

Real World Aspect Oriented Programming with Java



AspectJ Cookbook™

O'REILLY®

Russ Miles

Implementing Creational Object-Oriented Design Patterns

17.0 Introduction

With the release of the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley), the now infamous Gang of Four (GoF) design patterns became formally recognized as a useful practice in object-oriented (OO) software development.

The GoF book was split into three categories of design pattern: Creational, Structural, and Behavioral. The original GoF design patterns were designed to be implemented using the mechanisms available in most OO languages. Aspect orientation, when implemented using AspectJ, adds new mechanisms with which these patterns can be applied:

- The code that deals with the mechanics of a design pattern can be modularized out of the rest of the business logic so as to be less intrusive.
- In Java, inheritance relationships between classes must be used with care since Java allows only one inheritance relationship between two classes. Aspects can provide mechanisms by which the more generic pattern-oriented relationships can be applied separately from any core business relationships between classes. Therefore, aspects can remove the need for an abstract base class leaving the classes with as much freedom as possible to define the right relationships for their business logic without the design pattern getting in the way.
- A by-product of the above two advantages is that code is clearer and easier to understand because of the removal of pattern-focused relationships and logic from the business logic classes thanks to the modularization of the patterns into aspects.

All three categories of design pattern from the GoF book can benefit from an aspect-oriented implementation and this chapter focuses on the Creational patterns.



The pattern implementations described in the next three chapters are based on Jan Hannemann and Gregor Kiczales's work for the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA). Go to <http://www.cs.ubc.ca/~jan/AODPs/> to check out the original research.

17.1 Implementing the Singleton Pattern

Problem

You want to apply the singleton pattern using AspectJ.

Solution

The *Singleton* pattern allows the definition of a class as having one runtime instance within an application. A singleton is normally met by providing no default constructor for a particular class, often overridden as a protected constructor, so an object of the class cannot be directly instantiated by the application. Access to the singleton object is usually implemented by creating a static method that returns the single instance of the class.

Example 17-1 shows an abstract aspect that uses the Director aspect-oriented design pattern (see Chapter 23) to provide a generic foundation for applying the singleton pattern.

Example 17-1. Using an aspect to define the Singleton Pattern

```
public abstract aspect SingletonPattern issingleton()
{
    private Hashtable singletons = new Hashtable();

    public interface Singleton
    {
    }

    public interface NonSingleton
    {
    }

    // Pointcut to define specify an interest in all creations
    // of all Classes that extend Singleton
    pointcut selectSingletons() : call((Singleton +).new (..));

    // Pointcut to ensure that any classes in the Singleton inheritance tree
    // that are marked as Non Singletons are not included in the Singleton
    // logic.
    pointcut excludeNonSingletons() : !call((NonSingleton +).new (..));
}
```

Example 17-1. Using an aspect to define the Singleton Pattern (continued)

```
Object around() : selectSingletons() && excludeNonSingletons()
{
    Class singleton = thisJoinPoint.getSignature().getDeclaringType();

    synchronized(singletons)
    {
        if (singletons.get(singleton) == null)
        {
            singletons.put(singleton, proceed());
        }
    }

    return (Object) singletons.get(singleton);
}
}
```

Discussion

The SingletonPattern abstract aspect defines two roles: Singleton and NonSingleton. The roles are implemented as interfaces so that the abstract aspect can work with the singletons without worrying about implementation details.

Figure 17-1 shows the structure of the SingletonPattern abstract aspect with the interfaces and behavior that it defines to support the singleton design pattern.

