From AOP to UML - A Bottom-Up Approach

Mohamed Mancona Kandé, **Jörg Kienzle** and **Alfred Strohmeier**

Software Engineering Laboratory
Swiss Federal Institute of Technology Lausanne
CH - 1015 Lausanne EPFL
Switzerland
email: {Mohamed.Kande, Joerg.Kienzle, Alfred.Strohmeier}@epfl.ch

Abstract

This position paper takes a bottom-up approach that analyzes the suitability of UML for modeling aspect-oriented software, and compares it with the UML support for modeling object-oriented software. We first introduce the basic concepts of AspectJ, a state-of-the-art aspect-oriented programming language, and then take a naïve approach using standard UML, as it is, for modeling these concepts. As the limitations of current UML become apparent, we propose some extensions to UML to overcome these limitations.

1 Introduction

Aspect-oriented programming (AOP) is the name given to a set of techniques based on the idea that software is better programmed by capturing different "things that happen in a program" in different ways, and by encapsulating them into different modules [1]. This approach makes it possible to separately specify various *kinds of concerns* and localize them into separate units of encapsulation. One can deal with both the concerns and the modules that encapsulate them at different levels of abstraction, not only at the code-level [2]. Examples of kinds of concerns in a software system include functionality, emerging system-level properties, and other qualities. Representing certain kinds of software concerns is well supported by modeling languages, like UML [3], if these concerns can be localized on a single component of a system, such as a class. Modeling others kinds of concerns, e.g., logging, transactions and security, is more difficult, as they typically cut across the boundaries of many components of a system. In this work, we consider two issues that we believe need to be fundamentally addressed when providing support for modeling crosscutting concerns:

- Understanding what concerns cut across which representational elements and where they do so. Without this information, it becomes very hard to represent and reason simultaneously about the crosscutting structure and the behavior in the system.
- Provide a means to separate crosscutting from non-crosscutting concerns and encapsulate the former into aspects.

To tackle these issues, we take a bottom-up approach that starts with the key concepts used to represent crosscutting concerns in aspect-oriented programs. Based on this and on our experience in modeling object-oriented software with UML, we analyze the suitability of UML to support aspect-oriented software modeling.

One of the main elements of an aspect-oriented programming language is the *join point model*. It describes the "hooks" where enhancements may be added, determining thus the structure of crosscutting concerns. To support this model, AOP languages are required to provide means to identify join points, specify behavior at join points, define units that allow one to group join point specifications and behavior enhancements together, as well as means for attaching such units to a program [1].

A second important element in AOP is the "weaving" capability support. When using AOP languages, the programmer relies on the underlying AOP environment to weave or compose separate concerns together into a coherent program. Separating the representation of multiple kinds of concerns in programs in such a way promises increased readability, simpler structure, adaptability, customizability and better reuse.

UML is a standard modeling language used to create and document software artifacts. It includes many useful ideas and concepts that have their roots in various individual methods and theories. UML provides numerous modeling techniques, including several types of diagrams, model elements, notation and guidelines. These techniques can be used in various ways to model different characteristics of a software system. Key features of UML comprise: support for model refinement, extension mechanisms (stereotypes, tagged values, and constraints), and a language for expressing constraints (known as the object constraint language, OCL). UML has established itself as a well-accepted modeling language that provides adequate support for object-oriented and component-based software development [4].

Basically, there are various possibilities of using UML to model crosscutting concerns in a software system. For instance, join points can be represented in UML as model elements, but their effect can also be shown in different diagram types of UML, e.g., collaboration, sequence and statechart diagrams. Now, the question that

remains to be asked is how suitable is UML, in its current state, for modeling aspect-oriented software systems?

To answer this question, this paper takes a bottom-up approach, establishing parallels between AOP code and UML models. Section 2 introduces the key concepts of AspectJ, the aspect-oriented programming language we used for our experiment. Section 3 shows how some, but not all, AOP concepts can be modeled in standard UML. Section 4 identifies extensions to UML that allow us to better capture the essence of aspect-oriented modeling, and finally, in section 5 we draw some conclusions from this experiment.

2 AspectJ

AspectJ [5] is an aspect-oriented programming environment for the Java language. It has served as a basis for our experiment.

AspectJ defines the notion of *join point* as a well-defined point in the execution flow of a Java program. Join points include method and constructor calls or executions, field accesses, object and class initialization, and others. A set of join points is called a *pointcut*. To pick out a set of join points of a certain kind, a programmer uses a *pointcut designator* of a certain kind and pattern matching techniques to specify the method signature, classes, or packages of interest. Pointcuts can be further composed with boolean operations to construct other pointcuts.

