

Aspect-Oriented Programming—An Object-Oriented Extension

Thomas Østerlie

July 22, 2003

1 Introduction

Aspect-oriented programming is an extension to object-orientation with the goal of separating the concerns of domain logic from application logic. The essay aims at looking at aspect-oriented from a theoretical perspective, trying to place it within the existing framework provided by object-orientation. The first section is an introduction to object-orientation. Here I will present the most common elements of object-orientation, and show the multifaceted terrain that object-orientation is. Concluding this first section I present two different ways of viewing object-orientation, trying to bring some order to the material presented that far. The second section deals with aspect-oriented programming. First I introduce the topic, and then I relate it to object-orientation actual implementations. Concluding I arguing for its relevance as a complimentary technique to be added to the object-orientation toolbox.

2 Object-orientation

Object-oriented programming languages can be traced back to the SIMULA 67 programming language from the late 1960s (Holmevik 1995). SIMULA was created as a language to write simulations. SIMULA 67's key concept was to group data and their operations in distinct units, or objects. Pivotal in the language's evolution was the search for some kind of generalized process concept with record class properties. From this search came the idea of prefixing the process, or object as it was called. The object consisted of two layers: a prefix layer containing references to its predecessor and successor along with a number of other properties, and a main layer containing the attributes of the object in question. In addition to this important new feature, the concept of classes was introduced. This concept made it possible to establish class and subclass hierarchies of concatenated objects. With the two central concepts of encapsulation and inheritance, object-orientation was created.

Today, over 35 years after Simula 67, object-orientation is a lot more than just encapsulation and inheritance. Object-orientation has proven an attractive abstraction and reuse mechanism and is transfered to numerous other areas like software design and modeling, databases and distributed systems. But what is object-orientation really?

2.1 What is object-orientation?

Object-orientation is in essence the grouping of data and operations on these data. The data and operations are encapsulated in objects, representing a more or less natural unit of abstraction. Communication between objects is performed through *message passing*. Instead of having different objects invoking operations directly on each other, they send messages to each other. This emphasizes each object's independence, as objects themselves determine what operations to be invoked as the result of receiving a message. Message passing can therefore be understood as an encapsulation mechanism (Nierstrasz 1989).

Object-oriented systems and programming languages realize encapsulation differently. Common to them is that they formalize encapsulation and encourage programming in terms of objects rather than programs and data. How this is done, is embedded in the language's object model. Different object models have different ways of enabling encapsulation, depending on which properties of objects they need to encapsulate. There is no such thing as a formal, unified model for object-oriented programming languages. Object-orientation has instead grown as a collection of different techniques and technologies, leading to a number of object models. Still, a number of core concepts are shared by most object models, even though their actual realization may vary. Among the most important concepts are encapsulation, object classes, instantiation, inheritance, and polymorphy.

2.1.1 Encapsulation

Encapsulation as a concept is discussed earlier in this section. Object models implement encapsulation differently. The difference lies in how much of an object's implementation is hidden behind the interface. Python (<http://www.python.org>), for instance, has a lenient object model when it comes to encapsulating data. Every member variable in a Python object is publicly visible to every other object in the software system. Ruby's (<http://www.ruby-lang.org>) object model, on the other hand, does not expose any member variables outside of objects.

The encapsulation of operations is often implicit in object-oriented languages. Sending a message to an object is usually the same as calling the operation directly. Instead of explicit message passing to provide encapsulation of operations, object models provide access restrictions to operations. Operations on objects in Java, for instance, can be public, protected, private, or have package scope. Trying to call an operation without the correct access rights will lead to a compiler error. Aglets (<http://aglets.sf.net>), a Java based object-oriented Agent system, on the other hand, has an explicit message passing system. Explicit message passing is a mechanism for inter-object communication between agent object as agents rarely know about other objects' interfaces at compile time. The programmer has to explicitly define what operations are to be called when an agent object receives any given message in addition to providing some sort of error handling if illegal messages are received.

