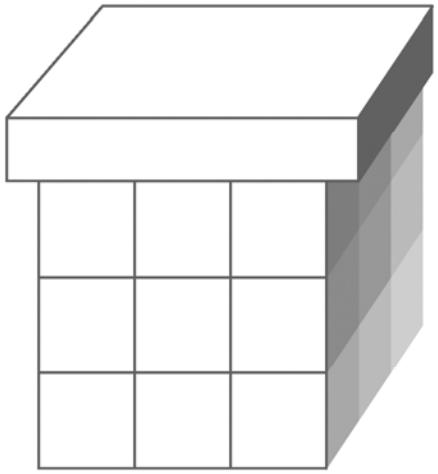
Chapter 5

Assembling Object Models



OBJECTIVES

After completing "Assembling Object Models," you will be able to:

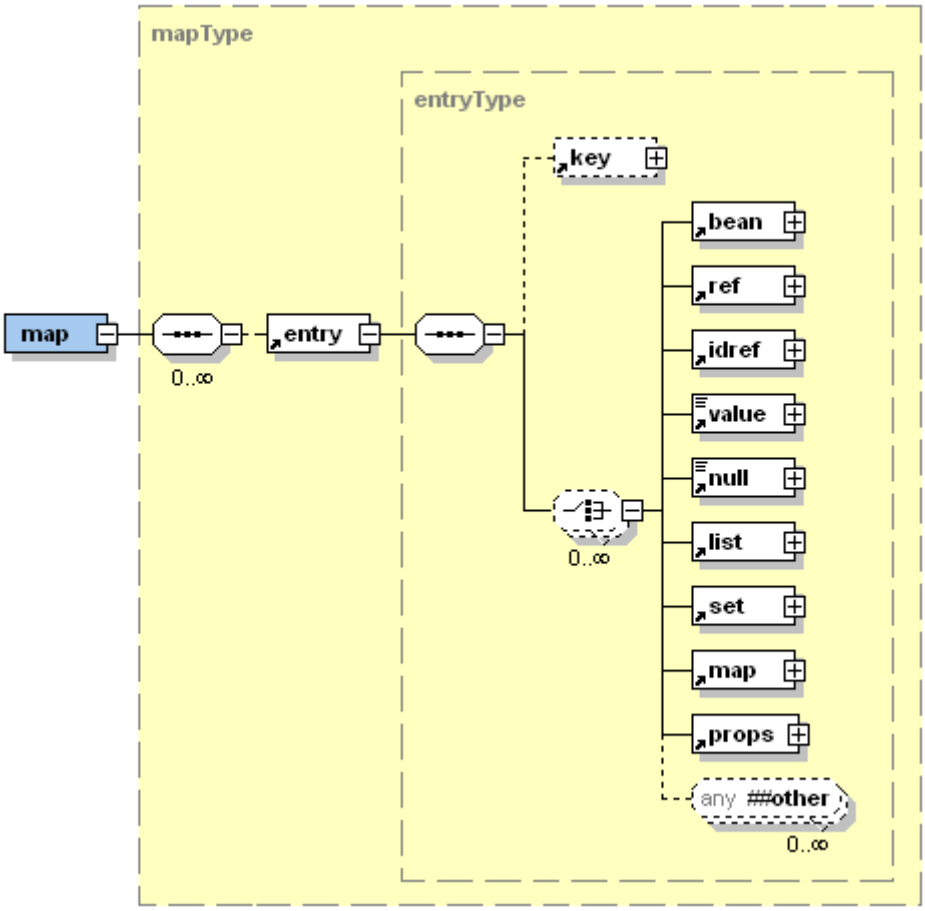
- **Populate bean properties that are collections and maps.**
- **Refer to sub-objects of defined beans using compound property names.**
- **Control the order in which declared beans are instantiated.**

Collections and Maps

- **Spring offers declarative support for properties that are collections or maps of values.**
- **Note that using compound property naming would not work here, because that syntax will never instantiate new objects.**
  - It assumes that the tree is already constructed and just lacks various leaf values; it assumes that any indexed property has already been allocated or "dimensioned" to its correct size.
  - This will be an issue later, when we work with command objects.
- **Spring supports various Collections API types, each by its own element name, used as a child of a property, constructor argument, or other collection type:**
  - **`<list>`** populates a **List**, with child elements **`<value>`**, **`<bean>`**, **`<ref>`**, or another collection type
  - **`<set>`** populates a **Set** with the same possible child elements
  - **`<map>`** populates a **Map**; this model is a little more complex
  - **`<props>`** populates a **Properties** object, much the way a map works but with fewer options

Populating Maps

- The model for the **<map>** element involves a child element for each **<entry>**:



  - These in turn have keys (either a **key** attribute or a **<key>** child element) and values (any of the usual suspects, as with lists).
- The **<props>** element can be populated in a similar fashion to a map object, with **<prop>** children instead of **<entry>**s.

