Dependency Injection and Unit Testing

by Christophe Verré

In recent years, Dependency Injection may have buzzed into your ears quite often. You may already know that it has nothing to do with drug addiction, but with some cool object-oriented concept. You may also have heard of the Spring Framework, a so-called Dependency Injection container. You might be under the impression that Dependency Injection equals Spring. But no, Dependency Injection is a simple concept which can be used anywhere, without any Dependency Injection container, and especially useful in unit testing. In this article, we will see

- what Dependency Injection is
- how to make a class Dependency Injection friendly
- why Dependency Injection can ease unit tests

Ladies and gentlemen, start your engine !

A simple car

Let's consider a simple example, using engines and cars. We'll leave classes and interfaces empty for clarity. A car has an engine, and we'd like that car to be equipped with JavaRanch's famous MooseEngine™.

We've got engines:

```
public interface Engine {
}
public class SlowEngine implements Engine {
}
public class FastEngine implements Engine {
}
public class MooseEngine implements Engine {
}
```

And we've got a car:

```
public class Car {
        private MooseEngine engine;
}
```

This is a great car, but it cannot have any other kind of engine, although there are other brands available on the market. We say that the Car

class is **tightly coupled** to the MooseEngine class. There's nothing wrong with it, the MooseEngine™ is great, but what happens if we finally decide to equip it with another engine ?

Programming with interfaces

You may have noticed that the MooseEngine™ was implementing the Engine interface. Other brands are also implementing the same interface. Let's think about it. When we designed our Car class, we thought that a "car" was equipped with an "engine". So let's rewrite the Car class to use the Engine interface instead:

```
public class Car {
        private Engine engine;
}
```

Programming with interfaces is an important concept in Dependency Injection. I can already hear you screaming, "*wait a minute! That's an interface you're using here, where is the concrete class? Where do you set it? I want my MooseEngine™ in my car*". We could set it the following way:

```
public class Car {
        private Engine engine = new MooseEngine();
}
```

But would that be useful ? It doesn't look much different from the first example. Our car is still **tightly coupled** to our beloved MooseEngine™. So? Where do we set (or inject) our car's engine ?

Introducing Dependency Injection

As its name says, Dependency Injection is all about injecting dependencies, or simply said, setting relations between instances. Some people refer to it as being the Hollywood principle "Don't call me, we'll call you". I prefer calling it the "bugger" principle: "I don't care who you are, just do what I ask". In our first example, a Car depended on an Engine's concrete class called MooseEngine. When an class A depends on another class B, and that B's implementation is set directly in A, we say that A is **tightly coupled** to B. As we've seen in the second example, we have decided to use the Engine interface instead of the MooseEngine™ concrete class to make the Car more flexible. Moreover, we decided **not to define the engine's concrete implementation**. In other words, we have made our Car class **loosely coupled**. The Car doesn't depend on any concrete class of Engine anymore. Where do we indicate which Engine to use then ? That's where Dependency Injection comes. Instead of setting the concrete Engine class in the Car class, the concrete Engine class is injected from the outside. How do we do that?

1. Using constructor based injection

One way to set dependencies is to pass the concrete implementation of the depending class to the constructor. Our Car class would become:

```
public class Car {
        private Engine engine;

        public Car(Engine engine) {
                this.engine = engine;
        }
}
```

We could then create a Car using any kind of engine. For example, one car using the great MooseEngine™ and another one using the crappy SlowEngine:

```
public class Test {

        public static void main(String[] args) {
                Car myGreatCar = new Car(new MooseEngine());
                Car hisCrappyCar = new Car(new SlowEngine());
        }
}
```

2. Using setter based injection

Another common way to set dependencies is to use setter methods. Using setter methods is recommended instead of the constructor when many dependencies need to be injected. Our car class would then be written that way:

```
public class Car {
        private Engine engine;

        public void setEngine(Engine engine) {
                this.engine = engine;
        }
}
```

It looks very similar to the constructor based injection, doesn't it? We would then implement the same cars we have used above the following way:

```
public class Test {

        public static void main(String[] args) {
                Car myGreatCar = new Car();
                myGreatCar.setEngine(new MooseEngine());
```

```
                Car hisCrappyCar = new Car();
                hisCrappyCar.setEngine(new SlowEngine());
        }
}
```

