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

Dean Wampler
Aspect Programming
aspectprogramming.com
dean@aspectprogramming.com

Draft

ABSTRACT

For aspect-oriented design (AOD) to become mainstream, appropriate design principles are needed to guide its proper use in real, evolving systems. Design principles should tell us what types of coupling are appropriate between aspects and the software entities they advise, what if any restrictions should exist on non-invasiveness, how can aspects be used in ways that preserve correct behavior in the advised entities, and how do aspects complement other design constructs? I examine these questions 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 introduces nuances in their interpretations. Conversely, the OOD principles suggest good AOD-specific principles. In particular, they have implications for noninvasiveness, which is aspect weaving without module modification, but with appropriate controls.

Categories and Subject Descriptors

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

General Terms

Design, Theory.

Keywords

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

1. INTRODUCTION

Aspect-Oriented Software Development (AOSD) is attractive because of its ability to modularize crosscutting concerns. However, for mainstream adoption, it must also promote good engineering principles for initial development and for ongoing evolution to address changing requirements [1].

To date, modularization of concerns like input & output (I/O), persistence, security, logging, performance, *etc.* have seen the most activity in Aspect-Oriented Design (AOD). These concerns are sometimes described as part of the "nonfunctional" requirements of a typical application, in contrast to the domain logic, which is represented by the functional requirements.

AspectJ [2] introduced the term *advice* for the code inserted by an aspect into a software entity[1], reflecting this emphasis on augmentation to an existing design [3], rather than a fundamental reworking of it[2].

These types of aspects usually require no modification of the entity, a property initially referred to as *obliviousness* [4]. Obliviousness usually works in these cases because the domains of the entity and the aspects are often disjoint.

However, partitioning the domain logic itself into aspects is more likely to introduce logic conflicts, because the domain entities will tend to overlap; similar objects play different roles in different aspects. Obliviousness is not sophisticated enough to address these design issues. For example, Hyper/J [5] and Composition Filters [6] both implement aspects separately and generate applications using a composition meta-language. How do we ensure that the composition process does not corrupt each aspect's behavior in isolation? Appropriate forms of "active" collaboration are needed to preserve logical correction during aspect composition to create applications.

Even for disjoint entities and aspects, an entity may need some control over possible advices, introductions, and join points, for reasons of program correctness, security, *etc.*

Recently, the term *noninvasiveness* (See, *e.g.,* [7]) was proposed as an alternative to obliviousness. It retains the notion of advice insertion without direct entity modification, but it recognizes the need for techniques of control.

Do AO applications evolve? Tourwé, *et al.* [1] argue that aspects written with current technologies tend to be tightly coupled to the rest of the application logic, leading to an *AOSD-Evolution Paradox*, where the initial version of an application using AOSD has better modularization than a comparable, non-AOSD implementation, but change to satisfy evolving requirements is harder, due to tight coupling of the aspects to the rest of the application. This coupling occurs because current ways of specifying *join points* tend to be very concrete and make explicit assumptions about program structure, *e.g.,* the package hierarchy, naming conventions, *etc.* Part of the paradox is that obliviousness only exists on the application side of the application-aspect relationship, while the aspects are not at all oblivious to the application they advise. This paradox is a practical barrier to AOSD adoption.

To explore noninvasiveness, good AOD, and issues of software evolution, I start with a set of object-oriented design and

[1] "Entity" refers generically to a module, package, class, aspect, function, *etc.*

[2] However, AspectJ advice is more powerful than the term might suggest. We frequently use AspectJ terminology because it is familiar.

packaging principles described by Martin [8]. AOD facilitates these principles in various ways, but these principles also suggest guidelines for good AOD itself, including effective controls and noninvasiveness. This analysis compliments the work of other authors who have examined design patterns [9] from an AO perspective [10-11]. I also consider the evolution of AO-based software, by examining the *AOSD-Evolution Paradox* in more detail.

2. Principles of Object-Oriented Design

Martin has described eleven principles of OO design and packaging [8]. Five deal specifically with class and interface design as they affect issues of evolution, reuse, and stability. Three focus on package cohesion and three focus on package coupling. They are summarized in Table 1.

In this section, I describe the principles, conventional OOD approaches to them, and how to support them using AOD techniques. Later, I examine what they tell us about good AOD.

2.1 OOD Design Principles

2.1.1 The Single Responsibility Principle[3] (SRP)

A class should have only one reason to change.

The SRP states that a class[4] with multiple reasons to change, reflecting multiple purposes or tangled concerns (AOSD terminology) is hard to change when requirements change, making the class *rigid*.

The rigidity comes from the coupling of concerns in the class. 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 static. Furthermore, since each concern usual has external dependencies, the dependencies become coupled, too. Hence, reuse is compromised in applications that don't need some of the concerns and their dependencies.

Note that the way the SRP is worded highlights the fact that a tangled entity that never needs to change poses only minor issues. The need for maintenance, evolution, and reuse make tangling a problem that needs to be solved.

The SRP is another way of stating the classic separations-of-concerns problem that AOSD, like OOSD before it, was invented to help solve. Hence, standard *refactoring* techniques from both disciplines apply, *e.g.,* [12].

2.1.2 The Open-Closed Principle (OCP)

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

Another form of *rigidity* exists when a change in one location causes a cascade of changes to other points in the system, which is a barrier to making the original change. These cascades also tend to result in *brittle* systems, because it is hard to find all the points where changes are required, so bugs often result from the changes.

