AOP@Work: Enhance design patterns with AspectJ, Part 2
Enable transparent composition and code-level reuse of complex patterns

Nicholas Lesiecki
May 17, 2005

Nicholas Lesiecki continues his discussion of the benefits of implementing design patterns with aspect-oriented techniques with this in-depth study of the Observer pattern. In this article of the AOP@Work series, he illustrates how AspectJ allows complex patterns to be converted into reusable base aspects, thus enabling framework authors to supply prebuilt libraries of patterns for developers to exploit.

View more content in this series

In Part 1 of this article, I examined two widely used OO design patterns from an aspect-oriented perspective. After demonstrating how the Adapter and Decorator patterns could be implemented in both a Java™ system and an AspectJ system, I considered the effects of each implementation in terms of code comprehension, reuse, maintainability, and ease of composition. In both cases, I found that a pattern that cuts across the modularity of the Java implementation could be drawn together into a single aspect in the AspectJ implementation. This co-location of logically related code made the pattern easier to understand, change, and apply. Looking at patterns in this light turns a frequent criticism of AOP -- that it will prevent developers from understanding the behavior of code by reading it -- on its head. In this second half of the article, I complete my comparison of Java language and AspectJ pattern implementations, with an in-depth look at the Observer pattern.

I've chosen to focus on the Observer pattern because it is the queen spider of the OO design patterns. It is widely used (especially in GUI applications) and forms a critical part of the MVC architecture. It tackles a complex problem and solves it relatively well. However, it also poses serious challenges in terms of implementation effort and code comprehensibility. Unlike the Decorator or Adapter patterns whose participants are mainly new classes created for the pattern, the Observer pattern requires that you invade existing classes in your system to support the pattern -- at least in the Java language.
Aspects can reduce the burden of invasive patterns like Observer, making pattern participants more flexible because they do not contain pattern code. Furthermore, the pattern itself can become an abstract base aspect letting developers reuse it by importing and applying it, rather than having to rethink the pattern every time. To see how these possibilities play out, I continue the format set by the first part of the article. I begin with an example problem and provide a general description of the Observer pattern. Then, I describe how to implement Observer in both the AspectJ and Java languages. After each implementation, I discuss what makes the pattern crosscutting and what effects that version of the pattern has on understanding, maintaining, reusing, and composing the code.

Click the Code icon at the top or bottom of this page to download the article source before continuing.

---

**About this series**

The AOP@Work series is intended for developers who have some background in aspect-oriented programming and want to expand or deepen what they know. As with most developerWorks articles, the series is highly practical: you can expect to come away from every article with new knowledge that you can put immediately to use.

Each of the authors contributing to the series has been selected for his leadership or expertise in aspect-oriented programming. Many of the authors are contributors to the projects or tools covered in the series. Each article is subjected to a peer review to ensure the fairness and accuracy of the views expressed.

Please contact the authors individually with comments or questions about their articles. To comment on the series as a whole, you may contact series lead Nicholas Lesiecki. See Resources for more background on AOP.

---

**The Observer pattern**

According to the Portland Pattern Repository Wiki (see the Resources section for details), the intent of Observer is to:

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."

This makes Observer appropriate for all sorts of notification needs. Consider the following:

- A bar chart can observe data-objects that it displays so as to redraw when those objects change.
- AccountManager objects can observe Accounts so that they can send an e-mail to a salesperson when the account changes its status.
- A billing service can observe song-play events in an online music store so as to bill the customer.

