally write the pointcut definition. We next show three commonly used pointcuts: execution, this, and target.

An *execution* pointcut activates just before a called method is executed. The syntax is the following:

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

Figure 5 shows message `pushContent(IReq req)` sent by `PushContentServlet` to `GenericPlugin`. This call triggers an implicit call to the `PluginLogger` aspect due to the match 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 (`proceed` signal). Once the method finishes, the control goes back to the aspect, which performs a recording through the `Recorder` class. Finally, the aspect returns to the `PushContentServlet` object.

A t*his* pointcut is commonly used in combination with other pointcuts, such as `execution`. We can select join points according to the object that activates the pointcut:

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

With the `this` pointcut, we can add a constraint to aspect intervention, indicating that only calls to method `pushContent` made by `PushContentServlet` will match the pointcut.

A *target* pointcut references the destination object of the pointcut:

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

Figure 6 shows the use of `target`, `this`, and `execution` pointcuts in conjunction. The result is that we select all executions of the `pushContent` method of the `GenericPlugin` class with `IReq` parameter, coming from a `PushContentServlet` object.

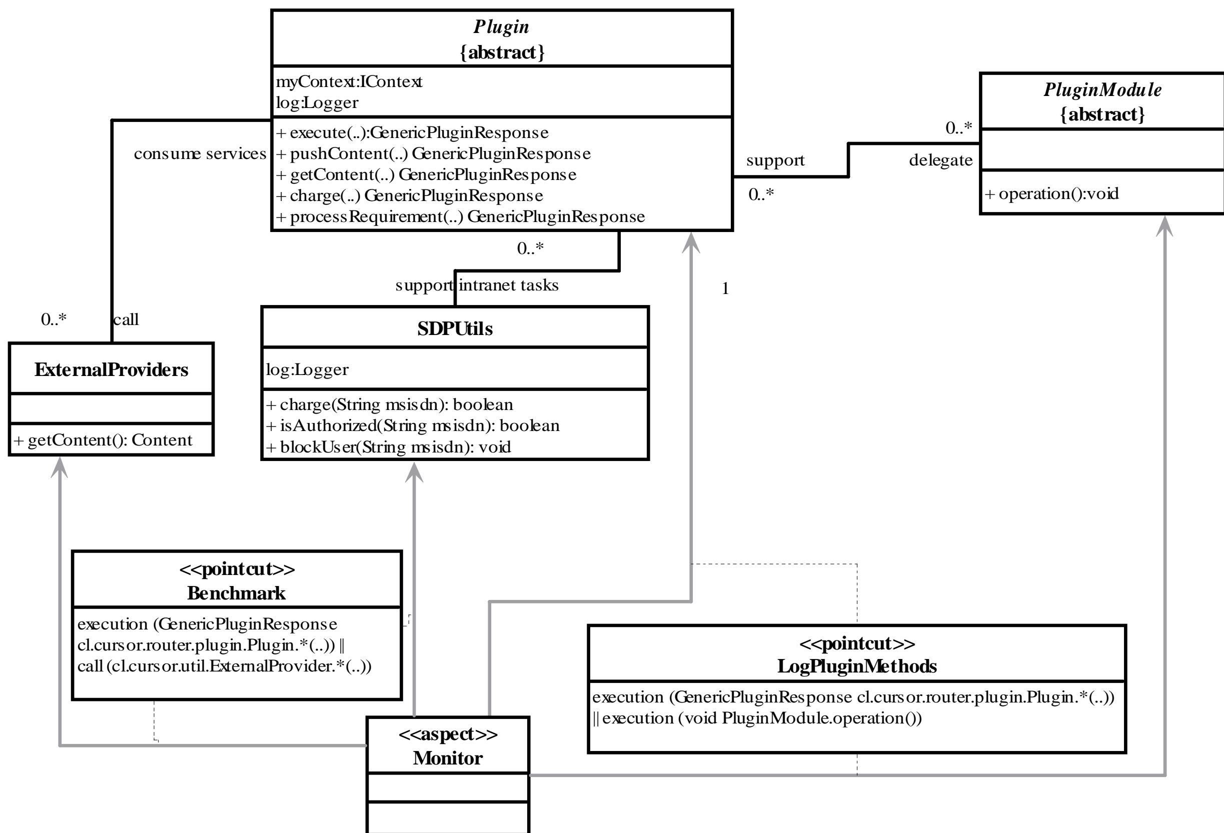

Figure 4 – Monitor aspect's static view: Third level. This level shows pointcuts as association classes.