Support for Generics

- **Spring starting with version 2.5 has solid support for Java-5.0 collection types that take advantage of generics.**
- **There is a real challenge here, and Spring 2.0 had some trouble:**
  - Remember that type parameters are **erased** at runtime – by the time the container and bean factory are in play, their target types are plain old **List**s and **Map**s, even if they were **List<String>** and **Map<Long,Set<String>>** when they were compiled.
  - It's not feasible to declare the full generic type of a collection as a class name in the configuration – after all, the angled brackets would be misinterpreted as XML markup!
- **There are ways around both of these difficulties, but even in spite of them, Spring will do a pretty good job of recognizing type parameters.**
  - If you use **`<list>`** to populate a **List<Integer>**, Spring will convert your numeric values to **Integer** objects – even though at runtime it could get away with piling up **String** objects that would later get you in trouble.
  - There are theoretical limits to Spring's powers of perception, but even **nested collection types** resolve neatly, as we'll see in a moment.

Declaring and Using Collections

EXAMPLE

- See **Collections_Step1** for a simple example of declaring the contents of collection-type properties.
- The class **cc.Holder** defines three properties:

```
List<String> names;
Map<String,Integer> frequencies;
Map<String,Set<String>> errors;
```

- The configuration file **Collections.xml** makes short work of this:

```
<bean id="Holder" class="cc.Holder" >
  <property name="names" >
    <list>
      <value>Sam</value>
      <value>Fred</value>
      <value>Jane</value>
    </list>
  </property>
  <property name="frequencies" >
    <map>
      <entry key="Rock" value="787" />
      <entry key="Paper" value="662" />
      <entry key="Scissors" value="1090" />
    </map>
  </property>
  ...
```

Declaring and Using Collections

EXAMPLE

– It even handles the nested type, a map of strings and sets of strings:

```
...
<property name="errors" >
  <map>
    <entry key="firstName" >
      <set>
        <value>Field is required</value>
      </set>
    </entry>
    <entry key="lastName" >
      <set>
        <value>Field is required</value>
      </set>
    </entry>
    <entry key="age" >
      <set>
        <value>Field is required</value>
        <value>Must be a positive number</value>
      </set>
    </entry>
  </map>
</property>
</bean>
```

- **cc.test.TestCollections** does very little, just instantiates the bean and dumps its values to the console. Run it now:

```
[Sam, Fred, Jane]
{Rock=787, Paper=662, Scissors=1090}
{firstName=[Field is required], lastName=[Field is
required], age=[Field is required, Must be a
positive number]}
```

A New Policy Language

DEMO

- We'll put our newfound facility with collections to work.
- The Java security architecture specifies a Java-like grammar for policy files:

```
grant codeBase "file:This.jar" signedBy "Fred"
{
  permission java.io.FilePermission
    "SomeFile.txt", "read";
  permission java.util.PropertyPermission
    "java.home", "read";
};
```

- Maybe an XML vocabulary would be better!
- Do your work in **Policy_Step1**.
  - The completed demo is in **Policy_Step2**.

1. If you like, review the code in **src/cc/security/DynamicPolicy.java**. The full implementation of a Java SE **Policy** is not important, but notice that the behavior of the class is based on a property **permissionsMap** that is a **Map<CodeSource,List<Permission>>**.
2. In **src/cc/security/test/TestPolicy.java**, the **main** method sets an instance of this class as the local security policy, installs a security manager, and then attempts to read a system property. Since reading system properties is a checked action, the policy will have to grant permission to the controller class for this action to succeed.

```
Policy.setPolicy
  ((Policy) factory.getBean ("SecurityPolicy"));
System.setSecurityManager (new SecurityManager ());

System.out.println ("Property java.home is " +
    System.getProperty ("java.home"));
```

A New Policy Language

DEMO

3. Run this class as a Java application, and see that the action does fail:

```
Checking (java.util.PropertyPermission java.home
read) for code base (file:/C:/Capstone/Spring/Demos
/Policy/build/classes/ <no signer certificates>)
Exception in thread "main"
java.security.AccessControlException: access denied
(java.util.PropertyPermission java.home read)
```

4. If you open **Policy.xml**, you'll see why: the policy bean is declared, but the **permissionsMap** is not populated, so it sits empty when the access controller checks for the necessary permission.