I'm going to focus on the last scenario as the example for this article. (The idea for the problem domain was borrowed from "Implementing Observer in .NET"; see the Resources section for details.) Suppose you're tasked with adding some features to an online music store. The store already features the ability to play songs individually as well as to assemble them into online Playlists. Customers can also view the lyrics for a given song. You need to add billing and
statistics. First, the system should track play and lyric-show events so as to charge the customer appropriately. Second, the system should update a list of the most frequently played songs for display in a "most popular" section. Listing 1 contains code for the core objects that already exist in the system:

**Listing 1. Core domain objects in the online music service**

```java
//common interface for items that
//can be played
public interface Playable { 
    String getName(); 
    void play(); 
}

public class Song implements Playable {
    private String name;
    public Song(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void play() {
        System.out.println("Playing song " + getName());
    }
    public void showLyrics(){
        System.out.println("Displaying lyrics for " + getName());
    }
}

public class Playlist implements Playable {
    private String name;
    private List songs = new ArrayList();
    public Playlist(String name) {
        this.name = name;
    }
    public void setSongs(List songs) {
        this.songs = songs;
    }
    public void play() {
        System.out.println("playing album " + getName());
        for (Song song : songs) {
            song.play();
        }
    }
    public String getName() {
        return name;
    }
}

public class BillingService{
    public void generateChargeFor(Playable playable) {
        System.out.println("generating charge for : " + playable.getName());
    }
}
```
Now that you've seen the core system, let's consider a pre-AOP implementation of Observer.

**A Java language Observer**

Although the implementations will differ markedly, there are some similarities between them. Regardless of how you implement Observer, you'll have to answer the following four questions in code:

- Which objects are Subjects and which are Observers?
- When should the Subject send a notification to its Observers?
- What should the Observer do when notified?
- When should the observing relationship begin and end?

I'll walk you through an OO implementation of the Observer pattern, using these questions as a framework.

**Role definition**

You begin by assigning roles through marker interfaces. The **Observer** interface defines a single method, `update()`, which corresponds to the action taken when a **Subject** sends a notification. A **Subject** takes on more responsibility; its marker interface defines two methods for tracking observers and one for notifying those observers of an event:

```java
public interface Subject {
    public void addObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Once you've defined these roles, you apply them to the appropriate players in the system.

**Applying the Observer role**

You can modify the **BillingService** to implement the Observer interface with a small amount of code:

```java
public class BillingService implements Observer {
    //...
    public void update(Subject subject) {
        generateChargeFor((Playable) subject);
    }
}
```

**Tracking and notifying observers**

Once that's done, you can move on to the two **Subjects**. Here are the modifications to **Song**:

```java
public class SongPlayCounter {
    public void incrementPlays(Song s) {
        System.out.println("Incrementing plays for " + s.getName());
    }
}
```
private Set observers = new HashSet();

public void addObserver(Observer o) {
    observers.add(o);
}

public void removeObserver(Observer o) {
    observers.remove(o);
}

public void notifyObservers() {
    for (Observer o : observers) {
        o.update(this);
    }
}

You now face the unpleasant task of cutting and pasting this implementation of Subject into Playlist as well. Factoring some of the implementation of Subject out into a helper class should help you stomach the design smell, but it won't completely eliminate it.

**Trigger events**

Now you've adapted your classes to their roles in the pattern. However, you still need to go back and fire notifications when the appropriate events occur. Song requires two additions and Playlist requires one:

```java
//...in Song
public void play() {
    System.out.println("Playing song " + getName());
    notifyObservers();
}

public void showLyrics(){
    System.out.println("Displaying lyrics for " + getName());
    notifyObservers();
}

//...in Playlist
public void play() {
    System.out.println("playing album " + getName());
    for (Song song : songs) {
        //...
    }
    notifyObservers();
}
```

Keep in mind that although the example system requires three state-change notifications, a real system might require many more. For example, I once worked on an application that required Observer-style notifications of changes to the state of a shopping cart. To make it happen, I applied notification logic to more than 10 locations in the cart and related objects.

With the players ready to participate in the pattern, all that's left to do is connect them.

**Starting the observation relationship**

For a BillingService to begin watching a Song or a Playlist, you need to add glue code that will call song.addObserver(billingService). This requirement is similar to the glue code requirements of Adapter and Decorator described in Part 1. In addition to affecting the players, the pattern
requires you to modify an unspecified part of your system to activate the pattern. Listing 2 contains code that simulates a client's interaction with the system and incorporates this glue code, which is highlighted. The listing also shows sample system output:

Listing 2. Client code that connects Subjects and Observers (as well as exercising the system)

