Dependency injection and aspect-oriented programming (AOP) are two key technologies that help to simplify and purify domain models and application layers in enterprise applications. Dependency injection encapsulates the details of resource and collaborator discovery, and aspects can (among other things) encapsulate the details of middleware service invocations -- for example, to provide transaction and security management. Because both dependency injection and AOP lead to simpler, more easily tested object-based applications, it is natural to want to use them together. Aspects can help to bring the power of dependency injection to a wider range of objects and services, and you can use dependency injection to configure the aspects themselves.

In this article, I show you how to combine the dependency injection of the Spring framework effectively with aspects written using AspectJ 5. I assume you have a basic knowledge of AOP (although if you don't, you'll find some good starting points in the Resources section), so I'll begin my discussion by analyzing the key roles and responsibilities involved in a dependency injection-based solution. From there, I'll show you how to configure singleton aspects via dependency injection. Because configuring non-singleton aspects shares much in common with configuring domain objects, I'll then look at a simple solution that applies to them both. I conclude the article by showing you how to use aspects for several advanced dependency injection scenarios, including interface-based injection and repeated injection.
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 Related topics for more background on AOP.

See Download for the article source and Related topics to download AspectJ or the Spring framework, which you will need to follow the examples.

What is dependency injection?

In the book Domain-Driven Design, Eric Evans argues persuasively for hiding objects from the details of their configuration and association establishment:

> Much of the power of objects rests in the intricate configuration of their internals and associations. An object should be distilled until nothing remains that does not relate to its meaning or support its role in interactions. This mid-life cycle responsibility is plenty. Problems arise from overloading a complex object with responsibility for its own creation.

Evans goes on to give an example of a car engine whose dozens of parts collaborate to perform the engine's responsibility. While you might be able to imagine an engine block capable of grabbing hold of a set of pistons and inserting them into its cylinders, such a design would significantly complicate the engine. Instead, a human mechanic or a robot assembles the engine, and the engine itself is concerned only with its operation.

While I took this example from a section of the book introducing the concept of factories for complex object creation, we can also apply it to explain the motivation for dependency injection techniques.

From collaboration to contract

Further reading

For the classic introduction to dependency injection, see Martin Fowler's “Inversion of Control Containers and the Dependency Injection Pattern.” For more about dependency injection using Spring, see Professional Java Development with the Spring Framework. Links for both are in Related topics.

For the purposes of this article, you can think of dependency injection as a contract between an object and the environment in which it executes. An object (playing any or all of the roles of ResourceConsumer, Collaborator, and ServiceClient) agrees not to go out searching for the resources it needs, partners it collaborates with, or services it uses. Instead, the object provides a means for these dependencies to be provided to it. In turn, the execution environment agrees to provide the object with all its dependencies before they are needed.
The way that the dependencies are resolved differs in different scenarios. In a unit test case, for example, the execution environment for the object is the test case itself, and the test setup code is tasked with satisfying the dependencies directly. In an integration test or when the application is in production, a *broker* is responsible for finding the resources that satisfy the object's dependencies and passing them to it. The broker role is often played by a lightweight container such as the Spring framework. Regardless of how dependencies are resolved, the object being configured is typically unaware of such details. In the second example, it would likely be unaware of the presence of the broker.

A broker such as the Spring framework has four key responsibilities, which I return to throughout the article:

- Determining that an object needs configuring (typically because it has just been created)
- Determining the dependencies of the object
- Finding the objects that satisfy those dependencies
- Configuring the object with its dependencies

Multiple strategies are possible for discharging these responsibilities, as you'll see from the variety of dependency injection solutions that follow.

**Dependency injection with Spring**

In a standard Spring deployment, the Spring container is responsible for both creating and configuring the core application objects (known as *beans*). Because the container both creates beans and plays the role of broker, it is trivial for the Spring container to determine that a bean has been created and needs configuring. The dependencies of the bean are determined by querying the application metamodel, which is typically expressed in XML as a Spring configuration file.