5. Now we'll start building up a declaration of a specific security policy. Start by setting the **permissionsMap** property to a map with one empty entry:

```
<bean id="SecurityPolicy"
      class="cc.security.DynamicPolicy" >
  <property name="permissionsMap" >
    <map>
      <entry>
      </entry>
    </map>
  </property>
</bean>
```

A New Policy Language

DEMO

6. Define the key for the entry as a **CodeSource** object. To create this bean you have to provide constructor arguments: the location of the code source as a string, and the list of parties who've digitally signed the code source. This latter argument is null, so we'll use a construct we haven't yet studied, but a very simple one: null is **<null/>**.

```
<entry>
  <key>
    <bean class="java.security.CodeSource" >
      <constructor-arg value=
        "file:/C:/Capstone/Spring/Demos
                  /Policy/build/classes/"
      />
      <constructor-arg >
        <null/>
      </constructor-arg>
    </bean>
  </key>
</entry>
```

– WARNING: case matters in the URL above – even the capital 'C'!

7. Grant the necessary permission to this code source, as one element in a list:

```
    </key>
    <list>
      <bean class=
              "java.util.PropertyPermission" >
        <constructor-arg value="java.home" />
        <constructor-arg value="read" />
      </bean>
    </list>
</entry>
```

A New Policy Language

DEMO

8. Test again, and see how your policy declaration flies:

```
(ouch) (oh, that's rude)
Failed to convert value of type [java.lang.String]
to required type
[java.security.cert.Certificate[]]; nested
exception is java.lang.IllegalArgumentException: No
matching editors or conversion strategy found
(oops)
```

9. The bean factory really didn't like something that we did! The gist of the error report that we get back is shown above: couldn't convert a string to an array of certificates ... ?

10. If you look at the constructors for **CodeSource**, there are two constructor overloads – one of which takes an array of certificates:

```
javap java.security.CodeSource

public class java.security.CodeSource extends
java.lang.Object implements java.io.Serializable{
    public java.security.CodeSource(java.net.URL,
java.security.cert.Certificate[]);
    public java.security.CodeSource(java.net.URL,
java.security.CodeSigner[]);
```

11. We don't really care which constructor we call, since we just want to pass **null** for either the certificate array or the signer array. But Spring cares! It cares deeply, and it can't make the decision based on a string value and a null.

A New Policy Language

DEMO

12. You'd figure a **type** attribute would be the solution here; but, strangely, what works is an **index** attribute. Set this as shown:

```
<constructor-arg value=
  "file:/C:/Capstone/.../Policy/build/classes/"
/>
<constructor-arg index="1" >
  <null/>
</constructor-arg>
```

13. Test now, and you should see that the policy is instantiated correctly – and, what's more, it works!

```
Property java.home is c:\Java7\jre
```

The Spring Utility Schema

- **Spring includes a number of other XML models for content that can appear in a beans configuration file, thanks to Spring's support for XML namespaces as extension points.**
- **We'll see models for AOP and transactions later in this course.**
- **Now, we'll look at the utility schema, which makes possible a few new configurations:**
  - **Beans that are collections** – note that so far we've only seen beans that hold collections as properties
  - Using a **property** of one bean **to initialize** a property on a another
  - Gaining access to **constants** defined on a Java class and treating those values as configurable beans
- **A configuration file that uses these new constructs must declare a namespace prefix for this separate schema and identify the schema location:**

```
<beans
  xmlns="...beans"
  xmlns:util=
    "http://www.springframework.org/schema/util"
  xmlns:xsi="..."
  xsi:schemaLocation=
    "http://www.springframework.org/schema/beans
      http://www.springframework.org/schema
                        /beans/spring-beans-2.5.xsd
     http://www.springframework.org/schema/util
      http://www.springframework.org/schema
                        /util/spring-util-2.5.xsd"
>
```

Sharing Information

EXAMPLE

- **What if we wanted to configure a second instance of our three-collection Holder class, and for it to have its own data for one of the collections but the same data for the other two?**
  - We could just clone the whole bean config and edit from there – a/k/a copy-and-paste "reuse."
  - It would be much better, more maintainable, to define those two shared collections just once each, and then re-use the information.
  - We'll look at two ways of doing that.
- **In Collections_Step2 are two new configuration files.**
- **SeparateBeans.xml defines each of the three collections (that were inner beans in the "Holder" bean before) as first-class beans of their own, using the utility vocabulary:**

```
<util:list id="globalNames" >
  <value>Sam</value>
  <value>Fred</value>
  <value>Jane</value>
</util:list>

<util:map id="globalFrequencies" >
  <entry key="Rock" value="787" />
  <entry key="Paper" value="662" />
  <entry key="Scissors" value="1090" />
</util:map>

<util:map id="globalErrors" >
  ...
```