The OCP is a solution to this problem. An entity can be open for extension, so its behavior can be changed, but it must be closed for modification, meaning its code cannot be changed. The OCP reduces rigidity and brittleness because preventing change in the original entity and its clients, as long as the interface itself doesn't change, prevents a cascade of changes.

Instead, new entities are added to create extensions. Typically, they implement an abstraction, such as an interface, exposed by the original entity. An appropriate mechanism for connecting a particular implementation to the client must also exist, such as a *Factory* [9].

An obvious example of an OCP-violating application is one that hard-codes conditional logic for the known types in a hierarchy, where unique action is taken depending on the object's class. Introduction of a new type forces updates to all such code blocks.

The usually OOD solution is to declare methods in the original abstraction that represent the variant behaviors and then to define concrete implementations of the methods in classes that satisfy the abstraction.

A second common approach is the *Template Method* pattern [9], where a base class defines a set of unimplemented methods and a concrete method that invokes them in a particular order, thereby defining a *protocol*. Subclasses define the unimplemented methods to specify the actual behavior.

However, as Martin points out, the OOD approaches to OCP still have 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 you could anticipate all possible changes, it would not be desirable to design the original entity for all such contingencies, as this would lead to overly-complicated entity interfaces, bloated and inefficient code, and an unacceptable implementation effort, all to support options for change that might never be used.

This is one of the reasons that frameworks have not been as successful as expected. Traditional frameworks face a *catch-22* situation; they need maximal flexibility to be useful to a broad range of projects, yet that same flexibility tends to make them bloated with lots of complexity and undesirable overhead, which is a barrier to adoption. Agile Programmers, particular those in the Extreme Programming (XP) camp, reject the traditional notion of designing-in program extension points ("hooks") to support potential extensions that might be needed in the future. Instead, they *refactor* the software to include those hooks when they are actually needed. [8,12]

Consider a simple example of geometric shapes that satisfies the OCT. It has an overridden `draw` method and a client that iterates through a list of shapes and draws them (Listing 1).

```
Shape.java:
interface Shape {
    public void draw();
    void drawAllShapes (Vector list);
};
```

[3] The names, acronyms and definitions are quoted from [8]. Note that the where "class" appears, "aspects" can be added to it. The descriptions and examples are adapted from [8].

[4] The SRP could also refer to aspects, of course. Packages or similar *collaborations* like modules and components aren't mentioned because they are discussed in several of the other principles.

```
Circle.java:
class Circle implements Shape {
   public void Draw() {… }
};

Square.java:
class Square implements Shape {
   public void draw() {… }
};

ShapeClient.java:
class ShapeClient {
…
   void drawAllShapes (Vector list) {
      Iterator  i = list.iterator();
      while (i.hasNext())  {
         Shape s = (Shape) i.next();
         s.draw();
      }
   }
}
```

Listing 1

The `drawAllShapes` method assumes that the shapes can be drawn in any order. Now suppose that a new client wants to draw shapes ordered by the number of vertices they contain, *i.e.,* circles (0), points (1), lines (2), triangles (3), *etc.* In our contrived example, assume the client can't order the list in advance, so the existing `drawAllShapes` method is not usable. There is no way for the client to query the shape for the number of vertices, so the original `Shape` interface has to be modified to declare an overridden method that returns the number of vertices. The client can then write a method for drawing shapes in order. Because the `Shape` abstraction changes, all clients must be changed or at least rebuilt[5].

The AOD solution is to extract interface elements that reflect different concerns and to make them into separate aspects. The `draw` method is part of an I/O concern that should be extracted into a separate set of aspects (most likely one per class in the `Shape` hierarchy). If the `draw` methods are added through introductions, they will reproduce the abstraction that existing clients already use, making modification to them unnecessary[6].

The number of vertices is an intrinsic property of a shape, so it makes sense to add a "get" method to the original abstraction with implementations in the hierarchy. We could either modify the hierarchy directly (preferred, though "painful") or use another set of aspect introductions. The latter approach has the virtual of allowing us to keep the original abstraction for those clients that don't care about the new functionality. This may be a practical requirement to minimize the impact to an existing system.

Hence, new aspects can be introduced to address new requirements, while still satisfying the OCP. Listing 2 shows an AspectJ example that refactors the original design in two ways. Clients of the original `draw` method still have it, but it has been refactored to a set of aspects. At the same time, the support for vertices has been added to the original abstraction (rather than using aspect introductions).

```
Shape.java:
interface Shape {
   public int getNumVertices();
}

Circle.java:
class Circle implements Shape {
   public int getNumVertices() { return 0;
};
}

Square.java:
class Square implements Shape {
   public int getNumVertices() { return 4;
};
}

DrawableShape[7].aj
aspect DrawableShape  {
   public void Shape.draw() { … }
};

DrawableCircle.aj:
aspect DrawableCircle  {
   public void Circle.draw() { … }
};

DrawableSquare.aj:
aspect DrawableSquare  {
   public void Square.draw() { … }
};

ShapeClient.java:
class ShapeClient { /* same as before! */  }

ShapeClient2.java:
Class ShapeClient2 {
   protected Vector
   sortByNumVertices (Vector v) { … };

   void drawAllShapes (Vector list) {
      Vector list2 = sortByNumVertices(v);
      Iterator  i = list2.iterator();
      while (i.hasNext())  {
         Shape s = (Shape) i.next();
         s.draw();
      }
   }
}
```

Listing 2

[5] Adding a new method doesn't change the part of the interface the existing clients care about, but it does change the "binary" footprint.