2.1.2 Object class and instantiation

The object class, or simply class, defines a set of data and the operations on these data. The class is a “cookie cutter” for objects, declaring the class’ data in form of member variables, operations in form of as functions and/or procedures, parent pointers, etc. needed to define a new object (Stein *et al.* 1989).

Closely connected to the idea of classes, is instantiation. Objects are cut from the template embedded in the class. This is instantiation. An instance of a class and objects are used synonymously. Data defined in the class may be shared between instances of that class (as with the static operator in Java), or each object may have its own instance of each variable. Some object models define strict templates, where data and operations cannot be gained or lost during execution, like Ada95 for instance, which is an object-oriented programming language with strict templates. Other object models allow an object to gain or lose data and operations. Java, on the other hand, allows the dynamic creation of both new classes and redefinition of existing classes through its serialization and class loader system.

The generic is a another variant of the “cookie cutter” idea, complimentary to classes. The class is a mechanism that affords reuse of data and operations on these data. The problem with classes is that the reuse of operations is tightly coupled with the data. The generic is a compile time parameterization making operations independent of the data. The generic defines the operation’s algorithm independent of its data as long as the data object defines and implements those operations called by the generic. The ordered linked list can be implemented as a simple generic. The ordered list has no idea of how to sort its data objects, but it will use the object’s < and > operators instead. As long as an object defines these two operators, a sorted linked list of these objects may be instantiated. C++ standard template library is an implementation of generics. Ada95 provides an even more advanced form of generics. Ada95’s generics allows for the parameterized instantiation of both objects and operations. The hash table for instance, is implemented as a generic where the user has to both supply the data object for the hash and the hashing operation itself (?).

Generics are often called static polymorphy as opposed to the dynamic polymorphy provided. Static polymorphy happens at compile time, while dynamic polymorphy takes place at runtime. The idea of polymorphy is that an operation may have several implementations. The use of polymorphy is very often connected to typing and inheritance, but it need not be so. Before progressing further on the subject of polymorphy, I need to explain the basic concepts behind inheritance and sub-typing.

2.1.3 Inheritance, sub-typing and polymorphy

(Nierstrasz 1989) summarizes the difference between object models’ implementation of inheritance with the following questions:

- Does inheritance occur statically or dynamically?
- What are the clients of the inherited properties?
- What properties can be inherited?

- Which inherited properties are visible to to client?
- Can inherited properties be overridden or suppressed?
- How are conflicts resolved?

“Client” in this context is the inheriting object.

The most common form of inheritance is the sharing and reuse of operations and data between hierarchically organized classes, also called class inheritance. The key idea of class inheritance is to provide a simple and powerful mechanism for defining new classes that inherits data and operations from existing classes. Inheritance is in that sense a reuse mechanism.

Inheritance and sub-typing are two concepts that many object models blend into each other. A type is superficially the same as an object class. The idea of typing is to eliminate an object’s need to protect itself from unexpected messages, as this is done at compile time. A language is strongly typed if type compatibility of all expressions representing values can be determined from the static program representation at compile or run-time (Wegner 1987). The language could be said to be type safe. Weakly typed languages are not completely type checked at compile time, and unchecked type violations may happen at run time.

A programming language may be typed without being object oriented, and object-oriented languages may be untyped like Smalltalk. However, many object models mix the idea of sub-typing and inheritance. Ada95 does this (?). A class is a user defined type in Ada95, and inheriting classes are considered subtypes of their ancestors. The traditional approach to type-checking with user-defined types is to insist that the types of expressions supplied to operations at runtime must correspond exactly to the expected type, is not applicable in object-oriented languages with dynamic binding. If a typed language is to support dynamic binding types serve as specifications for valid bindings and invocations. One type confirms to a second if some subset of its interface is identical to that of the second (Nierstrasz 1989). This way a subclass can pass as its superclass.

