Article

Spring 2.0: What's New and Why it Matters

Posted by Rod Johnson on Jan 15, 2007 11:00 PM
Community Java Topics Transactions Processing , Web Frameworks , AOP Tags Spring

Themes

Since the open source project began in February, 2003, the Spring Framework has gone from strength to strength. It has powered past 1 million downloads; become a de facto standard in a wide range of industries; and changed the development of enterprise Java applications.

Most important, it has developed a large and loyal user base, which understands its key values and has contributed feedback that has helped it to advance rapidly. Spring's mission always been clear:

- To provide a non-invasive programming model. As far as possible, application code should be decoupled from the framework.
- To provide a superior solution to in-house infrastructure, so that developers can focus on delivering business value rather than solving generic problems.
- To make developing enterprise applications as simple as possible, but enhancing, rather than sacrificing, power.

Spring 2.0, which went final in October, 2006, further advances these values. As the core team looked at the development feature set coming together before the December 2005 *Spring Experience Conference* in Florida, we realized that two key themes - Simplicity and Power - stood out as a the common thread in Spring 2.0, and remained faithful to Spring's beginnings.

Some decisions were easy. From the outset, we were clear that Spring 2.0 was going to be fully backward compatible, or as near to fully backward compatible as possible. Especially given Spring's position as a de facto standard in many enterprises, it was vital to avoid any disruption in user experience. Fortunately, because Spring has always gone to such pains to be non invasive, this goal was achievable.

As work on Spring 2.0 progressed through 10 months of development, we also needed to take into account several trends that became evident in Spring usage in 2005-2006:

- Spring is increasingly used by very large organizations, who are adopting it strategically rather than project-by-project. This imposes not merely a level of responsibility regarding backward compatibility, but a set of challenges relating to a demanding class of users.
- An increasing number of prominent third party software products are using Spring internally, and need the optimum in configurability and flexibility from the container. Examples here are many. To choose just a few:
  - The upcoming BEA WebLogic Server 10, which uses Spring and the Pitchfork Project to perform injection and interception.
  - BEA WebLogic Real Time (WLRT)-a high-end product from BEA targeted at applications such as front office trading, requiring low latency.
  - Numerous widely used open source products such as Mule, ServiceMix and the Apache JetSpeed portal container.
  - Enterprise vendors integrating their products with Spring such as GigaSpaces, Terracotta and Tangosol. Vendors in the grid space, in particular, are increasingly embracing Spring as the programming model of choice.
  - Oracle's SCA implementation and various other Oracle products.

So we needed to ensure that while Spring become even better for developers of business applications, we also catered towards the needs of these demanding users.

From 35,000 Feet

What's the big picture in Spring 2.0?

Spring 2.0 provides a wide range of enhancements, of which the most visible are probably:

- Configuration extensions: In Spring 2.0, Spring supports extensible XML configuration, enabling the development of custom elements that offer a new level of abstraction for generating Spring bean definitions. The XML extension mechanism also allows the provision of new tags to simplify many common tasks.
- Major enhancements in the AOP framework, which make it both more powerful and easier to use.
- Enhanced support for Java 5.
- The ability for Spring beans to be implemented in dynamic languages, such as Groovy, JRuby and Beanshell, while retaining all the services of the Spring component model, such as Dependency Injection, out of the box declarative services and AOP.
- Many new features including a Portlet MVC framework, "message driven POJOs," integration with new APIs including the Java Persistence API (JPA) and an asynchronous task execution framework.

A number of features below the surface are less evident, but still important:

- Further IoC container extension points that make it easy to build frameworks or products on top of Spring.
- Improvements in Spring's unique integration testing support.
- The provision of AspectJ aspects exposing core Spring functionality such as transaction management and Dependency Injection to users using both AspectJ and Spring.

Importantly, these features are designed to work together in a harmonious whole.

Outline

This article falls into two parts. In Part I (this article), we will cover the core container, XML configuration extensions, AOP enhancements and Java 5-specific features.

In Part II (posted in the coming months), we will cover messaging, support for dynamic languages, Java Persistence API and web tier enhancements. We will also look at a number of behind the scenes improvements.

Now let's look at some of the new features in more detail, illustrating them with code examples.

XML configuration extensions

One of the most visible enhancements in Spring 2.0 concerns XML configuration.

The Spring IoC container is actually independent of the representation of metadata, such as XML. Spring has its own internal metadata in the form of Java objects (BeanDefinition and subinterfaces). There is active research in alternatives to supplement XML configuration, such as Java configuration using annotations (http://blog.interface21.com/main/2006/11/28/a-java-configuration-option-for-spring/).

However, XML is most often used to configure Spring today, and thus is a major focus of configuration improvements in the Spring core.

The XML enhancements in Spring 2.0 neatly sum up the themes of simplicity and power: they simplify performing some common tasks, but also make additional advanced tasks possible.

Goals

Traditionally there has been a 1:1 relationship between Spring's XML configuration syntax and Spring's internal metadata. One element produced one BeanDefinition.

This is normally what we want, and is ideal for configuring application classes that the framework does not know about.

However, what if the framework should know more about a particular class that is likely to be used repeatedly-for example, a generic class such as JndiObjectFactory, which is used to look up objects from JNDI and bring them into a Spring container as injectable objects. What if several bean definitions only make sense if used together?

Thus a new form of abstraction adds important benefit.

Consider the following example, of a JNDI lookup:

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/jpetsore" />
</bean>
```

This is certainly better than the nasty old days of implementing Service Locators, but it's not perfect. We are always going to use the same bean class. And (at least unless we're using Java 5) there is no mechanism for indicating that the "jndiName" property is required, while other properties are optional.

Spring 2.0 adds a "jee" namespace out of the box, including the tag allows us to perform the same JNDI lookup as follows:

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>
```

This is simpler and clearer. It clearly expresses the intent. It's more concise. And the schema captures the fact that the jndiName property is mandatory, in turn facilitating tool support. Other optional properties are also expressed in the schema, as the following example shows:

```
<jee:jndi-lookup id="simple"
             jndi-name="jdbc/MyDataSource"
             cache="true"
             resource-ref="true"
             lookup-on-startup="false"
             expected-type="com.myapp.DefaultFoo"
             proxy-interface="com.myapp.Foo"/>
```

Note that while in such a relatively simple case, the attribute names are almost the same as the property names in the class being configured, this is not necessary. There does not need to be a naming correspondence, or a 1:1 correspondence between attributes and properties. We can also process subelement content. And, as I mentioned earlier, we can generate any number of bean definitions we want from an extension tag.

Now let's consider a more complex example, where we use the ability of a custom tag to generate more than one bean definition.

Since Spring 1.2, it has been possible to make Spring recognize the @Transactional annotation and automatically make affected beans transactional by proxying. This results in a simple deployment model-simply add annotated beans and they are automatically transactional-but setting it up required some distracting magic. Three bean definitions were needed for the necessary collaborating objects-a Spring AOP Advisor and TransactionInterceptor, as well as a DefaultAdvisorAutoProxyCreator to cause automatic proxying. These magic bean definitions constituted an incantation that could be used unchanged in different applications, but exposed more detail than a user needs to know:

```
<bean
    class="org.springframework...DefaultAdvisorAutoProxyCreator"/>

 <bean class="org.springframework...TransactionAttributeSourceAdvisor">
    <property name="transactionInterceptor
                    ref="transactionInterceptor"/>
</bean>

 <bean id="transactionInterceptor"
      class="org.springframework...TransactionInterceptor">
    <property name="transactionManager"
                    ref="transactionManager"/>
    <property name="transactionAttributeSource">
        <bean
class="org.springframework...AnnotationsTransactionAttributeSource">
            </bean>
    </property>
</bean>
```

This is the wrong level of abstraction. You don't need to see this level of detail concerning the workings of the Spring AOP framework to control transaction management, and the intent is less clear. Spring 2.0 provides a single tag in the new "tx" namespace that replaces all these definitions as follows:

```
<tx:annotation-driven />
```

This out of the box tag achieves the same result. The tag clearly expresses the intent-automatically recognize transaction annotations. Those three bean definitions will still be created by the extension tag behind the scenes, but that is (correctly) now a matter for the framework, not the user.