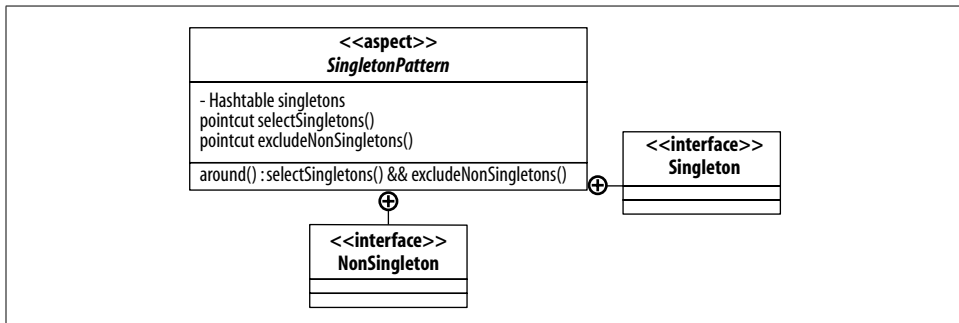


Figure 17-1. The structure of the SingletonPattern abstract aspect

The Singleton interface is applied by subspects of the abstract SingletonPattern aspect to any classes within the target application that are to be treated as singletons. Similarly, the NonSingleton interface is applied to classes that may pick up singleton behavior from their parent class by inheritance. If you decide the child class is not to be a singleton, then the NonSingleton interface can be employed so that the singleton characteristic of the parent is overridden.

Two pointcuts are declared to capture when classes that have the Singleton interface are instantiated. The selectSingletons() pointcut definition picks the calls to the constructors on classes that extend the Singleton interface. To support the need to turn off

the singleton behavior for subclasses of singletons, the `excludeNonSingletons()` pointcut is declared. This pointcut can be overridden by the specific aspects when you need to stop a subclass from being affected by a superclass's singleton behavior.

The `around()` advice captures calls to constructors on classes that have had the Singleton interface applied. The `around()` advice overrides the constructor to check that the type of the object being instantiated has not been created.

A lookup of the object's class being created is performed on the singletons hash table using the class information supplied by the `thisJoinPoint` variable. If the type of class is not present within the hash table, then its class is added and an object of that class is constructed by calling `proceed()`, which executes the original constructors logic. The `proceed()` call returns the constructed object and this is added with the class object to the hash table.

If the type of the class is present within the hash table, then no new objects need to be created. The singleton object is retrieved from the hash map according to its class and returned from the `around()` of advice as the result of the constructor call.

By default, an aspect in AspectJ is a singleton. The `SingletonPattern` aspect uses the explicit `issingleton()` instantiation policy to highlight this aspect's important behavioral characteristic.



The implementation of the singleton pattern in this recipe is thread-safe because the hash table is locked during the execution of the `around()` advice using a `synchronize` block for simplicity. This incurs a performance penalty when the singleton is accessed for the first time and created, but it means that the pattern can then be applied within a multithreaded application. A more efficient mechanism could be to use an instance of the `ThreadLocal` class as a variable inside the aspect to ensure that the locked check is only performed once for a single thread, as discussed in the article by Brian Goetz, available at <http://www.javaworld.com/javaworld/jw-11-2001/jw-1116-dcl.html>.

Example 17-2 shows how the abstract `SingletonPattern` aspect can be applied for a specific application.

Example 17-2. Applying the abstract `SingletonPattern` aspect to target application classes

```
public aspect PrinterSingleton extends SingletonPattern
{
    declare parents: Printer implements Singleton;

    declare parents: SpecializedPrinter implements NonSingleton;
}
```

Figure 17-2 shows how the PrinterSingleton aspect affects an applications classes.

