Aspect-Oriented Design Principles: Lessons from Object-Oriented Design

Dean Wampler
Object Mentor, Inc.
objectmentor.com
dean@objectmentor.com

ABSTRACT

For aspect-oriented design (AOD) to become mainstream, appropriate design principles are needed to guide its use in real, evolving systems. The principles should tell us what types of coupling are appropriate between aspects and the software entities they advise, how to use non-invasiveness effectively, how to preserve correct behavior in the advised entities, and how to use aspects with other design constructs. I examine these topics using several object-oriented design (OOD) principles, considered from an AOD perspective. I demonstrate how AOD contributes design solutions to satisfy these principles, while it also introduces nuances in their interpretations. I also derive several AOD-specific principles from the OOD principles.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming, Aspect-Oriented Programming.

General Terms

Design, Theory.

Keywords

Aspect-oriented programming, object-oriented programming, software design principles.

1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) is an effective technique for modularizing crosscutting concerns [1,3], but effective design principles are needed to create aspect systems that support long-term maintenance and evolution [2].

To date, aspects have mostly been used to modularize "nonfunctional" concerns like persistence, security, logging, caching, *etc.*, in contrast to the domain logic, specified by the functional requirements.

Aspects for a nonfunctional concern usually require no modifications of the target modules[1], because the concern's problem domain is usually orthogonal to the modules' domains. Hence, the advised modules are *oblivious* [4].

However, partitioning the domain logic itself into aspects is more likely to introduce logic conflicts, since they are no longer orthogonal. *Obliviousness* by itself does not address this design issue. Some aspect systems, *e.g.,* Hyper/J [5] and Composition Filters [6], handle this problem by composing applications from aspects, using merging heuristics to resolve potential conflicts. However, general design principles are needed to address this problem for all aspect systems.

Also, for reasons of program correctness, security, performance, *etc.*, a module may need to control access to its join points and prohibit some types of advice and introductions.

Real, successful systems evolve over time. Tourwé, *et al.* [2] showed that aspects written with early AOSD approaches tended to be tightly coupled to the rest of the application logic, leading to an *AOSD-Evolution Paradox.* While the initial version of an aspect-based application has better modularity than a comparable object-based implementation, tight coupling of the aspects to the rest of the application makes evolution harder. This coupling occurs when pointcuts refer to concrete program structure, like class and package names, that tend to be volatile. While the advised modules are oblivious to the aspects, the aspects are not at all oblivious to the modules they advise. This paradox has been a practical barrier to AOSD adoption, but it can be resolved by adherence to several of the dependency principles discussed in this paper.

The term *noninvasiveness* (See, *e.g.,* [7]) is now used to retain the notion of advice insertion without direct module modification, but with the recognition that techniques of control are sometimes required and naïve obliviousness is not adequate for all design problems.

Mezini and Kiczales [8] analyzed *aspect-aware interfaces*, a module's true interface to the system, which can only be known after accounting for all the aspects present in the system that might affect the module. Hence, reasoning about a module requires understanding the system context. However, AOP makes this explicit and provides tools for modular reasoning.

Sullivan, *et al.* [9-11] and Lopes and Bajracharya [12] examined the value of aspects that are constrained by "Design Rules" *vs.* the openness of obliviousness. They showed how such constraints actually improve the quality of the software, using a net-options value (NOV) model.

This paper addresses design problems, like those caused by a naïve application of obliviousness, in terms of well known Object-Oriented Design principles cataloged by Martin, *et al.* [13], adapted for aspects. The analysis complements the work of other authors who have examined a few of these principles [17], as well as design patterns [14] from an AO perspective [15-17].

2. Principles of Object-Oriented Design

Martin, *et al.* [13] cataloged eleven principles of OO module design and packaging that promote reduced coupling (dependencies) and improved cohesion, leading to software that is more adaptable to changing requirements. The first five deal specifically with class and interface design as they affect

[1] I sometimes use the term "module" generically for classes, aspects, *etc.*

evolution, reuse, and stability. Three more cover package cohesion and three cover package coupling. They are summarized in Table 1.

In this section, I describe the principles and their AOD implications. The names, acronyms and definitions are adapted from [13]. Since aspects are class-like modules in many ways, all the principles also apply to them. Aspects also provide new techniques for supporting the principles and aspects introduce nuances into their interpretations.

The discussion is based on an example "shapes" hierarchy. Inessential details are elided for brevity, such as file names, `public` keywords, constructors and comparison methods[2].

```
package shapes;  // for all the classes…
interface Shape {
    double getArea();
    void draw();
}

class Point {
    double getX() {…}
    double getY() {…}
}

abstract class Polygon implements Shape {
    Point  getVertex(index i) {…}
    void   draw() {…}
    String toString() {…}
}

class Triangle extends Polygon  {
    double getArea() {…}
}

abstract class NinetyDegreeParallelogram
extends Polygon  {
    double getArea() {…}
}

class Square extends NinetyDegreeParallelogram {…}

class Rectangle
extends NinetyDegreeParallelogram {…}

abstract class ClosedCurve implements Shape {…}

class Circle extends ClosedCurve {
    double getRadius() {…}
    Point  getCenter() {…}
    double getArea() {…}
    void   draw() {…}
    String toString() {…}
}

class Ellipse extends ClosedCurve {
    double getApogeeRadius() {…}
    double getPerigeeRadius() {…}
    Point  getFocus1() {…}
    Point  getFocus2() {…}
    Point  getCenter() {…}
    double getArea() {…}
    void   draw() {…}
    String toString() {…}
}
```