Yet, polymorphy itself is not dependent on sub-typing. Ruby, for instance, is a dynamically typed language. If strong and weak typing is about the whether all expressions are checked for typing, dynamic typing, along with its opposite, static typing, says something about when type checking is performed. A language is dynamically typed if type errors cannot be detected until run-time, and it is statically typed if the compiler can detect all type errors without actually running the program (?). That Ruby is dynamically typed affords a pure form of polymorphy. As everything is an object in Ruby, no checks are made to ensure the object’s class when objects are sent as parameters with method calls. Instead of checking if an object implements a given method at compile time, messages are sent to the object and the runtime system has to handle the situation where the message is unhandled by the callee.

2.2 What makes an programming language object-oriented?

In the preceding section I have shown that even object models implementing the same object-oriented techniques, differ in the way they implement these techniques. Instead of presenting a unified view of object-orientation, I have tried to present the multiplicity and heterogeneity that makes up the terrain of object-orientation. The question then is what makes an object-oriented programming language? Several attempts have been made to provide rules to say something about what makes an object-oriented language (Stein *et al.* 1989) (Nierstrasz 1989) (Wegner 1987). Common ground for these attempts is a conclusion that a complete definition of what it means to be object-oriented is not possible. Instead of trying to come up with an exhaustive classification scheme, I will present two views on the common foundations of object-orientation.

Stein *et al.* (1989) argues that object-orientation differs from other programming approaches in that it allows for extension rather than modification. At the heart of object-orientation lies the two fundamental mechanisms of *templates* and *empathy*. Templates create new objects in their own image, providing guarantees about the similarity of group members. Empathy, on the other hand, allows an object to act as if it were some other object, thus providing sharing of state and behavior. They are fundamental in that they cannot be defined in terms of one another, and that most object-oriented languages can be described largely in terms of the ways in which they combine these mechanisms. What separates different object-oriented programming languages is how they use templates and empathy to allow for extension and sharing.

Empathy underlies inheritance, in that an object may “borrow” its superclass’ attribute—data or operation—if it does not provide that attribute itself.

We say object A empathizes with object B for message M if A doesn’t have its own protocol for responding to M, but instead responds to M as though it were borrowing B’s response protocol. A borrows just the response protocol, but not the rest of B. That is, any time B’s response protocol requires a message to be sent to SELF (or a variable to be looked up), it is sent to A, not B otherwise, A and B respond in the same way.

Not only does this apply to the concept of inheritance, but also sub-typing and polymorphy. The second fundamental mechanism, templates, can provide distinctions between types of object models. The object class is a type of template, and so is generics. By determining the parameters of *when* and *how* an object model empathizes, and *for*, *what* and *how* (Stein *et al.* 1989) classifies different object models as shown in table 1.

In many cases object orientation is seen as an essential enabler for reuse (Morisio *et al.* 1999). Nierstrasz (1989) argues that reuse follows from the core object-oriented principle: *encapsulation*. Classes, instantiation, inheritance, polymorphy and generics are technologies for packaging objects in such a way that they can conveniently be reused without modification. Classes and instantiation affords reuse of similar data structures and their operations. Inheritance is strictly a re-usability mechanism for sharing behavior between objects. Polymorphism enhances software reuse by making it possible to implement soft-

Language	Determination of empathy			Template mechanisms	
	When	How	For	What	How
Actors	runtime	explicit	per object		none
Delegation	runtime	both	per object		none
Self	runtime	implicit	per object	templates	nonstrict
Simula	compile time	implicit	per group	classes	strict
Smalltalk	object creation time	implicit	per group	classes	strict

Table 1: Various languages and their attributes

ware that can operate on not only existing software, but also with objects that are added later. Generics affords the reuse of algorithms.

In this sense any programming language providing mechanisms to exploit encapsulation can be said to be more or less object-oriented—*more of less* being a central term here. Instead of providing a set of hard qualifiers, Nierstrasz (1989) argues that it is more fruitful to argue to what degree a programming language can be said to be object-oriented. The measure of how object-oriented a programming language is, can be based on the homogeneity of its object model. In Ruby, for instance, everything is an object, even class objects are objects. Conversely Java have both objects and primitive types like int, long, byte, etc. In that sense Ruby can be understood as more object-oriented than Java.