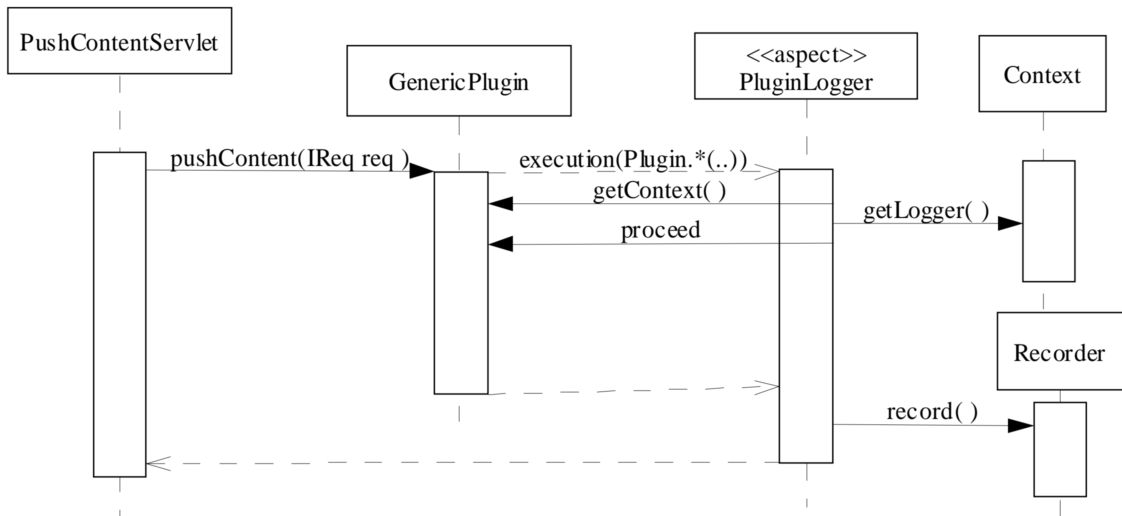

Figure 5 – Sequence diagram: *execution* pointcut.

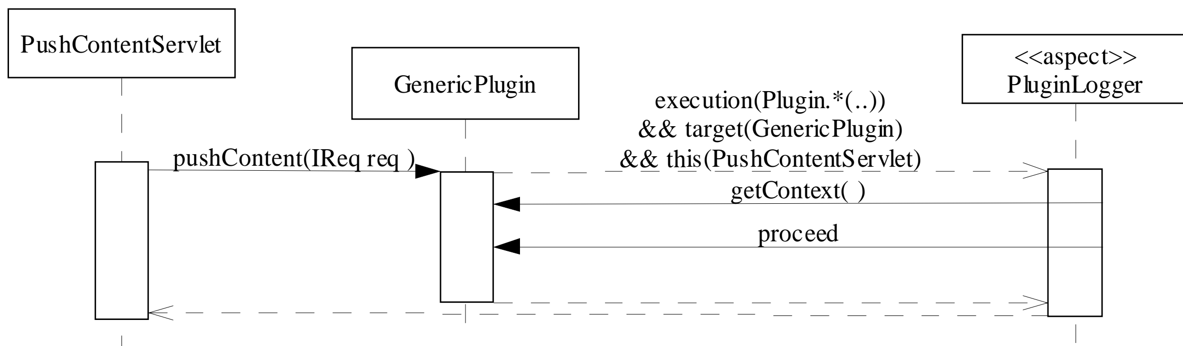

Figure 6 - Sequence diagram: `this` and `target` pointcuts.

b) Aspect Internal Behaviour

We use state diagrams to represent the internal behavior of aspects. State diagrams show *what* an aspect does when a pointcut is reached and *when* it does it. Figure 7 shows the behavior of the benchmark aspect. Initially. the aspect is *waiting* for a pointcut to be reached; when the pointcut is reached, a transition is triggered. We use a dotted arrow to represent this transition, to imply that the triggering event is the matching of a 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 4, the `Benchmark` pointcut). The second element indicates when does the advice occur; its possible values are `before`, `after`, and `around`.