```java
public class ObserverClient {
    public static void main(String[] args) {
        BillingService basicBilling = new BillingService();
        BillingService premiumBilling = new BillingService();

        Song song = new Song("Kris Kringle Was A Cat Thief");
        song.addObserver(basicBilling);

        Song song2 = new Song("Rock n Roll McDonald's");
        song2.addObserver(basicBilling);
        //this song is billed by two services,
        //perhaps the label demands an premium for online access?
        song2.addObserver(premiumBilling);

        Playlist favorites = new Playlist("Wesley Willis - Greatest Hits");
        favorites.addObserver(basicBilling);

        List songs = new ArrayList();
        songs.add(song);
        songs.add(song2);
        favorites.setSongs(songs);

        favorites.play();
        song.showLyrics();
    }
}
```

//Output:
playing playlist Favorites
Playing song Kris Kringle Was A Cat Thief
generating charge for: Kris Kringle Was A Cat Thief
Playing song Rock n Roll McDonald's
generating charge for: Rock n Roll McDonald's
generating charge for: Rock n Roll McDonald's
generating charge for: Favorites
Displaying lyrics for Kris Kringle Was A Cat Thief
generating charge for: Kris Kringle Was A Cat Thief

Analysis of the Java language Observer

As you can probably sense from the implementation steps, Observer is a heavyweight pattern. This analysis explores some of the current and potential impacts of the pattern on the system:

Is a better OO Observer possible?

This article focuses on the Java language for its OO pattern implementations. You may wonder whether more feature-rich OO languages could deal better with some of the headaches induced by Observer's roles. The .NET implementation from which this article draws inspiration uses delegates and events to drastically reduce the burden on its version of the Song and Playlist classes. (To compare the implementations, see the Resources section.) Similarly, an OO language that supports multiple inheritance could limit the burden on Song to mixing in a Subject support class. Either technique helps quite a lot, but neither
Crosscutting: In the Music Store application, the billing observation concern involves both the billees (Song and Playlist) and the biller (the BillingService). The Java language implementation also drags a glue-code location along to the party. Because it affects three distinct classes that have a purpose other than this observation relationship, billing observation represents a crosscutting concern.

Understanding: You can begin by looking at the system from the point of view of the domain objects. The OO implementation of Observer requires modification to the domain classes affected by the pattern. As you saw in the implementation classes, these modifications were not trivial. In the case of Subjects, you needed to add several methods to each class for it to play its role in the pattern. What were once simple abstractions of domain concepts (playlists, songs) become choked with the machinery of the Observer pattern. (Refer to "Is a better OO Observer possible?" for why some non-Java languages may make this easier.) You can also examine the system from the point of view of the pattern. In the Java language implementation, the pattern runs the risk of "disappearing into the code." You can't meaningfully construct a mental model of the pattern without examining all three facets of the pattern: the Observer(s), the Subject(s), and the client code that connects the two. Even a savvy developer, spotting one part of the pattern, will go looking for the others, but there's no "BillingObserver" module that brings the pieces together for you.

Reusing: To reuse this pattern, you must reimplement it from scratch. Usually you can make use of prebuilt support classes (java.util contains Observer and Observable for instance) that fill out some parts of the pattern for you, but the bulk of the labor remains for you to do.

Maintaining: To help you think through the maintenance costs of Observer, consider the need to avoid double billing. If you glance back at the output generated by the first implementation of the pattern (see Listing 2), you'll see the application bills songs played within the context of a playlist. This shouldn't surprise you because that's exactly what you wrote the code to do. The marketing department, however, would like to provide aggregate pricing for playlists to encourage bulk purchases. In other words, they want only the playlist billed, not the songs within it.

You can easily imagine the trouble this will cause for the traditional implementation of Observer. You could modify each play() method to accept a parameter indicating that it had been called by another play() method. Alternatively, you could maintain a ThreadLocal to track the information. In either case, you would need to conditionally call notifyObservers() only when the invocation context warranted. These modifications would add weight and complexity to already beleaguered pattern participants. Because the change affects multiple files, bugs could slip in during the redesign.

Composing: Another scenario you should consider is observation by distinct observers. As you might recall from the setup, the music service should track the most popular songs. However, statistics gathering applies only to song plays, not to lyric viewings or playlist plays. This means that the SongCountObserver watches a slightly different set of operations from the billing observer.