Sharing Information

EXAMPLE

- The utility schema defines **<util:list>**, **<util:set>**, and **<util:map>**.
- Each has the same effect: it defines whichever type of collection object as a named bean, rather than as a property of some other bean, and populates the collection with the same child elements as **<list>**, **<set>**, and **<map>** as shown earlier.
- This makes it possible to configure the "Holder" bean using bean references:

```
<bean
  id="Holder"
  class="cc.Holder"
  p:names-ref="globalNames"
  p:frequencies-ref="globalFrequencies" />
  p:errors-ref="globalErrors"
/>
```

  - ... and for a second bean "Reuser" to share two of them while defining its own **frequencies** map:

```
<bean id="Reuser" class="cc.Holder" >
  <property name="names" ref="globalNames" />
  <property name="frequencies" >
    <map>
      <entry key="Heads" value="501" />
      <entry key="Tails" value="499" />
    </map>
  </property>
  <property name="errors" ref="globalErrors" />
</bean>
```

Sharing Information

EXAMPLE

- **Build and test using program arguments to identify the configuration file and bean to instantiate and dump:**

```
run SeparateBeans.xml Holder

[Sam, Fred, Jane]
{Rock=787, Paper=662, Scissors=1090}
{firstName=[Field is required], lastName=[Field is
required], age=[Field is required, Must be a
positive number]}

run SeparateBeans.xml Reuser

[Sam, Fred, Jane]
{Heads=501, Tails=499}
{firstName=[Field is required], lastName=[Field is
required], age=[Field is required, Must be a
positive number]}
```

- **CrossReference.xml takes a different approach, using the `<util:propertyPath>` element.**
  - This uses a compound property expression in its **path** attribute to find a bean property defined elsewhere in the configuration.
  - It then exposes that property as a named bean, which can then be instantiated by the factory, or referenced normally by other beans.

Sharing Information

EXAMPLE

- So, in this file, the "Holder" bean is the same as it was in **Step1**.
- The two collections we want to share are exposed as beans:

```
<util:property-path id="usefulNames"
                    path="Holder.names" />
<util:property-path id="usefulErrors"
                    path="Holder.errors" />
```

- The new "Reuser" bean now has something to latch onto using a property reference:

```
<bean id="Reuser" class="cc.Holder" >
  <property name="names" ref="usefulNames" />
  <property name="frequencies" >
    <map>
      <entry key="Heads" value="501" />
      <entry key="Tails" value="499" />
    </map>
  </property>
  <property name="errors" ref="usefulErrors" />
</bean>
```

- Test this configuration and see we get the same effect:

```
run CrossReference.xml Reuser

[Sam, Fred, Jane]
{Heads=501, Tails=499}
{firstName=[Field is required], lastName=[Field is
required], age=[Field is required, Must be a
positive number]}
```

The Firing Sequence

LAB 5A

Suggested time: 15-30 minutes

In this lab you will improve the CMS from the previous lab, by making the firing sequence configurable using a list bean. The controller code will reduce to a very simple loop over the configured bean, which in turn will link in the transformers and pipes.

In optional steps you can also flesh out the full set of CMS pathways, adding a number of XML-to-HTML transformations in parallel with the text formatting you've already done.

Detailed instructions are found at the end of the chapter.

Autowiring to Multiple Beans

EXAMPLE

- A neat trick we've not yet had opportunity to demonstrate involves auto-wiring an array or list of object references.
- An **autowire**d (or **@Autowired**) property usually resolves to the one and only bean in its context that matches the desired type.
- But if the type of the property is an array or a list (with a type argument – this doesn't work for Java-1.4 **List**s), then the context will populate that array or list with all beans found in the context that match the autowiring criteria.
- In **Transformer_Step11**, we use autowiring to get the firing sequence for free.
  - We do need a new Java class, because, sadly, we can't auto-wire a **<util:list>** in this way. See **src/FiringSequence.java**; a **javap**-style listing is shown below:

```
public class FiringSequence
{
  public void setSequence (List<XSLTransformer> s);
  public List<XSLTransformer> getSequence ();
}
```

  - We autowire that **sequence** property, and can drop the (somewhat redundant) **<util:list>** that we had going before. See **CMS.xml**:

```
<bean id="firingSequence" class="FiringSequence"
  autowire="byType" />
```

Autowiring to Multiple Beans

EXAMPLE

- **The controller has been augmented a bit, to make it easy to see what transforms are actually added to the sequence when it is autowired:**

```
List<XSLTransformer> firingSequence =
  ((FiringSequence) beanFactory.getBean
    ("firingSequence")).getSequence ();
for (XSLTransformer transformer : firingSequence)
{
  System.out.println
    (... transformer.getSourcePath() ... + " --> "
      + ... transformer.getResultPath () ...);

  transformer.transform ();
}
```

  - For instance, you may wonder, not if we'll get all the transforms into the list, but whether they'll appear in the desired order ...