[2] A complete version of the example is available at http://www.aspectprogramming.com/papers/aosd2007/

Listing 1

Each `Shape` can return its area, draw itself, and return a string representation. The objects are read-only at this point. Each shape is either a `Polygon` or a `ClosedCurve`. Concrete `Polygons` include `Squares`, `Rectangles`, and `Triangles`, while `Circles` and `Ellipses` are concrete `ClosedCurves`. Some methods are implemented in abstract helper classes, while others are implemented in the concrete classes, as indicated. Note that `Squares` and `Circles` are not subclasses of `Rectangles` and `Ellipses`, respectively. This is discussed later in the Liskov Substitution Principle section.

For a simple example like this, a pure object-oriented design would be adequate in most practical cases. However, the AOP techniques discussed will be most valuable in larger design problems with long-term maintenance, reuse, and enhancement needs. Also, while the example is for Java and AspectJ, the principles should be valid for most AOP systems.

2.1 The OOD Design Principles

2.1.1 The Single Responsibility Principle (SRP)

A class or aspect should have only one reason to change.

A class that mixes multiple concerns, each of which is an axis of potential change, effectively couples the concerns. If the class needs to evolve along one concern axis, the changes often compromise the class's ability to support the other concerns, even when they remain fixed. Changing one concern also imposes *accidental* changes on clients of the class which don't depend on that concern. Hence, it is difficult to modify the class, making it *rigid* and reuse is compromised in applications where a dependent is forced to accept changes in features that it doesn't need. Note that the definition emphasizes *change*; a tangled module that never needs to change poses no practical problems.

The SRP is the OOD solution to the classic "separation-of-concerns" problem. The SRP splits orthogonal state and behavior into separate classes, but it usually isn't sufficient when a crosscutting concern interacts with other concerns in fine-grained ways.

The shapes example exhibits a common SRP problem, while drawing shapes and converting to string formats is useful, it is incidental to the "true nature" of shapes. Hence, it is cross-cutting, especially since the details of these operations can vary depending on the context. String representations could be in XML or another format, for example. Drawing depends on the graphics libraries in use. As shown, using a typical non-aspect approach, one variant of each concern is implemented in an invasive way. An alternative approach would be to use a design pattern like *Visitor* [14]. I will demonstrate an AOP alternative shortly.

2.1.2 The Open-Closed Principle (OCP)

Software entities (classes, aspects, modules, functions, etc.) should be open for extension, but closed for modification.

If a change in one location causes a cascade of changes to other points in the system, those cascades result in *brittle* systems, because it is hard to find all those points where changes are required. This situation is another form of *rigidity*.

The OCP is a design strategy that minimizes this problem. An entity should be closed for modification, meaning its code cannot be changed, yet open for extension, through subclassing or composition. The OCP reduces rigidity and brittleness because preventing change in the original entity reduces a cascade of changes in dependents.

An example OCP violation is conditional logic that switches on the known classes in a hierarchy (or a "type code"), where a unique action is taken for each case. Introduction of a new class forces updates to all such code blocks. Instead, overloaded methods should be added to the class hierarchy that implement the variant behaviors (*e.g.,* `draw()` in the example). The conditional logic collapses to a single polymorphic method invocation. However, what if the behavior is actually crosscutting and doesn't really belong in the hierarchy?

A related technique that supports the OCP is the *Template Method* pattern [14], where a base class implements a concrete method that defines a *protocol* and which calls one or more abstract methods to complete the details. Subclasses implement the abstract methods to fill in the appropriate behaviors.

However, as discussed in [13], the OCP still has one limitation; it is not possible to anticipate all changes that clients might want. A new client requirement might not be satisfied by the existing abstraction. This will force the abstraction to change, which will probably cause a cascade of client changes.

Even if we could anticipate all possible changes, it would not be desirable to design the original module for all such contingencies, as this could lead to SRP violations, overly-complicated interfaces, bloated and inefficient code, and increased implementation effort, all to support options for change that might never be used.

Let us return to the example and use aspects to refactor it in ways that better support both the SRP and the OCP.

As it stands now, the `Shape` hierarchy satisfies the OCP because we can easily add new shapes without modifying any existing code[3]. Still, let us refactor the design to extract the crosscutting `toString()` and `draw()` "features". For brevity, unchanged classes are omitted.

```
package shapes;
interface Shape {  // draw() removed
    double getArea();
}

abstract class Polygon implements Shape {
    Point getVertex(index i) {…}
}

class Circle extends ClosedCurve {
    double getRadius() {…}
    Point  getCenter() {…}
    double getArea() {…}
}

class Circle extends ClosedCurve {
    double getApogeeRadius() {…}
    double getPerigeeRadius() {…}
    Point  getFocus1() {…}
    Point  getFocus2() {…}
    Point  getCenter() {…}
    double getArea() {…}
}
```

