Guidelines for Aspect-Oriented Design

Christina von Flach G. Chavez\textsuperscript{1} Carlos J.P. Lucena\textsuperscript{2}

\textsuperscript{1}UFBA, Computer Science Department
Av. Adhemar de Barros, s/n − 40170-110, Salvador, BA, Brazil
\textsuperscript{2}PUC-Rio, Computer Science Department
R. Marquês de Sã o Vicente, 225 − 22453-900, Rio de Janeiro, RJ, Brazil
email: flach@im.ufba.br, lucena@inf.puc-rio.br

Abstract

This position paper presents some principles for the design of aspect-oriented systems and preliminary guidelines for aspect-oriented design using the \texttt{aSideML} modeling language.

Since aspect-oriented programming is a young discipline, with clear focus on implementation issues and programming style and naturally, with few guidelines available to support aspect-oriented design and modeling, our informal assessment of principles and the initial set of guidelines set up the ground for discussion and also constitute a contribution to the area.

1. Introduction

Aspect-oriented design (AOD) is a relatively young area, and design knowledge, including generic design patterns and guidelines, is expected to emerge as practice advances. Some AOD-related principles and properties have already emerged. However, there are some well-known principles in software engineering, currently recommended for the design of object-oriented systems, that may be affected by the concepts and pragmatics of aspect-oriented programming (AOP). These sets of principles, conventional or emergent, can be used as a starting point to suggest general guidelines for AOD. Guidelines for AOD document modeling experience that may be useful in different categories of domains and facilitate the implementation using aspect-oriented languages.

In this position paper we present some principles for the design of aspect-oriented systems and preliminary guidelines for AOD using the \texttt{aSideML} modeling language \cite{2}. The guidelines presented here have been derived from our experience in using AspectJ \cite{1} for implementing different kinds of applications (e.g. the Portalware MAS \cite{8}) and also from some canonical examples available from the AspectJ team and the practitioners’ community.

This paper is organized as follows. In Section 2 we present some principles used in object-oriented design in order to discuss the way AOD deals with them. In Section 3 we present some guidelines that are explained in plain text, and illustrated using the \texttt{aSideML} modeling language. Section 4 presents related work and Section 5 presents final remarks.

2. Principles

Software design is replete with principles and techniques for managing \textit{module dependencies} with the goal of promoting easy evolution and reuse. Some general principles are presented in Table \ref{tab:principles}, and principles used in object-oriented design are presented in Table \ref{tab:oop_principles}.
Principle | Definition
---|---
Separation of concerns | Every important issue (or concern) should be considered in isolation [5].
Low coupling | Every module should communicate with as few others as possible [27].
Weak coupling | If two modules communicate at all, they should exchange as little information as possible [20].
Information Hiding | All information about a component should be private to a component unless it is specifically declared public [22].
Cohesion | Logically related components should be grouped together [27].

Table 1: General Principles.

Structured design exhibits a particular kind of dependency structure, that starts at the top and points down towards details. *High level modules depend upon lower level modules*, which depend upon yet lower level modules.

Object-oriented design presents a dependency structure in which *the majority of dependencies point towards abstractions*. Moreover, the modules that contain detailed implementation are no longer depended upon, rather they depend themselves upon abstractions. Therefore, the dependency structure provided by structured design has been inverted [19].

<table>
<thead>
<tr>
<th>Principle</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Open Closed Principle (OCP)</td>
<td>A module should be open for extension but closed for modification [20].</td>
</tr>
<tr>
<td>The Liskov Substitution Principle (LSP)</td>
<td>A derived class may substitute a base class [18].</td>
</tr>
<tr>
<td>The Dependency Inversion Principle (DIP)</td>
<td>High-level modules should not depend upon low-level modules. Both should depend upon abstractions [19].</td>
</tr>
<tr>
<td>Stable Dependencies Principle (SDP)</td>
<td>Depend on the direction of stability [19].</td>
</tr>
<tr>
<td>The Interface Segregation Principle (ISP)</td>
<td>Many client specific interfaces are better than one general purpose interface [19].</td>
</tr>
<tr>
<td>The Law of Demeter (LoD)</td>
<td>Each unit should have knowledge only about closely related units [16].</td>
</tr>
</tbody>
</table>

Table 2: Principles related to Object-Orientation.