- **Test this final version of the application:**

```
Input/Listings.xml --> Output/Summary.txt
Input/Listings.xml --> Output/Summary.html
Input/Listings.xml --> Output/Detail.txt
Input/Listings.xml --> Output/Detail.html
...
PIPE --> Output/Statistics.html
PIPE --> Output/AccessibleStatistics.txt
PIPE --> Output/AccessibleStatistics.html
```

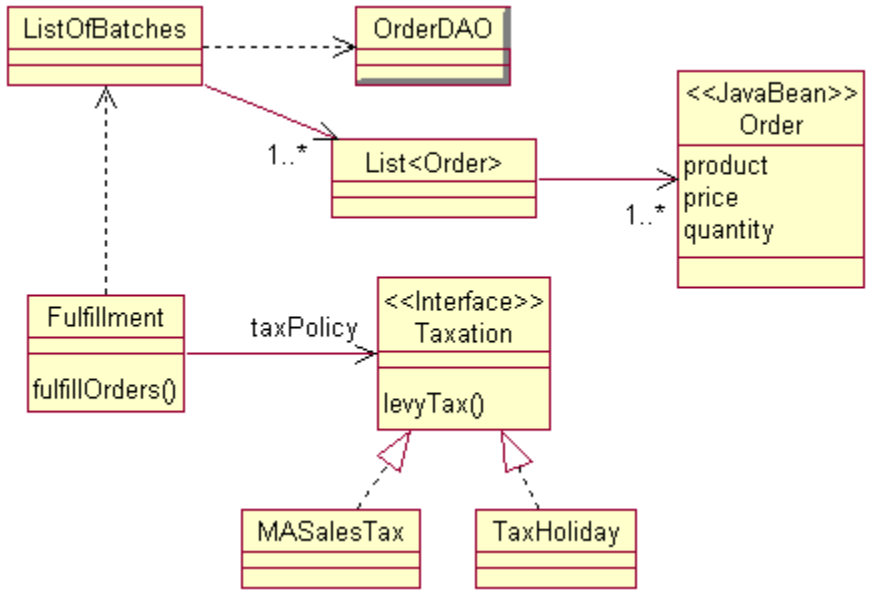
  - The controller doesn't echo all the configuration information, but there's enough here to confirm that the objects were assembled in the order in which they were declared, which is good news.

Wholesale Beans

LAB 5B

Suggested time: 30-45 minutes

In this lab you will complete the implementation of an application that processes product orders for a wholesale distribution company. Client retailers maintain files containing regular orders for their products of interest, and one of the jobs is to configure the list of these "feeds." A fulfillment engine processes these, also applying a locale-specific tax policy, and produces a sales record. This is then post-processed to produce an HTML report.



Detailed instructions are found at the end of the chapter.

Order of Instantiation

- **Often the order in which various beans are created is unimportant to the application logic.**
- **But sometimes it matters very much that bean A is created and initialized before bean B gets a chance to use it.**
- **Different IoC containers will pursue different policies in this regard.**
- **A BeanFactory will instantiate a defined bean only on a call to getBean.**
  - Supporting beans are then instantiated **on demand** – that is, as necessary to satisfy dependencies of beans as they are created.
  - This might be called a **lazy** instantiation policy.
- **We'll see that an application context is more eager to create beans, publishing all singletons when created itself.**
  - This is an **eager** instantiation policy.

Controlling Instantiation Timing

- **Various attributes can be defined in the configuration to customize the timing by which the container will create your beans.**
- **The primary means of control over the timing of object creation is the lazy-init attribute, which will convert a singleton bean from eagerly instantiated to instantiated-on-demand.**
  - The default behavior is true (lazy) for a simple bean factory, false (eager) for an application context.
  - You can also define **default-lazy-init** for an entire configuration unit.
- **Another issue arises when there is a bean dependency that isn't apparent from the information in the configuration file.**
  - Perhaps bean A requires bean B to exist prior to a call it makes on a third bean, or that a property on B be set before A makes a call to one of B's methods.
  - None of this would show up in a bean definition for A or B.
- **In such cases you can explicitly state the dependency with the depends-on attribute.**
  - As in A **depends-on="B"**.
- **In the JavaBean itself, you can implement an instantiation hook method – by any of the three techniques discussed in Chapter 3.**
  - This can be used to trigger additional bean loading, potentially resulting in a domino chain of beans loading one another.