To accomplish this, you would have to modify Song to maintain a separate list of Observers that were only interested in play() notifications (not in lyric operations). Then, the play()
method would raise this event independently of the billing event. The OO pattern has thus far guarded against direct dependency of the concrete subjects on the concrete observers, but it seems impossible to avoid in this case since song must raise discrete events for each type of observer. Faced with a second type of observer, the Java language implementation simply breaks down.

As you might expect, the AspectJ Observer offers elegant solutions to the scenarios posed in this analysis. Before I examine them, I'll walk you through how AspectJ handles the base pattern.

The AOP design patterns project

My fascination with design patterns and AspectJ began with the Aspect-Oriented Design Pattern Implementation project (see Resources) sponsored by the University of British Columbia. As I mentioned in the first part of this article Jan Hanneman and Gregor Kiczales investigated the 23 GoF patterns and created reference implementations of each in both the Java language and in AspectJ. The results are fascinating to study, with 17 of the original patterns improving when implemented with AOP. Even more exciting is the Pattern Library. The project authors were able to extract reusable abstract aspects from 13 of the patterns and have made them freely available under the Mozilla Public License. The ObserverProtocol aspect shown in this article was taken from that project. See the Resources section to download the patterns.

An AspectJ Observer

As with the other patterns I've considered in this article, the intent and even basic structure of Observer remains the same when implemented in AspectJ. However, there is a critical difference. By using aspect-oriented inheritance, you can reuse those portions of the pattern that are common to all implementations while customizing the pattern to suit your needs. Keep in mind the four questions that an Observer implementation must answer:

- Which objects are Subjects and which are Observers?
- When should the Subject send a notification to its Observers?
- What should the Observer do when notified?
- When should the observing relationship begin and end?

The abstract pattern aspect

Much like an abstract class, you must extend an abstract aspect before putting it to use. The abstract aspect can define abstract pointcuts (pointcuts with a name but without a body) and then define concrete advice on those pointcuts. This means that the super-aspect specifies the behavior and the subaspect controls where that behavior applies. Additionally, abstract aspects can define abstract methods that a subaspect must override, just as an abstract class can. With these facts in mind, take a look at the ObserverProtocol aspect from The AOP design patterns project.

Role definition

Just as with the other implementation, roles are denoted by marker interfaces. However, in the case of the AspectJ implementation, the interfaces are empty (and could probably be replaced by annotations in AspectJ 5). The interfaces appear as members of ObserverProtocol:
/**
 * This interface is used by extending aspects to say what types
 * can be Subjects.
 */
public interface Subject{

/**
 * This interface is used by extending aspects to say what types
 * can be Observers.
 */
public interface Observer{}

Tracking observers

Rather than forcing the participants to track observers, the aspect centralizes this responsibility. Listing 3 contains the code that implements this part of the pattern. The general idiom should be familiar to you from the AOP implementation of Decorator in Part 1. Again, the aspect uses a lazily initialized Map to track state specific to an object. (And again, there are alternatives to this pattern that involve instantiation models, but they're a little beyond the scope of this article.) Notice one interesting element of Listing 3 -- the addObserver and removeObserver methods are public. This means that code elsewhere in the system has the option of programatically determining participation in the pattern:

Listing 3. ObserverProtocol manages the Observers watching a Subject

/**
 * Stores the mapping between Subjects and
 * Observers. For each Subject, a LinkedList
 * is of its Observers is stored.
 */
private WeakHashMap perSubjectObservers;

/**
 * Returns a Collection of the Observers of
 * a particular subject. Used internally.
 */
protected List getObservers(Subject subject) {
    if (perSubjectObservers == null) {
        perSubjectObservers = new WeakHashMap();
    }
    List observers = (List)perSubjectObservers.get(subject);
    if (observers == null) {
        observers = new LinkedList();
        perSubjectObservers.put(subject, observers);
    }
    return observers;
}

/**
 * Adds an Observer to a Subject.
 */
public void addObserver(Subject subject, Observer observer) {
    getObservers(subject).add(observer);
}

/**
 * Removes an observer from a Subject.
 */
public void removeObserver(Subject subject, Observer observer) {
    getObservers(subject).remove(observer);
Notifying observers

To implement the actual update of the observers, ObserverProtocol uses a loop very similar to the one you employed in the Java language implementation. However, it invokes the notification very differently. First the aspect defines an abstract pointcut:

```java
protected abstract pointcut subjectChange(Subject s);
```

This pointcut exists to be concretized by subaspects. ObserverProtocol goes on to define a piece of advice that responds to join points selected by `subjectChange()`:

```java
after(Subject subject) returning : subjectChange(subject) {
    for (Observer observer : getObservers(subject)) {
        updateObserver(subject, observer);
    }
}
```

```java
protected abstract void updateObserver(Subject subject, Observer observer);
```

For each `Observer` of the changed subject, the aspect calls `updateObserver()`, an abstract method that it also defines. This way, concrete aspects can define what it means to receive an update.

The concrete subsaspect

At this point, you've seen all that ObserverProtocol has to offer you. To apply the pattern, you need to extend the aspect and provide specific answers to the four central Observer questions posed at the beginning of this discussion. Once you have downloaded ObserverProtocol, you can begin by creating your aspect and declaring that it extends ObserverProtocol. As you might expect, the AspectJ compiler helpfully informs you that you have not completed the aspect definition. You must concretize the `subjectChange` pointcut and implement the `updateObserver` method. These reminders serve to guide you in applying the pattern to your system.

Role assignment

With AspectJ you can assign roles to the pattern participants without modifying the `Song`, `Playlist`, and `BillingService` classes directly, as shown here:

```java
declare parents : Playable extends Subject;
declare parents : BillingService implements Observer;
```

The compiler does not compel you to insert these declarations, but you would find it impossible to use the rest of the aspect's machinery without them, since ObserverProtocol defines its pointcuts and methods in terms of these marker interfaces.

Trigger events

In perhaps the most important part of the aspect, you concretize the `subjectChange` pointcut to define those operations that constitute a notification-worthy event. In this case, the pointcut
specifies the execution of the `play()` method on any class that implements the `Playable` interface (both `Song` and `Playlist` do so). The pointcut also selects the execution of the `showLyrics()` method on `Song`:

```java
pointcut titleUse(Playable playable) :
    this(playable)
    && (execution(public void Playable+.play()) ||
        execution(public void Song.showLyrics()))
;
public pointcut subjectChange(Subject subject) :
    titleUse(Playable) && this(subject);
```

### Responding to updates

Much like the Java language version, to respond to an update, you implement an update method. The difference is that here you are adding the method to the aspect, not to the `BillingService` directly, which remains ignorant of its participation in the pattern:

```java
public void updateObserver(Subject s, Observer o){
    BillingService service = (BillingService)o;
    service.generateChargeFor((Playable)s);
}
```

### Starting the observation relationship

As I mentioned earlier, the presence of the public `addObserver` and `removeObserver` on the aspect make it possible to configure the aspect programatically. To start the observation of songs and playlists, you could duplicate the Java language implementation of the client and replace each occurrence of `song.addObserver(basicBilling)` with `ObserverBillingPolicy.aspectOf().addObserver(song, basicBilling)`.

However, to keep the pattern's presence as unobtrusive as possible, it makes sense to begin the relationship automatically, using advice, as shown here:

```java
private BillingService defaultBillingService =
    new BillingService();

pointcut playableCreation(Subject s) :
    execution(public Playable+.new(..))
    && this(s);

after(Subject s) returning : playableCreation(s){
    addObserver(s, defaultBillingService);
}
```

This pointcut and advice begin the billing observation immediately on the creation of a `Playable`, using the default `BillingService` configured in the aspect. However, the `ObserverProtocol` allows for flexibility here. You can imagine further advice that applies supplemental billing for certain songs, or that retrieves the billing plan associated with the current user of the system. Similarly,
advice could be used to automatically discontinue the relationship once the song becomes ineligible for further billing.

**Analysis of AspectJ Observer**

Some of the design observations I'm about to make should sound familiar since they echo those made about the previous two patterns discussed in this article (see Resources). However, the design contrasts are particularly sharp in Observer, as you'll note:

- **Understanding**: From the point of view of the participants, the AOP version of Observer is simpler because it requires nothing special of Song, Playlist, or BillingService. Removing this dependency from the participants makes them easier to understand in terms of their primary abstraction as well as making them more flexible and reusable. For instance, they can be reemployed in other systems that do not require the Observer pattern. They can also participate in other patterns without the fear that a mass of pattern code will overwhelm the core responsibilities of the classes.

  From the point of view of the pattern, the AOP version is, again, simpler. Rather than relying on you to assemble the pattern fragments in your head, the AspectJ implementation offers an observerProtocol aspect that lays out the general structure of the pattern. The concrete aspect defines how the pattern applies to this system. Both modules are tangible, and from the point of view of the pattern, complete. So, if you are using AJDT and notice that, say the play() method is advised by an observer aspect, you can follow the link that AJDT provides you and understand the complete picture. (The Resources section includes a link to a great article on how AJDT can assist you in navigating and comprehending AspectJ code.)

  The generation of notification "events" modularizes particularly well in the AspectJ implementation. In the traditional implementation, these calls exist in three separate places. No direct clue exists as to why certain operations would raise events or which other operations might also do so. In the AOP pattern, a named pointcut communicates the common property of the operations (the use of a title). The specification of the pointcut indicates which other join points the pattern affects.

- **Reusing**: The potential for reusing the observerProtocol aspect is high. In fact, reusing pattern aspects is half the reason I wrote this article. As a developer, any time I can rely on the expertise of a framework author rather than rolling my own implementation, I'm happy. Aspects provide the ability to reuse crosscutting code, code that before was so entwined with the details of its implementation that it had no independent existence. You'll see a good example of reuse in the Composing section of the analysis.

- **Maintaining**: To see a concrete example of how aspects make this system easier to evolve, hark back to the difficulty of excluding songs from double billing in the Java version of the system. In order to fulfill this requirement in AspectJ, you need only edit the subjectChange() pointcut. The following code makes use of the cflowbelow() pointcut to exclude play() executions that occur within the control flow of another play() execution:
This modification is trivial, particularly because it only affects one file. Furthermore, it makes the intent of the new policy clear. Other modifications such as adding new Subjects or subject change operations require similarly straightforward changes to the aspect, not coordinated changes to multiple files.

- **Composing**: Recall that the Java-language implementation of Observer also had problems coping with a second instance of the pattern applied to (some of) the same participants. To implement this requirement in AspectJ, you simply extend the abstract aspect a second time with new definitions for the pointcuts and abstract methods. Since each aspect manages its own list of observers and subjects, and each aspect also defines its own subjectChange pointcut, the two aspects do not collide. Each operates essentially independently of the other. You can examine the code for SongCountObserver in Listing 4:

### Listing 4. A second Observer aspect affects the same participants