> **A Spring 2.0 extension tag can define its own attributes and subelement structure as necessary. The developer defining the namespace is in full control. A NamespaceHandler processing a Spring 2.0 extension tag can create any number of bean definitions from an extension tag.**

In order to use an extension tag, it is necessary to use XML schema rather than DTD, and import the relevant namespace. The default namespace should be the beans schema. The following example the "tx" namespace, allowing the use of **<tx:annotation-driven>**:

```
<?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
```

> **Extension tags can be mixed with regular bean definitions, and any number of namespaces can be imported and used in the same document.**

Out-of-the-box namespaces

Spring 2.0 provides several namespaces out of the box. The most important are:

- Transaction management ("tx"): Making Spring beans transactional becomes significantly easier in Spring 2.0, as we have seen. It also becomes easier to define "transaction attributes" mapping transactional behavior onto methods.
- AOP ("aop"): Specialized tags allow AOP configuration to be much more concise in Spring 2.0 than previously, without the IoC container needing to depend on the AOP framework.
- Java EE ("jee"): This simplifies working with JNDI and other Java EE APIs, as we have seen. EJB lookups gain still more than JNDI lookups.
- Dynamic languages ("lang"): Simplifies the definition of beans in dynamic languages-a new feature in Spring 2.0.
- Utils ("util"): Simplifies loading java.util.Properties objects and other common tasks.

In Spring 2.1 and beyond, further namespaces will be added to bring this simplicity to new areas. Namespaces simplifying Spring MVC and JPA usage will probably be the first to be added to the core.

Third party configuration extensions

Being an *extension* mechanism, the most important possibilities around Spring 2.0 namespaces will be outside the Spring core.

Many products build on Spring, and their configuration can become simpler using namespaces. A good example is Acegi Security for Spring (to be rebranded Spring Security in early 2007), which requires the definition of several collaborating beans to configure. A Spring 2.0 namespace will make this much simpler.-again, more clearly expressing the intent.

Many products integrate closely with Spring, and the benefits in this case are obvious. Tangosol's integration for Coherence is a case in point.

Other potential examples include products with Spring-friendly configuration such as IBM's ObjectGrid. Although ObjectGrid does not presently use Spring internally, it is designed to be configured via JavaBeans to make it easy to integrate into Spring-based applications. An extension schema would make this much simpler.

> **It is possible for an XML document to use an extension tag as the top level element. This avoids the need to prefix the extension schema elements with a namespace, meaning that the configuration can look "native" and not Spring-centric. (Normally the element is in the default namespace, so traditional Spring bean definitions do not need prefixing.)**

Over time, as with JSP custom tags, experience will point to general purpose tags with proven value. We expect users to create libraries of namespaces to benefit the community.

Implementing an XML extension

Implementing namespaces is relatively easy. It is a three step process:

1. Define your XML schema. This is the hardest step, and requires appropriate tooling. There are no constraints on the schema, although of course you need to be able to see how it will guide the generation of BeanDefinition metadata at runtime.
2. Implement the NamespaceHandler interface to generate BeanDefinitions from the elements and attributes in your schema.
3. Edit a special registration file, spring.handlers, to make the new NamespaceHandler class known to Spring.

The spring.handlers file that ships with Spring shows how the "standard" namespaces are configured:

```
http\://www.springframework.org/schema/util=org.springframework.beans.factory.xml.UtilNamespaceHandler
http\://www.springframework.org/schema/aop=org.springframework.aop.config.AopNamespaceHandler
http\://www.springframework.org/schema/lang=org.springframework.scripting.config.LangNamespaceHandler
http\://www.springframework.org/schema/tx=org.springframework.transaction.config.TxNamespaceHandler
http\://www.springframework.org/schema/jee=org.springframework.ejb.config.JeeNamespaceHandler
http\://www.springframework.org/schema/p=org.springframework.beans.factory.xml.SimplePropertyNamespaceHandler
```

You can have multiple spring.handlers files in different /META-INF directories. Spring will merge them at runtime.

Once you've followed these steps you can use your new extension.

The NameSpaceHandler interface is not difficult to implement. It takes W3C DOM Elements and generates BeanDefinition metadata by processing them. Spring parses the XML: your code merely has to walk the tree.

```
public interface NamespaceHandler {
     void init();
     BeanDefinition parse(Element element, ParserContext parserContext);

      BeanDefinitionHolder decorate(Node source, BeanDefinitionHolder definition, ParserContext parserContext);
  }
```

The parse() method is the most important, and is responsible for adding BeanDefinitions to the context supplied.

Note also the decorate() method. A NamespaceHandler can also decorate an enclosing bean definition, as shown in the following syntax for creating scoped proxies:

```
<bean id="scopedList" class="java.util.ArrayList" scope="request">
    <aop:scoped-proxy/>
 </bean>
```

The <aop:scoped-proxy> tag decorates the regular <bean> element that contains it; given access to the BeanDefinition, it can modify it.

To simplify the generation of BeanDefinition metadata, Spring 2.0 introduces a handy new BeanDefinitionBuilder class, offering a fluent, builder-style API. The best guide to getting started implementing NamespaceHandlers is those shipped in the Spring core. The UtilNamespaceHandler is a relatively simple example; the AopNamespaceHandler an advanced example that parses a sophisticated subelement structure.

Best practices: When should your define your own namespace?

Just because you have a hammer doesn't mean that everything is a nail. As we've seen, Spring 2.0's XML extension mechanism delivers great value in many cases. However, it should not be used without good reason. Because an XML extension tag is a new abstraction, it also provides something new to learn. Spring's regular XML format is already familiar to tens of thousands of developers, and intuitive even to those new to Spring. Spring XML files provide a well-understood map or blueprint of an application's structure. This will not necessarily be the case if configurations make heavy use of unfamiliar custom tags.

Let's consider some relevant experience in this area. JSP custom tags are a good example. Ultimately they produced real benefit, in the form of well-designed tag libraries resulting from experience such as JSTL, the Struts and Spring MVC tag libraries. But in the early years they produced abominations that could even obfuscate JSPs. (I can speak from experience here, as I implemented one or two such myself.)

Think of namespace handlers as an important new extension point and valuable new abstraction for Spring. They are great for those building third party products on top of Spring; they are useful for very large projects. It will soon be hard to imagine life without them. But end users should be cautious about implementing, rather than consuming, them.

> **Of course the extension tags provided out of the box with Spring, such as the aop, tx and jee namespaces, will soon be a core part of the Spring configuration vocabulary, as widely understood as the element. You should certainly use these in preference to the more verbose traditional means of accomplishing the same thing.**

Syntax sugar

The move to schema also allows a little shortcut, using attributes instead of subelements for property values. These attributes are not validated, but because we are using XML schema, rather than a DTD, we can still retain all other validation. As the attribute names are property names, XML validation wouldn't add anything anyway; this is a problem of Java-based validation, rather than XML structure.