Lazy Instantiation

DEMO

- **In Wholesale_Step4, we'll experiment with instantiation policies and declarations in the Wholesale application.**
  - The completed demo is in **Wholesale_Step5**.

1. Build and run the application, which is just as we left it at the end of Lab 5C.

```
Total sales:  10,075.65
```

2. Now, for each of four classes – **cc.sales.Fulfillment**, **cc.sales.ListOfBatches**, **cc.sales.MASalesTax**, and **cc.tools.xml.XSLTransformer** – add code to print a line to the console when the class is instantiated. (For **MASalesTax** you'll need to create an explicit no-argument constructor for this purpose.)
3. Build and test again, and see the order in which the objects are created.

```
MASalesTax instantiated.
Fulfillment instantiated.
ListOfBatches instantiated.
Total sales:  10,075.65
XSLTransformer instantiated.
```

4. Not much to surprise us here ... the tax object has to be created before the fulfillment object can be configured, but otherwise this is create-on-demand.

Lazy Instantiation

DEMO

5. Open **src/cc/sales/test/TestWholesale.java**, and replace **DefaultListableBeanFactory** with **FileSystemXmlApplicationContext**.

```
BeanFactory factory =
  new FileSystemXmlApplicationContext
    ("SalesObjects.xml");
```

   – Remember, you'll have to import this class from **org.springframework.context.support**.

6. Try it again:

```
Transformer instantiated.
MASalesTax instantiated.
Fulfillment instantiated.
ListOfBatches instantiated.
Total sales:  10,075.65
```

7. Hmm! An application context will eagerly instantiate and configure all singleton beans. There is no guarantee of the order of instantiation, but it seems that this factory works from the top of the configuration file to the bottom.
8. What if the transformer object pre-loaded some or all of its information? Say we configure it with a source file **TotalSales.xml** and it tries to read in the file contents as soon as we give it the filename. In this order of instantiation, there's no file to read!

Lazy Instantiation

DEMO

9. We can convert the transformer back to a create-on-demand policy by declaring **lazy-init** for it:

```
<bean
  id="PostProcessor"
  class="cc.tools.xml.XSLTransformer"
   lazy-init="true"
>
```

10. Test now:

```
MASalesTax instantiated.
Fulfillment instantiated.
ListOfBatches instantiated.
Total sales:  10,075.65
Transformer instantiated.
```

SUMMARY

- **The ability to assemble graphs of objects, of arbitrary complexity, completes the IoC container, and it's with these features that the value of IoC really comes home:**
  - Dependency injection
  - Populating collections
  - Generally, being able to hide all the details of object configuration behind a few strings, perhaps even one string, the bean name for the top object of a tree
- **As we want to control object creation, we may also want control over the timing of object creation, and Spring provides a few means to influence the behavior of the IoC factory.**
  - However, when a sequence of creation events must be strictly followed, it's a good idea to implement that sequence yourself, perhaps in an Abstract Factory; then, publish that factory to the IoC container.

The Firing Sequence

LAB 5A

In this lab you will improve the CMS from the previous lab, by making the firing sequence configurable using a list bean. The controller code will reduce to a very simple loop over the configured bean, which in turn will link in the transformers and pipes.

In optional steps you can also flesh out the full set of CMS pathways, adding a number of XML-to-HTML transformations in parallel with the text formatting you've already done.

Lab project: **Transforms_Step8**

Answer project(s): **Transforms_Step9** (intermediate)
Transforms_Step10 (final)

Files: * to be created
CMS.xml
src/cc/tools/xml/test/TestTransformation.java

Instructions:

1. If you didn't complete the previous lab, you might want to build and test the starter code, and observe the results in the **Output** directory.
2. In **CMS.xml**, add support for the utility schema by defining the **util:** namespace prefix and identifying the standard schema location – see the earlier page in the chapter entitled "The Spring Utility Schema" for the exact URI and location.
3. Define a new bean using the **`<util:list>`** element, and call it "firingSequence".
4. Add one **`<ref>`** child element to this list for each transformation in the sequence, with a **bean** attribute giving the name of the corresponding transformer bean. In other words, reproduce the sequence that's currently carried out in a series of expressions in **TestTransformation.java** – but now it's just a simple list of beans.
5. Remove the code in the **try** block of **TestTransformation.java**'s **main** method.
6. In that now-empty **try** block, derive a reference to a bean called "firingSequence" and downcast it to a **List<XSLTransformer>**.
7. For each element in the list, call **transform**.
8. Build and test, and you should see the application working as before, producing the same six output files. But now, you can put **TestTransformation.java** away; everything you need to say about CMS pathways – transformers, pipes, input and output paths, and the firing sequence – is configured for the application via the IoC container.

