Injecting objects in sealed code

Dependency Injection

*With examples in Java*

Dhanji R. Prasanna

MEAP Release: February 2008

Softbound print: August 2009 | 352 pages

ISBN: 193398855X

Not all the code you work with is under your control. Many third party libraries come in binary form and cannot be altered to work with dependency injectors. Adding annotations, refactoring with providers or builders, is out of the question. We'll call this **sealed** code. (Don't confuse this with the C# keyword.)

So if we have no control over sealed code, what can be done to make it work with dependency injection? One answer might be to find a way *not* to use annotations.

**Injecting with externalized metadata**

Recall some of the early Spring XML configuration. It eliminates the need for annotations, right off the bat. In essence, moving configuration metadata from source code to an external location (the XML file). Listing 1 shows a sealed class injected purely with externalized metadata.

**Listing 1: A sealed class injected via external configuration**

```java
public class Sealed {
    private final Dependency dep;

    public Sealed(Dependency dep) {
        this.dep = dep;
    }
}
<!-- XML injector configuration -->
<beans>
    <bean id="sealed" class="Sealed">
        <constructor-arg>
            <bean class="Dependency"/>
        </constructor-arg>
    </bean>
</beans>
```
Here Sealed did not have to change, and the injector configuration is straightforward. It even works with setter injection. See figure 1.

![Diagram](image)

**Figure 1:** External metadata allows you to inject classes in sealed code easily

But sealed code often throws you more curve balls than this. It may have completely private constructors and only expose a static factory method. This is not uncommon in library code (see listing 2).

**Listing 2: A sealed class injected via external metadata**

```java
public class Sealed {
    private final Dependency dep;
    
    private Sealed(Dependency dep) {
        this.dep = dep;
    }
    
    public static Sealed newInstance(Dependency dep) {
        return new Sealed(dep);
    }
}
<!-- XML injector configuration -->
<beans>
    <bean id="sealed" class="Sealed" factory-method="newInstance">
        <constructor-arg><bean class="Dependency"></constructor-arg>
    </bean>
</beans>
```

Listing 2 shows how Spring is able to call on factory methods just as though they were constructors (the `<constructor-arg>` element now passes arguments to the factory). If, however, it is an unfriendly factory that completely encapsulates construction,

```java
public class Sealed {
    private final Dependency dep;
    
    Sealed(Dependency dep) {
        this.dep = dep;
    }
    
    public static Sealed newInstance() {
        return new Sealed(new Dependency());
    }
}
```
Then the situation is a bit more tricky. There is no obvious way to provide the constructor with an instance of Dependency. Custom or mock implementations have been removed from the equation. Also making testing very difficult. Here's another scenario where the XML falls over:

```java
public class Sealed {
    private Dependency dep;

    public Sealed() {
    }

    public void dependOn(Dependency dep) {
        this.dep = dep;
    }
}
```

This class accepts its dependency via setter injection—however the setter method does not conform to Spring's naming convention. Rather than `setDependency()` it is called `dependOn()`. Remember we can't change any of this code—it is sealed.

Even if there were some way around it, misspelling method names can easily cause you much chagrin. The Adapter pattern provides a better solution.

**Using the Adapter Pattern**

The Adapter is yet another design pattern from the *Gang of Four* book. It allows you to alter the behavior of existing objects by extending them. The following adapter allows you to inject classes whose constructors are hidden (so long as it is not `private`):

```java
public class SealedAdapter extends Sealed {
    @Inject
    public SealedAdapter(Dependency dep) {
        super(dep);
    }
}
```

The call to `super(dep)` chains to Sealed's hidden constructor. Since `SealedAdapter` extends `Sealed`, it can be used by any dependent of `Sealed` transparently. And it benefits from dependency injection (see figure 2).

```
Figure 2: SealedAdapter helps Sealed to be injected transparently
```

The same solution applies when you have a `package-local` constructor too. Except that you build `SealedAdapter` into the same package as `Sealed`.

Similarly, this works with the unconventional setter methods:

```java
public class SealedAdapter extends Sealed {
    @Inject
```
public sealedadapter(Dependency dep) {
    dependon(dep);
}

Or a more convoluted example with both combined:

public class SealedAdapter extends Sealed {
    @Inject
    public SealedAdapter(Dependency dep1, Dependency dep2, Dependency dep3) {
        super(dep1);
        dependon(dep2);
        relyon(dep3);
    }
}

Using an adapter:

- any infrastructure logic is cleanly encapsulated within SealedAdapter
- typing mistakes and mis-spellings are caught at compile time
- adapters can be used directly by dependents of Sealed, requiring no code changes (unlike a provider, for example)
- adapters can be tested and used in the same fashion as the original sealed class.

Contrast the use of an adapter with a provider with these points in mind. Adapters can sometimes be more verbose but they have no impact on client code. And can even be passed back to other sealed classes transparently. Something that's hard to do with providers.

**TIP**

If you're particularly clever, you can bridge the gap between adapter and provider to solve the reinjection problem. Do this by overriding each method of the original component and looking up a new instance as needed with a provider. Then delegate all methods to that instance. It leaves minimal impact on dependent code and is a compelling alternative.

One other interesting option for injecting sealed code is the Builder pattern. You can incrementally add dependencies to a builder and then let it decide how to construct the sealed component. Builders are particularly useful with unconventional setters, or if there are several constructors or factories to choose from.