While pointcuts allow the programmer to identify join points, the *advice* constructs define additional code to run at those join points. An advice contains a code-fragment that executes either *before*, *after* or *around* a given pointcut. Finally, *aspects* are provided which, very much like a class, group together methods, fields, constructors, initializers, but also named pointcuts and advice. Aspects are intended to be used for implementing a crosscutting concern.

3 Supporting Aspect-Oriented Modeling

This section addresses UML support for object-oriented modeling, and discusses the appropriateness of current UML for modeling of aspect-oriented software systems.

Using UML for Object-Oriented Modeling

Modeling object-oriented concepts is well supported by UML. Consider, for example, a simple banking system, which the following Java classes Account and Customer are part of:

```
public class Account {
  private int balance = 0;
  public void withdraw (int amount) {...};
  public void deposit (int amount) {...};
  public int getBalance() {...};
}
```

```
public class Customer {
  public String name;
  // inside some method (a is an account)
  a.withdraw(50);
}
```

To model the static and dynamic structure shown in the Java code, two types of UML diagrams could be used, i.e., a collaboration diagram and a class diagram, each providing a different view. The collaboration diagram shown in Figure 1 illustrates a behavioral view that focuses on the interaction between a Customer object c invoking a method of an Account object a. Collaboration diagrams typically give you an idea about how objects work together, showing both the participant objects (part of the structural characteristics) and the interactions between them (the messages they exchange). In our example, c calls the method withdraw of a, which is shown in the figure by the arrow labeled with withdraw(50).

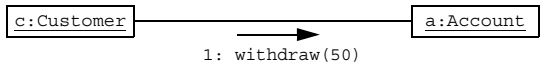

Figure 1: A UML Collaboration Diagram

As in this case no further information is given on how the customer c gets to know the account a, we must assume that they are statically linked together. This results in an association link between the two classes, as depicted by the class diagram in Figure 2. Thus, Figure 2 shows a static structure view of (this part of) the system with a particular focus on how the classes relate to each other.

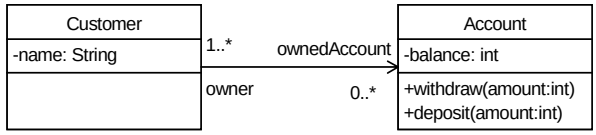

Figure 2: A UML Class Diagram

In Figure 1, each interacting object plays a different role. UML allows us to specify this by attaching role names to both association ends, shown as owner and ownedAccount in Figure 2. Multiplicity constraints can also be attached to association ends in order to express the possible number of occurrences of the association. Note that this information is not shown in the Java code.

As the simple banking system evolves, the requirements change. Developers might be asked to add the following new feature to the system: every access to an account object should be logged, recording the name of the accessing customer and the type of access on a log file. This logging feature is a typical example of a crosscutting concern, which can not easily be represented in an object-oriented design. The concern will inevitably be scattered throughout the model and / or entangled with other features [6][7][8]. Adding such a feature is best supported by aspect-oriented software development.

Using UML for Aspect-Oriented Modeling

The most intuitive join points defined by AspectJ are method calls (not including super calls). The pointcut designator that allows a programmer to pick out a set of method calls based on the static signature of a method has the form:

```
pointcut someName : call(Signature);
```

Using the pointcut construct, it becomes straightforward to write an aspect that intercepts all calls to object instances of a certain class, performs some preprocessing, proceeds with the call, and finally does some postprocessing.

In the example below, the Logging aspect intercepts all method calls made by customers on an account object and logs the access to a file:

```
public aspect Logging {
  private PrintWriter Account.myLog;
  public void Account.setLog(String fileName) {
    myLog = new Log(fileName);
    myLog.println
      ("This is the logfile for account " +
       this);
  }
  pointcut MethodCall(Customer c, Account a) :
    call (public * Account.*(..))
      && this(c) && target(a);
  after (Customer c, Account a) :
          MethodCall(c, a) {
    a.myLog.println(c + " called " +
      thisJoinPoint.getSignature().getName());
    a.myLog.flush();
  }
}
```

The Logging aspect introduces a new method, called setLog(), into the Account class. This new method associates an account object with a log of type PrintWriter that writes the logging information into a text file. The reference to the log is stored in the private attribute named myLog. A UML class diagram that captures this change from the static structure point of view is shown in Figure 3.