   (This is the intermediate answer in the **Step9** directory.)

The Firing Sequence LAB 5A

Optional Steps

9. To exercise this new freedom, try adding another transformation to the system. For each of the XML-to-text transformations currently in use, there is a corresponding XML-to-HTML transformation that will produce a simple web page. For example, if you add the following bean:

```
<bean id="summaryHTML" class="cc.tools.xml.XSLTransformer" >
  <constructor-arg value="Transforms/SummaryHTML.xsl" />
  <property name="sourcePath" value="Input/Listings.xml" />
  <property name="resultPath" value="Output/Summary.html" />
</bean>
```

   ... and add a reference to this bean to the firing sequence, you'll see a new file **Summary.html** in the **Output** directory when you run the application again. (And of course there's no longer any need to rebuild.)

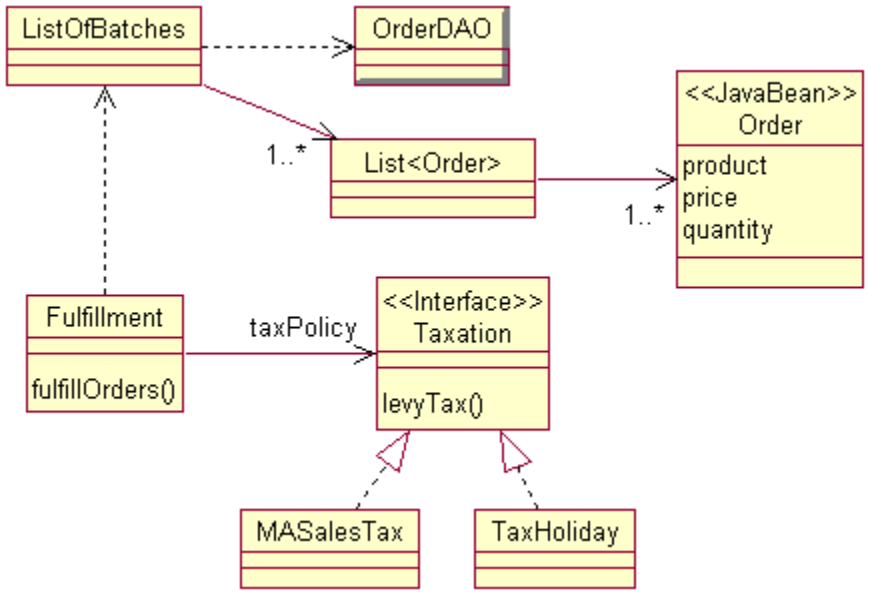
10. Add a similar bean to carry out a detail-HTML transformation, and you'll be able to click the links in the summary page to see sections of the detail page. You can add all six HTML transformations, side-by-side with the existing text transformations, and see the various pages in the **Output** directory.

    (This is the final answer in the **Step10** directory.)

Wholesale Beans

LAB 5B

In this lab you will complete the implementation of an application that processes product orders for a wholesale distribution company. Client retailers maintain files containing regular orders for their products of interest, and one of the jobs is to configure the list of these "feeds." A fulfillment engine processes these, also applying a locale-specific tax policy, and produces a sales record. This is then post-processed to produce an HTML report.



Lab project: **Wholesale_Step1**

Answer project(s): **Wholesale_Step2** (intermediate)
Wholesale_Step3 (intermediate)
Wholesale_Step4 (final)

Files: * to be created
SalesObjects.xml *
src/cc/sales/test/TestWholesale.java
src/cc/sales/ListOfBatches.java *

Instructions:

1. Review the domain model as implemented so far. The **ListOfBatches** class doesn't exist yet; it will bring the whole persistent model of **Order** objects into play, with the help of the **OrderDAO**. The **Fulfillment** engine combines a batch of orders with a configurable tax policy to derive final sales information, which it saves to a file.

Wholesale Beans LAB 5B

2. Open the **TestWholesale** source file and implement **main** to carry out the task of loading configured feeds and processing them. First, derive a **BeanFactory** based on the file **SalesObjects.xml** – you'll create this in a moment to satisfy the requirements of this Java class.
3. Get a **Fulfillment** object called "FulfillmentEngine".
4. Get a **List<List<Order>>** called "OrdersToProcess", and pass this to the fulfillment engine's **fulfill** method.
5. Run **ant** to check your code with the compiler.
6. Now create your **SalesObjects.xml** configuration file. Get the usual shell content from a previous example, and declare your "FulfillmentEngine" bean. This class' constructor takes two arguments: a **Taxation** object and a filename to which it will write the XML record of orders filled. Use an inner bean of class **cc.sales.MASalesTax** for the first argument, and for the second pass "TotalSales.xml".