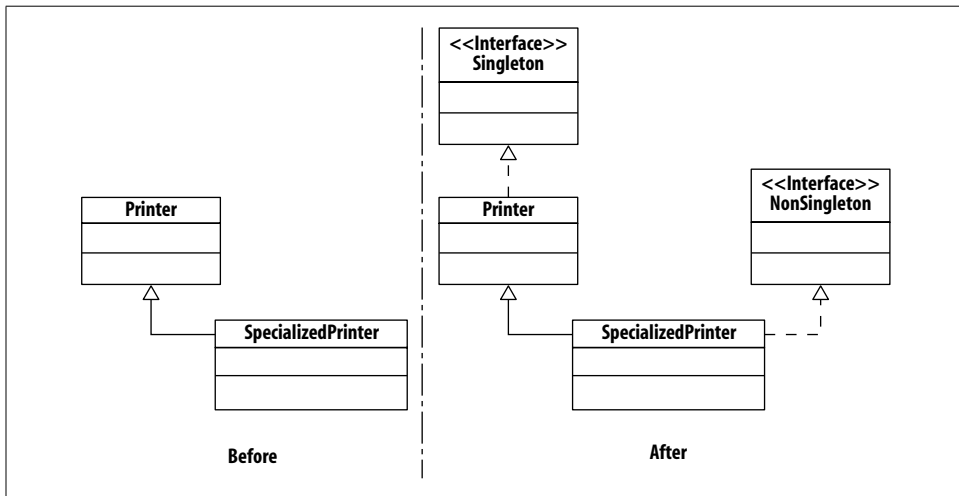


Figure 17-2. How the static application structure is affected by the PrinterSingleton aspect

Figure 17-3 shows how the new singleton behavior of the Printer class behaves in an example application.

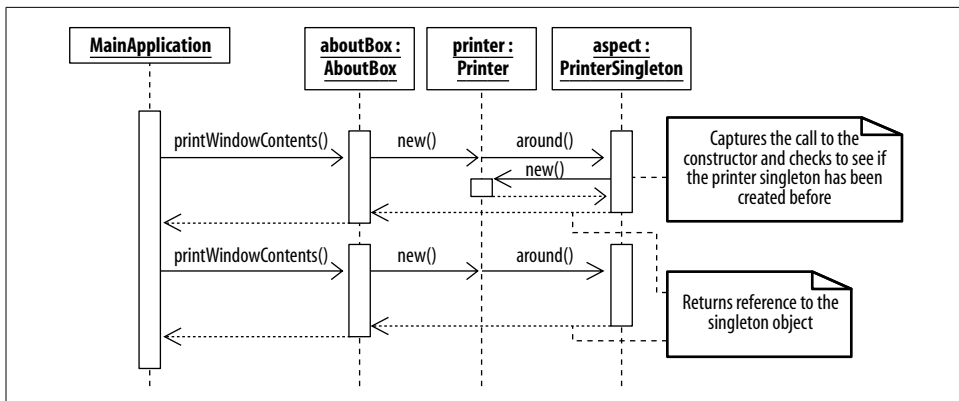


Figure 17-3. Using the new singleton behavior of the Printer class

See Also

Recipe 13.4 provides more detail on the `proceed()` call and its usage within `around()` advice; Recipe 7.1 describes the `call(Signature)` pointcut when capturing calls to a classes constructor; more information on aspect instantiation policies is available in Chapter 14; the Director aspect-oriented design pattern is explained in Recipe 23.3.

17.2 Implementing the Prototype Pattern

Problem

You want to apply the prototype pattern using AspectJ.

Solution

The *prototype* pattern supports the creation of duplicate objects based on an original object, the prototype.

Example 17-3 uses the Director aspect-oriented design pattern (see Chapter 23) to define the generic behavior needed to apply this pattern.

Example 17-3. Using an aspect to define the prototype pattern

```
public abstract aspect PrototypePattern
{
    protected interface Prototype
    {
    }

    public Object Prototype.clone() throws CloneNotSupportedException
    {
        return super.clone();
    }

    public Object cloneObject(Prototype object)
    {
        try
        {
            return object.clone();
        }
        catch (CloneNotSupportedException ex)
        {
            return createCloneFor(object);
        }
    }

    protected Object createCloneFor(Prototype object)
    {
        return null;
    }
}
```

Discussion

The abstract `PrototypePattern` aspect defines the `Prototype` interface that can be applied to any class within the target application that is to be a prototype. Those classes are extended with a `clone()` method to support prototype duplication.

Figure 17-4 shows the structure of the `PrototypePattern` abstract aspect with the interfaces and behavior that it defines to support the prototype design pattern.