<X>ToString.aj files: // separate aspect files

```
package shapes.tostring; // for all "toString()"…
aspect PolygonToString {
    String Polygon.toString() {
    StringBuffer buff = new StringBuffer();
    buff.append(getClass().getName());
    … append name and area fields …
    … append each line, as "from" and "to" points
    return buff.toString();
    }
}

aspect CircleToString {
    String Circle.toString() {...}
}

aspect EllipseToString {
    String Ellipse.toString() {...}
}
```

Drawable.java:

```
package drawing;
interface Drawable {
    void draw();
}
```

Drawable<X>.aj files: // separate aspect files

```
package shapes.drawing; // for all "draw()"…
import drawing.Drawable;
abstract aspect DrawableShape {
    declare parents: Shape implements Drawable;

    void Shape.draw () {
      String drawCommand = makeDrawCommand();
      // send command to graphics engine...
    }
    String Shape.makeDrawCommand() {
      return getClass().getName() + "\n" +
        makeDetails("\t");
    }
    abstract String
    Shape.makeDetails (String indent);
}

aspect DrawablePolygon extends DrawableShape {
    String  Polygon.makeDetails (String indent){…}
}

aspect DrawableCircle extends DrawableShape {
    String Circle.makeDetails (String indent){…}
}

aspect DrawableEllipse extends DrawableShape {
    String Ellipse. makeDetails (String indent){…}
}
```

DrawLogger.aj:

```
package drawing.logging;
aspect DrawLogger {
    after (Drawable d):
      call (void Drawable+.draw()) && target(d) {
      // log the draw operation
    }
}
```

Listing 2

[3] The exceptions are the places where decisions are made about which shapes to instantiate, *e.g., Factories* [14].

The `toString()` and `draw()` methods have been moved to separate aspects, where *intertype declarations* (ITD's) are used to extend the shape classes with the new methods.

The `Polygon.toString()` method is sufficient for all of `Polygon`'s subclasses. Separate implementations are needed for `Circle.toString()` and `Ellipse.toString()` (details not shown).

For the `draw()` method, I introduce a `Drawable` interface and make `Shape` implement it[4]. The interface provides an important benefit; if we write pointcuts that reference only narrow abstractions like this one, we greatly reduce the fragile coupling caused by the AOSD-Evolution Paradox. Notice that the `DrawLogger` aspect depends only on the `Drawable` abstraction. It has no dependency on the `Shape` hierarchy and therefore it requires no modifications when the `Shape` hierarchy changes, thereby satisfying the OCP.

The `draw()` method is implemented using *Template Method* [14].

Having separate aspects for these methods looks similar to the *Visitor* pattern [14]. However, unlike Visitor, no modifications to the original class hierarchy are required to "accept" visitor objects.

Hence, aspects give us a powerful tool for supporting the OCP. We can extend the behavior without modification of the original classes. Even though intertype declarations (ITD's) are used to introduce new methods into the classes, we don't *manually* modify the shapes code itself. Hence, *ITD's are not OCP-violating modifications.*

The Single Responsibility Principle (SRP) is better supported by this refactoring because the shapes classes are now concerned only with their essential structural properties and behaviors; they are closer to pure domain objects. They could be reused in a wider variety of contexts, with aspects and ITD's used to add new context-dependent state and behaviors, as needed, to support implementation concerns. Also, the overall application structure is more *cohesive,* because each concern is better localized.

The drawback of this approach is that state and behavior for a particular class are no longer defined in a single place. Indeed, this approach is somewhat radical for statically-typed languages. Hence, good tooling is helpful to understand a module's behavior as modified by the aspects in the system [8].

However, developers using languages that allow classes to be reopened for modification (Ruby is a recent and popular example), routinely implement concerns in separate "modules" and reopen classes to incorporate those modules and to make other modifications to the classes.

How do you decide when to put state and behavior in the class definition *vs.* in separate modules, using aspects or other mechanisms? The Common Closure Principle (CCP) provides guidance, which I will discuss later. For now, note that excessively fine-grained modularization spreads information too thin, compromising cohesion and comprehension.

[4] No "`Stringable`" interface is used because `Object` already defines the `toString()` method.

2.1.3 *The Interface Segregation Principle (ISP)*

Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies.

There is a tendency for services to offer fat interfaces with clusters of methods, each of which serves a particular type of client. Any one client will ignore the other method clusters. However, changes to the interface force unwanted changes on clients who aren't using the affected methods.

The solution is for a client to only depend on the narrowest, possible interface that meets its needs. The best way to define that interface is for the client to define it, since the client understands its needs best.

This segregation of fat interfaces is the interface analog of the SRP for classes and aspects. Pointcuts that only depend on such interfaces are less affected by the AOSD-Evolution Paradox. The `Drawable` interface is a good example of a minimal interface.

2.1.4 *The Liskov Substitution Principle (LSP)*

Subtypes must be substitutable for their base types.

If a program *P* depends on the behavior defined by a base class *B* and *D* is considered a derived class of *B,* then instances of *D* must not alter the behavior defined by *B* in ways that break *P*. The LSP says that, under these circumstances, instances of *D* are substitutable for instances of *B*. This is a more precise definition of inheritance than the vague "is a" relationship. Note that substitutability is context dependent. In another program *P2, D* objects may not be substitutable for *B* objects.