Aspect-oriented design (AOD) can be defined as an approach to software design that supports separation of concerns along two dimensions of decomposition: a component-based dimension and an aspect-based dimension. AOD encompasses the process of *aspect-oriented decomposition* (in addition to object-oriented and algorithmic decomposition) and *new design principles and properties*. AOP techniques interfere with the principles described above by providing a new kind of *modular dependency*: the dependency between aspects and components.

The aspect-oriented decomposition of AspectJ applications, for example, yields to a dependency structure in which the dependencies start at the aspect modules and point towards components (Figure 1). AspectJ also promotes the *aspect-base dichotomy*, that is, the adoption of a clear distinction between *base components* and aspects. Furthermore, it requires *quantification* support and may promote some level of component *obliviousness*.

*Quantification* and *obliviousness* were proposed as the core characteristic of AOP, and have been used to define whether a language is aspect-oriented or not [6]. These new properties may
be supported by AOD, although nowadays many researchers in the AOSD community would refute one or both of "aspect-base dichotomy" and "obliviousness".

**AOP and Design Principles.** We provide our preliminary view about the way AOP techniques based on the aspect-base dichotomy interfere with existing design principles, and their potential to reduce or improve the dependencies in a system, in order to further characterize AOD.

**Separation of Concerns** AOP technologies propose innovative abstractions and mechanisms to improve (or advance) the separation of concerns provided by the object-oriented paradigm. Separation of Crosscutting Concerns (SCCC) is a new design principle [15] promoted by AOD.

**Coupling** AOP supports low coupling among components. Aspects depend on components and components may not depend on aspects when the obliviousness property holds. Aspects modularize features that otherwise would be scattered and tangled up with several components. Therefore, the use of aspects may minimize dependencies among components. Low coupling among aspects is also required. In practice, when several aspects affect the same single model, aspects may be coupled to each other in several ways.

**Cohesion** AOP supports modules that are more cohesive. Cohesion is a measure of the binding between the elements of a single module. Aspects localize concern-related elements that otherwise would be scattered and tangled up with several component modules; these modules tend to be more cohesive.

**Information Hiding** Whenever possible, AOP promotes information hiding. Information hiding (also known as encapsulation) has received considerable interest in the context of OOP. In particular, the encapsulation of objects that are composed through inheritance has been discussed repeatedly [24]. Some approaches to AOP provide means to break the encapsulation of the implementation details of concerns and allow aspects to handle classes’ private features. On the other hand, aspects encapsulate and localize features that otherwise would be scattered and tangled up with several components.

**OCP** AOP promotes OCP. The *OCP principle* helps designers to create modules that are extensible, without being changed. This means that new features can be added to existing
code, without changing the existing code and by only adding new code. With AOP, the base classes are the closed part and aspects are the open part [21].

**LSP** AOP may help to promote LSP between classes. The *LSP principle* helps the designer to promote *strict inheritance*, that is, inheritance that prescribes behavior compatibility between classes and their subclasses. The LSP goes beyond the simple checking of signatures, taking into account the behavior of operations (as defined by their pre and post conditions) and the invariants defined by classes. Aspects may localize redefinitions and additions that eventually break the LSP, that otherwise would be organized around subclasses. However, clients may also have their behavior enhanced to deal with the objects that are enhanced by these aspects. Aspects are also useful to modularize pre and post conditions to be composed to class operations.

**DIP** AOP may promote DIP. The *DIP principle* guides the designer to depend upon interfaces and abstract classes, since concrete things change more frequently. No dependency should target a concrete class. Aspects should be composed as higher as possible in a class hierarchy [21].