[6] However, you have to be careful about issues like "binary compatibility", depending on the aspect system in use.

[7] At a naïve level, aspects are the *adjectives* that modify the object *nouns.* Therefore, I use aspect names that are adjectives with the name of the object they modify appended to avoid "namespace" conflicts. I will leave it to the reader to decide if this naming convention makes sense. (What about aspects that advise multiple classes in several hierarchies?)

Using aspect refactoring and introductions reduces the size of the original abstraction (the dubious service method `drawAllShapes` is gone from `Shape`), yet makes it more flexible for future changes without breaking existing clients. However, since aspects involve interface or class modification, do they violate the OCP? Technically they don't, since the actual source code is not changed, just the run-time structure. Still, we are inserting new methods and state into the entity's run-time structure. Is that safe?

First, practically speaking, for some aspect systems and languages, adding a new aspect will require rebuilding the clients, which strict OCP seems to oppose. For situations where this is acceptable, modifications through aspects won't violate the OCP if they preserve the *contract* of the software entity, in the sense of Bertrand Meyer's Design by Contract (DbC) [13] principle. I will return to the role of DbC in AOD in more depth in subsequent sections.

Returning to aspect-based refactoring and modularization, how far should the designer go with such fine-grained modularization? AOD probably won't eliminate the problem that, at some level, extreme modularization will result in obfuscation because information gets spread over many entities, and its benefits in flexibility.

Multi-Dimensional Separation of Concerns [5] attempts to strike the appropriate balance by embracing the partitioning of the domain model into a set of concerns, each of which focuses on a particular "feature". Concerns are composed together to create applications. Jacobson has proposed that use cases are these domain-logic concerns [14].

2.1.3 The Liskov Substitution Principle (LSP)

Subtypes must be substitutable for their base types.

Programs that depend on a base class *B* will likely break if a derived class *D* is used that in some way does not conform to the behavior defined by *B*. More specifically, if a program *P*'s behavior is unchanged by the substitution of *D* for instances of *B*, then *D* is considered a subtype of *B*.

Most OO languages restrict the possible violations of the LSP. Most allow derived classes to *add* to the public interface defined in a parent class, but not *remove* items from it, *e.g.,* by declaring a parent's public method private. Removing a method in *D* and using it in place of a *B* would break clients expecting the missing method.

Most LSP violations not prevented by language restrictions are also violations of the OCP. A common example is the abuse of run-time type identification (RTTI) facilities in languages like Java and C++. For example, suppose a function takes an argument of type *B* and it has to take action based on the actual subtype of an object passed in for the argument. This would violate the OCP, since the method would probably misbehave if an object of a new derived class D', unknown to the function, were passed to it.

A more subtle OCP violation is suggested by the description of the LSP above. Class *D* is considered a subclass of *B* if the behavior of program *P* is invariant. This implies two things: (i) that substitutability, or the "IS-A" relationship, is a statement about *behavior,* not *structure*, and (ii) that substitutability is actually in the context of the client using the classes. Martin demonstrates these points using the common, but questionable, assertion that a `Square` is a subclass of a `Rectangle`.

Structurally, a `Square` has the same properties as a `Rectangle`. That is, you can define a square using a `Rectangle`'s four points. However, as far as a `Rectangle` is concerned, its width and height can vary independently, while a `Square` constrains them to be equal. Naïvely, a `Square` could simply set its width equal to its height any time the one or the other is changed. However, consider the following client program of a `Rectangle`.

```
Void testSanity (Rectangle& r) {
    r.SetWidth(4);
    r.SetHeight(5);
    assert (r.GetArea()  == 20);
}
```

Listing 3

This client of a `Rectangle` assumes the area should equal the height times the width, as explicitly set in the preceding lines. However, this fails for `Squares`, because a `Square` doesn't obey the *contract* for a `Rectangle`'s *behavior*, even though it is *structurally* the same.

The example also illustrates that the validity of the domain's behavior can't be determined in isolation, but only in relation to the expectations of its clients! The LSP and the OCP both show the importance of understanding and maintaining an entity's contract, relative to client expectations. I will discuss below the implications of the LSP for advice and introductions.

2.1.4 The 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 upon details. Details should depend upon abstractions.

A common flaw in layered architectures is for the upper layers to depend directly on the details of the layers just below them. These dependencies are *transitive;* if layer A depends on layer B and layer B depends on layer C, then layer A depends on layer C. This creates the perverse situation where high-level application and context-setting modules are both *fragile* in the face of change 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 Martin [6].

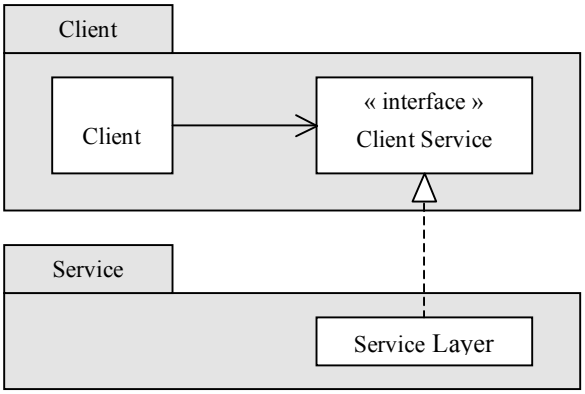

Figure 4

Note that the interfaces are actually defined in the client layer, not the serving layer, as is commonly seen. This has two benefits. First, It allows the client to define exactly what services it needs, nothing more or less. This supports the Interface Segregation Principle (ISP), which is discussed below. The second benefit is that each layer is completely portable, as long as a replacement subordinate layer implements the client-defined interface.