Imagine the following Java object, with two simple properties and a dependency on an associated object:

```
public class Person {
  private int age;
  private String name;
  private House house;
  public void setAge(int age) {
        this.age = age;
  }

  public void setName(String name) {
        this.name = name;
  }

  public void setHouse(House house) {
        this.house = house;
  }

  }
```

This can be configured using XML as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean class="com.interface21.spring2.ioc.Person"
     p:name="Tony"
     p:age="53"
     p:house-ref="number10"
        />

   <bean class="com.interface21.spring2.ioc.House"
     id="number10"
     p:name="10 Downing Street"
     />

  </beans>
```

Note how the properties can be supplied using attributes, rather than elements. This works through the magic of the special namespace: "p". This namespace is not validated, but allows the use of attributes with names matching Java property names.

With simple types we simply use the property name in the "p" namespace, as in "p:name". When injecting references to other Spring beans, use the "-ref" suffix, as in "p:house-ref".

This shortcut syntax is particularly compelling when you want to use autowiring. For example, consider the following variant:

```
<bean class="com.interface21.spring2.ioc.Person"
      autowire="byType"
      p:name="Tony"
      p:age="53"
 />
```

Here we have not set the "house" property, as autowiring can take care of that. You could even use default-autowire at the <beans> element level to have autowiring across the entire file.

The following snippet from Spring 1.0 or 1.1 usage illustrates how much Spring configuration has reduced the minimum number of angle brackets in the last two major releases (1.2 and 2.0):

```
<bean class="com.interface21.spring2.ioc.Person">
    <property name="name"><value>"Tony"</value></property>
    <property name="age"><value>"53"</value></property>
    <property name="house"><ref local="number10" /></property>
 </bean>
```

In Spring 1.2 we introduced the "value" and "ref" attributes, instead of requiring subelements of in most cases, while in Spring 2.0 it's possible to use attributes pure and simple.

Of course, the traditional XML forms continue to work. Use them when the property value is complex and not legal or readable as an attribute value. And, of course, there is no need to rewrite existing configuration files.

Besides XML config extensions, there are a number of other new features in the Spring IoC container.

Other IoC container enhancements

New bean scopes

Along with XML extensions, the most important new IoC container feature is the addition of custom scopes for bean lifecycle management.

1. Background

Spring has traditionally offered two scopes for beans: **Singleton** and **Prototype** (or non-singleton).

A Singleton bean is a singleton in the context of the owning container. There will be exactly one instance during the lifetime of the container, and when the container is closed down it will issue events to any singleton beans that are interested in destruction events-for example, to close down any managed resources like pooled connections.

A Prototype bean is created whenever it is referenced through injection into another bean, or in response to a getBean() call on the owning container. In this case, the bean definition does not correspond to a single object, but is a recipe for creating an object. Each created instance will be identically configured, but have a distinct identity. The Spring container will not hold onto a reference to a Prototype; its lifecycle is the responsibility of the code that obtained it.

In Spring 2.0 we add the ability for custom scopes. These can be given any name. A custom scope typically corresponds to a backing store that can manage object instances. In this case, Spring provides its familiar programming model, enabling injection and lookup, and the backing store provides instance management of the scoped objects.

Typical backing stores are:

- HTTP session
- Clustered cache
- Other persistent store

2. Web scopes

The most common requirement for this feature concerns transparently storing objects in the HTTP session in web applications. This is supported out of the box in Spring 2.0. Because this requirement is common, it's a good basis for an example.

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

Here we specify "session" scope instead of the default "singleton". Scopes can be given any name, but "session" and "request" are provided out of the box for use in web applications.

When we call getBean() with the name "userPreferences" Spring will transparently resolve the UserPreferences object from the current HTTP Session. If no UserPreferences object is found Spring will create one. Injection enables a blueprint for a preconfigured UserPreferences that can then be customized for the user in question.

To get this working, you need to define a filter as follows in your web.xml file. This will handle a ThreadLocal binding behind the scenes so that Spring knows which HTTP Session object to look in:

```
<web-app>
   ...
   <listener>
     <listener-class>
org.springframework.web.context.request.RequestContextListener></listener-class>
   </listener>
   ...
 </web-app>
```

This addresses lookup. But we normally prefer an API-less injection style. What happens if we want to inject the "userPreferences" bean into other beans, which have a longer lifecycle? For example, what if we want to work with individual UserPreferences objects in a singleton Spring MVC Controller like this:

```
public class UserController extends AbstractController {
    private UserPreferences userPreferences;
    public void setUserPreferences(UserPreferences userPreferences) {          this.userPreferences = userPreferences;
        }

        @Override
   protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response)
                       throws Exception {
        // do work with this.userPreferences
        // Will relate to current user  }
  }
```

In this case, we effectively want "just in time" injection, where the reference to the shorter-lived UserPreferences object is resolved at the point of use in the injectee.

> **There is a common misconception that Spring injection is static and hence necessarily stateless. This is untrue. Because Spring has a sophisticated proxy-based AOP framework, it can hide lookups at runtime providing such "just in time" functionality. Because the IoC container controls what is injected, it can inject a proxy that hides the necessary lookup.**

We can instruct the container to perform such proxying easily, as follows. We decorate the userPreferences bean definition with a subelement, <aop:scoped-proxy>.

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
            <aop:scoped-proxy/>
    </bean>

     <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userController" class="com.mycompany.web.UserController">
         <!-- a reference to the proxied 'userPreferences' bean -->
        <property name="userPreferences" ref="userPreferences"/>

     </bean>
 </beans>
```

Now the resolution will occur dynamically as expected; the userPreferences instance variable in the UserController will be a proxy that resolves the correct target UserPreferences from the HTTP Session. We don't need to interact with the HTTP Session directly, and can easily unit test the UserController without a mock HttpSession object.

Another way to get "just in time" injection is to use a lookup method. This is a technique available since Spring 1.1 to enable a long-lived (usually singleton) bean to depend on a potentially shorter-lived bean. The container can override an abstract (or concrete) method to return the result of performing a getBean() call at the time when the method is invoked.

In this case we do not define an instance variable in the injectee, but an abstract method returning the required object. The method may be public or protected:

```
public abstract class UserController extends AbstractController {
   protected abstract UserPreferences getUserPreferences();

   @Override   protected ModelAndView handleRequestInternal(HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        // do work with object returned by getUserPreferences()
        // Will relate to current user  }
  }
```

In this case, there is no need to use a scoped proxy. We change the bean definition of the injectee, not the bean being injected. The XML configuration will look as follows:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session" />

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userController" class="com.mycompany.web.UserController">
     <!-- a reference to the proxied 'userPreferences' bean -->
        <lookup-method name="getUserPreferences" bean="userPreferences" />

     </bean>
</beans>
```

This mechanism requires CGLIB on the classpath.

3. Other possibilities

Naturally, in true Spring fashion, the underlying mechanism is pluggable, and not tied to the web tier. For example, a Tangosol Coherence scope can be used as follows, with the "datagrid" namespace provided by Tangosol and Interface21:

```
<?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"

 xmlns:datagrid="http://schemas.tangosol.com/schema/datagrid-for-spring"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                                          http://www.springframework.org/schema/aop
 http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
                                          http://schemas.tangosol.com/schema/datagrid-for-spring
 http://schemas.tangosol.com/schema/datagrid-for-spring/datagrid-for-spring.xsd">

       <datagrid:member/>

       <bean id="brian" class="Person" scope="datagrid">
       <aop:scoped-proxy/>

              <property name="firstName" value="brian" />
              <property name="lastName" value="oliver" />
              <property name="age" value="21" />
       </bean>
 </beans>
```

Declaring a bean to be scoped within a "datagrid" means that the management of state for the bean is performed by Coherence, in a Clustered Cache, rather than on the local server. Of course, the bean is instantiated and injected by Spring, and can benefit from Spring services. We anticipate that scoped beans will be useful in the context of SOA and batch processing, and expect to add further scopes out of the box in future Spring releases.

4. Custom scopes

Defining your own custom Scope is straightforward. The Reference Manual explains the process in detail (http://static.springframework.org/spring/docs/2.0.x/reference/beans.html#beans-factory-scopes). You will need a strategy for identifying how to resolve the object in the current scope. Typically, this will involve a ThreadLocal, as the web scopes do behind the scenes.

Type inference

If you're running Java 5, Spring 2.0 can benefit from new capabilities like generics. For example, imagine the following class:

```
public class DependsOnLists {
    private List  plainList;

    private List<Float> floatList;

    public List<Float> getFloatList() {

       return floatList;
        }

  public void setFloatList(List<Float> floatList) {
    this.floatList = floatList;
        }
  public List getPlainList() {
    return plainList;
        }

 public void setPlainList(List plainList) {
    this.plainList = plainList;
        }

 }
```

The "plainList" property is an old style collection, while the "floatList" property has a typed indicated.

Consider the following Spring configuration:

```
<bean class="com.interface21.spring2.ioc.DependsOnLists">
      <property name="plainList">
           <list>
              <value>1</value>
              <value>2</value>
              <value>3</value>
            </list>
      </property>
      <property name="floatList">
            <list>
               <value>1</value>
               <value>2</value>
               <value>3</value>
            </list>
      </property>
 </bean>
```

Spring will correctly populate the "floatList" property with floats, rather than Strings, as it is smart enough to realize that it needs to perform the type conversion.

The following test illustrates this:

```
public class GenericListTest extends
AbstractDependencyInjectionSpringContextTests {

    private DependsOnLists dependsOnLists;

    public void setDependsOnLists(DependsOnLists dependsOnLists) {
     this.dependsOnLists = dependsOnLists;
   }

    @Override
   protected String[] getConfigLocations() {
     return new String[] { "/com/interface21/spring2/ioc/inference.xml" };
   }

    public void testLists() {
     List plainList = dependsOnLists.getPlainList();
     List<Float> floatList = dependsOnLists.getFloatList();
     for (Object o : plainList) {
         assertTrue(o instanceof String);
         System.out.println("Value='" + o + "', class=" +
                                 o.getClass().getName());
                 }
     for (Float f : floatList) {
         System.out.println("Value='" + f + "', class=" +
                                 f.getClass().getName());
     }
   }

  }
```

The output will look as follows:

```
Value='1', class=java.lang.String
Value='2', class=java.lang.String
Value='3', class=java.lang.String
Value='1.0', class=java.lang.Float
Value='2.0', class=java.lang.Float
```

```
Value='3.0', class=java.lang.Float
```

New extension points

There are a host of new extension points in the IoC container, including:

- Additional PostPostProcessor hooks, providing further power to projects such as Pitchfork to process custom annotations or perform other operations at any point during the instantiation and configuration of Spring beans.
- The ability to add arbitrary metadata to BeanDefinition metadata. This is useful to add information that is not meaningful to Spring itself but which can be processed by frameworks built on Spring, or products integrated with Spring such as clustering products.

These topics mainly concern power users and those writing products using Spring, so are beyond the scope of this article. But it is important to understand that Spring 2.0 enhancements are not all at the surface; a lot of hard work has been done underneath.

A variety of enhancements have also been made to support integration with OSGi, enabling the Spring OSGi integration project, which integrates the power of dynamic module management with OSGi with the Spring component model. This work will continue in Spring 2.1, with the packaging of Spring's JAR files as OSGi bundles.

AOP enhancements

One of the most exciting enhancements in Spring 2.0 concerns Spring AOP, which becomes both simpler to use and far more powerful, primarily through ability to leverage functionality from the sophisticated and mature AspectJ language, while remaining a pure Java proxy-based runtime.

We have always believed that AOP (Aspect Oriented Programming) is important. Why? Because it provides us with a new way of thinking about program structure that solves many important problems which are not solved by pure OOP-enabling us to implement certain requirements in a modular, rather than scattered, fashion.

To understand the benefits, let's consider some things that we can express in requirements but not implement directly in pure OO code. Enterprise developers use a common vocabulary that enables them to communicate clearly. For example, terms such as *service layer, DAO layer, web layer or web controller* need no explanation.

Many requirements are expressed in terms of this vocabulary. For example:

- The service layer should be transactional
- When a DAO operation fails the SQLException or other persistence technology-specific exception should be translated to ensure that DAO interfaces do not provide a leaky abstraction
- Service layer objects should not call the web layer, as layers should depend only on the layer immediately below them
- An idempotent business service that fails with a concurrency related failure can be retried

While all these requirements are realistic and drawn from experience, they cannot be elegantly addressed using pure OOP. Why? There are two main reasons:

- These terms from our vocabulary make sense, but they are not *abstractions*. We cannot program using terms; we need abstractions.
- All are examples of what are called *crosscutting concerns*. A crosscutting concern, when implemented in a traditional OO approach, cuts across many classes and methods. For example, imagine applying retry logic in the event of particular exceptions being encountered across the DAO layer. This concern cuts across many DAO methods, and would require many separate modifications to implement in a traditional manner.

AOP is a technology that solves such problems, by *modularizing crosscutting concerns* and allowing us to express terms from the common vocabulary as abstractions we can program against. These abstractions are called *pointcuts*, and I will explain them in a little more detail shortly. This approach produces major benefits, such as:

- Reduction in lines of code due to the elimination of cut and paste style duplication. This is particularly beneficial in try/catch/finally idioms such as exception translation and performance monitoring.
- The ability to capture such requirements in a single code module, improving traceability.
- The ability to fix bugs in such functionality in a single place, rather than requiring revisiting many points in the application.
- Ensuring that crosscutting concerns do not obscure mainline business logic-a real danger as different concerns add up as development progresses
- Better separation of responsibilities between developers and teams. For example, retry functionality can be coded by a single developer or team, rather than requiring coding across multiple subsystems by many developers.

So AOP is important, and we wanted to offer the best solution available.

Spring AOP is undoubtedly the most widely used AOP technology. It owes this position to the following strengths:

- Near zero cost of adoption.
- Offering true pointcuts, thus deserving the term AOP rather than mere interception.
- Offering a flexible framework that supports usage in a variety of ways, programmatically and through XML.

However, prior to Spring 2.0, AOP in Spring had some drawbacks:

- Only simple pointcuts could be expressed without writing Java code. There was no *pointcut expression language* allowing sophisticated pointcuts to be expressed concisely in strings, although RegexpMethodPointcutAdvisor allowed simple regular expression-based pointcuts to be defined.
- XML configuration could become complex when configuring complex AOP usage scenarios. The generic element was used to configure the AOP classes; while this was great for consistency, offering DI and other services to aspects as well as classes, it was not as concise as a dedicated configuration approach.
- Spring AOP was not suited for advising fine-grained objects-objects need to be Spring-managed or proxied programmatically.
- The performance overhead of a proxy-based approach can be an issue in a small minority of cases.
- Because Spring AOP separates the *proxy* and the *target* (the object being decorated or *advised*), if a target method invoked a method on the target, the proxy would not be used, meaning that the AOP advice would not apply. The pros and cons of using a proxy-based approach to AOP are beyond the scope of the article: there are some definite positives (such as being able to apply different advice to different instances of the same class), but this is the major negative.

To enhance this important area in Spring 2.0, we wanted to build on its strengths, while addressing the weaknesses.

Goals

The first two weaknesses are the most significant. They both relate to pointcuts. The last three occur less often in practice for Spring users, and if they prove problematic, we recommend using AspectJ. (As you will see, this is now a straightforward progression from Spring AOP.)

The XML configuration extensions solved one of the key challenges. Because we wanted to keep the design of Spring modular, we couldn't provide AOP-specific tags in the Spring DTD in the past-hence needed to rely on generic configuration that could become verbose in this case. With Spring 2.0, this problem went away, as XML schema, unlike DTD, allows for extension. We could provide an AOP namespace that appeared to make the IoC container aware of AOP constructs, but without compromising modularity.

AOP terminology 101: Understanding pointcuts and advice

Let's briefly revise some AOP terminology. These concepts should be familiar to you if you have used Spring AOP-the concepts are identical, it's merely a different, more elegant and powerful, way of expressing them.

A **pointcut** is a matching rule. It identifies a set of points in the execution of a program where an aspect should apply. Such points are referred to as joinpoints. As an application runs, joinpoints fly by, such as the instantiation of objects and the invocation of methods. In the case of Spring AOP (all versions), the only joinpoint supported is the execution of a public method.

Advice is behavior that can be applied by aspects to joinpoints. Advice can apply before or after a joinpoint. The full range of advice types is:

- Before advice: Advice invoked before the joinpoint. For example, logging that the method call is about to take place.
- After returning advice: Advice invoked if the method at the joinpoint returns normally.
- AfterThrowing advice (called ThrowsAdvice in Spring 1.x): Advice invoked if the method at the joinpoint throws a particular exception.
- After advice: Advice invoked after the joinpoint, regardless of the outcome. Rather like finally in Java.
- Around advice: Advice that can completely control whether to invoke the joinpoint. Used, for example, to wrap a method call in a transaction, or time the execution of a method.

Aspects combine pointcuts and advice into a modular solution addressing a particular crosscutting concern.

Don't worry if this seems a bit abstract at this point: code examples will make it all clearer shortly.

> For further discussion of AOP fundamentals in the context of Spring 2.0 and AspectJ, see Adrian's excellent article on infoq, "Simplifying Enterprise Applications with Spring 2.0 and AspectJ."

Why AspectJ pointcut expressions?

So far the concepts we have discussed are fundamental AOP concepts, not specific to Spring AOP or AspectJ, and already available in Spring 1.x. So why did we choose to align with AspectJ in Spring 2.0?

Given that we needed a pointcut expression language, the choice was simple. AspectJ has a well thought out, rigorously defined and well documented pointcut language. It has recently had a comprehensive overhaul to take advantage of Java 5 syntax when running on Java 5. There is not only excellent reference material, but many books and articles explaining it.

We do not believe in reinventing the wheel, and defining our own pointcut expression language was unjustifiable. Further, it is clear that AspectJ is the only other mainstream AOP technology besides Spring 2.0, since the merger of AspectWerkz into the AspectJ project in early 2005. So critical mass was a consideration as well as technical excellence.

New XML syntax

The new AOP namespace allows Spring XML configuration to specify AspectJ pointcut expressions, targeting advice to any Spring bean with methods matched by the pointcuts.

Consider the Person class we looked at above. This has an age property, and also a birthday method, which increments that property:

```
public void birthday() {
      ++age;
 }
```

Let's suppose we have a requirement that whenever a birthday method is called, we should send a birthday card to the person in question. This is a classic example of a crosscutting requirement: it isn't part of our mainline business logic, but is a separate concern. Ideally, we want to be able to modularize that functionality without affecting the Person object.

Let's now consider the advice. Actually posting out a birthday card, or even sending an e-card, is of course going to be the main work of this method. However, for the sake of this article we're interested in the triggering infrastructure, not the mechanics of sending birthday cards. So we'll simply use console output. This method needs access to the Person whose birthday it is, and we would like it invoked whenever the birthday method is called. Our simple advice implementation is as follows:

```
public class BirthdayCardSender {
   public void onBirthday(Person person) {
      System.out.println("I will send a birthday card to " +
             person.getName() + "; he has just turned "
             person.getAge());
   }

}
```

Essentially we want a kind of Observer mechanism on the Person, but without modifying the Person to be observable. Note also that the BirthdayCardSender object is used as an aspect in this case, but does not need to implement any framework-specific interfaces. This allows the important possibility of using existing classes as aspects, further extending Spring's non-invasive programming model to embrace potential aspects as well as regular classes.

With Spring 2.0 we can use the BirthdayCardSender as an aspect as follows. First, we define the BirthdayCardSender class as a bean. This is trivial:

```
<bean id="birthdayCardSenderAspect"
      class="com.interface21.spring2.aop.BirthdayCardSender" />
```

We could dependency inject this object or apply any other Spring component model services if we wanted.

Next, we add AOP configuration to tell Spring to invoke the onBirthday() method of the BirthdayCardSender bean whenever the birthday() method is called on a Spring-managed Person. This is done using the new <aop:config> and <aop:aspect> tags. First we must import the out of the box AOP namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
 <beans xmlns="http://www.springframework.org/schema/beans"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:aop="http://www.springframework.org/schema/aop"
     xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans.xsd
     http://www.springframework.org/schema/aop
 http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">
```

Next we use the new AOP tags as follows. This is the complete configuration to apply this aspect:

```
<aop:config>
     <aop:aspect id="sendBirthdayCard" ref="birthdayCardSenderAspect">
         <aop:after
            method="onBirthday"
            pointcut="execution(void com.interface21..Person.birthday()) and
 this(person)"
         />
     </aop:aspect>
 </aop:config>
```

The <aop:after> tag applies an after advice. It specifies the method to call-onBirthday()-which is the advice. It specifies *when* to call the method-the pointcut-in the form of an AspectJ pointcut expression.

The pointcut expression is the key. Let's look at it more closely.

```
execution(void com.interface21..Person.birthday()) and this(person)
```

The execution() prefix indicates that we are matching the execution of a method. That is, we are changing the behavior of the method.

The content of the execution() clause defines the method to match. We could write an expression here that matched a set of methods, on many classes. (In fact, that is a more common and more valuable use: when advice matches just one method, externalizing it is less essential.) Finally, we *bind* the object being invoked to the argument of the onBirthday() method. This narrows the pointcut, which can only match execution on a Person object-and provides an elegant way to get at the invoked object without any lookup required. We can use argument binding to bind method parameters, return types and exceptions as well as targets if we wish. Getting rid of the lookup code should sound familiar: this is effectively *injection* of the dependency on the type being advised! In the spirit of Spring, it removes an API, incidentally it making it easy to unit test the advice method.

Now any Person defined in the Spring context will be automatically proxied *if the pointcut expression matches one or more methods*. Beans whose classes do not contain matches for this pointcut will be unaffected, as will objects matched by the pointcut that are not instantiated by the Spring container. This aspect configuration is designed to be added to the existing beans configuration shown in the IoC samples above, either in an additional XML file or in the same file. As a reminder, the Person is defined as follows:

```
<bean class="com.interface21.spring2.ioc.Person"
     p:name="Tony"
     p:age="53"
     p:house-ref="number10"
 />
```

No configuration is required to make the Person or other bean definitions eligible for advice. When we invoke the birthday() method on a Person object configured in the Spring context, we see output like the following:

```
I will send a birthday card to Tony; he has just turned 54
```

@AspectJ syntax

Advice is always contained in Java methods. But so far we have seen pointcuts defined in Spring XML.

AspectJ 5 also provides an elegant solution for defining aspects where advice is contained in methods and pointcuts in Java 5 annotations. The pointcut expression language is the same as in AspectJ's own syntax, and the semantics are identical. But in this style-called the @**AspectJ** model-aspects can be compiled using javac.

> **With @AspectJ syntax, framework-specific annotations are in the aspects, not business logic. There remains no need to introduce annotations to the business logic to drive the aspects.**

This annotation-driven programming model was pioneered by AspectWerkz, which in early 2005 merged into the AspectJ project. With AspectJ itself, @AspectJ aspects are applied by *load time weaving*: a class loader hook modifies the byte code of classes being loaded to apply the aspects as necessary. The AspectJ compiler also understands such aspects, so there is a choice of implementation strategy.

Spring 2.0 provides an additional option for @AspectJ aspects: Spring can use its proxy-based AOP runtime to apply such aspects to Spring beans.

Let's look at how the same functionality in our earlier example can be implemented using this style.