Also, substitutability is primarily a *behavioral* trait, not a *structural* one. To see this, recall that the example does not treat `Square` as a subclass of `Rectangle` nor `Circle` as a subclass of `Ellipse`, even though informally the "is a" relationship seems valid in these two cases. *Currently*, since none of the shapes allow modification (they are immutable), once you have a `Square`, you *can* use it anywhere you need a `Rectangle`. However, consider what happens when we extend the shapes to be mutable.

```
package shapes;
class InvalidPointException {…}

class Polygon {
    list<Point> getVertices(); // package private
    …
}
```

Mutable<X>.aj files:

```
package shapes; // SAME package; see discussion
import shapes.Point;
import shapes.InvalidPointException;
import shapes.Polygon;

aspect MutablePolygon {
    void Polygon.setVertex(int i, Point v)
        throws IndexOutOfBoundsException,
               InvalidPointException {
        if (v == null)
            throw new InvalidPointException();
        this.getVertices().set(i, v);
    }
}
```

Listing 3

We use an aspect to add a setter method to `Polygon`. This forces a few other refactorings. In AspectJ, methods introduced into a class must obey the same scoping rules that apply to other methods in the system, by default[5]. In other words, the introduced methods can only access public or package private members in the class and only the latter if the class and aspect are in the same package. In this case, the `Mutable` aspects must be added to the `shapes` package, not a different package as we did for the other aspects. Also, so the aspect can modify the vertices, a package private[6] `Polygon.getVertices()` method is added so the aspect can modify the vertices, without exposing this method to clients of `Polygon` outside the package.

So, the mutability enhancement is an example where the OCP doesn't quite succeed, but only a small, backwards-compatible modification to an existing class is necessary. *This characteristic of AspectJ suggests that, when the OCP and aspect-aware class design issues are considered we should consider making all private members package private instead, especially if we expect to use ITD's.*

Other details of this enhancement are omitted for brevity, such as the details of the mutability enhancement for `Ellipses` and `Circles` and the requirement that changing one vertex in a `Square` or `Rectangle` requires others to change so the angles remain 90 degrees!

With the enhancement defined, let us return to the LSP. Now consider the following unit test for `Rectangles`.

```
public class RectangleTest extends TestCase {
    public void testPerpendicularSideLengths(){
      Point zerozero = new Point(0,0);
      Point zerotwo  = new Point(0,2);
      Point fivezero = new Point(5,0);
      Point fivetwo  = new Point(5,2);
      // C'tor arguments are the vertices
      Rectangle r = new Rectangle(
        zerozero, zerozero, zerozero, zerozero);
      assertEquals(0, r.getArea());

      r.setVertex(1, zerotwo);  // change them
      r.setVertex(2, fivetwo);
      r.setVertex(3, fivezero);
      assertEquals(zerozero, r.getVertex(0));
      assertEquals(zerotwo,  r.getVertex(1));
      assertEquals(fivetwo,  r.getVertex(2));
      assertEquals(fivezero, r.getVertex(3));
      assertEquals(10, r.getArea());
}
```

Listing 4

What if the test instantiates a `Square` instead of a `Rectangle`? The test will now fail. *In the context of the test*, where modifications are expected, the LSP says that a `Square` is *not* a valid substitute for a `Rectangle` and hence not a subclass.

While some potential LSP violations are prevented by language restrictions, others are not. A common example is switching on object type, mentioned previously when discussing the OCP. Suppose a method *m* takes a parameter of type *B* and it has conditional logic to perform different work based on the actual class of the parameter. Introducing a new derived class *D'* of *B* would break this method, unless it is suitably modified.

The LSP is the primary theoretical basis for *Design by Contract* (DbC) [18], a technique for defining an *executable* form of a module's contract of use. DbC is one way of quantifying substitutability. (Unit tests are another, as I demonstrated above for `Rectangles` *vs*. `Squares`.)

Design by Contract stipulates three characteristics of a contract.

- *Preconditions* for a method must be true before it can execute, *i.e.,* constraints on the method parameters, object state, and global data. They define what the method requires in order to work successfully.
- *Postconditions* must be true when the method returns, *i.e.,* what the method guarantees to accomplish, assuming the preconditions were met.
- *Invariants* define state invariants satisfied by the object within the atomicity of calls to the visible methods.

The contract also has interesting properties under inheritance. As stated by Meyer [18],

> A routine redeclaration [in a derivative] may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger[7].

A redeclaration can weaken the precondition or strengthen the postcondition because neither change violates the LSP. The new "routine" is still substitutable for the original routine.

Aspects modify this picture. The *effective* contract of an object combines the object's contract in isolation and the effects of the aspects that advice it or make intertype declarations into it. This is another way of discussing modular reasoning for *aspect-aware interfaces* [8].

Hence, when adding aspects to an *existing* system, the aspects must obey the contracts of the objects they affect or else the aspects will break the program[8]. Hence, the aspects+object must behave exactly like instances of subclasses of the object's class. I will revisit this topic in Section 3, when I discuss aspect-specific principles.

Finally, notice that `before` advice can be used to test preconditions, `after` advice can be used to test postconditions, and `around` advice can be used to test invariants. While a contract is an integral part of a module, how it is *used* is sometimes a crosscutting concern. In fact, aspects are an excellent tool for testing and enforcing contracts (See, *e.g.,* [19-21]).