The objects that satisfy the bean's dependencies are other beans managed by the container. The container acts as a repository for these beans so that they can be looked up by name (or created if need be). Finally, the container configures the new bean with its dependencies. This is usually done via setter injection (calling setter methods on the new bean passing in the dependencies as arguments), though Spring supports other forms of injection such as constructor injection and lookup-method injection. (See [Related topics](#) to learn more about dependency injection using Spring.)

**Dependency injection for aspects**

Like any other object, aspects can benefit from configuration through dependency injection. In many cases, it's good practice to implement an aspect as a lightweight *controller*. In this case, the aspect determines when some behavior should be executed but delegates to a collaborator to perform the actual work. For example, you could configure an exception-handling aspect with an exception-handling strategy object. The aspect would detect when exceptions were thrown and delegate to the handler to handle them. Listing 1 shows a basic *RemoteException* handling aspect:
Listing 1. RemoteException handling aspect

```java
public aspect RemoteExceptionHandling {
    private RemoteExceptionHandler exceptionHandler;

    public void setExceptionHandler(RemoteExceptionHandler aHandler) {
        this.exceptionHandler = aHandler;
    }

    pointcut remoteCall() : call(* *(..) throws RemoteException+);

    /**
     * Route exception to handler. RemoteException will still
     * propagate to caller unless handler throws an alternate
     * exception.
     */
    after() throwing(RemoteException ex) : remoteCall() {
        if (exceptionHandler != null)
            exceptionHandler.onRemoteException(ex);
    }
}
```

Go to the source!
If you want to see the RemoteExceptionHandling aspect configuration in action, download the [article source](#) and run the RemoteExceptionHandlingTest test case from the testsrc folder.

Now I'd like to configure my aspect with a particular exception handling strategy using dependency injection. For this, I can use the standard Spring approach, but with one caveat. Normally Spring is responsible for both creating and configuring beans. AspectJ aspects, however, are created by the AspectJ runtime. I need Spring to configure the aspect that AspectJ has created. For the most common case of singleton aspects such as the RemoteExceptionHandling aspect above, AspectJ defines an aspectOf() method that returns the aspect instance. I can tell Spring to use the aspectOf() method as a factory method for obtaining the aspect instance. Listing 2 shows the Spring configuration for the aspect:

Listing 2. Spring configuration for an aspect

```xml
<beans>
    <bean name="RemoteExceptionHandlingAspect"
        class="org.aspectprogrammer.dw.RemoteExceptionHandling"
        factory-method="aspectOf">
        <property name="exceptionHandler">
            <ref bean="RemoteExceptionHandler"/>
        </property>
    </bean>

    <bean name="RemoteExceptionHandler"
        class="org.aspectprogrammer.dw.DefaultRemoteExceptionHandler">
    </bean>
</beans>
```

I want to be sure my aspect is configured before any remote exceptions are thrown. In the sample code, I'm using a Spring ApplicationContext that ensures this is the case, because it automatically pre-instantiates all singleton beans. If I were using a plain BeanFactory, then calling preInstantiateSingletons would achieve the same result.
Dependency injection for domain objects

Configuring singleton aspects is as easy as configuring any other bean inside a Spring container, but what about aspects that have other lifecycles such as perthis, pertarget, or even perflow aspects? Aspect instances that have a lifecycle other than singleton cannot be pre-instantiated by the Spring container; instead, they are created by the AspectJ runtime in accordance with the aspect declaration. So far, the broker (Spring) has known that an object needed configuring because it created that object. If I want to perform dependency injection of non-singleton aspects, I need to use a different strategy to determine that an object has been created that needs configuration.