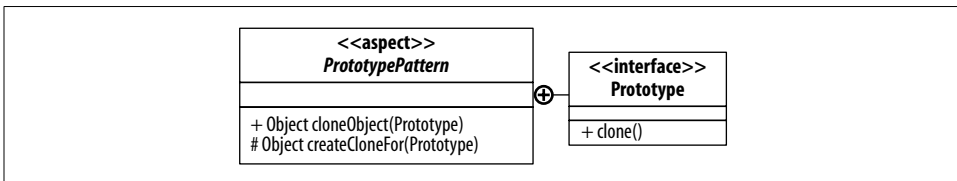


Figure 17-4. The structure of the `PrototypePattern` abstract aspect

A `clone()` method is the Java mechanism for implementing a deep copy of the object. Some base classes may not support being cloned; the `PrototypePattern` aspect provides the `createCloneFor(Prototype)` method, so it can be overridden by sub-aspects to perform specific cloning operations that the generic aspect will not know.

Example 17-4 shows how the abstract `PrototypePattern` aspect can be applied for a specific application.

Example 17-4. Applying the abstract `PrototypePattern` aspect to a target application

```

public aspect GraphicPrototypes extends PrototypePattern
{
    declare parents : Graphic implements Prototype;

    declare parents : MusicalNote implements Prototype;

    declare parents : Staff implements Prototype;

    protected Object createCloneFor(Prototype object)
    {
        if (object instanceof MusicalNote)
        {
            return new MusicalNote(
                ((MusicalNote) object).getX(),
                ((MusicalNote) object).getY());
        }
        else if (object instanceof Staff)
        {
            return new Staff(((Staff) object).getX(), ((Staff) object).getY());
        }
        else
        {
            return null;
        }
    }
}
  
```

Figure 17-5 shows an example of the effects that the `GraphicsPrototype` aspect has on an application's class.

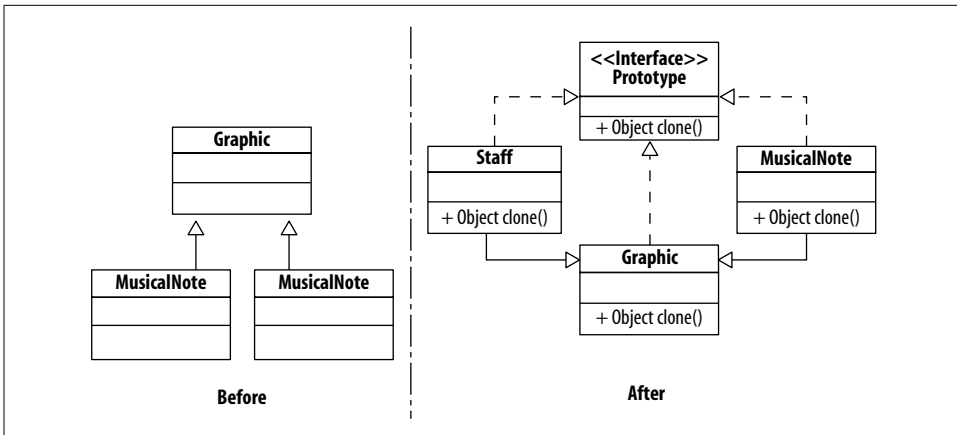


Figure 17-5. How the static application structure is affected by the GraphicsPrototype aspect

Figure 17-6 shows how the new prototype pattern behavior interacts in an example application.