2.1.5 The Dependency Inversion Principle (DIP)

> *(i) High-level modules should not depend on low-level modules. Both should depend on abstractions.*

[5] Unless you use the `privileged` keyword, which should be used with caution, as it bypasses Java's access protection model

[6] As indicated by a comment in the source; recall that I have been suppressing the `public` keyword

[7] As [13] also remarks, "weaker" means that the derivation can choose not to enforce all the original preconditions. However it can add new ones

[8] For a new application, there is no such constraint on the contract

(ii) Abstractions should not depend upon details. Details should depend upon abstractions.

The last principle in this section covers a common flaw seen, *e.g.,* in layered architectures, where classes in the upper layers depend directly on the details of classes in the layers below them. These dependencies are *transitive;* if *A* depends on *B* and *B* depends on *C*, then *A* depends on *C*. This means the high-level application and context-setting modules are *fragile* because they depend on volatile details and they can't be reused easily with different lower layers.

The solution is for both layers to depend on an abstraction, as shown in Figure 4, adapted from [13].

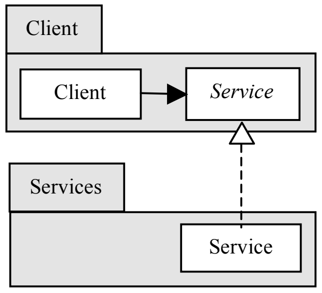

Figure 1

Note that the *Service* interface is defined in the Client layer, not the Services layer, as is usually the case. This has two benefits. First, it allows the client to define exactly the abstraction it needs, nothing more or less (the Interface Segregation Principle). Second, each layer is completely portable, as long as a replacement subordinate layer implements the client-defined interface.

If the layer dependency is actually a tangled concern, then it can be factored out of the top layer completely into an aspect. For example, if the Client needs to persist state to a database provided by the Services layer, then the dependency is actually a tangled concern that may be refactored as shown in the left-hand diagram in Figure 2.

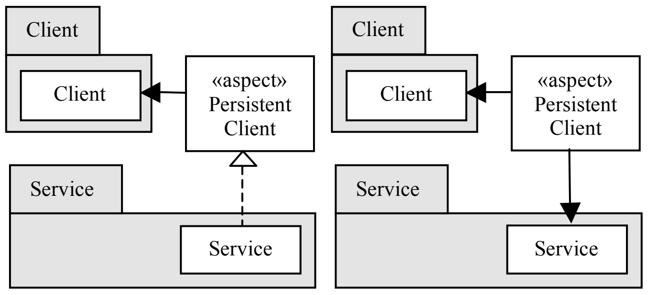

Figure 2

The Client is now decoupled completely from the Services. The aspect observes state changes in the Client and persists the changes. The aspect also defines the abstraction (not shown) that a particular persistence service needs to implement. Hence, the Services dependencies are structurally the same, but the Client is more modular and decoupled.

A different approach is shown in the right-hand diagram in Figure 2. The Services layer no longer implements a "client" interface. Instead, the aspect advices and invokes the Services directly. This variation is more like an aspect implementation of the *Adapter* pattern [14].

Technically, this approach has recreated the DIP violation, this time in the aspect. However, the violation is likely much more localized and hence maintainable, which may be sufficient for real projects.

This approach may also violate the Stable Dependencies Principle (see below), which states that modules should only depend on more stable modules (because an unstable dependency introduces instability into the dependent). This can be avoided if the aspect depends only on stable, generic abstractions in both layers.

Package Cohesion and Coupling Principles

In addition to the five principles just discussed, there is a set of three principles for package cohesion (internal structure) and three for package coupling (inter-package structure) [13].

Most have straightforward implications for aspects and aspects help implement the principles. All the principles are summarized in Table 1. Here, I discuss only the packaging principles with nontrivial aspect implications. (See also the discussion in [17])

In general, because some aspects have pervasive scope, consider carefully how to package them with respect to the modules they advise. On the other hand, as the examples in the previous section demonstrate, aspects can reduce coupling, often by making dependencies more localized, and aspects can make modules and packages more granular.

2.1.6 The Common Closure Principle (CCP)

The classes in a package should be closed together against the same kinds of changes. A change that affects a closed package affects all the classes in that package and no other packages.

The CCP is the package analog of the Single Responsibility Principle (SRP). Since systems evolve, localizing related changes to a single package and making that package cohesive enough that it has only one concern will make it easier to change and to release an updated version when needed. The "closure" part of the CCP relates to the Open-Closed Principle (OCP). Closing a module to modification is not always possible when unanticipated requirements emerge. However, if the changes are isolated, then the impact of change is reduced.

Since aspects make it easier to support the SRP and the OCP, they support the CCP. Also, packages tend to be smaller and more cohesive, as demonstrated by the shapes example.

Earlier, I asked when should functionality be defined in the class declaration vs. in separate aspects, using ITD's? The CCP suggests that the latter approach can yield higher cohesion and lower coupling. The potential trade-off is reduced comprehension as the class members are less localized.

2.1.7 The Stable Dependencies Principle (SDP)

Depend in the direction of stability.