Non-singleton aspects aren’t the only kinds of objects created outside the control of the Spring container that would benefit from externalized configuration. For example, domain entities that need access to repositories, services, and factories (see Related topics) would reap the same benefits from dependency injection as do container-managed beans. Recall the four responsibilities of a broker:

- Determining that an object needs configuring (typically because it has just been created)
- Determining the dependencies of the object
- Finding the objects that satisfy those dependencies
- Configuring the object with its dependencies

I still want to use Spring to determine the dependencies of an object, to find the objects that satisfy those dependencies, and to configure the object with its dependencies. However, I need another way to determine that an object needs configuring. In particular, I need a solution for objects that can be created at arbitrary points in the execution of the application, outside of the Spring container’s control.

The SpringConfiguredObjectBroker

I call an object to be configured by Spring a SpringConfigured object. The requirement follows that after creating a new SpringConfigured object, I should ask Spring to configure it. A SpringConfiguredObjectBroker backed by a Spring ApplicationContext should do the job, as shown in Listing 3:

Listing 3. @SpringConfigured object broker

```java
public aspect SpringConfiguredObjectBroker
    implements ApplicationContextAware {
    private ConfigurableListableBeanFactory beanFactory;

    /**
     * This broker is itself configured by Spring DI, which will
     * pass it a reference to the ApplicationContext
     */
    public void setApplicationContext(ApplicationContext aContext) {
        if (!(aContext instanceof ConfigurableApplicationContext)) {
            throw new SpringConfiguredObjectBrokerException("ApplicationContext "] + aContext +
```
"] does not implement ConfigurableApplicationContext"
};
} this.beanFactory = ((ConfigurableApplicationContext)aContext).getBeanFactory();
}

/**
* creation of any object that we want to be configured by Spring
*/
pointcut springConfiguredObjectCreation(
    Object newInstance,
    SpringConfigured scAnnotation
) : initialization((@SpringConfigured *).new(..))
&& this(newInstance) && @this(scAnnotation);

/**
* ask Spring to configure the newly created instance
*/
after(Object newInstance, SpringConfigured scAnn) returning
: springConfiguredObjectCreation(newInstance,scAnn)
{
    String beanName = getBeanName(newInstance, scAnn);
    beanFactory.applyBeanPropertyValues(newInstance,beanName);
}

/**
* Determine the bean name to use - if one was provided in
* the annotation then use that, otherwise use the class name.
*/
private String getBeanName(Object obj, SpringConfigured ann) {
    String beanName = ann.value();
    if (beanName.equals("")) {
        beanName = obj.getClass().getName();
    }
    return beanName;
}

Inside the SpringConfiguredObjectBroker

I'll walk through the parts of the SpringConfiguredObjectBroker aspect in turn. First, the aspect implements the Spring ApplicationContextAware interface. The broker aspect is itself configured by Spring (that's how it obtains the reference it needs to the application context). Having the aspect implement the ApplicationContextAware interface ensures that Spring knows to pass it a reference to the current ApplicationContext during configuration.

The pointcut springConfiguredObjectCreation() matches the initialization join point of any object with the @SpringConfigured annotation. Both the annotation and the newly created instance are captured as context at the join point. Finally, the after returning advice asks Spring to configure the newly created instance. A bean name is used to look up the configuration information for the instance. I could provide this as the value of the @SpringConfigured annotation, or the default of using the class name can be used.

The aspect implementation itself could be part of a standard library (an aspect like this will in fact be shipped with future releases of Spring), in which case all I need to do is annotate types whose instances are to be configured by Spring, as shown here:
@SpringConfigured("AccountBean")
public class Account {
    ...
}

You can create instances of such types under program control (for example, as the result of a query to the database), and they will have all of their Spring-configured dependencies managed for them automatically. See the article source for examples of the @SpringConfigured annotation being used. Note that while I chose to use an annotation for this example (because it is a very natural way to provide the bean name), a marker interface would make it possible to use the approach with Java™ 1.4 and below.

As I discussed at the start of this section, the SpringConfigured technique works not only for domain entities, but for any object created outside the control of the Spring container (for objects created by Spring itself there is no need to add the extra complication). You can configure any aspect, regardless of its lifecycle, this way. For example, if you define a percflow aspect, then AspectJ creates a new aspect instance each time the associated control flow is entered, and Spring configures each of the aspects as it is created.