A simple heuristic for the DIP is "depend on abstractions", which implies three things.

1. No variable should hold a pointer or reference to a concrete class.
2. No class should derive from a concrete class.
3. No method should override an implemented method in any of its base classes.

Some violations of this heuristic are fine, such as dependencies on concrete but very stable classes (*e.g.,* Java's `String` class).

Nordberg [11] shows how aspects can solve examples of dependencies that violate the DIP and the Acyclic Dependency Principle (ADP), discussed below, such as the well-known Visitor Pattern [9]. He also argues that one of the reasons that component-based development (CBD) has not been successful is because software parts have dependencies on connectors that are typically both concrete and unstable, in contrast to mechanical and electrical "CBD", where the connectors are well standardized and stable. Dependencies on unstable connectors make component development and assembly infeasible.

If the layer dependency is actually a tangled concern, then it should be factored out of the top layer. If so, then untangling the concern may eliminate the dependency. For example, if the Client layer in Figure 4 needs to persist state to a database in the Service layer, then the dependency is actually a tangled concern that may be refactored as shown in Figures 5 and 6.

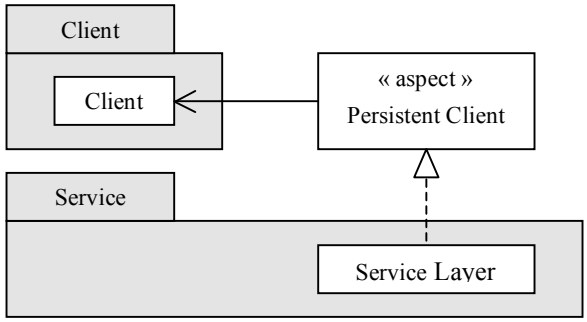

Figure 5

In Figure 5, the Client module is now decoupled completely from the Service module. All the dependencies are in the aspect. Here, I assume that aspect still defines an abstraction that the Service module implements. Hence, the modules are decoupled accept at one point, which is easier to maintain and modify when required.

A different approach is shown in Figure 6, where the notion of an interface is removed and the aspect now depends directly on both the Client and the Service modules. Technically, this violates the DIP again. Furthermore, it probably also violates the Stable Dependencies Principle (SDP), discussed below, which states that modules should only depend on stable modules, because an unstable dependency introduces instability in the dependent.

Recalling the *AOSD-Evolution Paradox* discussed in [1], the coupling of the aspect in Figure 6 is concrete in two directions, making *integration aspects* unstable and rigid. Unfortunately, such aspects are very common. However, if the aspects are small enough, changes will be easy to make when required. Hence, the rigidity should be manageable. As usual in real-world projects, we don't need perfect solutions to all design problems, just "good enough" solutions.

However, in practical terms violations of our principles may be tolerable if they are well isolated and small enough that the trouble they cause is nominal. In the example, the persistence aspect may meet these criteria and the violations of the DIP and the SDP may be tolerable. We should only solve problems worth solving!

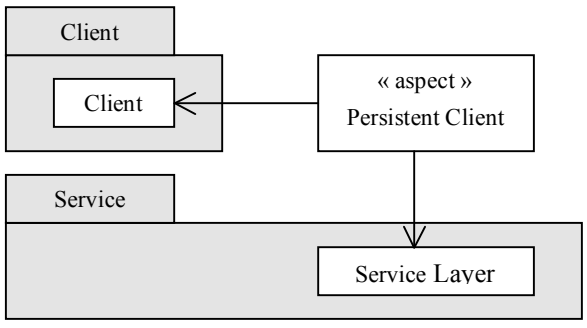

Figure 6

Consider other recent approaches to decoupling concerns. The various J2EE specifications [15] tried to decouple concerns with

mixed success[8]. "Application assemblers" join modules together and configure properties for a fixed list of known concerns, such as persistence and transactions, using XML-based property files called deployment descriptors. This is done without modifying the source code of the modules, making them more reusable.

A more successful approach to decoupling of aspects and satisfying the DIP has been the recent emergence of lightweight containers for enterprise applications that exploit two concepts called *Inversion of Control* (IoC) and *Dependency Injection (DI)* [16].

IoC is fairly common in frameworks. Instead of objects managing their own lifetimes or creating custom, *ad hoc* manager objects, this chore is inverted by having a container handle lifecycle management, usually according to a well-defined protocol, but will requiring container-specific lifecycle code to be embedded in the objects[9].

Dependency Injection is a special type of IoC that adds facilities for making the container populate the dependencies of objects, the properties (attributes) and references to other objects, rather than requiring each object or an *ad hoc* manager to populate the dependencies.

For example, the Spring framework [16] has containers that can manage plan-old Java objects (POJOs) without requiring the objects to embed container-specific lifecycle code or to use JNDI to locate dependencies[10]. Instead, the containers use the constructors and the JavaBean properties [17] of the object as the client-defined interface that the container "implements"[11], as in our layer example. Hence, Spring and similar IoC/DI systems satisfy the DIP. Also, a basic AO system in Spring provides weaving of concerns like persistence and transaction management. IoC/DI plus aspect support almost completely remove requirements for coupling between application objects, containers, and supporting libraries. This greatly improves comprehension (by supporting the SRP), testing, reusability, and maintenance.