Since changes to dependencies cause a ripple effect to clients, dependencies should point from less stable to more stable packages. Similarly, a package that depends on many other packages is inherently unstable because it is susceptible to change any time one of its dependencies changes.

Note that a package of aspects with pervasive scope can break this principle and the CCP if the aspects are coupled too closely to concrete and volatile details in other packages. This suggests that, *the more pervasive the scope of an aspect, the more abstract its dependencies should be*. Failure to do this is another source of the AOSD-Evolution Paradox [2].

2.1.8 The Stable Abstractions Principle (SAP)

A package should be as abstract as it is stable.

The SDP tells us to depend in the direction of stability. What if we need flexibility in the stable packages? The solution is the Open-Closed Principle (OCP). We design classes and aspects that allow extension without modification. Stability is achieved by putting the stable abstractions in separate packages from the implementations, which are less stable. Any dependencies point only to the stable abstraction packages. *Factories* [14] or other mechanisms are used to satisfy the dependencies with actual implementations, but the clients only know about the abstractions.

Abstract aspects that other aspects extend should also depend only on abstractions and they should be packaged with other stable abstractions.

3. Aspect-Oriented Design

In the previous sections, I summarized the OOD principles from [13] and how they are supported by AOD. Now I return to AOD itself and discuss further how the OOD principles lead us to some aspect-specific design principles. I then discuss noninvasiveness from the perspective of what we have learned.

3.1 Principles of Good AOD

First, AOD refines several of the OOD principles.

3.1.1 The Updated Open-Closed Principle (OCP')[9]

Software entities (classes, aspects, modules, functions, etc.) should be open for extension, but closed for source and contract *modification*

Through intertype declarations (ITD) and advice, aspects actually modify "entities", but in a controlled way. *Ad hoc* manual editing is still discouraged. Because a form of modification still occurs, the principle also emphasizes that the original *contract* of the entity must be preserved, even though this requirement is really covered by the LSP'.

3.1.2 The Updated Liskov Substitution Principle (LSP')

Subtypes must be substitutable for their base types.

Aspects plus base types must be substitutable for the base types.

As far as the LSP is concerned, a base type modified by an aspect must obey the same contract rules as a subtype of the base type. This means that the preconditions can be relaxed, the postconditions can be strengthened, but the invariants must be preserved.

So, the OCP' and the LSP' constrain aspects to maintain the invariance of the module's contract. This leads us to a set of AOD-specific subordinate principles that clarify the LSP' for aspects.

3.1.3 The Advice Substitution Principle (ASP)

Before *advice must support the same or weaker preconditions of the join point it advices.*

After *advice must support the same or stronger postconditions of the join point it advices.*

Around *advice must support the same or weaker preconditions of the join point it advices and the same or stronger postconditions of the join point.*

All advice must support the invariants of the join point.

The ASP clarifies the second part of the LSP', which implies that advice and introductions are effectively a derivation (in the subtyping sense) at a join point. Specifically, `before` advice is a derivation that can change the "initial" behavior, but not the "final" behavior, while the opposite is true of `after` advice. Both behaviors are potentially affected by `around` advice.

Note that the `after` advice principle also applies for exception handling cases, because the thrown exception is also part of the postcondition contract, albeit for abnormal termination.

What about multiple modifications introduced simultaneously? A tricky issue with aspects is avoiding aspect collisions, caused by mutually incompatible advices or introductions. Two or more *superimposed* aspects that are orthogonal should have no affect on each other. Each must separately obey the ASP.

In the general case of superimpositions, most aspect systems provide a precedence mechanism to eliminate arbitrary execution order. The ASP rules follow the precedence rules. If aspect *A* has higher precedence than Aspect *B* and both advise join point *J*, `before` advice for *A* is executed first, followed by `before` advice for *B*, followed by *J*. To satisfy the ASP and hence the LSP', the preconditions of *A*'s `before` advice must support the preconditions of *B*'s `before` advice or weaker preconditions, which must be equal to or weaker than *J*'s preconditions. Also, *A*'s advice must satisfy *B*'s invariants, which must satisfy *J*'s invariants.

Similarly for `after` advice, *J* is executed first, followed by `after` advice for *B*, followed by `after` advice for *A*. Hence, the postconditions of *A*'s `after` advice must support the postconditions of *B*'s `after` advice or stronger postconditions, which must be equal to or stronger than *J*'s postconditions.

The rules for `around` advice combine the rules for `before` advice and `after` advice.

Finally, note that most *non-functional* concerns are often orthogonal to the domain logic and therefore tend to obey the ASP by default. It is when overlapping concerns are discussed, such as the partitioning of domain logic, that the ASP becomes more important.

3.1.4 The Introduction Substitution Principle (ISP2[10])

An Introduction must conform to the contract of the advised module and, if called by advice, it must conform to the ASP of the advice.

This is a corollary to the ASP for introductions, which have an interesting nuance. If an introduction doesn't affect existing join

[9] "OCP prime"

[10] "ISP2", since "ISP" is already taken.

points, *i.e.,* it represents orthogonal state and behavior, it only needs to satisfy the invariants of the advised module[11]. However, if an introduction is invoked from an advice that modifies a join point, then it implicitly affects the join point and therefore the introduction is subject to the same contract as the advice in which it is used.

