Software Engineering: Design

Aspect-Oriented Programming and Modularity

Christian M. Meyer

Software Technology Group
Darmstadt University of Technology

January 29, 2006

---

1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a programming paradigm introduced in the 1990s by Gregor Kiczales and his team at the Xerox Palo Alto Research Center [PARC]. Kiczales developed the first and still most popular aspect-oriented programming language *AspectJ* [AspJ].

The paradigm allows to define some well-defined points within the control flow of the application. These **join points** can be queried using an sql-like syntax and combined using Java's default logical operators. Such a combination of join points is called a **pointcut**. Besides the pointcuts, **advices** can be specified, to define how a given pointcut is processed. In *AspectJ*, the advices `before`, `after` and `around` may be used. An **aspect** holds several advices and their pointcut(s).

The advice can be seen as a method, which is not called manually, but every time, the current control flow reaches a point, that matches the advice's pointcut. In this case the code within the advice is invoked. An example for Aspect-oriented programming is given below, using the Java language extension *AspectJ*:

```
1:  public aspect SomeAspect {
2:
3:    public pointcut allMethods(): execution(* *(..));
4:    public pointcut intGetters(): call(int mypackage.*());
5:
6:    before(): allMethods() {
7:      System.out.println(thisJoinPoint);
8:    }
9:
10:   after() returning(int result): intGetters() {
11:     System.out.println("int: " + result);
12:   }
13:
14: }
```

The advantage of aspect-oriented programming is the combination of concerns that are spatiotemporal divided, so called **cross-cutting concerns**. These cross-cutting concerns stands orthogonally upon other concerns and cannot be easily extracted into an own module.

This article deals with a modularity claim which is pesented in section 2. For details on the *AspectJ* syntax, [Lad03] and [Bhm05] are recommended, both books give also a good overview of the aspect-oriented paradigm.

2 Claim

By now, the following claim will be discussed:

> *AOP brings improved support for the following properties in particular:*
>
> - *the modular continuity criterion,*
> - *the direct mapping rule, and*
> - *the linguistic modular units principle.*

Bertrand Meyer described in his book *Object-Oriented Software Construction* [Myr97] five criteria, five rules and five principles to encourage a good modular design. Although Meyer concentrated on object-oriented programming, three of his suggestions are applied to aspect-oriented programming in the claim above. In the following each statement of the claim will be examined separately.

3 Modular continuity

The modular continuity criterion says:

> *A method satisfies Modular Continuity if, in the software architectures that it yields,a small change in a problem specification will trigger a change of just one module, or a small number of modules.* [Myr97]

It is sufficient to determine the advantages of using aspect-oriented programming, because the AOP-paradigm can be applied to any other languages, so if the use of an aspect would be disadvantageous, the structures of the underlying paradigm can be used.

A good object-oriented decomposition benefit modular continuity, but there are still some situations, where a small change of the specification leads to a big change in the desgin. A management system for a book- and graphic-store will serve as an example: An abstract class `Product` generalizes the two product types `Book` and `Graphic`. Furthermore, there is a `Customer` class, which will be used in a billing system later on. Consider that each class has a name/title field as well as some internal remarks.

Now imagine the shop owner changes the specification, such that there are different users of the application. Each user has one of the roles *Manager* or *Assistant*. A `User` class will suffice the new specification and the modular continuity rule. But what about some access restrictions for each role? Only the *Manager* should now be able to read the internal remarks of products and customers. Object-oriented design needs to change the `Product` and `Customer` class (and many many others in a bigger system). The modular continuity rule is no longer fulfilled.

Aspect-oriented programming allows to add only one aspect to handle the specification change with a minimum of code:

```
1:  public aspect AccessRestrictions {
2:
3:    public pointcut managerOnly(): execution(* Product.getInternalRemarks())
4:        || execution(* Customer.getInternalRemarks());
5:
6:    before(): managerOnly() {
7:      if (!currentUser.isManager())
8:        throw new SecurityException("Access is restricted to managers only");
9:    }
10:
11: }
```

As a conclusion could be stated, that aspect-oriented programming can improve the modular continuity rule. Whenever the specification change is a cross-cutting concern, an object-oriented realisation forces the change of several classes, while a single aspect could implement the new concern.

But AOP can also violate the continuity rule in some special cases. Imagine some methods in a class are used in pointcut definition of several aspects. Now the class has to be extended to realize a specification change. A new method is added, that contains all necessary changes. If the name of the method is very similar to the existing methods, the probability is high, that one or more pointcuts will match to the invocation of the new method. If this was not intended, several aspects have to be changed too, so the continuity rule is no longer fulfilled.

4 Direct mapping

The direct mapping rule is defined as:

> *The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modeling the problem domain.* [Myr97]

In the object-oriented decomposition, the business objects are usually direct mapped into classes, e.g. the product classes in the example above. However, the business workflows are more difficult. Most of the activities are implemented in the business object classes, for example saving the product info to the database. Composite workflows, like loading a product, changing it's price and saving the new information, use those instance methods and form a new method or a new class, depending on its complexity. So the direct mapping rule is not every time completely fulfilled.

In a system, that is designed as described above, the only possibility, that remains for cross-cutting concerns, is dividing it's functionality and implement them within the defined business objects and workflows. So cross-cutting concerns are seldom direct mapped using this design method.

Figures 1 and 2 show the module structure of Apache Tomcat [Tmct]. Red bars indicate the position of one concern in a module. In figure 1, the XML parsing concern is highlighted. The functionality, that is needed to realize this concern, is concentrated in one module, so it is direct mapped and suffices Meyer's rule. Figure 2 displays the logging concern, which is separated into quite a lot of modules. It hence is not direct mapped and violates Meyer's rule.