Interface-based injection

So far, I've used the bean definition read by the Spring container to determine the dependencies of an object. A variation on this theme uses contract interfaces for a client to declare its requirements. Suppose the Account entity from the previous section required access to the AccountOperationValidationService. I could declare an interface as shown in Listing 4:

Listing 4. Client interface

```java
public interface AccountOperationValidationClient {
    public void setAccountOperationValidationService(
            AccountOperationValidationService aValidationService);
}
```

Now any object needing access to the AccountOperationValidationService simply has to implement this interface and declare itself as a client. Using an aspect similar to the one I developed in the previous section, I can match all initialization join points for client objects implementing this interface. That takes care of the first broker responsibility: determining when an object needs to be configured. The second responsibility is made explicit in the interface: the dependency that must be satisfied is the validation service dependency. I'm going to use an aspect to dependency inject all clients of the validation service. The easiest way for the aspect to get a hold of the appropriate service to inject is to inject that service into the aspect itself! Listing 5 shows an example:

Listing 5. Service injector aspect

```java
/**
 * ensure that all clients of the account validation service
 * have access to it
 */
```
public aspect AccountOperationValidationServiceInjector {
    private AccountOperationValidationService service;

    /**
     * the aspect itself is configured via Spring DI
     */
    public void setService(AccountOperationValidationService aService){
        this.service = aService;
    }

    /**
     * the creation of any object that is a client of the
     * validation service
     */
    pointcut clientCreation(AccountOperationValidationClient aClient) :
        initialization(AccountOperationValidationClient+.new(..)) &&
        this(aClient);

    /**
     * inject clients when they are created
     */
    after(AccountOperationValidationClient aClient) returning :
        clientCreation(aClient) {
            aClient.setAccountOperationValidationService(this.service);
        }
}

This solution gives me two levels of control. The actual definition of the service itself is provided in
the Spring configuration file, as for example in the XML fragment of Listing 6:

**Listing 6. Service injector configuration**

```xml
<beans>
    <bean name="AccountOperationValidationServiceInjector"
        class="org.aspectprogrammer.dw.
        AccountOperationValidationServiceInjector"
        factory-method="aspectOf">
        <property name="service">
            <ref bean="AccountOperationValidationService"/>
        </property>
    </bean>

    <bean name="AccountOperationValidationService"
        class="org.aspectprogrammer.dw.
        DefaultAccountOperationValidationService">
    </bean>
</beans>
```

Clients of the service only have to implement the `AccountOperationValidationClient` interface
and they are automatically configured with the current instantiation of the service as defined by
Spring.

**Repeated injection**

**Lookup method injection in Spring**

Lookup method injection is an advanced feature supported by the Spring container whereby
the container overrides abstract or concrete methods on managed beans, to return the
result of looking up another named bean in the container. The lookup will typically be of a non-singleton bean. The bean that looks up the dependency defines a lookup method with a declared return type of the type of the bean to be looked up. The Spring configuration file uses the `<lookup-method>` element inside a bean definition to tell Spring what bean instance to return when the lookup method is called. See Related topics to learn more about this technique. A Spring AOP proxy with a HotSwappable target source offers another approach.

So far, I've looked at solutions that involve configuring objects immediately after their instantiation. In some cases, however, the objects that a client needs to collaborate with will change at run time. For example, by interacting with the system, a sales team might be able to dynamically change the pricing strategy and seat allocation strategy for an online booking application. The booking service interacting with the pricing strategy and seat allocation strategy would need the implementations of those strategies that were current at the time of making a booking, not the versions that were current when the booking service itself was first instantiated. In such cases, it is possible to defer injection of a dependency until a client first needs it, and to re-inject the client with the most up-to-date version of the dependency each time it is referenced.