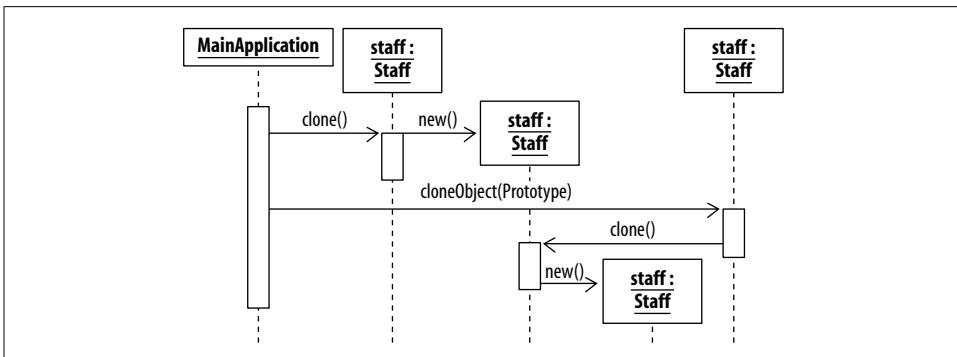


Figure 17-6. Using the new prototype behavior of the Staff class

See Also

Using the declare keyword to affect a class hierarchies static structure is explained in more detail in Recipe 16.2; the Director aspect-oriented design pattern is explained in Recipe 23.3.

17.3 Implementing the Abstract Factory Pattern

Problem

You want to apply the abstract factory pattern using AspectJ.

Solution

The *abstract factory* pattern supports groups of related classes being instantiated while shielding the clients of the factory from the exact implementations.

To implement the abstract factory pattern using AspectJ, create an aspect that can remove the abstract factory pattern's reliance on an abstract base class using static cross-cutting techniques to provide default implementations of the factory methods.

Discussion

Creating a generic factory using abstract aspects makes no sense because the factory contains methods specific to the objects that can be created. The only advantage an AspectJ implementation offers to this design pattern is the ability to remove the reliance on an abstract base class for the abstract factory and replace it with a simple interface. This means that the specialized factories can inherit from other more appropriate classes rather than having use their one allowed inheritance relationship to support the design pattern.

Example 17-5 shows how the abstract factory pattern could be applied to the factory needs of a simple application by providing default implementations of the methods declared in a Java interface.

Example 17-5. Applying the abstract factory pattern using aspects

```
public interface ComputerFactory
{
    public Computer createPentiumProcessorComputer();

    public Computer createComputerWithHardDisk(HardDisk hardDisk);
}

public aspect DefaultComputerFactoryImplementation
{
    public Computer ComputerFactory.createPentiumProcessorComputer()
    {
        Processor processor = new Processor("Pentium 4 : 9089085043");
        Motherboard motherboard = new Motherboard("019283", processor);
        HardDisk hardDisk = new HardDisk("738947");
        FloppyDisk floppyDisk = new FloppyDisk("93746");
        Computer computer = new Computer("12345", motherboard, hardDisk, floppyDisk);
        return computer;
    }

    public Computer ComputerFactory.createComputerWithHardDisk(HardDisk hardDisk)
    {
        Processor processor = new Processor("Pentium Standard : 123478");
        Motherboard motherboard = new Motherboard("434244", processor);
        FloppyDisk floppyDisk = new FloppyDisk("432434");
    }
}
```

Example 17-5. Applying the abstract factory pattern using aspects (continued)

```
        Computer computer = new Computer("56789", motherboard, hardDisk, floppyDisk);
        return computer;
    }
}
```

See Also

Chapter 16 contains recipes that show how static cross-cutting techniques can be used to extend classes and provide default implementations of interface elements.

17.4 Implementing the Factory Method Pattern

Problem

You want to apply the factory method pattern using AspectJ.

Solution

The *factory method* pattern is similar to the abstract factory pattern in that it provides mechanisms by which the exact implementation of an object is decoupled from the clients of the factory. However, the factory method provides a single method for instantiating different implementations of a single interface.

The abstract class that contains the abstract factory method is implemented by specialized classes that explicitly override the factory method to provide mechanisms for instantiating different implementations of the desired object.

To implement the factory method pattern using AspectJ, use the same mechanisms as the abstract factory pattern to create an aspect that can remove the factory method pattern's reliance on an abstract base class using static cross-cutting techniques, providing a default implementation of the factory method or methods.