3.1.5 The Pointcut Inversion Principle (PIP - DIP for Aspects)

Pointcuts should not depend on concrete details; they should depend on abstractions.

This extension of the DIP recognizes that pointcuts are a form of dependency and therefore they should only use abstractions.

Most pointcut languages use regular-expression or similar "query-like" formalisms. This is problematic, because a method name change, for example, requires a more sophisticated analysis to find any affected pointcuts. Tool support for this analysis is limited.

A number of approaches are being investigated for expressing pointcuts in more abstract ways, including logic meta programming (see *e.g.,* [23]) and logical query languages (*e.g.,* [24]).

Until join point abstraction mechanisms mature, several pragmatic solutions help. One solution is to isolate and localize the "bad" coupling and thereby make it more manageable.

A better solution is to write join points that refer only to existing abstraction conventions, *e.g.,* Java interfaces and annotations (Java 5). It may be necessary to refactor existing target code to make pointcuts easier to specify using abstractions. This may appear to violate obliviousness, but refactoring is already an integral part of "agile" development processes, because it promotes adaptable and reusable software, in general.

Indeed, "aspect awareness" is now seen as important for good design [8-12] and aspects should be regarded as first-class design constructs along with classes and interfaces.

3.1.6 The Pointcut Scope Principle (PSP)

The more pervasive the scope of a pointcut, the more abstract it should be.

Leaving the LSP', this PSP is a practical consequence of the CCP and the SDP. A pointcut with pervasive scope must be abstract. Otherwise, the package it contains is too volatile because it depends on too many volatile details in other packages.

3.2 Noninvasiveness

In general, modern languages and frameworks impose controls to prevent unauthorized or ill-advised use of modules. For example, most OO languages have scoping and protection constructs to control access to state information and to restrict behavior, while supporting extension through derivation and composition. Many application frameworks provide security mechanisms to prevent unauthorized activity, intentional or accidental. To achieve mainstream adoption, aspect systems have to evolve beyond naïve *obliviousness* for the same reasons. The idea of *noninvasiveness* was developed to allow aspect weaving without code modification while permitting access controls and general "awareness".

Since advice and introductions must obey the contracts of the join points they advise, the contracts must be explicit enough to constrain the behavior of potential advice and ITD's. A necessary extension is for contracts to be able to define access restrictions on allowed join points [11]. Hence, contract specification, in the generic sense, is an important characteristic of aspect-awareness.

AspectJ follows the protection model of Java, although the `privileged` keyword allows bypassing the access restrictions. It should be used only in carefully controlled circumstances. The language access protection helps, but it is insufficient; I may wish to prevent any advice inside a "critical" method for performance or other reasons, for example.

When restricting the types of advice and introductions, the hardest conditions to specify are those that involve detailed or subjective information about the context of the join point. Furthermore, it is of course not possible to anticipate all conceivable aspects that might be used, so the constraints need to be general enough to affect a reasonably large set of known and potential types of advice.

4. Further Work

While the general principles discussed here are universal, the details probably reflect some Java/AspectJ biases. Further analysis of these ideas for general AOP theory and in other language contexts would be useful. For example, I have only briefly considered *symmetric* AOP systems, *e.g.,* [5].

More work is required to understand the roles of contracts in aspect interfaces, how they should constrain allowed advice and ITD's, and how to define them in a practical, yet effective ways [11]. The discussion of contracts, the OCP' and the LSP' only partially address this topic.

Finally, while I discussed how aspects influence the principles and are constrained by them, I have not explored what these principles say about aspect typing theory itself. More investigation is needed.

5. Conclusions

By examining some well-known principles of good object-oriented design, I demonstrated how aspects support and refine them. I also discussed what these principles tell us about good aspect-oriented design. In particular, I discussed the role of contracts as constraints on how aspects are used in order to preserve program correctness, security, *etc*., thereby supporting the goals of noninvasiveness. Along the way I examined weaknesses in some common aspect design techniques, which lead to problems such as the *AOSD-Evolution Paradox.*

6. ACKNOWLEDGMENTS

My thanks to my colleagues at Object Mentor, New Aspects of Software, and the contributors on "aosd-discuss" for stimulating discussions.

7. REFERENCES

[1] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J., Aspect-oriented programming.

[11] This is one reason it is often easier to use introductions, rather than advice, to extend entity behavior without violating the OCP.

European Conference on Object-Oriented Programming (ECOOP), 1997, 220-242.

[2] Tom Tourwé, Johan Brichau and Kris Gybels, On the Existence of the AOSD-Evolution Paradox, *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies* (Boston, Massachusetts, USA, March 17-21, 2003).

[3] AspectJ. http://www.aspectj.org/.

[4] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns*, OOPSLA 2000, 2000.

[5] Ossher, H. and Tarr. P. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. *Proceedings of the Symposium on Software Architectures and Component Technology*. Kluwer, 2000.

[6] Bergmans, L. and Aksit, M. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM,* 44(10:51-57, October 2001.

[7] "Obliviousness Principle in Aspect-Oriented Software Development," *aosd-discuss* thread, http://server2.hostvalu.com/pipermail/discuss_aosd.net/2003-August/000617.html.

[8] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on software engineering*, 2005.

[9] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. *SIGSOFT Softw. Eng. Notes*, 26(5):99–108, 2001.