2.1.5 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 convenience, for services to offer fat interfaces with clusters of methods. Each cluster serves a particular client type. Any one client will ignore the other clusters. The problem is that changes to the interface in the uninteresting methods can affect the clients. This is another manifestation of tangled concerns.

The solution is for the clients to define the interfaces. They will include just the services the client needs, as discussed in the previous section on the Dependency Inversion Principle.

Of course, aspects are especially useful for both untangling concerns and in localizing interfaces where they are actually needed. As in our previous example, if the interface used by a client is actually for a concern, it may be extracted to an aspect where the interface is collocated with the glue code that joins the client and server modules together.

2.2 Package Cohesion Principles

The previous set of principles focused mostly on classes, interfaces, and their relationships. Now, I briefly review the principles in [8] that deal with groupings of classes and interfaces, organized as packages in languages like Java.

2.2.1 The Release-Reuse Equivalency Principle (REP)

The granule of reuse is the granule of release.

A practical issue with software is the release and maintenance processes. Developers release updates periodically and at the same time, clients seek ways to reuse modules. Since some classes will always have some dependencies on others, the developer should package together dependent classes into a release "granule" and clients should expect that they will need to reuse the entire granule or none of it. It is not realistic to expect to pick and choose pieces of a release for reuse.

When aspects are used, they should be part of the release-reuse granule if they are tightly coupled to entities within it. Similarly, an update to a granule of aspects may force an update to its dependencies.

2.2.2 The Common Reuse Principle (CRP)

The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.

Classes that form tight collaborations are natural choices for grouping. However, classes with weak coupling should not be packaged together. Suppose a package depends on only one class in a different package, the dependency is still to the entire package, because that is the granule of reuse, according to the REP. Therefore, it is best to package together classes that are so closely linked that they are inseparable and a dependency to one is effectively a dependency to all. This will result in small, well-focused packages.

Separating concerns as aspects makes it easier to create small-well-focused packages, but care must be taken in designing the coupling between these otherwise, well-focused packages.

2.2.3 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 version of the Single Responsibility Principle (SRP), which applied to individual classes. CCP is a practical principle; change is required as systems evolve and localizing the change to one package and making that package cohesive enough that it has only one concern, will make it easier to change and to release an updated package when needed.

The "closure" part of the CCP relates to the Open-Closed Principle (OCP). As we've seen, closing an entity to modification is not always possible when unanticipated requirements emerge.

---

[8] Many ideas in J2EE can be viewed as imperfect precursors to better AOSD approaches.

[9] The Enterprise Java Bean (EJB) container also manages lifecycles of "beans", but it requires container-specific lifecycle code to be embedded in the bean implementation.

[10] Although, when special circumstances require it, container APIs are available for implementing customized behavior.

[11] Reflection is used to determine the client's interface and XML files are used to specify how to satisfy the dependencies.

However, if the changes are limited to a few packages, then the impact of change is reduced.

As discussed before, aspects make it easier to support the SRP and the OCP. Therefore, by extension, they help support the CCP.

2.3 Package Coupling Principles

The package coupling principles focus on dependencies between packages.

2.3.1 The Acyclic Dependencies Principle (ADP)

Allow no cycles in the package dependency graph.

If you graph the dependencies between packages, it should form a directed acyclic graph (DAG). When there are cycles in the graph, all the packages in the cycle must be built, tested, and released together. They are effectively one large package. In a DAG structure, building, testing, and releasing a package only requires the sequence of its dependent packages down to leaf nodes, which are packages with no dependencies. In fact, this structure makes it easy to rebuild the application by building and testing the leaf-node packages first, then their immediate dependents, *etc*., up to the top-level package upon which no other package depends.

When cycles occur, they can be removed by applying the Dependency Inversion Principle (DIP) or by factoring out dependent classes into a new package so that a DAG structure is restored. Of course, refactoring to aspects can assist in this process.

An interesting implication of the ADP is that top-down package design based on a functional decomposition usually doesn't work, because it becomes necessary to introduce packages that may not have an obvious association with the domain logic. Instead the package structure has to evolve to facilitate the *buildability* of the application.

2.3.2 The Stable Dependencies Principle (SDP)

Depend in the direction of stability.

As demonstrated previously, it is harder to justify changing a package if it has a lot of dependents, since a change will force changes, or at least rebuilding, of the dependent packages. Therefore, to minimize this ripple effect, 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.

2.3.3 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), which tells us to design classes that promote extension without modification. Abstract classes or interfaces satisfy the OCP.

The stability requirement is achieved by having the dependencies point to packages that contain only abstractions, which are relatively stable, while other, less-stable packages provide concrete implementations of the abstractions, which tend to be less stable. In an application that supports the SAP, the only dependents of the concrete packages should be one or a few "factories" that glue the application together.

3. Aspect Design

In the previous sections, I summarized the OOD principles from [8] and how they are supported by AOD. Now I return to AOD itself and discuss how the OOD principles lead us to AOD-specific principles. I then discuss noninvasiveness from the perspective of what we have learned.