In another dimension, DIP is related to SDP, the *Stable Dependencies Principle* [19]: “*Depend on the direction of stability*”. If we consider that functional classes are more stable than aspects, then the SDP prescribes the dependency of aspects upon classes.

**ISP** AOP may promote ISP. The *ISP principle* states that, if a class has several clients, rather than overloading the class with all the methods that the clients need, the designer shall create specific interfaces for each client and multiply inherit them into the class. Aspects support the separation and modularization of views that may be later woven to several classes.

**LoD** The LoD is a design principle that states that a method should only contain message sends to self, local instance variables, and/or method arguments. By following the LoD, we avoid long sequences of accessing methods; however, a direct consequence of the application of the LoD is a large number of the small “information-passing” methods. AOP techniques such as Adaptive Programming [17] deal with these consequences and therefore, promote the LoD.

### 3. Guidelines

In this Section, we present one rule and some guidelines for AOD and use the aSideML aspect diagrams to illustrate them. The presented guidelines may promote one or more AOD principles discussed earlier.

**The aSideML.** The aSideML is a modeling language for specifying and communicating aspect-oriented designs. It provides notation, semantics and rules with the main purpose of addressing the conceptual modeling of a system in terms of aspects and crosscutting within the UML Object Model. The complete specification of the aSideML can be found at [2].

In the aSideML, an aspect is drawn as a dashed rectangle, with a diamond symbol containing the aspect name, that touches a rectangle that is used to describe local features (see Figure 2). A *crosscutting interface* is a named set of crosscutting features (structural or behavioral) that characterizes the crosscutting structure and behavior of aspects. A crosscutting interface is declared inside aspects and is drawn as a solid-outline rectangle with compartments separated by horizontal lines. *Crosscutting features* are listed in different compartments, depending on the
kind of enhancement they support. The Additions compartment lists data and operations to be introduced in classes, the Refinements compartment lists crosscutting operations to be combined before, after or before/after class operations and the Redefinitions compartment lists crosscutting operations that override class operations.

3.1. A General Rule for AOD

Try to maintain a stand-alone object model, which aspects extend [13]. The adoption of this general rule promotes the aspect-base dichotomy and obliviousness.

3.2. Aspects and Obliviousness

Aspects may add crosscutting features to base classes that may be explicitly used by other classes or even aspects. In such cases, the classes and aspects are not oblivious about the former aspect.

We present one aspect-oriented solution to deal with the undesirable dependency between classes and aspects whenever an explicit call to an introduced method is made from a class. The introduced operation should not be explicitly invoked from the other class. The proposed solution promotes low coupling and obliviousness.

Scenario. The base class that is extended by the addition, also has a method that can be extended by a crosscutting operation that contains an explicit call to the new method.

Intent. Minimize dependencies between classes and aspects; avoid explicit calls to introduced methods from base classes.

Solution.

Introduce additional operations in one class and advice an original method in the same class. Use a crosscutting interface to modularize the addition and the refinement. Removing the aspect from the model does not leave inconsistencies.

Structure.

Figure 2: Operation Addition and Refinement using One Crosscutting Interface
3.3. Aspects for Collaboration

A collaboration in an object-oriented program often involves several different collaborating classes. However, since collaborations are not first-class citizens in object-oriented design, they either are localized into one method on one of the classes, or are divided into methods on each of the classes involved. The latter solution scatters the collaboration across multiple classes, making it difficult to adapt when the collaboration changes.

Aspects can be used to localize collaborations among participating classes. In this case, aspects affect classes heterogeneously, and therefore, require the support of two or more crosscutting interfaces.

**Intent.** Localize a collaboration among classes and emphasize the participant roles.

**Solution.**

Define an aspect with one crosscutting interface for each participant role.

The aspect localizes the interactions among participating classes and rules that govern those interactions. The crosscutting interfaces localize the behavior related to each kind of participant.