3 Aspect-oriented programming

The current object-oriented paradigm builds software systems by decomposing the referent system into objects. The problem is that many requirements do not decompose neatly into behavior centered on a single locus (Elrad *et al.* 2001). These crosscutting concern needs to be encapsulated in separate abstractions. Aspect-oriented programming is based on the idea that computer systems are better programmed by separately specifying the various concerns (properties or areas of interest) of a system and some description of their relationships, and then relying on mechanisms in the underlying environment to weave or compose them together into a coherent program.

The rest of this section is organized as follows. The first subsection describes the problem of crosscutting concerns. The second explains how the problem of crosscutting concerns is addressed by aspect-oriented programming. The third subsection shows an actual implementation of the aspect-oriented programming ideas.

3.1 The problem of crosscutting concerns

A crosscutting concern can be understood as method calls appearing in several places throughout an application. The essence of aspect-oriented programming is that in any object-oriented program, even if it's well designed, there will be some method calls that appear in several places throughout the program. If you change the call to that method, or change the error handling of that method, you may need to make changes in all of the

occurrences of this call. If the change is subtle, the compiler may not catch it, which could lead to annoying run-time errors that might be difficult to track down.

One good way to understand crosscutting concerns is with an illustration. Consider the UML for a simple figure editor, as depicted in figure 1, in which there are two concrete classes of `FigureElement`, `Point` and `Line`. These classes manifest good modularity, in that the source code in each class is closely related and each class has a clear and well-defined interface. But consider the concern that the `Display` should be notified whenever a figure element moves. This requires every method that moves a figure element to do the notification. The crosscutting box in the figure is drawn around every method that must implement this concern, just as the `Point` and `Line` boxes are drawn around every method that implements those concerns. Notice that the box for `DisplayUpdating` fits neither inside of nor around the other boxes in the figure—instead it cuts across the other boxes. This is called a crosscutting concern.

3.2 The response

The goal of aspect-oriented programming is to make designs and code more modular, meaning the cross cutting concerns are localized rather than scattered throughout the software system, and have well-defined interfaces with the rest of the system. This provides the usual benefits of modularity, including making it possible to reason about different concerns in relative isolation, making them (un)pluggable, amenable to separate development, and so forth.

Using just object-oriented programming, the implementation of crosscutting concerns tends to be scattered out across the system, just as it would be here. Using the mechanisms of aspect-oriented programming, the implementation of `DisplayUpdating` can be modularize into a single aspect. Because crosscutting behavior can be implemented in a single modular unit, it makes it easier for to think about it as a single design unit. In this way, having the programming language mechanisms of aspects lets the software developer think in terms of aspects at the design level as well.

It is important to understand that crosscutting is relative to a particular decomposition. Crosscutting concerns of a design cannot be neatly separated from each other. A basic design rule is to represent significant concerns as first-class abstractions in the language. This allows them to be composed and extended. In the figure editor example, there are two important design concerns: representing the graphical elements and tracking the movement of graphical elements. In the figure, classes are used to model the first concern. This allows them to be extended using aggregation and inheritance. Also, every graphical class encapsulates its internal data structure. The second concern requires tracking movements to also be represented as a separate class. However, the first choice makes this difficult because the movement functionality is part of the behavior of graphical classes. We could have designed the system around the tracking functionality; in that case, the graphical functionality would crosscut the tracking classes.