Discussion

A specialized aspect is shown in Example 17-6 that provides a default implementation for the factory method.

Example 17-6. Applying the factory method pattern using aspects

```
public interface ComputerCreator
{
    public Computer createComputer(String serial);
}

public aspect DefaultComputerCreatorImplementation
{
    public void ComputerCreator.createComputerAndPrintInventory(String serial)
    {
```

Example 17-6. Applying the factory method pattern using aspects (continued)

```
        System.out.println("Inventory of computerparts:");
        System.out.println(this.createComputer(serial).toString());
    }
}
```

Traditionally, the `ComputerCreator` interface in this solution would be an abstract class. However, static cross-cutting techniques can provide more freedom when applying the factory method design pattern by removing the need for the abstract base class.

See Also

Chapter 16 contains recipes that show static cross-cutting techniques for extending classes and providing default implementations of interface elements; Recipe 20.3 shows how to decouple the decision as to which implementation classes of an interface are instantiated solely using aspects.

17.5 Implementing the Builder Pattern

Problem

You want to apply the builder pattern using AspectJ.

Solution

The *builder* pattern captures the complex steps that may be needed in the creation of a particular object. The steps are implemented as methods on the builder class; after each of the required steps has been completed, then the builder can be called to create the resulting built object.

To implement the builder pattern using AspectJ, create an aspect that adds to the top-level builder class a field to store the build result and a method to access that result using static cross-cutting techniques. This enables the builder to be an interface and not an abstract class.

Discussion

The builder pattern can be implemented using aspects, as shown in Example 17-7.

Example 17-7. Applying the builder pattern using aspects

```
public interface TextPhraseBuilder
{
    public void buildHeader(String title);

    public void buildBody(String content);
}
```

Example 17-7. Applying the builder pattern using aspects (continued)

```
    public void buildFooter(String closingContent);

    public String getResult();
}

public aspect TextPhraseBuilderDefaultImplementation
{
    public StringBuffer TextPhraseBuilder.result = new StringBuffer();

    public String TextPhraseBuilder.getResult()
    {
        return result.toString();
    }

    /**
     * Declares a compiler error that gets reported if other classes
     * (except Builders or this aspect) try to access the result variable.
     */
    declare error : (
        set(public StringBuffer TextPhraseBuilder +.result)
        || get(public StringBuffer TextPhraseBuilder +.result))
        && !(within(TextPhraseBuilder +)
        || within(TextPhraseBuilderDefaultImplementation)) :
        "variable result is aspect protected. use getResult() to access it";
}
}
```

The `TextPhraseDefaultImplementationBuilder` aspect provides a default implementation of the `getResult()` method. This frees the specialized builders from exhausting their single inheritance relationship with an abstract base class. The `getResult()` method provides access to the `result` field that is also added to the interface and its implementing classes to provide a place to store the result of the builder.

Ideally, the `result` field would be declared protected since it is only used internally by the `TextPhraseBuilder` and its subclasses. The AspectJ compiler will not allow protected fields to be introduced on an interface, `public` is the only option.

This leads to a second problem. How can the pattern's use be constrained so direct access of the public `result` field is flagged to the developers as the wrong mechanism by which to obtain the field's value? The solution is provided by another powerful feature of AspectJ, the ability to define new compile time checking and error notifications.

Until aspect orientation, the compiler had been a fairly rigid piece of software, not to be tampered with by the developers. However, a few circumstances occur when interaction with the compiler can influence its validation of the code could be useful and this feature is available with AspectJ.

The `TextPhraseBuilderDefaultImplementation` aspect defines an error to be triggered by the compiler should your code attempt to access the newly added `result` attribute

directly. This will provide a final check that the rules of the pattern are being followed. Although the problem being protected against is a disadvantage of the aspect-oriented implementation of the design pattern, the problem is reduced by incorporating this compile time check.

See Also

Chapter 16 provides specific recipes showing static cross-cutting techniques for extending classes, providing default implementations, and extending the compilers capabilities.