The aspect class contains the same advice method body as the BirthdayCardSender class, but is annotated with the org.aspectj.lang.annotation.Aspect annotation to identify it as an aspect. Further annotations in the same package define advice methods.

```
@Aspect
public class AnnotatedBirthdayCardSender {

   @After("execution(void com.interface21..Person.birthday()) and this(person)")
  public void onBirthday(Person person) {
     System.out.println("I will send a birthday card to " +
           person.getName() + "; he has just turned " +
           person.getAge());
  }
}
```

This combines the pointcut with the advice method, making the aspect a complete module. @AspectJ aspects, like all AspectJ aspects, can contain any number of pointcuts and advice methods.

In Spring XML, we define this as a bean again, and add an additional tag to cause aspects to be applied automatically:

```
<aop:aspectj-autoproxy />

   <bean id="birthdayCardSenderAspect"
      class="com.interface21.spring2.aop.AnnotatedBirthdayCardSender" />
```

The aspectj-autoproxy tag tells Spring to recognize @AspectJ aspects automatically and apply them to any beans in the same context that their pointcuts match.

Choosing between XML and @AspectJ style

Which of these methods-XML or annotation-is best? Firstly, as the annotations are in the aspects, not core business logic, the cost of switching isn't high. The decision usually comes down to whether it makes sense to externalize the pointcut expression from Java code altogether.

Consider the annotation style if your aspect is domain-specific: that is, the pointcuts and advice are closely linked, and the advice is not generic and likely to be used repeatedly in different scenarios. For example, birthday card sending is a good candidate for the annotation style, as it's specific to a particular application class (Person). However, a performance monitoring aspect might be used differently in different applications and the advice methods might best be decoupled from the pointcuts, with pointcuts living more naturally in XML configuration.

Use XML if:

- You are unable to use Java 5, and have no choice. Spring 2.0's AOP enhancements, except for processing @AspectJ syntax, work on Java 1.3 and 1.4 as well as Java 5, although you won't be able to write pointcut expressions matching annotations or other Java 5 constructs.
- You might want to use the advice in different contexts.
- You want to use existing code as advice, and don't want to introduce AspectJ annotations into it: for example, introducing an Observer behavior invoking a method on an arbitrary POJO.

Programmatic usage

You can also create AOP proxies programmatically, using @AspectJ aspects, as follows:

```
Person tony = new Person();
tony.setName("Tony");
tony.setAge(53);

AspectJProxyFactory ajpf = new AspectJProxyFactory(tony);
ajpf.addAspect(new AnnotatedBirthdayCardSender());
Person proxy = ajpf.getProxy();
```

AnnotatedBirthdayCardSender will automatically be recognized as an @AspectJ aspect, and the proxy will decorate the target with the behavior it defines. This style does not require the Spring IoC container.

Programmatic proxy creation is useful when writing infrastructure code and tests, but is not often used in regular business applications.

Cool things you can do with AspectJ pointcuts

What we've seen so far has just scratched the surface.

Let's look at some of the more advanced capabilities of AspectJ pointcut expressions, in Spring XML, @AspectJ style (or, of course, the AspectJ language itself):

- Parameter, target, exception and return value binding.
- Benefiting from type safety, where types in the method signature are specified in the pointcut.
- Composition of pointcut expressions to build sophisticated expressions.
- Reuse of pointcut expressions.

We've already seen target binding. Let's take an example of parameter binding:

```
@Aspect
public class ParameterCaptureAspect {

  @Before("execution(* *.*(String, ..)) && args(s)")
  public void logStringArg(String s) {
    System.out.println("String arg was '" + s + "'");
  }
}
```

The args() clause binds to argument(s) in methods being matched, implicitly narrowing to methods that have a first argument of type String. Because the pointcut binds the first argument, which must be of type String, casting is unnecessary in the advice method.

Type safety flows naturally from this mechanism. Advice targeted with this pointcut can never be invoked inappropriately, with an argument of the wrong type,

or no matching argument.

To give an example of the superiority of the mechanism, here's how this would look in a Spring 1.x MethodBeforeAdvice. As with an AOP Alliance MethodInterceptor (the most common interface implemented in Spring 1.x applications), we need to look into an array of arguments to find the argument we're looking for:

```
public class ParameterCaptureInterceptor implements MethodBeforeAdvice {
   public void before(Method method, Object[] args, Object target) throws Throwable {
      if (args.length >= 1 && method.getParameterTypes()[0] == String.class) {
          String s = (String) args[0];
          System.out.println("String arg was '" + s + "'");
      }
   }
}
```

We could use a Spring AOP Pointcut to do away with the guard in the MethodBeforeAdvice, but as mentioned before, this would probably require writing Java code. Having a guard in the interceptor is slower than using a pointcut, because it does not allow the AOP runtime to optimize out advice that can never be called.

Here we can see how far true AOP is removed from interception, and why it's both simpler and more powerful. EJB 3.0 interception is significantly worse again than Spring's first generation AOP functionality, as it lacks a real pointcut mechanism, meaning that both ClassCastException and ArrayIndexOutOfBoundsException are likely risks. It would also be necessary to use Around advice (an interceptor) rather than Before advice, because EJB 3.0 does not offer specialized advice types. Also, the need to provide an InvocationContext object makes it much harder to unit test advice methods.

> **The power of the AspectJ pointcut expression language is not merely about allowing sophisticated constructs. It also plays an important role in avoiding the potential for errors and making applications more robust. It can also significantly reduce the amount of code required, by removing the need for guard code in advice methods.**

Composition and reuse are characteristics of real languages. One of AspectJ's primary goals is to offer them for pointcut expressions.

Lets look at pointcut reuse in practice.

@AspectJ syntax (like Spring 2.0's AOP XML format) allows us to define named pointcuts. In @AspectJ syntax, we use the @Pointcut annotation on a void method, as follows:

```
@Pointcut("execution(public !void get*())")
 public void getter() {}
```

The @Pointcut annotation allows us to define pointcut expressions and, where necessary, the number and type of arguments to be bound by the pointcut. The method name is used as the name of the pointcut.

The above pointcut matches JavaBean getters. Note the strength of the matching here: we are not merely dealing with wildcards, but language semantics. A naïve approach would view a getter as any method whose name starts with "get". This pointcut is much more robust as it asserts that a getter is public, has a non-void return (!void) and no arguments (indicated by the empty parentheses for arguments).

Let's add a pointcut to match methods returning int:

```
@Pointcut("execution(public int *())")
 public void methodReturningInt() {}
```

Now we can express advice in terms of these pointcuts. Our first example simply references our first named pointcut, "getter":

```
@After("getter()")
   public void getterCalled(JoinPoint jp) {
      System.out.println("Method "  + jp.getSignature().getName() +
            " is a getter");
   }
```

However, now things get interesting. Here we apply advice to a getter that returns int by ANDing our two pointcuts into a single expression:

```
@After("getter() and methodReturningInt()")
 public void getterCalledThatReturnsInt(JoinPoint jp) {
    System.out.println("ANDing of pointcuts: Method " +
   jp.getSignature().getName() +
          " is a getter that also returns int");
 }
```

ANDing means that both pointcuts must apply. ORing means that one or other must apply. We can build expressions of whatever complexity we require.