[10] K. J. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, H. Rajan. On the criteria to be used in decomposing systems into aspects. In *ESEC/FSE'05: Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering.*

[11] Griswold, W.G., K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai and H. Rajan, Modular Software Design with Crosscutting Interfaces, *IEEE Software, Special Issue on Aspect-Oriented Programming*, January/February, Volume 23, Number 1, 2006, pp. 51-60.

[12] C. V. Lopes and S. K. Bajracharya, An analysis of modularity in aspect oriented design. In *Proceedings of AOSD 2005* (Chicago, IL, USA, March 14-18, 2005). ACM Press, New York, NY , 2005, pp. 15–26.

[13] Martin, R., Newkirk, J., and Koss, R. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, Upper Saddle River, NJ, 2003.

[14] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns; Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[15] Hannemann, J. and Kiczales, G. Design Pattern Implementation in Java and AspectJ. In *Proceedings of OOPSLA '02* (Seattle, Washington, USA, November 4-8, 2002). ACM Press, New York, NY, 2002, 161-173.

[16] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., and von Staa, A., Modularizing Design Patterns with Aspects: A Quantitative Study, *Proceedings of AOSD 2005* (Chicago, IL, USA, March 14-18, 2005). ACM Press, New York, NY , 2005, pp. 3-14.

[17] Nordberg, M. E. Aspect-Oriented Dependency Inversion. *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[18] Meyer, B., *Object-Oriented Software Construction*, 2nd edition. Prentice Hall, Saddle River, NJ, 1997.

[19] *Contract4J: Design by Contract for Java*. http://contract4j.org.

[20] Skotiniotis, T. and Lorenz, D. Cona -- Aspects for Contracts and Contracts for Aspects. http://www.oopsla.org/2004/ShowEvent.do?id=594.

[21] *Barter – Beyond Design by Contract*. http://barter.sourceforge.net/.

[22] De Volder, K. and D'Hondt, T. Aspect-Oriented Logic Meta Programming, *Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99. LNCS 1616*. Springer-Verlag, 1999, pp. 250-272.

[23] *JQuery, a Query-Based Code Browser*. http://jquery.cs.ubc.ca/.

[24] Larochelle, D., Scheidt, K. and Sullivan, K. *Join Point Encapsulation.* http://www.cs.virginia.edu/~eos/papers/encapsulation.pdf.

Name	**Definition†**	**AOD Perspective**
Single Responsibility Principle (SRP)	A class or aspect should have only one reason for change. (*I.e.,* it should do only one thing.)	Tangling of concerns is a common source of SRP violations, *e.g.,* an "entity" class that also handles its own persistence and transactional behavior. Aspects provide additional tools for supporting the SRP.
Open-Closed Principle (OCP)	Classes and aspects should be open for extension, but closed for modification.	The word "closed" is refined to mean closed for *manual* source modification. Aspects modify the entity in a controlled way, but they must obey the join-points' *contract*. This is easier for "orthogonal" state and behavior changes.
Liskov Substitution Principle (LSP)	Subtypes must be substitutable for their base types. (LSP is the basis for *Design by Contract* [18])	Factoring out crosscutting concerns reduces the likelihood of LSP violations. Aspects must preserve the contract expected by *existing* clients of the module.
Interface Segregation Principle (ISP)	Clients should only depend upon methods that they use. Interfaces belong to clients. (SRP for interfaces.)	Aspects provide additional ways to integrate services with clients.
Dependency Inversion Principle (DIP)	(i) High-level modules should not depend on low-level modules. Both should depend on abstractions. (ii) Abstractions should not depend on details. Details should depend on abstractions.	DIP violations are the biggest contributor to the AOSD-Evolution Paradox problem, when pointcuts use concrete join point details. Hence, pointcuts should only reference abstractions. For dependencies that are concerns not related to the domain logic, extraction into aspects localizes the coupling to the aspects themselves.
Release-Reuse Equivalency Principle (REP)	The granule of reuse is the granule of release.	Aspects that are closely coupled to packages may need to be part of the release "granule". Special care is required when packaging aspects with *pervasive* scope.
Common Reuse Principle (CRP)	The classes and aspects in a package are reused together. If you reuse one of them in a package, you reuse them all.	Aspects promote the "SRP for packages", but also require careful packaging due to dependencies on other packages.
Common Closure Principle (CCP)	The classes and aspects in a package should be closed together against the same kinds of changes. A change that affects a closed package affects all the classes and aspects in that package and no other packages.	Aspects promote having packages with one concern. An AOSD system will tend to have more packages, but they will be smaller, more cohesive, and with less coupling between them.
Acyclic Dependencies Principle (ADP)	Allow no cycles in the package dependency graph.	Aspects are one tool for breaking cycles, *e.g.,* by supporting the DIP.
Stable Dependencies Principle (SDP)	Depend in the direction of stability.	Aspects that don't depend on abstractions contribute to the AOSD-Evolution Paradox.
Stable Abstractions Principle (SAP)	A package should be as abstract as it is stable.	The SAP applies to aspects, too.

†Adapted from [13]. The definitions reflect enhancements for aspects. Note that the first 5 form an acronym: SOLID.

Title 1: Object-Oriented Design Principles [13], extended for Aspects