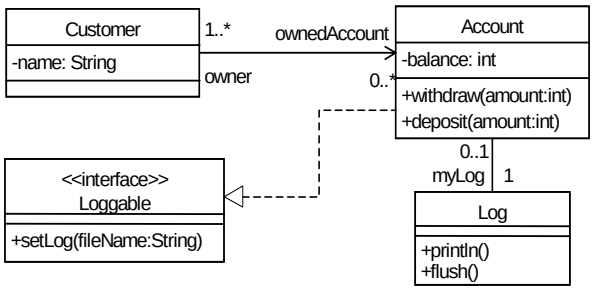

Figure 3: Class Diagram for Account Logging Aspect

To capture the interaction going on during a method call with an aspect, we need to modify the collaboration diagram presented in Figure 1 to explicitly show the method call interception.

A typical way of modeling interception of method calls, when using UML, is to add a new class, whose instance will be interposed between the interacting objects. This technique is illustrated in Figure 4. An interceptor object i is placed between the customer and the account objects. In addition to implementing the logging feature itself, the object i must:

- Offer an interface that supports all methods of the account object that are invoked by the customer object.
- Provide a mechanism to forward intercepted calls to the account object.

Instead of calling the account object directly, the customer object now actually calls the interceptor object (1:withdraw(50)). In our example, the logging action is performed in the after advice. Therefore, the message 1:withdraw(50) is first forwarded to the account object as 1.1:withdraw(50), and upon return the call is logged by sending the message 1.2:println(..) and 1.3:flush() to the log file associated with the account object.

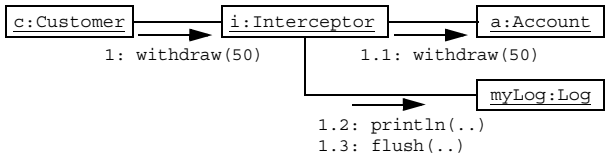

Figure 4: Collaboration Diagram With an Interceptor

Discussion

UML, in its current state, allows us to capture the structure and interactions of our aspect-oriented program. However, the resulting model presents some major drawbacks:

- Crosscutting concerns can not be well modularized: The design of the logging capability is scattered throughout the diagrams. It is partially modeled in both the Loggable interface and the new association between the Account and the Log class, as shown in the class diagram. In the collaboration diagram, it is represented by the Interceptor object, and the messages it exchanges with the other objects. Note also that the two diagrams are inconsistent, since the Interceptor class does not appear in the class diagram.
- According to the diagrams, there is no difference between modularization by class and by aspect. The basic concepts of AspectJ, such as pointcuts, introduction and advice, are not explicitly modeled.
- The model does not capture the fact that the interception of the call is done transparently. In Figure 4, the code of the Customer class must be modified to call the interceptor object instead of calling the account object. As a consequence, any permanent attributes referencing account objects must be changed to reference interceptor objects.

- The model does not show that the logging aspect is a "pluggable" entity, i.e., that the account can be used with or without logging, depending on the application semantics. Both diagrams give the impression that aspects are static entities, although, in reality, aspects are configured at weave-time, and triggering them can be based on various kinds of execution flows or conditions.

These drawbacks actually come with no surprise. During design, the aim is to model a specific solution for a given problem. As a result, the standard UML diagrams model the static and dynamic structure of the Java code *after* the weaving process, since it is only in its woven form that the code implements the logging capability.

Unfortunately, this leads to some form of abstraction inversion, since the nicely separated logging code in the implementation is scattered throughout the design model. The only solution to this problem, in our opinion, is to separate concerns also during design, and to define a design-weaving process.

In summary, our intention in this section was to address two essential issues:

1. explain how a code-driven design approach allows one to understand some key characteristics of aspect-oriented modeling and compare them to object-oriented modeling;
2. argue for the use of advanced separation of concerns (e.g., separating crosscutting concerns from non-crosscutting concerns) as a technique to complement object-oriented modeling with the notion of aspects.

4 Extending UML for AOSD

Code-driven design, as presented in the previous section, limits the ability to understand various kinds of concerns, since it enables expressing the crosscutting nature of software concerns from only one single perspective (a low-level, static and textual view of the system). This makes it difficult to integrate aspects with other software artifacts and to reason about modules of crosscutting concerns from different perspectives or viewpoints. Indeed, developing aspect-oriented software requires thinking of an aspect as an abstraction that defines a certain interaction context, and offers behavior that can vary depending on certain conditions at run-time. Fulfilling such a requirement necessitates the ability to understand and describe the system from multiple viewpoints.