**Description.**

The aspect names a set of interaction points in its crosscutting interfaces. Each crosscutting interface describes a separate participating class and may enhance it in order to perform a specific role in the collaboration. Aspect’s local state and behavior are used to specify the collaboration protocol.

Aspects for collaboration can act as software connectors and insert structure and behavior in precise, unanticipated ways into the parts they connect. We refer to this crosscutting flavor as the collaborative dimension of crosscutting.

**Example.**

The Telecom (c.f. [14]) is a system that comprises a simple model of telephone connections to which timing and billing features are added using aspects. The Timing feature is concerned with timing the connections between customers and keeping the total connection time per customer. If a connection is completed, a timer must be started. When a connection is dropped, the timer must be stopped. The time elapsed during a connection must be associated to the payer (the customer that made the call). Figure 3 presents the Timing aspect.

![Figure 3: The Timing aspect.](image)

3.4. Aspects for Evolution

**Intent.** Evolve a base class so that it provides new services, or is able to play a new role, without invasive changes.
Solution.

Define an aspect with one crosscutting interface that implements an interface or extends a class. Use the Additions compartment to declare the attributes and methods to be implemented.

Example.

Figure 4 illustrates the use of aspects to make points cloneable elements [1]. The aspect will be enhancing classes denoted by `CloneableElement`.

3.5. Aspects for Views

**Intent.** Address the need for subjective views to objects.

A class may have many clients. Instead of defining one large general interface to serve them all, the client’s needs may be localized into smaller, separated views, specific for each kind of client. Those views are associated to a base class, and implemented there.

Subjective aspects can be used to address the need for this segregated views to objects, enforcing the ISP principle.

**Solution.** Aspects can be used to support interface segregation.

- A class provides a default view that is available for every client.
- An aspect localizes new private attributes and methods that correspond to a new view using a crosscutting interface. These features are introduced in the base class (the server class).
- The same aspect, by means of another crosscutting interface, enhances other classes (the clients) with additional behavior that allows them (and only them) to use the private services introduced into the server class.

**Example.**

Subjective aspects also support context-dependent behavior enhancements, such as the need to log, to monitor or to give access to some behavior only when observed under a certain perspective, or invoked by a certain caller. Aspects may crosscut only selected method calls, depending on the caller type, and provide the additional behavior in a per-client basis.

4. Related Work

The AspectJ community provided several contributions in the form of idioms, recipes, rules and patterns for the AspectJ language [9, 11, 12, 10]. They describe solutions which proved to be useful in constructing past systems with AspectJ. Some patterns have also been proposed for
specific domains or concerns such as multi-agent systems [7], distribution [25], etc. These solutions are based on experience and shall increase with the development of new aspect-oriented systems.

Aspect-oriented design and modeling have been addressed by several research groups. Stein’s Aspect-Oriented Design Model (AODM) [26] presents a design model that complies with the semantics of AspectJ. Clarke and colleagues have done significant work on AOD; their former research on the Composition Patterns [3] approach evolved to Theme/UML [4], with the goal of providing a “standard AOSD design language”. A comparison between the aSideML, the AODM and Composition Patterns can be found at [2].

5. Final Remarks

We have presented some principles for the design of aspect-oriented systems and preliminary guidelines for AOD using the aSideML modeling language. Ongoing work includes documenting other common uses for aspects at the design level (aspects for contract enforcement, for incremental extension, etc.).

Measuring structural design properties of software artifacts, such as coupling, cohesion, separation of concerns, and others derived from the principles listed in Section 2, is a promising path towards early quality assessments of AOP systems. In [23], we have presented a framework, which is based on a suite of metrics and a quality model, to assist the assessment of aspect-oriented software that deals with new abstractions and new dimensions of coupling and cohesion introduced by AOD. Further experimental investigation is required to assess the way AOP interferes with other important principles.

Since AOP is a young discipline, with clear focus on implementation issues and programming style and few guidelines available to support AOD, our informal assessment of principles and the initial set of guidelines also constitute a contribution to the area.

References