Both ANDing and ORing are demonstrated in the following complete aspect:

```
@Aspect public class PointcutReuse {

    @Pointcut("execution(public !void get*())" )
   public void getter() {}

    @Pointcut("execution(public int *())" )
   public void methodReturningInt() {}

    @Pointcut("execution(public void *(..))" )
   public void voidMethod() {}

    @Pointcut("execution(public * *())" )
   public void takesNoArgs() {}

    @After("methodReturningInt()" )
   public void returnedInt(JoinPoint jp) {
     System.out .println("Method "  + jp.getSignature().getName() +
           " returned int" );
 }

    @After("getter()")
   public void getterCalled(JoinPoint jp) {
     System.out .println("Method "  + jp.getSignature().getName() +
           " is a getter" );
 }

    @After("getter() and methodReturningInt()" )
   public void getterCalledThatReturnsInt(JoinPoint jp) {
     System.out.println("ANDing of pointcuts: Method " +
jp.getSignature().getName() +
           " is a getter that also returns int");
 }

    @After("getter() or voidMethod()" )
   public void getterOrVoidMethodCalled(JoinPoint jp) {
     System.out .println("ORing of pointcuts: Method "  +
jp.getSignature().getName() +
           " is a getter OR is void" );
   }

  }
```

This will produce the following output, showing ORing and ANDing of pointcut expressions:

```
Method getName is a getter
 ORing of pointcuts: Method getName is a getter OR is void
 ORing of pointcuts: Method birthday is a getter OR is void
 Method getName is a getter
 ORing of pointcuts: Method getName is a getter OR is void
 Method getAge returned int
 Method getAge is a getter
 ANDing of pointcuts: Method getAge is a getter that also returns int
 ORing of pointcuts: Method getAge is a getter OR is void
 I will send a birthday card to Tony; he has just turned 54
```

Pointcut composition can also be done in Spring AOP XML. In that case, use "and" and "or" in place of "&&" and "||" operators to avoid problems with XML attribute values.

Reusing AspectJ library aspects for power users

It is possible to reuse AspectJ pointcut expressions written in the AspectJ language itself and compiled into a JAR file. If you use Eclipse, you can develop such aspects using the AJDT plugin. Alternatively, if you're already using AspectJ, you may already have such aspects and want to reuse them.

As an illustrate, I'll rewrite part of our earlier example, putting the pointcut in an AspectJ aspect:

```
public aspect LibraryAspect {

    pointcut getter() :
                execution(public !void get*());
 ...
 }
```

This aspect is written in the Aspect language, so needs to be compiled by the AspectJ ajc compiler. Note that we can use keywords for *aspect* and *pointcut*.

We can reference this aspect in an @AspectJ aspect to be used in Spring as follows. Note that we use the FQN of the aspect, which will normally be packaged in a JAR file on the classpath:

```
@Aspect
 public class PointcutReuse {
    @After("mycompany.mypackage.LibraryAspect.getter()")
   public void getterCalled(JoinPoint jp) {
      System.out.println("Method " + jp.getSignature().getName() +
             " is a getter");
   }
```

This class, on the other hand, can be compiled using javac and applied by Spring.

You can also reference AspectJ aspects in Spring XML. As you can see, Spring 2.0 AOP and AspectJ are very closely integrated, although Spring AOP provides a complete runtime without the need to use the AspectJ compiler or weaver.

Using AspectJ library aspects is a best practice if you have very complex pointcut expressions, as in that case the AspectJ language and tool support is compelling.

Best practices

So what does this mean to Spring users?

Hopefully you agree that AOP solves important problems in enterprise software, and you have realized how much more powerful and elegant the AspectJ programming model is compared to any alternative for interception or AOP.

You don't need to migrate your existing Spring MethodInterceptors or other advice implementations to the new programming model: they will still work fine. But going forward, the new programming model should be preferred and is far more compelling.

> **If you're using ProxyFactoryBean or TransactionProxyFactoryBean to configure proxies one at a time, you will find autoproxying (which becomes much easier and more natural in Spring 2.0) can significantly reduce the volume of your Spring configuration, and better divide work between team members.**

What does the runtime look like?

While the usage model looks different, it's important to remember that concepts-joinpoint, pointcut and advice-are exactly the same as in Spring AOP and the AOP Alliance API that Spring has implemented since 2003. Spring AOP has always had interfaces for these concepts.

Perhaps more surprisingly, the implementation it's actually pretty much the same under the covers. The new programming model, leveraging AspectJ constructs, is built on top of the existing Spring AOP runtime. Spring AOP has always been very flexible, so this required no significant changes. (The org.springframework.aop.framework.Advised interface can still be used to query and modify the state of AOP proxies, as in Spring 1.x.)

This means that you can mix and match AOP Alliance and Spring AOP aspects with AspectJ-style aspects: particularly important if you want to leverage existing aspects.

Let's add to the example of programmatic proxy creation to illustrate this. We will add a traditional Spring AOP style MethodInterceptor:

```
Person tony = new Person();
 tony.setName("Tony");
 tony.setAge(53);

  AspectJProxyFactory ajpf = new AspectJProxyFactory(tony);
 ajpf.addAspect(new AnnotatedBirthdayCardSender());
 Person proxy = ajpf.getProxy();
 ajpf.addAdvice(new MethodInterceptor() {
 public Object invoke(MethodInvocation mi) throws Throwable {
   System.out.println("MethodInterceptor: Call to " + mi.getMethod());
   return mi.proceed();
   }
 });
```

This produces the following output, with output from the MethodInterceptor (which, without a pointcut, matches all method invocations) interspersed with output from the BirthdayCardSender written in @AspectJ style.

```
MethodInterceptor: Call to public void
 com.interface21.spring2.ioc.Person.birthday()
 MethodInterceptor: Call to public java.lang.String
com.interface21.spring2.ioc.Person.getName()
 MethodInterceptor: Call to public int
com.interface21.spring2.ioc.Person.getAge()
 I will send a birthday card to Tony; he has just turned 54
```

Toward AOP unification

Spring 2.0 brings a new and welcome unification to the world of AOP. For the first time, the implementation of an aspect is independent of its deployment model. Every one of the @AspectJ examples we've seen can be compiled using the AspectJ compiler or applied using AspectJ load time weaving, as well as being applied by Spring. In the spirit of Spring, we have one programming model that can span different runtime scenarios.

And if you wish to adopt AspectJ itself, there is a natural progression due to Spring bringing AspectJ pointcut expression concepts to a wider audience.

When should you use AspectJ? The following are some of the indications:

- You want to advise fine grained objects, which may not be instantiated by the Spring container.
- You want to advise joinpoints other than the execution of public methods, such as field access or object construction.
- You want to advise self-invocation in a transparent way.
- When an object needing to be advised is invoked very many times and no proxy performance overhead is acceptable. (Be careful to benchmark before making a decision on this basis: the overhead of Spring AOP proxying is all but undetectable in normal use.)
- You want to use AspectJ's ability to declare warnings or errors to be flagged by the compiler. This is particularly useful for architectural enforcement.

It's not necessarily an either/or choice. It is possible to use both AspectJ and Spring AOP simultaneously: they do not conflict.

Java 5

Spring 2.0 remains backward compatible with Java 1.3 and 1.4. However, an increasing number of new features target Java 5.

Some of these, such as type inference, as discussed, come for free. Others require a choice on your part. Let's quickly look at some of them.

New APIs

A number of new APIs offer Java 5 functionality over core functionality that continues to run on earlier versions of Java.

In particular:

- SimpleJdbcTemplate: A new class alongside the familiar JdbcTemplate, this makes JDBC usage still simpler.
- AspectJProxyFactory: A new class alongside ProxyFactory that is designed for programmatic creation of proxies using @AspectJ aspects.

The number of such classes will increase over time.