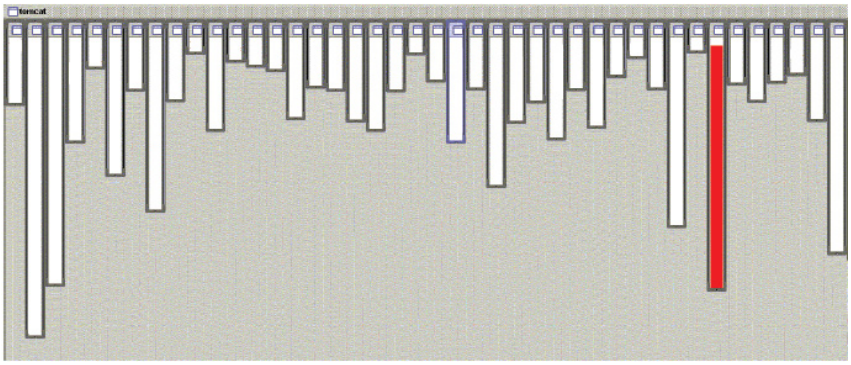

Figure 1: The XML parsing concern in Apache Tomcat

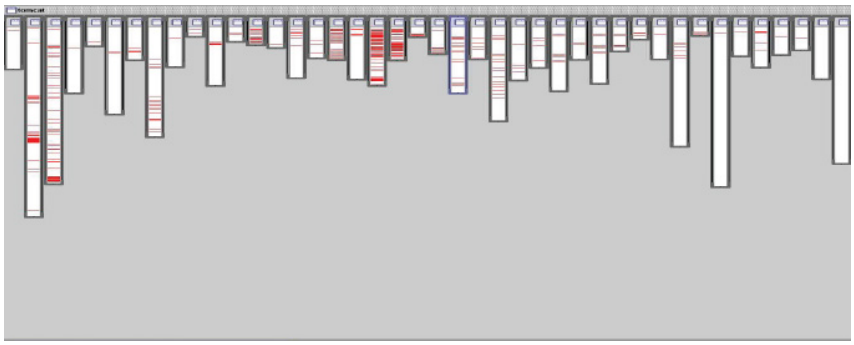

Figure 2: The logging concern in Apache Tomcat

Aspect-oriented programming allows to design cross-cutting concerns as a single module (an aspect) to fulfill the direct mapping rule. It is even possible to design some object-oriented patterns, containing cross-cutting concerns. The following code illustrates how the visitor pattern could be direct mapped to a single aspect:

```
1:  public aspect PriceObserver {
2:
3:    private ArrayList charts;
4:
5:    public pointcut attachChart(): call(PriceChart.new(..));
6:    public pointcut detachChart(): call(void PriceChart.hide());
7:    public pointcut notifyChart(): call(* Product.setPrice(..));
8:
9:    public PriceObserver() {
10:     charts = new ArrayList();
11:   }
12:
13:   after() returning(PriceChart chart): attachChart() {
14:     System.out.println("Attaching price chart.");
15:     charts.add(chart);
16:   }
17:
18:   before(PriceChart chart): detachChart() && target(chart) {
19:     System.out.println("Detaching price chart.");
20:     charts.remove(chart);
21:   }
22:
23:   after(): notifyChart() {
24:     Iterator iter = charts.iterator();
25:     while (iter.hasNext())
26:       ((PriceChart) iter.next()).notify();
27:   }
28:
29: }
```

Every creation of `PriceChart` enables the observation of the product list. Invoking `setPrice(..)` within the `Product` class leads to a notification of the price chart. In the given aspect all necessary parts of the observer patterns are combined, so there is no need of inform the product classes, that they are observed, even the attaching can be put aside.

5 Linguistic modular units

Meyer's linguistic modular units criterion says:

> *Modules must correspond to syntactic units in the language used.* [Myr97]

Cross-cutting concerns have no corresponding syntactic unit in object-oriented programming languages, so they are separated and distributed among other classes.

Aspects yield a new syntactic structure for these cross-cutting concerns and improve so the linguistic modular units criterion. Their usually class-like syntax, fits perfectly with the object-oriented paradigm.

Modules can be defined by using a namespace or package hierarchy. This hierarchy can also be applied to aspects, so a module then contains classes *and* aspects.

References

[AspJ] *AspectJ,*
http://www.eclipse.org/aspectj

[Bhm05] Böhm, Oliver: *Aspektorientierte Programmierung mit AspectJ*,
Heidelberg, dpunkt Verlag, 2005, ISBN: 3-89864-330-1

[GoF95] Gamma et. al.: *Design patterns : elements of reusable object-oriented software*,
Reading/Mass., Addison-Wesley, 1995, ISBN: 0-201-63361-2

[Lad03] Laddad, Ramnivas: *AspectJ in Action – practical aspect-oriented programming*,
Greenwich/CT, Manning Publications, 2003, ISBN: 1-9301-1093-6

[MBH05] Mezini, Mira/Bockisch, Christoph/Haupt, Michael: *Software Engineering Design*,
Lecture in the winter term 2005/2006, Darmstadt University of Technology,
http://www.st.informatik.tu-darmstadt.de

[Myr97] Meyer, Bertrand: *Object-Oriented Software Construction*,
2nd. Edition, Upper Saddle River/NJ, Prentice Hall, 1997, ISBN: 0-1362-9155-4

[PARC] *Xerox Palo Alto Research Center,*
http://www.parc.com

[Tmct] The Apache Software Foundation: *Apache Tomcat,*
http://tomcat.apache.org