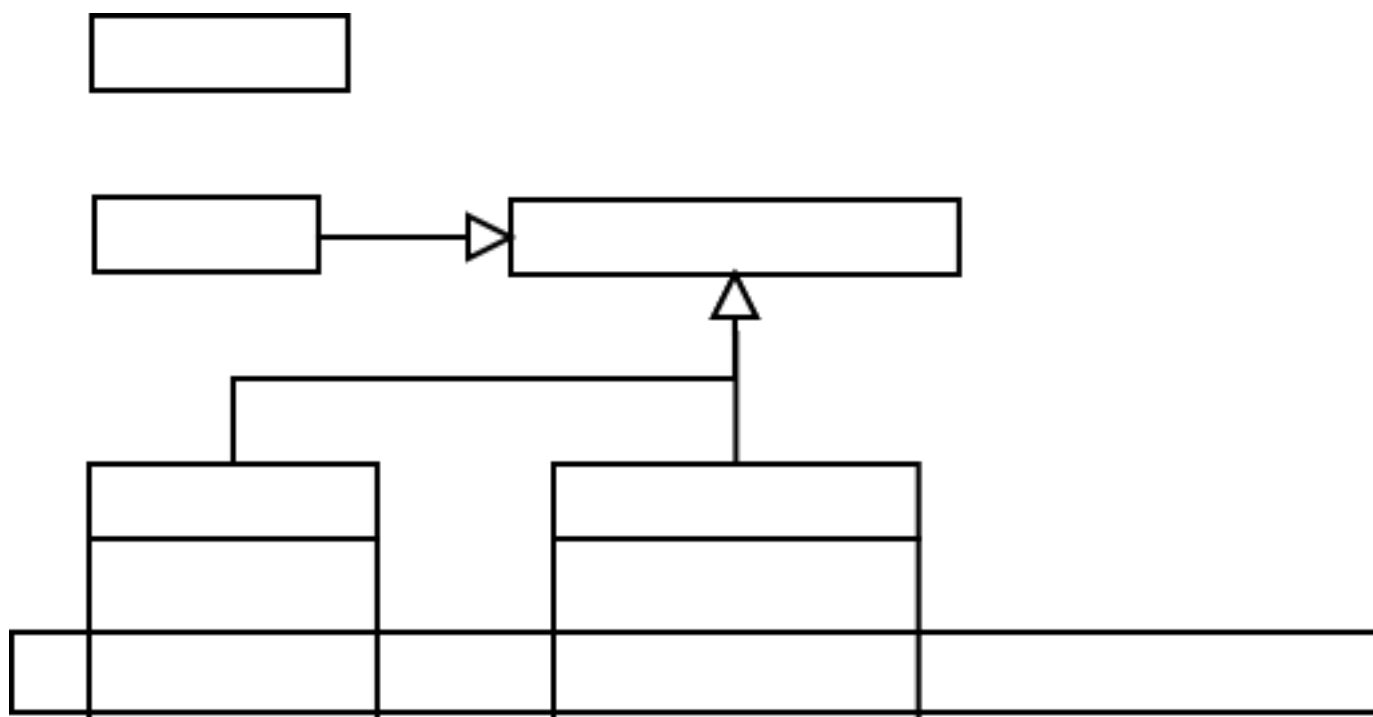


Figure 1: Aspect crosscut classes in a simple figure editor

3.3 AspectJ—A practical example

AOP languages use five main elements to modularize crosscutting concerns: a join point model describing the “hooks” where enhancements may be added; a means of identifying join points; a means of specifying behavior at join points; encapsulated units combining join point specifications and behavior enhancements; and a method of attachment of units to a program. Think of join points as nodes or edges in some graph. The graph can be a dynamic call graph (a UML interaction diagram), a class graph (a UML class diagram), an object graph (a UML object diagram). An appealing way to define join points is to use succinct specifications: Instead of enumerating the join points, short descriptions can be filled in with information from elsewhere to create the list of join points. Using traversal strategies leads to loose coupling between the structural and the numerous behavioral concerns.