SimpleJdbcTemplate is illustrative. Let's look at its effect on calling an aggregate function. Using JdbcTemplate prior to Java 5, we need to wrap bind variables in an array, as follows:

```
jdbcTemplate.queryForInt("SELECT COUNT(0) FROM T_CLIENT WHERE TYPE=? AND CURRENCY=?",
   new Object[] { new Integer(13), "GBP" }
 );
```

If we are using Java 5, autoboxing removes a little of the noise, as we no longer need the primitive wrapper type. This simply comes from the language feature, rather than requiring Spring to offer anything new:

```
jdbcTemplate.queryForInt("SELECT COUNT(0) FROM T_CLIENT WHERE TYPE=? AND CURRENCY=?",
   new Object[] { 13, "GBP" }
 );
```

However, by embracing Java 5 we can get rid of the need for an object array altogether. The following example shows how SimpleJdbcTemplate uses varargs for bind variables, meaning that the developer can provide any number of them without an array:

```
simpleJdbcTemplate.queryForInt("SELECT COUNT(0) FROM T_CLIENT WHERE TYPE=? AND CURRENCY=?",
   13, "GBP"
 );
```

As an added benefit, we no longer need to distinguish between the case with bind variables and that without. While this required two methods on JdbcTemplate, to avoid the need to pass an empty Object array to the overloaded method taking literal SQL, with SimpleJdbcTemplate the framework code can check the length of the varargs. Thus the following example invokes the same method:

```
simpleJdbcTemplate.queryForInt("SELECT COUNT(0) FROM T_CLIENT");
```

There are also more significant benefits. Generics make signatures clearer and eliminate casts. For example, the queryForMap() method on JdbcTemplate returns a Map from column name to column value in the ResultSet. This becomes much clearer when it is an explicit part of the method's signature on SimpleJdbcTemplate:

```
public Map<String, Object> queryForMap(String sql, Object... args)
                        throws DataAccessException
```

This is still clearer with the method that returns a list of such maps:

```
public List<Map<String, Object>> queryForList(String sql, Object ... args)
                               throws DataAccessException
```

An added goal of SimpleJdbcTemplate is to provide only those methods that are used most frequently. JdbcTemplate is one of the biggest classes in Spring and has many methods, some of which are for more esoteric purposes. In such advanced cases, language syntax sugar is likely to be less important, in terms of the total problem complexity. For such cases, SimpleJdbcTemplate wraps a JdbcTemplate instance that can be accessed through the getJdbcOperations() method.

To support the usage model of extending a DAO support class, Spring 2.0 provides the SimpleJdbcDaoSupport class, offering a preconfigured JdbcTemplate. SimpleJdbcTemplate is also just as easy to instantiate or inject directly as JdbcTemplate, merely providing a javax.sql.Datasource implementation-the starting point for all Spring's support for relational data access. Like JdbcTemplate, SimpleJdbcTemplate can be used as a class library without other use of Spring.

Annotations

We first introduced annotations, as an optional feature, in Spring 1.2, and we've gradually added more.

I've already mentioned the use of annotations for AOP, from AspectJ. These provide an elegant way of expressing pointcuts and advice in a single code module.

Spring also provides a number of its own annotations, for areas such as transaction management. The following were introduced in 1.2:

- @Transactional: Marks a class, interface or method as transactional.
- Various annotations including @ManagedResource in the org.springframework.jmx.export.annotation package, identifying operations, attributes and objects to export for JMX management.

The following are the most significant new annotations in 2.0:

- @Configurable: Indicates that a particular object should be dependency injected using Spring after construction, although it was not instantiated by Spring. Drives an AspectJ DI aspect, described in Part II of this article.
- @Required: Indicates a required JavaBean setter method. To activate this enforcement, define a RequiredAnnotationBeanPostProcessor in your application context. *In the spirit of its non-invasive programming model, and ability to work with existing classes, Spring can also enforce the use of other annotations to indicate a required property, by configuration of the RequiredAnnotationBeanPostProcessor.*
- @Repository: Identifies a DAO object as representing the Repository pattern (in Domain Driven Design terminology). Spring 2.0 provides an aspect (PersistenceExceptionTranslationAdvisor) that can automatically convert technology-specific exceptions from objects annotated with @Repository into Spring's generic DataAccessException hierarchy.

The Spring integration testing superclasses for JPA testing, such as the new AbstractJpaTest and the generic superclass AbstractAnnotationAwareTransactionalTests, now provide support for annotations such as @Repeat (causing a test to be repeated) and @ExpectedException

(indicating that the test should throw a particular exception, and fail if it does not). Unfortunately due to JUnit 3's design being based on concrete inheritance, these useful annotations are not available to other tests using Spring. As takeup of JUnit 4 increases we will provide a version of our integration tests that should be able to open this functionality up to other users.

What if you want to interpret your own annotations? Of course in this case, Spring's many extension hooks will help out. For example, you could write a BeanPostProcessor that identifies methods with a given annotation, in the same way as the RequiredAnnotationBeanPostProcessor works. The Pitchfork project (http://www.interface21.com/pitchfork), used in the forthcoming WebLogic 10, implements JSR-250 annotations and EJB 3.0 interception annotations on top of Spring using these extension points.

It's also worth noting that the AspectJ pointcut expression language offered in Spring 2.0 has extremely powerful annotation matching. It's easy to write pointcuts that target annotations. For example, the following pointcut expression will match any method annotated with an annotation in a Spring framework package:

```
execution(@(org.springframework..*) * *(..))
```

The following expression will match any class that is annotated with the @Transactional annotation:

```
@within(org.springframework.transaction.annotation.Transactional)
```

AspectJ 5 was a major extension of the AspectJ language and an impressive engineering effort to keep up to date with the evolution of the underlying language, and pointcut expressions can also match other Java 5 constructs such as generics and varargs.

To learn more

In the next article in this series, I will be discussing:

- Dynamic language support, and how the Spring component model becomes cross-language in Spring 2.0.
- Messaging and asynchronous invocation support.
- Data access and integration with the new Java Persistence API (JPA).
- Spring MVC enhancements, mainly focused around ease of use.
- The new Spring Portlet MVC framework.
- The new possibility of Dependency Injection for domain objects.

In the meantime, to read more about the topics I've discussed today, I suggest the following resources:

- The Spring Reference Manual, which has always been pretty good but has been dramatically improved in Spring 2.0. http://static.springframework.org/spring/docs/2.0.x/reference/index.html.
- To better understand the power of AspectJ, there are several good AspectJ books. I recommend AspectJ in Action by Ramnivas Laddad (Manning, 2003) and Eclipse AspectJ by Adrian Colyer, Andy Clement, George Harley and Matthew Webster (Addison-Wesley, 2005).
- To understand the changes in AspectJ, a number of books are on the way, but in the meantime the AspectJ Development Kit Developer's Notebook is very useful, especially for the chapter on "An Annotation Based Development Style". See http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html

The sample code accompanying this article can be downloaded HERE.

4 comments **Watch Thread** **Reply**

Acegi by craig schneider Posted Jan 19, 2007 10:25 AM
Excellent article, Rod! by David Main Posted Jan 24, 2007 9:10 AM
2nd article by Martin Gilday Posted Jan 28, 2007 12:05 PM
nicest article but... by Joshua H. Song Posted Feb 9, 2007 1:03 AM

Sort by date descending

Acegi

Jan 19, 2007 10:25 AM by **craig schneider**

> Many products build on Spring, and their configuration can become simpler using namespaces. A good example is Acegi Security for Spring (to be rebranded Spring Security in early 2007)

Where can I find out more information on this? Are the Acegi package names going to change again and will they be included in spring.jar?

Reply

Excellent article, Rod!

Jan 24, 2007 9:10 AM by **David Main**

I just wanted to say thanks for putting this together, Rod. Besides providing ever more details on the uses and value of Spring, it is well written... which seems to characterize much of the documentation associated with Spring. And that makes Spring all the more accessible.

Reply

2nd article

Jan 28, 2007 12:05 PM by **Martin Gilday**

A very useful article. Even though we have been using Spring 2 for some time now this highlighted some areas we had missed. Will the 2nd article also be posted on InfoQ?

Reply

nicest article but...

Feb 9, 2007 1:03 AM by **Joshua H. Song**

your story about spring 2.0 reminds me of this short story.. A: do you know how to store an elephant in your fridge? B: i don't know A: simple. first, open the fridge's door, second, push the elephant into the fridge, third, close the door. B: WOW. is spring 2.0 really worthy to be versioned as 2.0? oohhhh i'm getting fuzzy in this world of two dot o.

Reply