3.1 Principles of Good Aspect Design

The OOD principles can be applied to AOD in some obvious ways, which we won't discuss in detail here, but which are summarized in the last column of Table 1. For example, each principle should be rephrased to include the word "aspect" where the word "class" (or entity) appears. Also, aspects should follow the same packaging recommendations, *etc.*

However, aspects suggest a definition for a new Open-Closed Principle (OCP' – "prime"):

3.1.1 The New *Open-Closed Principle (OCP')*

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

That is, aspects actually modify the entity they advice or provide introductions, but in a specific and controlled way (manual editing is still not permitted, for example). Aside from possibly having to rebuild clients (depending on the aspect system), these modifications are acceptable as long as they do not violate the original *contract* of the entity.

So, the OCP', along with the Liskov Substitution Principle (LSP), constrain aspects to maintain the invariance of the entity's contract. This leads us to our first AOD-specific principle.

3.1.2 The Advice Invariance Principle (AIP)

Advice must conform to the contract of the advised join points.

An extensible entity defines a contract, which includes an abstraction (*e.g.,* an interface) and constraints on its use that must be respected by extensions, whether those extensions are subclasses, new entities conforming to an interface, or aspects.

The abstraction is typically a set of method signatures and sometimes data objects that clients may use. Language-specific AO systems keep the signatures invariant. (If they tried to change the signatures, run-time or compile-time errors would usually occur.)

The constraints on use of the abstraction are best described using Meyer's Design by Contract [13], which stipulates three kinds of contracts.

- *Preconditions* for a method must be true before it can execute. They define what the method requires in order to do its job successfully. Typically, they are constraints on the arguments to the method, object state, and/or relevant global data.
- *Postconditions* must be true when the method returns. They define what the method guarantees to accomplish, assuming the preconditions were met.
- *Invariants* define state invariants satisfied by the entity within the atomicity of calls to its client-visible methods.

(The invariants may be temporarily violated within the methods themselves[12].)

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. DbC is a programmer's crosscutting concern. Aspects are an excellent tool for testing and enforcing contracts (See, *e.g.,* [18,19]).

With regards to derived-classes and satisfying the LSP, Meyer has started,

> 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[13].

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

Recall our previous example of how a `Square` may not be substitutable for a `Rectangle`. This is actually a postcondition violation [8]. When setting the width, for example, of a `Rectangle`, the postconditions state that the width of the `Rectangle` now must equal the width passed as an argument and the height must be unchanged. `Square` relaxes the latter condition by changing the height to match the width.

I noted before the observation in [8] that substitutability is about *behavior* not *structure.* Contracts constrain behavior to within an allowed "range", but they rarely require "behavior invariance". If they did, extension would be impossible!

Advice is effectively a derivation 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. Therefore, the AIP in more detail states the following.

- `Before` advice must support the preconditions of the advised join point or weaker preconditions and it must also support the entity's invariants.
- `After` advice must support the postconditions of the advised join point or stronger postconditions and it must support the entity's invariants. Note that this also applies to the special type of `after` advice used for exception handling clauses, because the thrown exception is also part of the postcondition contract, albeit for abnormal termination.
- `Around` advice must support the preconditions of the advised join point or weaker preconditions. It must support the postconditions of the advised join point or stronger postconditions. It must support the entity's invariants.

What about multiple extensions introduced simultaneously? A tricky issue with aspects is avoiding aspect collisions, caused by mutually incompatible advices or introductions. Two or more *superimposed* aspects [2] that are disjoint have no affect on each other. Hence, each must separately obey the AIP.

In the case of non-disjoint superimpositions, each aspect either advises join points in one or more of the other aspects, in the original entity, or both. Most aspect systems provide a precedence mechanism to eliminate arbitrary execution order. Circular dependencies among aspects are conceptually possible but often forbidden because they lead to infinite recursions at run time. At the very least, they violate the Acyclic Dependencies Principle (ADP).

The rules for non-disjoint aspects 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 LSP and the AIP, 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. To satisfy the LSP and the AIP, 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` and `after` advices.

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

3.1.3 The Introduction Invariance Principle (IIP)

> *An Introduction must conform to the invariants of the advised entity, and if used in advice, it must conform to the contract of the advice.*

This is a corollary to the AIP for introductions, which have an interesting nuance. If an introduction doesn't affect existing join points, which is usually the case, it only needs to satisfy the invariants of the modified entity[14]. 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 invariants as the advice in which it is used.

3.1.4 The Join Point Inversion Principle (JIP - DIP for Aspects)

> *(i) Join points should not depend on low-level modules. Both should depend on abstractions.*
>
> *(ii) Abstractions should not depend upon details. Details should depend upon abstractions.*

[12] However, this has implications for reentrant systems.

[13] [15], p 573. As [8] also remarks, "weaker" means that the derivation can choose not to enforce all the original preconditions. However it can add new ones.

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

As discussed in the Introduction, [1] points out that an *AOSD-Evolution Paradox* exists in today's aspects systems because aspects tend to be tightly coupled to the entities they advice, making evolution difficult. This occurs because most join point languages in use are based on structural information about the join points, such as naming conventions and package structure, rather than the logical patterns of the software. This is clearly a violation of the Dependency Inversion Principle (DIP), which says that dependencies should be based on abstractions, not concrete details.

However, note that OOD has the same issue, to a degree. You can't have a Banking application depend on an "account-like" class; the best you can do is to define an explicit dependency to an Account *abstraction*, which will hopefully change rarely. This is the practical goal of both the Open-Closed Principle (OCP) and the DIP. However, the problem of explicit dependencies in AO system is more difficult because many of them are specified as sets using regular-expression or similar mechanisms. So, a name change requires a more sophisticated analysis ability to find matches in join points.