Using Dependency Injection for Unit Testing

If you compare the first example of the Car class, and the one using setter based injection, you might think that some extra steps were needed to make the Car class using Dependency Injection. That's right, you had to write a setter method. But you can feel how rewarding this extra work is when unit testing. If you are not confident about what unit tests are, I recommend this Campfire Story on Evil Unit Tests. Our Car example is so simple that it does not illustrate well how useful Dependency Injection is for unit testing. We'll leave cars, and consider the example provided in the forementioned Campfire Story, especially the part on using mocks for unit testing. We have a servlet class using a remote EJB to register animals in a farm:

```
public class FarmServlet extends ActionServlet {

        public void doAction( ServletData servletData ) throws Exception {
                String species = servletData.getParameter("species");
                String buildingID = servletData.getParameter("buildingID");
                if ( Str.usable( species ) && Str.usable( buildingID ) ) {
                        FarmEJBRemote remote = FarmEJBUtil.getHome().create();
                        remote.addAnimal( species , buildingID );
                }
        }
}
```

You have already noticed that FarmServlet is **tightly coupled** to the FarmEJBRemote instance, which was retrieved via a call to "FarmEJBUtil.getHome().create()". It makes it very hard to test. When unit testing, we don't want to use any database. We don't want to access an EJB server either. That would make unit tests both difficult to execute and slow. So in order to unit test the FarmServlet class smoothly, we'd better make it **loosely coupled**. To remove the tight dependency between FarmServlet and FarmEJBRemote, we could use a setter based injection:

```
public class FarmServlet extends ActionServlet {
        private FarmEJBRemote remote;

        public void setRemote(FarmEJBRemote remote) {
                this.remote = remote;
        }

        public void doAction( ServletData servletData ) throws Exception {
```

```
                String species = servletData.getParameter("species");
                String buildingID = servletData.getParameter("buildingID");
                if ( Str.usable( species ) && Str.usable( buildingID ) ) {
                        remote.addAnimal( species , buildingID );
                }
        }
}
```

In the real deployment package, we will make sure that an instance of FarmServlet's remote member will be properly injected via "FarmEJBUtil.getHome().create()". In our unit test, we will use a dummy mock class to act like a FarmEJBRemote. In other words, we will make a mock class implementing FarmEJBRemote:

```
class MockFarmEJBRemote implements FarmEJBRemote {

        private String species = null;
        private String buildingID = null;
        private int nbCalls = 0;

        public void addAnimal( String species , String buildingID )
        {
                this.species = species ;
                this.buildingID = buildingID ;
                this.nbCalls++;
        }

        public String getSpecies() {
                return species;
        }
        public String getBuildingID() {
                return buildingID;
        }
        public int getNbCalls() {
                return nbCalls;
        }
}

public class TestFarmServlet extends TestCase {

        public void testAddAnimal() throws Exception {
                // Our mock acting like a FarmEJBRemote
                MockFarmEJBRemote mockRemote = new MockFarmEJBRemote();
                // Our servlet. We set our mock to its remote dependency
                FarmServlet servlet = new FarmServlet();
                servlet.setRemote(mockRemote);
```

```
                // just another mock acting like a ServletData
                MockServletData mockServletData = new MockServletData();
                mockServletData.getParameter_returns.put("species","dog");
                mockServletData.getParameter_returns.put("buildingID","27");

                servlet.doAction( mockServletData );
                assertEquals( 1 , mockRemote.getNbCalls() );
                assertEquals( "dog" , mockRemote.getSpecies() );
                assertEquals( 27 , mockRemote.getBuildingID() );
        }
}
```

That's it! We've got an easy to test FarmServlet.
To sum up some important points:

- Use interfaces instead of using concrete classes to illustrate a dependency.
- Avoid to implicitly set the concrete implementation of a dependency in the class itself.
- Concrete implementation of a dependency may be set in various ways, including constructor based injection or setter based injection.
- Dependency injection makes unit testing very flexible.