The basic techniques for this scenario involve either field-level injection or getter-method overrides. Before getting into the example, let me stress once more that I'm looking at injection techniques for objects created outside of the Spring container's control. For objects created by Spring, the Spring container already offers simple mechanisms to address these needs.

### Field-level injection

In the example below, you can see how to apply a field-level injection for deferred or repeated injection. A `get` join point of a field lets me determine when to do the injection and the type of the field determines the dependency to inject. So, for example, if a client declares a field like this

```java
private PricingStrategy pricingStrategy;
```

and somewhere in a client method I find the code

```java
this.pricingStrategy.price(.....);
```

then the execution of that code at run time leads to a `get()` join point for the `pricingStrategy` field that I can use to inject the current pricing strategy implementation, as shown in Listing 7:

### Listing 7. Field-level injection example

```java
public aspect PricingStrategyInjector {
    private PricingStrategy currentPricingStrategy;
    public void setCurrentPricingStrategy(PricingStrategy aStrategy) {
        this.currentPricingStrategy = aStrategy;
    }

    /**
     * a client is trying to access the current pricing strategy
     */
    pointcut pricingStrategyAccess() :
        get(PricingStrategy *) &&
```
 Within(PricingStrategyInjector); // don’t advise ourselves!

/**
 * whenever a client accesses a pricing strategy field, ensure they
 * get the latest...
 */
PricingStrategy around() : pricingStrategyAccess() {
    return this.currentPricingStrategy;
}
}

See the article source to observe this technique in action.

**The service-location strategy**

An alternative to repeated injection is to instead use a more conventional technique to inject a client with a service-location strategy implementation. For example:

```java
public interface PricingStrategyLocator {
    PricingStrategy getCurrentPricingStrategy();
}
```

At the expense of defining an additional interface and making the client code a little more verbose, this approach certainly has the edge for me when it comes to code clarity.

**In conclusion**

In this article, I’ve looked at dependency injection as a contract between an object and the environment in which it executes. The object agrees not to go out searching for the resources it needs, partners it collaborates with, or services it uses. Instead, the object provides a means for these dependencies to be provided to it. In turn, the execution environment agrees to provide the object with all of the dependencies it needs, before the object needs them.

I discussed the four key components of a dependency injection solution, which must be addressed by the broker acquiring dependencies on the object’s behalf. Finally, I looked at a number of different approaches to satisfying these requirements. It should be clear that if you can use the Spring container to both instantiate and configure your objects, you should do so. For objects created outside of the control of the Spring container, such as some domain objects or aspects with a non-singleton instantiation model, I recommend using the @SpringConfigured annotation or similar. This technique lets you fully externalize all configuration information into a Spring configuration file.

I wrote the examples for this article using the latest milestone build of AspectJ 5 (as of October 2005) and Spring 1.2.4. Download the complete working examples to begin experimenting with the ideas I’ve discussed. The test cases under the testsrc directory are a good starting point.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source code</td>
<td>j-aopwork13.zip</td>
<td>26 KB</td>
</tr>
</tbody>
</table>
Related topics

- "Secrets of lightweight development success, Part 3: The emergence of Spring" (Bruce Tate, developerWorks, June 2005): An overview of the Spring framework.
- "AOP@Work: Unit test your aspects" (Nicholas Lesiecki, developerWorks, November 2005): Use dependency injection techniques to unit test your aspects.
- "Inversion of Control Containers and the Dependency Injection Pattern" (Martin Fowler, January 2004): The classic introduction to dependency injection.
- Domain-Driven Design: Tackling Complexity in the Heart of Software (Eric Evans; Addison-Wesley, August 2003): A good starting point for understanding the motivation for dependency injection techniques.
- Professional Java Development with the Spring Framework (Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu; Wiley, July 2005): Learn more about dependency injection with Spring.
- The Spring home page: Download the Spring framework.
- The AspectJ home page: Download the latest AspectJ compiler and tools.
- The AOP@Work series: Read the complete series for practical applications of AOP.