   You may want to validate your configuration file at this point.
7. Now, how to derive a list of lists of **Order** objects? You know how to build a list, and even a list of lists. But how to get those orders? They are loaded from prepared files by the **OrderDAO**, and so perhaps that would be the approach: let **OrderDAO.load** be the factory method. Ah – but here's a weak spot in Spring's factory support. You can declare a static factory method on the target class itself, or a separate factory bean with an instance method – but not a static method on a separate factory bean.

   You could refactor **OrderDAO** somehow, but probably the best approach is to wrap it in a new JavaBean. This is where **ListOfBatches** comes in.
8. Create this new class, in **src/cc/sales/ListOfBatches.java**. It's job is to be a JavaBean that takes a list of strings, processes a file of the given name for each one, and thus creates a list of lists of orders (since each file holds a list of orders and you'll handle multiple files). This list of lists will be available as a property on the bean.

   So, given a list of strings called **filenames**, and a private field **batches** of type **List<List<Order>>**, the interesting bit is

```
for (String filename : filenames)
    batches.add (OrderDAO.load (filename));
```

   ... and your constructor should be signed to throw the **java.io.IOException**.

Wholesale Beans LAB 5B

9. When your new class is complete and compiled, return to **SalesObjects.xml** and declare a bean of that type called "OrdersToProcess". Now, your job here has gotten easier: just give this bean a constructor argument which is a list of the following filenames:

    Feeds/Apples.txt
    Feeds/Furniture.txt
    Feeds/MixedBag.txt
    Feeds/PartyFavors.txt

    Now, if you test at this point, things won't go so well – remember why? We left a little disconnect back down the road ...

10. Remember that **TestWholesale.java** still expects this bean to be a **List**, not a bean that wraps a list. So it will fail when it tries to downcast the bean returned by the factory. Instead, change the code there to correctly derive a **ListOfBatches**, and then when calling **fulfill**, pass the results of a call to **getBatches** on that bean, rather than the bean itself. As in:

```
engine.fulfill (lob.getBatches ());
```

11. Now run **TestWholesale**. You should see complete results now, which include a line to the console and a new file **TotalSales.xml**.

```
run

Total sales:  10,075.65
type TotalSales.xml
  ...
  <Sale>
    <Product>Sideboard</Product>
    <Price>1800.0</Price>
    <Quantity>1</Quantity>
    <Total>1903.5</Total>
  </Sale>
</Sales>
```

    This is the intermediate answer in **Wholesale_Step2**.

Optional Steps

12. Let's give autowiring a try. Make the tax policy a top-level bean – but don't bother giving it a name, just keep the class as defined and leave it at that.

13. Now, remove the first constructor argument from the "FulfillmentEngine" bean.

14. Define **autowire="constructor"** on that bean. The idea is that there's only one bean defined that could satisfy the dependency, given its type, so it should be found by the container when we ask it to auto-wire.

Wholesale Beans LAB 5B

15. Give this a try:

```
Total sales:  10,075.65
```

    This is the intermediate answer in **Wholesale_Step3**.

16. Finally, let's try wiring up one of our XSLT transformers to the output of this application. There is a prepared transform in **SalesReport.xsl**, and you can copy the **XSLTransformer** class and the **TransformerObjects.xml** configuration from **Transformer_Step5**.

17. Edit **TransformerObjects.xml** to reflect new inputs and outputs. The source file is now **TotalSales.xml**; the transform is **SalesReport.xsl**, and you can state any filename you want for the HTML output.

18. Now, you could have **TestWholesale** load a second bean factory from this new file, but instead, try a new trick: import the file into your existing configuration. As the first child element of **<beans>**, add:

```
<import resource="TransformerObjects.xml" />
```

19. Add code to **TestWholesale.java** to get the transformer bean and to call its **transform** method. Import the package **cc.tools.xml** into this source file.

Wholesale Beans LAB 5B

20. Build and test. After the application runs, you should see a new HTML file in the lab directory, and in a browser it will look like this:

Sales Report

Product	Price	Quantity	Total
Candles	$2.25	60	$142.76
Portmanteau	$2,400.00	1	$2,538.00
Ottoman	$500.00	2	$1,057.50
Vanity	$1,200.00	2	$2,538.00
Baldwin	$50.20	4	$212.35
Winesap	$48.00	8	$424.06
Macoun	$56.50	6	$358.49
Horns	$.75	80	$63.45
Butler	$720.00	1	$761.40
Hats	$1.50	48	$76.14
Sideboard	$1,800.00	1	$1,903.50
Total sales			**$10,075.65**

This is the final answer in **Wholesale_Step4**; the output file is **Report.html**.