A number of approaches are being investigated for expressing join points in a more abstract way, including logic meta programming (see *e.g.,* [18]) and logical query languages (*e.g.,* [19]). Shorter-term options exploit the new annotation features in languages like Java and C# to indicate meta-information about entities that might be useful for join point matching. However, you still have to choose stable and meaningful annotations and you have to anticipate all possible annotations of interest to potential aspects.

Until join point abstraction mechanisms mature, three pragmatic solutions are useful. The first is to write join points that refer to abstractions, such as interface details, whenever possible, and to minimize references to concrete details.

The second solution is to make the problem small enough that it is easy to solve manually. Recall the discussion of the DIP earlier where I showed how aspects support it by completely extracting concern-related dependencies as separate aspects. The interface that would otherwise be defined by the client module is packaged in an aspect along with part of the implementation that would have been provided by the serving module. This module can have undesirable couplings to two or more unstable, concrete packages, but hopefully the instability is now well localized and manageable.

The third pragmatic solution is dismissed by [1]; refactor the modules to make them easier to use with aspects. This rejection is based on a view that obliviousness is an imperative, whereas noninvasiveness is now considered more appropriate. Hence, aspect awareness of some degree is now seen as important for good design. Furthermore, aspects should be regarded as first-class design constructs along with classes and other forms.

In this context, it is interesting to consider the MDSoC perspective [5], that real world objects are inherently subjective because they belong to different concern hierarchies. Any one object will appear in different forms in different concern "dimensions". It is perhaps true that there is always a dimension in which a particular "conceptual" pointcut is easy to express succinctly and abstractly, not unlike the way that some transformations in mathematics render difficult problems into a simpler and more tractable form, without loss of information. However you approach the problem domain, the Single Responsibility Principal (SRP) tells us that it is good to refactor our designs into single concerns.

So, we can expect that good design will be a combination of established OOD principles, as discussed in [8], in combination with an awareness of the types of advice that might reasonably be applied to a particular entity. This awareness will govern the structure chosen for modules and result in software that is more maintainable and adaptable for new needs.

3.2 Aspect Subtypes

Applying the Liskov Substitution Principle (LSP) to aspects raises the question, what is a "subaspect" for an aspect? More specifically what does substitutability mean for aspects?

In general substitutability means that I can insert a subaspect for any occurrence of the original, abstraction-defining aspect and program behavior is unchanged, within the constraints of the original abstraction.

I will mostly focus on AspectJ as an example, but the arguments should be adaptable to other systems. AspectJ gives aspects class-like properties, such as the ability to define abstract aspects, with abstract pointcuts and methods that are implemented in concrete "subaspects". Abstract advice is not supported, but this is a minor limitation. Based on the OCP and LSP, the concrete implementations must respect the contracts for the abstract methods, pointcuts, and advices. This also means that the Advice Invariance Principle (AIP) must apply to concrete advice, following the same rules as superimposed advice, where the concrete advice has precedence over the abstract advice, by default (following the same model in OOSD for overridden methods).

For our purposes, aspects differ from classes in two important respects. The first is the concept of join points (and by extension, pointcuts) where advice is applied and the second is a different lifecycle model.

Because of the *adjectival* nature of aspects, they have no purpose apart from their affect on other entities[15]. Their lifecycle model reinforces this fact; they are never instantiated directly by clients, as standalone entities. Instead, they are managed by the system and most instances are actually "singletons" [9], where one instance is used for all join points it advices[16].

By existing only in relation to other entities, and recalling that the LSP states that substitutability is defined by the affect a substitution has on the larger program's behavior, the implication is that an aspect is substitutable for another aspect if the *collaboration* between the aspect and the advised entities is equivalent within the constraints of the larger program's requirements.

This then implies that abstract pointcuts need a "signature", so they can constrain concrete implementations. In AspectJ, they have no signature and hence no constraining power. That is, they offer no contract. Instead, abstract pointcuts should themselves be declared as abstractions with a clear contract.

---

[15] This statement isn't quite correct, as stated, for the MDSoC perspective.

[16] There are usually ways to instantiate aspects on a "per object" basis, *etc*.

Do pointcut contracts have any sort of *weakening* or *strengthening* behavior, like preconditions and postconditions, under derivation? This is a question that requires further study, because it seems to depend on the larger context of how the aspect-entity collaboration satisfies the program's-own contract. Also, when considering common usage today, there is no *a priori* requirement for a pointcut to include more or fewer join points under derivation. Since abstract pointcuts have zero join points in AspectJ, concrete derivations always include 0 or *more* join points!

Another implication is that the subaspects advice must be substitutable in the join points. Again, the resulting behavior must satisfy the program's contract. Note that the set of join points could be different from the original pointcut.

3.3 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 restrict behavior, while supporting extension through derivation or composition. Many application frameworks provide security mechanisms to prevent unauthorized activity, intentional or accidental. To see mainstream adoption, AO systems have to evolve beyond naïve *obliviousness* for the same reasons. *Noninvasiveness* permits access controls while maintaining the principle of not requiring module modifications.

Since advice and introductions must obey the contracts of the join points they advise, as discussed previously, the contracts must be explicit enough to constrain the behavior of potential insertions, as well as to impose access restrictions on potential join points.