AspectJ, a preprocessor implementing an aspect-oriented extension to Java, provides a join point model of dynamic join points, in which join points are certain well-defined points in the execution flow of the program. An aspect is a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, constructors, and initializers. In addition aspects introduces the named pointcut and advice which both are part of the aspect model. Continuing with the figure editor example above, an aspect can look like this:

```
aspect DisplayUpdating {
    pointcut move() :
        call(void Point.setX(int)) ||
```

```

    call(void Point.setY(int)) ||
    call(void Line.setP1(int)) ||
    call(void Line.setP2(int));
after(): move() {
    updateDisplay();
}
void updateDisplay() {
    <the appropriate code to update the display>
}
}

```

A named pointcut defines a set of join points in the program flow. `move` is the only named pointcut in the `DisplayUpdating` aspect. The named pointcut relies on the notion of a join point. The join point model in an aspect-oriented language provides the common frame of reference that makes it possible to define the structure of crosscutting concerns. In AspectJ, a join point is defined by a method signature. `void Point.setX(int)` refers to the `setX` method in class `Point` being called with an `int` parameter. Whenever this method is called, the pointcut is activated.

The pointcut is the glue that ties the aspect model with the object model. As explained in the figure editor example above, after the position of a line or a point is updated the display needs to be updated to reflect the object's new position. AspectJ defines a join point either before or after a method has been called. The advice takes care of the timing. In the `DisplayUpdating` aspect, the advice declares that `updateDisplay` is to be called after the `move` pointcut is activated. In addition to declaring the timing, the advice also declares which of the aspect's methods to be called when the pointcut is activated.

The figure editor example above shows how the `DisplayUpdating` aspect modularize the display updating functionality, and has a clean interface with the rest of the system. The functionality is encapsulated and localized to one aspect instead of being spread across the two classes. This affords reuse by providing functionality for more orthogonal code. Orthogonality means that unrelated elements are expressed independently. A somewhat simple explanation of the concept is to say an operation is orthogonal if it has no side-effects. Continuing with the figure editor example, if calling `Point.setX(int)` in addition to setting the point object's X-axis also redrew the whole object on the display, the redrawing breaks the method's orthogonality. As observed by Poulin (1995), separating application specific and domain specific code affords better reuse. Separating concerns through aspects seems a reasonable way to achieve this goal.

4 Conclusion

Traditional object-orientation has not been able to meet the challenges of crosscutting functionality. Crosscutting functionality has been scattered across classes and objects throughout the software system instead of being encapsulated and found at one place. If a crosscutting concern cannot be treated as a single module, its adaptability and reusability are likely

to be reduced. In this sense traditional object-orientated languages have lead to less encapsulation in a certain kind of problem domain. Aspect-oriented programming addresses this, by providing a mechanism for encapsulating crosscutting concerns and connecting them to existing object systems.

Instead of viewing aspect-oriented programming as a new programming paradigm, it is better understood as a complimentary technique to those already found under the umbrella of object-orientation. Instead of replacing object-orientation, aspect-oriented further increases object-orientation's ability to abstract and encapsulate data and operations belonging together.

References

- ELRAD, Tzilla, Robert E. FILMAN, and Atef BADER, 2001: *Aspect-oriented programming: Introduction*. Communications of the ACM, **44**(10), pp. 29–32. ISSN 0001-0782. 6
- HOLMEVIK, Jan Rune, 1995: *The history of simula*. Accessed March 26 2003.
URL <http://java.sun.com/people/jag/SimulaHistory.html> 1
- MORISIO, Maurizio, Michel EZRAN, and Colin TULLY (eds.), 1999: *Introducing reuse in companies: A survey of european experiences*. 5
- NIERSTRASZ, Oscar, 1989: *Object-oriented concepts, databases and applications*, chap. Survey of object-oriented concepts, pp. 3–21. Addison-Wesley and ACM Press Frontier series. Addison-Wesley. 2, 3, 4, 5, 6
- POULIN, Jeffrey S., 1995: *Populating software repositories: Incentives and domain-specific software*. Journal of Systems and Software, (30), pp. 187–199. 9
- STEIN, Lynn A., Henry LIEBERMAN, and Henry UNGAR, 1989: *Object-oriented concepts, databases and applications*, chap. A shared view of sharing: The treaty of Orlando, pp. 31–48. Addison-Wesley and ACM Press Frontier Series. Addison-Wesley. 3, 5
- WEGNER, Peter (ed.), 1987: *Dimensions of object-based language design*. 4, 5