COMPSCI 335 Distributed Objects, Services and Programming Week 2 Lab Work

Ian Warren

July 26, 2014

The aim of this week's lab work is for you to demonstrate that you can work with the Spring container and that you can solve some simple problems using aspect-oriented solutions.

Suggested tasks

Study the lecture materials on inversion of control, aspect-oriented programming and the associated source code. The following tasks do not need knowledge from the second AOP lecture; any AOP solutions should be developed programmatically using the `ProxyFactory` class. Some of the tasks, however, require you to read beyond the lecture material – for example, using the Spring reference manual or other Web-based resources. In addition, you will need to use Javadoc documentation for the JDK and Spring as necessary to complete the work.

Task one: Explore Spring's Dependency Injection support

The lecture material has introduced the basics of using the Spring container, in particular specifying components and their dependency injection requirements. Spring offers considerably more configuration options, and this task involves finding out about a couple of the more common cases.

Package `com.aucklanduni.spring.labs` contains a few classes that model a simple shopping application. `Product` represents a type of product that is offered for sale; `Store` represents a store with products; and `Cart` represents a shopping cart to which users can add products. In addition, `ShoppingMain` is a program that exercises the shopping classes.

When injecting dependencies, Spring allows *collections* in addition to simple values and bean references to be injected. Investigate how you can express dependency injection requirements involving collections, and create an XML document to inject some `Product` instances into a `store` component.

As discussed in lectures, Spring's `bean` tag declares a bean. What the tag actually says, however, is that you are creating a template for bean creation – and not the actual bean itself. When a bean is requested, by `getBean()` or a reference from other beans, Spring decides which bean instance should be returned according to the bean's *scope*. The default scope, which is generally appropriate, is `Singleton` scope. In this case, it's always the *same* bean instance that's returned whenever requested. However, in other cases, `Singleton` doesn't make sense. In the Shopping application, for example, it's intended that each user will have their own shopping cart bean. Look into how you can set a bean's scope, and add necessary content to the XML file to allow a new instance of the bean to be created when requested.

Task two: Implement a simple authenticator

Package `com.aucklanduni.spring.labs.basicauthenticator` contains an incomplete application, comprising 4 entities: `SecureBean`, `DefaultSecureBean`, `SecurityManager` and `UserInfo`.

SecureBean is a simple interface that has an associated implementation (DefaultSecureBean); a SecureBean is a resource that needs to be protected against unauthorised access. SecurityManager tracks the currently logged on user, who is represented by a UserInfo object.

Develop the application by implementing authentication as a cross-cutting concern. Specifically, 2 extra classes are necessary: one to implement the authentication concern, and the other to provide the executable program that uses the services of ProxyFactory as necessary. In addition, you should create a XML document to create necessary beans and bind them together. The application should behave by checking the identity of the logged-on user when making a SecureBean call; if there is no logged-on user, or if the currently logged-on user is not authorized, a SecurityException should be generated and the SecureBean call aborted.

For simplicity, when implementing the authentication check, use a hard-coded UserInfo object to represent a single user who is authorized to operate on the SecureBean resource.

Task three: Advise an arbitrary object to make it threadsafe

In addition to the cross-cutting concerns that have been discussed in lectures, concurrency control is a classic cross-cutting concern. Many existing classes have not been designed to be threadsafe, but with AOP we can effectively make them so without needing to modify them.

Develop an AOP-based solution to allow any existing class to be protected against simultaneous access from different threads. In other words you want to ensure mutually exclusive access (via method invocation) to an instance of an existing class.

Package com.aucklanduni.labs.simplelock includes some support code. Counter is a simple interface that represents a counter, whose value can be incremented and retrieved. This interface is implemented by DefaultCounter. SimpleLockMainWithProxyFactory is an executable program class that you can use to test your implementation. Currently, the program creates a DefaultCounter along with a specified number of threads. The threads run concurrently, incrementing the counter. Since DefaultCounter is not threadsafe, its state (counter value) will likely be corrupted at run-time. As part of your implementation, you will need to edit SimpleLockMainWithProxyFactory to use ProxyFactory.

Task four: Perform argument validation on setter methods

For this task, you should develop an AOP-based solution for validating actual arguments during invocation of setter methods on arbitrary objects. In particular, if a method is being invoked on some target object, and the name of the method is prefixed with set then any arguments should be checked for being null references. Where a null argument is detected, your solution should abort the invocation and throw an IllegalArgumentException.

A relatively easy solution to this task relies on implementing the checks for determining whether the invoked method is a setter method and whether any arguments are null in an Advice implementation. However, you should avoid such an implementation for two reasons. First, the advice code becomes less reusable – the advice's sole responsibility should be to throw an IllegalArgumentException; it shouldn't be concerned with *where* the advice is to be applied (this is the job of a *pointcut*). Second, without using an explicit pointcut, the proxy implementation is inefficient because it performs the checks on *every* invocation. When using pointcuts, the proxy implements checking more efficiently, by not carrying out checks on methods that don't need to be advised (for example, methods other than setters in this case).

Since this problem relies on method invocation information that is not available until run-time (i.e. actual arguments), you'll need to look at Spring's DynamicMethodMatcherPointcut class. Similarly, to work with pointcuts, you can use Spring's DefaultPointcutAdvisor.

Task five: Transform and log exceptions

Develop a solution that transforms any of Java's unchecked exceptions (i.e. those that are descendants of RuntimeException) to a new subclass of RuntimeException, CustomUncheckedException.

CustomUncheckedException stores state that describes the underlying exception (cause) plus details of the method, actual arguments and target object concerned.

Package com.aucklanduni.spring.labs.translation contains the CustomUncheckedException class in addition to an executable program (ExceptionTranslationMain), an interface (Target) and an associated implementation (DefaultTarget). DefaultTarget has one method named generateAnError() that always throws an unchecked exception. The provided classes can be used to help test your solution.

In addition to transforming unchecked exceptions to CustomUncheckedExceptions, your solution should also automatically log the exception. Exception logging is yet another example of a cross-cutting concern and is appropriately handled as an aspect.

Task six: Investigate another container role

This task doesn't involve any software development. You simply need to study the 3 source files in package com.aucklanduni.spring.labs.event and the associated events.xml document.

Based on the studying the files provided, consider the following questions. What concept is being implemented as a Container service? Can you think of a real application scenario where this service would be useful?

Resources

- http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/ This is the Spring reference manual and the definitive source of information for Spring. It can be supplemented with tutorials as needed.

- http://docs.spring.io/spring/docs/3.2.9.RELEASE/javadoc-api/ Javadoc documentation for the Spring framework.

- http://docs.oracle.com/javase/tutorial/essential/concurrency/ Task 3 requires basic knowledge of threading in Java. The pages titled "Processes and Threads" through to "Intrinsic Locks and Synchronization" inclusive are sufficient for the lab work.

- http://docs.oracle.com/javase/7/docs/api/ The Javadoc documentation for Java.

- http://stackoverflow.com As noted last week, this site is often useful when you have specific questions.

Supplementary tutorials can be found by using Web search engines.