Language-specific aspect systems, like AspectJ, respect the protection model of the language they extend, as long as workarounds are avoided, like AspectJ's `privileged` keyword for bypassing Java's access controls, except in very special and careful circumstances. This approach may be sufficient for access restriction contracts, which are perhaps less important than restricting the *types* of advice permitted. For example, inserting a "no-op" advice in an inappropriate join point is essentially harmless.

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 all current and future types of advice that are relevant.

Languages like Java and C# are adding annotation support that can be used to indicate meta-information about the software. The language-specific AO systems will be extended to exploit annotations as join point discriminators. Appropriate annotations might convey sufficient context information that is harder to articulate with traditional join point primitives, which tend to rely on concrete structural details.

Table 2 shows a few examples of possible contracts on advice.

#	Type of Contract	Description
1	All advice	Allow anything
2	No advice	Reject all advice insertions
3	No advice with memory usage > M	Reject all advice that consumes more than M units of memory
4	No advice with execution time > N	Reject all advice that takes more than N units of time to run
5	No advice that calls API X	Don't permit advice to call into a particular API (*e.g.,* system or I/O calls)
6	No advice that accesses data D	Prevent advice that accesses sensitive data

Table 2

Examples 3 and 4 illustrate performance ("non-functional") requirements. Examples 5 and 6 can also arise to support performance, as well as security constraints (detection of "aspect viruses"). Note that these four constraints would require a combination of compile-time and run-time analysis and enforcement.

In fact, while work remains to characterize all the types of advice that might be subject to constraints, the examples here are all enforceable through aspects written in the common aspect systems available today. So, part of a software entity's contract should be a set of its own aspects that constrain and enforce the possible "external" aspects [20]. Furthermore, *these contract aspects should always take precedence over all external aspects*.

4. Conclusions

By examining some well-known principles of good object-oriented design, I have demonstrated how aspects support them, but also what these principles tell us about good aspect-oriented design. In particular, I have discussed the role of contracts between aspects and entities as constraints on how aspects are used in order to preserve program correctness, security, *etc*., thereby supporting the principal of noninvasiveness. Along the way I examined some weaknesses in current aspect systems, including the *AOSD-Evolution Paradox*. Finally, I commented on the nature of aspect derivation.

5. ACKNOWLEDGMENTS

My thanks to my colleagues at BridgePort Networks and the contributors on "aosd-discuss" for stimulating discussions.

6. REFERENCES

[1] 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).

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

[3] Katara, M. and Katz, S. Architectural Views of Aspects. *Proceedings of AOSD 2003* (Boston, Massachusetts, USA, March 17-21, 2003). ACM Press, New York, NY, 2003, 1-10.

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

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

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

[10] 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.

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

[12] Fowler, M. *Refactoring.* Addison-Wesley, Reading, MA, 2000.

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

[14] Jacobson, I. Use Cases as Aspects. *Invited talk, AOSD 2003* (Boston, Massachusetts, USA, March 17-21, 2003).

[15] J2EE Technology. http://java.sun.com/j2ee/.

[16] Johnson, R. and Hoeller, J. *J2EE Development without EJB.* Wiley, Indianapolis, IN, 2004.

[17] JavaBeans Technology. http://java.sun.com/products/javabeans/.

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

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

[20] 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.

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

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

TLA*	Name	Definition	AOD Perspective
SRP	The Single Responsibility Principle	A class should have only one reason for change.	Another way of stating the problem of tangled concerns, which aspects help solve. Aspects should also obey the SRP ("A class *or aspect*….").
OCP	The Open-Closed Principle	Software entities (classes, modules, functions, *etc.*) should be open for extension, but closed for modification.	The entities should be closed for *source* modification. Aspects can modify the source in a controlled way, but must object the join-points' *contracts*.
LSP	The Liskov Substitution Principle	Subtypes must be substitutable for their base types.	Factoring out concerns reduces the likelihood of LSP violations that result when a client expects a certain behavior that wasn't anticipated by the module designer. The offending concern can be replaced with a concern that meets client expectations, thereby restoring substitutability of the hierarchy in the client's domain.
DIP	The Dependency Inversion Principle	(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.	For dependencies that are concerns not related to the domain logic, extraction into aspects localizes the coupling to the aspects themselves, making management of the dependency more tractable and enhancing reuse of the original, decoupled modules.
ISP	The Interface Segregation Principle	Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies.	Same as for the DIP. Extraction of concerns from clients further decouples them from services and also localizes the "client" interface" in the aspect and the code that uses the service.
REP	The Release-Reuse Equivalency Principle	The granule of reuse is the granule of release.	Aspects tend to yield smaller, less broadly-coupled packages. However, aspects that are closely coupled to packages may need to be part of the release granule.
CRP	The Common Reuse Principle	The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.	Similar to the REP, aspects promote the "SRP for packages", but also require careful packaging due to dependencies on the packages.
CCP	The Common Closure Principle	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.	Aspects promote having packages with one concern. An AOSD system will tend to have more, smaller, well-focused packages.
ADP	The Acyclic Dependencies Principle	Allow no cycles in the package dependency graph.	Aspects are one tool for breaking cycles. Aspect packages should also be acyclic.
SDP	The Stable Dependencies Principle	Depend in the direction of stability.	Aspects can localize dependencies and provide stable abstractions. Aspects should also depend only on abstractions.
SAP	The Stable Abstractions Principle	A package should be as abstract as it is stable.	Aspects should try to obey the SAP, but current join point languages are too concrete.

*The three-letter acronyms, names and definitions are quoted from [8].

Title 1: Object-Oriented Design Principles and Aspects