To overcome the shortcomings of current modeling techniques, aspects need to be treated as first-class citizens in advanced modeling languages. We propose to define an aspect as a UML model element that modularizes crosscutting concerns at various levels of abstraction, not only at the code-level.

Consider, for example, the logging aspect introduced previously. To capture the logging aspect in a single module using UML, we need to introduce a representational unit that encapsulates the role of the interceptor object as well as the interactions between the participant objects of the classes `Customer`, `Account` and `Log`.

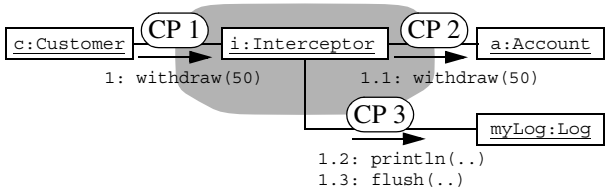

Figure 5: Identifying Connection Points

Let's go back to our collaboration diagram. Figure 5 is similar to figure 4, except that the additional structure introduced by the logging aspect is highlighted in grey. We will now proceed and make this grey part a first-class citizen, and determine what must be part of this aspect, and what not.

Clearly, the participants `c`, `a`, and `myLog` are not part of the aspect, since they perform computation on the outside. It is the role of the interceptor object to mediate the interactions between these objects, linking them together. The interceptor therefore should be part of the aspect. The mediation itself happens at the connection points highlighted in figure 5 by `CP 1`, `CP 2` and `CP 3`.

The aspect is executed when reaching `CP 1`, that is when a call join point is reached. From the interceptor point of view, `CP 1` is an incoming (or passive) connection point. The control flow enters the aspect from the outside. `CP 2` and `CP 3` on the other hand are outgoing (or active) connection points. The control flows from the aspect to the outside. `CP 2` has a special relationship with `CP 1`, since the calls that enter through `CP 1` can exit through `CP 2`. Therefore, they both present the same signatures, with opposite flow directions. They are *conjugated*. To actually perform logging, the aspect requires an instance of the `Log` class to be present, providing the `println` and `flush` methods, which are called at execution time. This is done through the connection point `CP 3`. The signature of `CP 3` is completely independent from `CP 1` and `CP 2`.

By making the notion of connection points explicit, we are now able to define stand-alone aspects and reason about them as a particular kind of UML collaboration. They clearly separate crosscutting interaction concerns from computation as performed by participant objects. The connection points being well-defined parts of the aspect, it is possible to bind them to individual objects. In UML, a binding corresponds to the concept of attaching a classifier role to an association role end. In our case, the association role end corresponds to a connection point. However, in contrast to standard UML, our model focuses on explicit modeling of the association role end as part of the aspect construct, allowing one to clearly separate modeling of interaction concerns from the "dominant" structure of classifier roles.

We propose to define a new stereotype of UML collaboration to support aspect-oriented modeling with

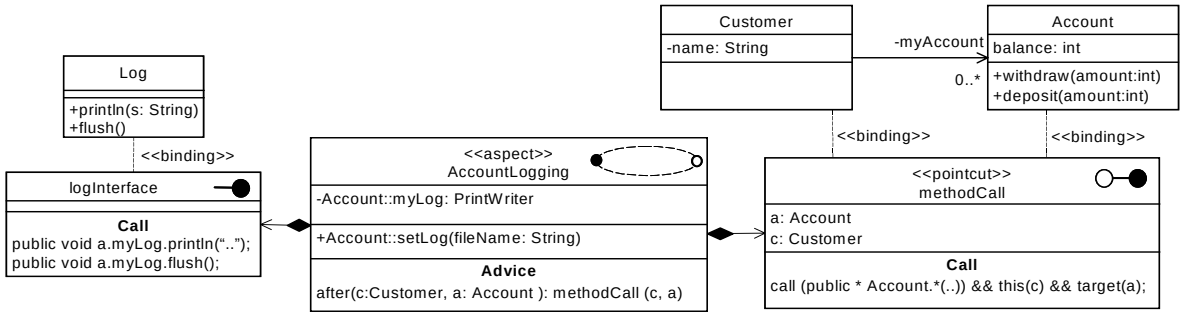

Figure 6: The `Logging` Aspect Model using a UML Collaboration Stereotype

UML. This enables us to instantiate the new *aspect classifier* to model interaction concerns in an explicit and reusable way.

Figure 6 illustrates this idea. It consists of four types of elements: the *aspect* itself, the associated *connection points*, normal UML classes, and the *binding* relationship.