```java
public aspect SongCountObserver extends ObserverProtocol {
    declare parents : Song extends Subject;
    declare parents : SongPlayCounter implements Observer;

    pointcut titleUse(Song song) :
        this(song)
        && execution(public void Song.play());

    public pointcut subjectChange(Subject subject) :
        titleUse(Song) && this(subject);

    public void updateObserver(Subject s, Observer o) {
        SongPlayCounter counter = (SongPlayCounter) o;
        counter.incrementPlays((Song) s);
    }

    // could be injected by a dependency injection
    // framework like Spring
    private SongPlayCounter defaultCounter = new SongPlayCounter();

    pointcut songCreation(Subject s) :
        execution(public Song.new(..))
        && this(s);

    after(Subject s) returning : songCreation(s){
        // different songs could be tracked in different statistical sets...
        addObserver(s, defaultCounter);
    }
}
```
Because in an AspectJ system multiple patterns (and even instances of the same pattern) can compose transparently, AspectJ systems avoid some of the problems that come from "pattern density." This can enable more use of the best practices that design patterns embody without the fear that the weight of the implementation will overwhelm the code. (Stay tuned for Wes Isberg's upcoming article in the AOP@Work series for more on using aspect-orientation to avoid pattern density in OO design patterns.)

**Conclusion**

At their core, design patterns are really design problems. Because programmers are clever, these problems have been solved many times. Because programmers are lazy (in a good way!) they would rather not repeat the effort of solving these design problems again and again. The contribution of the GoF book (and previous pattern work) was to surface general solutions to certain problems that had no better expression in the languages of the day. The existence of these cumbersome (if invaluable) solutions could almost be interpreted as a challenge, a challenge to find better (more understandable, reusable, maintainable, composable) means of solving the underlying problems.

Some OO patterns are difficult to use because they address problems that are crosscutting. For these problems, AOP offers solutions that go right to the heart of the issue. By gathering together the crosscutting behavior into a single module, the behavior becomes easier to understand, modify, and reuse. Because the participants in this type of aspect-oriented pattern do not depend on the pattern code, they also become more flexible and reusable. In this article, I've pointed out practical examples of scenarios and modifications that are much easier when pattern-problems are solved in an aspect-oriented way.

The new solutions take various forms. AOP can radically simplify some patterns (such as Decorator) to the point where they scarcely deserve pattern status. Other patterns (such as Observer) take on new life as reusable library modules. One day, you may find it as natural to extend and implement a pattern aspect as it is to use a class from the Collections framework today.

**Are there patterns specific to AOP?**

An intriguing area of further study is design patterns that solve recurring problems within aspect-oriented languages. You can see a hint of this in this article, where two aspects used a lazily initialized map to loosely associate state with an object. In addition to this, AOP-specific patterns (or candidate patterns) are beginning to emerge within the user community. Ramnivas Laddad describes several such patterns, such as Worker Object and Wormhole, in his book *AspectJ in Action* (see Resources). Stefan Hanenberg and Arno Schmidmeier have proposed several interesting candidates such as "Template Advice" and "Pointcut Method" in their paper "Idioms for Building Software Frameworks in AspectJ." Only time will tell whether these emerging patterns are useful and straightforward idioms or "workarounds for things that are broken in AspectJ." For example, at this year’s AOSD conference, I attended a session that discussed the possibility of
better language support for the "lazily initialized map" idiom. An examination of the patterns of today may very well drive the language features of tomorrow.

Acknowledgments

This article owes much to Wes Isberg, Mik Kersten, Ron Bodkin, and Ramnivas Laddad who reviewed earlier drafts and provided helpful insights and corrections.
Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-aopwork56code.zip</td>
<td>142 KB</td>
</tr>
</tbody>
</table>

© Copyright IBM Corporation 2005

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)