The rest of the diagram is a standard state diagram. 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 a `proceed` signal, as seen in Figure 6). Once the original method finishes, the aspect stops the timer, records the elapsed time, and unconditionally transitions to the initial *Waiting* state.

4. 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. [2] 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 behaviour of aspects.

Stein et al. [10] 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.

Suzuki et al. [7] describe the design of introductions, i.e., structural modifications of system classes, through a new UML meta-class. However, they do not explain how to represent pointcuts; and, in our opinion, it is not clear that introductions is a desirable feature of aspects. Also, they focus on the interchangeability of aspect models between development tools. But in the particular industrial setting presented in this work, where the use of CASE tools is not common, this issue is not relevant.

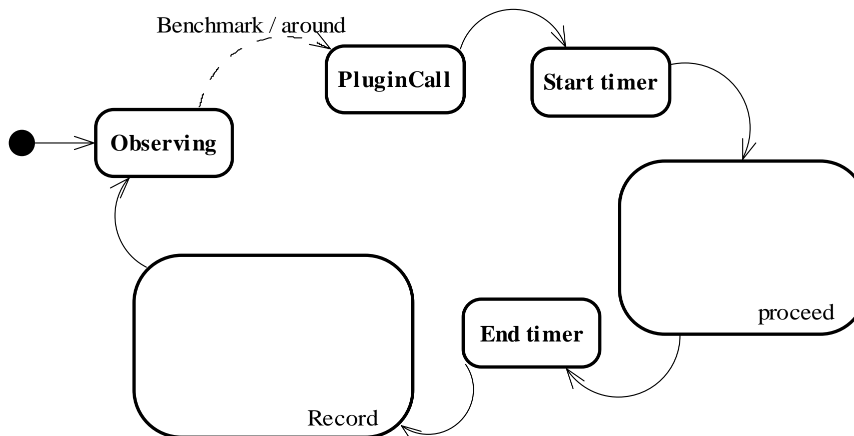

Figure 7 - Benchmark pointcut state diagram

Grassi et al. [6] propose 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.

The 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. promote a total separation of aspects from the base system; but, in our opinion, aspects are intrinsically tied to the base system, so it is natural to represent them in the same diagram.

Kandé et al. [8] propose a collaboration stereotype to represent introductions and the interactions with the rest of the system through the use of connection points. This approach merges the dynamic and static dimensions of the aspect in the same diagram, generating a complex model. Furthermore, it is not clear how to represent other pointcuts with the notation.

5. 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 static view, and a dynamic view. For the static 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 aspect and 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 static views in three stages, increasingly more detailed. 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.

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

We are now 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.

References

[1] AspecJ Team, The AspectJ Programming Guide. http:// www.eclipse.org/aspectj/doc/released/progguide/

[2] Baniassad, E. et al., Discovering early aspects*, IEEE Software* Jan./Feb. 2006.

[3] Harel D. Meaningful Modeling: What's the semantic of "semantics", *IEEE Software*, Oct. 2004.

[4] Mellor, Stephen J. Adaptig Agile Approaches to Your Project Needs. *IEEE Software* May/June 2005.

[5] Deubler M., Krüger I. Modeling Crosscutting Services with UML Sequence Diagrams, In L. C. Briand and C. Williams, editors, MoDELS, volume 3713 of LNCS, pages 522--536. Springer, Oct. 2005.

[6] Grassi, V. Sindico, A. UML modeling of static and dynamic aspects. *Aspect oriented modeling*, Oct. 2006.

[7] Suzuki, J., Yamamoto, Y. Extending UML with Aspects: Aspect support in the design phase. *3rd Aspect-Oriented Programming Workshop, ECOOP'99*, Lisbon, Portugal, June 1999.

[8] Kandé, M., Kienzle, J., Strohmeier, A. From AOP to UML—A bottom-up approach. *2nd Intl. Workshop on Aspect-Oriented Modeling with UML*, (UML2002), Dresden, Germany, April 2002.

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

[10] Stein, D., Hanenberg, S., Unland, R. An UML-based aspect-oriented design notation for AspecJ. *Proc. 1st Intl. Conf. on Aspect-oriented software development*, Enschede, The Netherlands, April 2002.

[11] JBoss Aspect Oriented Programming,

http://labs.jboss.com/portal/jbossaop

[12] Aspect J, Eclipse, http://www.eclipse.org/aspectj/

[13] Spring framework

http://www.springframework.org/docs/reference/aop.html