The aspect itself, highlighted in the figure by a dotted oval, specifies the actions to be performed at the connection points and along the interconnections, along with static information that will be woven into the objects of the classes it binds to. For example, in Figure 6, the first two compartments of the aspect represent two elements of the logging feature, an attribute and a method, which need to be woven into the `Account` class at binding time. The *Advice* compartment defines an action to be executed after returning from a method call to an account object instance. This action encapsulates the actual code calling the log.

The connection points are shown in Figure 6 using white and black circles. White circles are incoming connection points. Black circles are outgoing connection points. A white and a black circle connected by a line means that the connection points are conjugated. A connection point is the construct designed to model pointcuts, but it is general enough to cover also other kinds of points of interactions.

Connection points can be composed to build the interface of an aspect in the same way multiple UML interfaces can be combined to form the interface of a class. However, there are two major differences between a connection point and a UML interface: first, connection points can be instantiated, whereas UML interfaces cannot; second, UML interfaces define signatures of operations, while connection points declaratively specify invocations to these operations and the compositions thereof. Moreover, connection points allow us to define attributes. These are typically used as parameters for binding objects to the connection points of the aspect.

Examples of connection points shown in Figure 6 are `logInterface` and `methodCall`.

The `logInterface` connection point (`CP 3` in Figure 5) is outgoing, meaning that the aspect will make use of it during execution. It specifies what the aspect requires from its environment during execution. In an implementation, it maps to a Java interface or abstract class. Its properties, i.e., attributes and methods, are shown in the attribute and *Call* compartment.

The `methodCall` connection point is a special form of connection point that groups together two conjugated connection points (`CP 1` and `CP 2` shown in Figure 5). It directly maps to a call pointcut in `AspectJ`. The pointcut definition can be shown in the *Call* compartment.

At weave-time, objects bind to the connection points offered by aspects. The binding relationship specifies what class of objects an instance of the aspect can be bound to. In our example, the aspect must be bound to instances of the classes `Customer`, `Account` and `Log`. Therefore, our aspect could, at weave-time, interconnect customer `c` with account `a` and the log object named `mylog`.

5 Conclusion

In this paper we have analyzed the suitability of UML for modeling aspect-oriented software. We have taken a bottom-up approach, starting from a small piece of code as found in aspect-oriented programming languages, and then tried to model the design of this code using standard UML.

Since UML does not define the idea of weaving, the nicely separated concerns in the aspect-oriented program ended up scattered throughout the design model.

We then showed that by making aspects first-class citizens, we can nicely separate crosscutting concerns. The aspect models the interconnections between participants and coordinates their interactions. The mediation is performed through well-defined connection points. Connection points are the basis for pluggability, since they specify the aspect interface, and hence determine what objects the aspect can connect at weave-time.

6 Acknowledgements

Mohamed Kandé and Jörg Kienzle have been partially supported by the Swiss National Science Foundation project FN 2000-057187.99/1.

7 References

[1] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher: "Discussing Aspects of AOP". *Communications of the ACM **44**(10)*, pp. 33–38, October 2001.

[2] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr.: "N Degrees of Separation: Multi-Dimensional Separation of Concerns". In *Proceedings of the 1999 International Conference on Software Engineering*, pp. 107 – 119, Los Angeles, CA, USA, 1999, IEEE Computer Society Press.

[3] J. Rumbaugh, I. Jacobson, and G. Booch: *The Unified Modeling Language Reference Manual*. Object Technology Series, Addison Wesley Longman, Reading, MA, USA, 1999.

[4] G. Booch, J. Rumbaugh, and I. Jacobson: *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Massachusetts, USA, 1 ed., 1999.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold: "An Overview of AspectJ". In *15th European Conference on Object–Oriented Programming (ECOOP 2001)*, pp. 327 – 357, June 18–22, 2001, Budapest, Hungary, 2001.

[6] M. Aksit, L. Bergmans, and S. Vural: "An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach". In O. L. Madsen (Ed.), *6th European Conference on Object–Oriented Programming (ECOOP '92)*, pp. 372 – 395, Utrecht, The Netherlands, June 1992, Lecture Notes in Computer Science **615**, Springer Verlag.

[7] S. Clarke, W. Harrison, H. Ossher, and P. Tarr: "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code". In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 325 – 339, Addison–Wesley, 1999.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin: "Aspect-Oriented Programming". In M. Aksit and S. Matsuoka (Eds.), *11th European Conference on Object–Oriented Programming (ECOOP '97)*, pp. 220 – 242, Jyväskylä, Finland, 1997, Lecture Notes in Computer Science **1241**, Springer Verlag.