**Spring Dependency Injection**

**1. Dependency Injection**

Java components / classes should be as independent as possible of other Java classes. This increases the possibility to reuse these classes and to test them independently of other classes(Unit Testing). To decouple Java components from other Java components the dependency to a certain other class should get injected into them rather that the class itself creates / finds this object.

A class A has a dependency to class B if class uses class B as a variable.

If dependency injection is used then the class B is given to class A via

* the constructor of the class A - this is then called construction injection
* a setter - this is then called setter injection

The general concept between dependency injection is called Inversion of Control. A class should not configure itself but should be configured from outside.

A design based on independent classes / components increases the re-usability and possibility to test the software. For example, if a class A expects a Dao (Data Access object) for receiving the data from a database you can easily create another test object which mocks the database connection and inject this object into A to test A without having an actual database connection.

A software design based on dependency injection is possible with standard Java.

Spring just simplifies the use of dependency injection by providing a standard way of providing the configuration and by managing the reference to the created objects.

**2. Spring Overview**

The Spring Framework is a very comprehensive framework.

The fundamental functionality provided by the Spring Container is dependency injection. Spring provides a light-weight container, e.g. the Spring core container, for dependency injection (DI).

This container lets you inject required objects into other objects. This results in a design in which the Java class are not hard-coupled. The injection in Spring is either done via setter injection of via construction injection.

These classes which are managed by Spring must conform to the JavaBean standard.

In the context of Spring classes are also referred to as beans or as spring beans.

The Spring core container:

* handles the configuration, generally based on annotations or on an XML file (XMLBeanFactory)
* manages the selected Java classes via the BeanFactory

The core container uses the so-called bean factory to create new objects. New objects are generally created as Singletons if not specified differently.

**3. Spring Installation**

Download Spring from [**http://www.springframework.org/download.**](http://www.springframework.org/download) Select the -with-dependencies.zip to get also all required plugins. At this time of writing I downloaded the version Spring Framework 2.5.5.

The folder "dist" contains the Spring container "spring.jar". The folder lib contains additional require libraries. A minimal Spring application requires the spring.jar, commons-logging.jar (from \lib\jakarta-commons) and log4j\*.jar (from \lib\log4j).

**4. Datamodel**

We will later use the following datamodel for the example.

Create a Java project "de.vogella.spring.di.model" and create the following packages and classes.

**package** writer;

**public** **interface** IWriter {

**public** **void** writer(String s);

}

**package** writer;

**public** **class** Writer **implements** IWriter {

**public** **void** writer (String s){

System.out.println(s);

}

}

**package** writer;

**public** **class** NiceWriter **implements** IWriter {

**public** **void** writer (String s){

System.out.println("The string is " + s);

}

}

**package** testbean;

**import** writer.IWriter;

**public** **class** MySpringBeanWithDependency {

**private** IWriter writer;

**public** **void** setWriter(IWriter writer) {

**this**.writer = writer;

}

**public** **void** run() {

String s = "This is my test";

writer.writer(s);

}

}

The class "MySpringBeanWithDependency.java" contains a setter for the actual writer. We will use the Spring Framework to inject the correct writer into this class.

**5. Using dependency injection with annotations**

As of Spring 2.5 it is possible to configure the dependency injection via annotations. I recommend to use this way of configuring your Spring beans. The next chapter will also describe the way to configure this via XML.

Create a new Java project "de.vogella.spring.di.annotations.first" and include the minimal required spring jars into your classpath.

Copy your model class from the de.vogella.spring.di.model project into this project.

You need now to add annotations to your model to tell Spring which beans should be managed by Spring and how they should be connected.

Add the @Service annotation the MySpringBeanWithDependency.java and NiceWriter.java. Also define with @Autowired on the setWriter method that the property "writer" will be autowired by Spring.

**Tip**

@Autowired will tell Spring to search for a Spring bean which implements the required interface and place it automatically into the setter.

**package** testbean;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** writer.IWriter;

*@Service*

**public** **class** MySpringBeanWithDependency {

**private** IWriter writer;

*@Autowired*

**public** **void** setWriter(IWriter writer) {

**this**.writer = writer;

}

**public** **void** run() {

String s = "This is my test";

writer.writer(s);

}

}

**package** writer;

**import** org.springframework.stereotype.Service;

*@Service*

**public** **class** NiceWriter **implements** IWriter {

**public** **void** writer(String s) {

System.out.println("The string is " + s);

}

}

Under the src folder create a folder META-INF and create the following file in this folder. This is the Spring configuration file.

<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:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http:*//www.springframework.org/schema/beans/spring-beans-2.5.xsd*

http:*//www.springframework.org/schema/aop*

http:*//www.springframework.org/schema/aop/spring-aop-2.5.xsd*

http:*//www.springframework.org/schema/context*

http:*//www.springframework.org/schema/context/spring-context-2.5.xsd">*

<context:component-scan base-**package**="testbean" />

<context:component-scan base-**package**="writer" />

</beans>

**Tip**

 You can also configure the log4j logger (this is optional) by copying the following file into the source folder.

log4j.rootLogger=FATAL, first

log4j.appender.first=org.apache.log4j.ConsoleAppender

log4j.appender.first.layout=org.apache.log4j.PatternLayout

log4j.appender.first.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n

Afer this setup you can wire the application together. Create a main class which reads the configuration file and starts the application.

**package** main;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** testbean.MySpringBeanWithDependency;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("META-INF/beans.xml");

BeanFactory factory = context;

MySpringBeanWithDependency test = (MySpringBeanWithDependency) factory

.getBean("mySpringBeanWithDependency");

test.run();

}

}

If you run the application then the class for the IWriterInterface will be inserted into the Test class. By applying the dependency injecting I can later replace this writer with a more sophisticated writer.

As a result the class Test does not depend on the concrete Writer class, is extensible and can be easily tested.

**6. Using dependency injection with XML**

The following example will demonstrate the usage of the dependency injection via xml. The example will inject a writer into another class.

**Tip**

 I think annotations rock in general, therefore I recommend not to use the XML configuration but the annotation one. If you have good reason to use the XML configuration please feel free to do so.

Create a new Java project "de.vogella.spring.di.xml.first" and include the minimal required spring jars into your classpath.

Copy your model class from the de.vogella.spring.di.model project into this project.

Under the src folder create a folder META-INF and create the following file in this folder. This is the Spring configuration file.

<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:context="http://www.springframework.org/schema/context"

xsi:schemaLocation="http://www.springframework.org/schema/beans

http:*//www.springframework.org/schema/beans/spring-beans-2.5.xsd*

http:*//www.springframework.org/schema/aop*

http:*//www.springframework.org/schema/aop/spring-aop-2.5.xsd*

http:*//www.springframework.org/schema/context*

http:*//www.springframework.org/schema/context/spring-context-2.5.xsd">*

<bean id="writer" **class**="writer.NiceWriter" />

<bean id="mySpringBeanWithDependency" **class**="testbean.MySpringBeanWithDependency">

<property name="writer" ref="writer" />

</bean>

</beans>

Again, you can now wire the application together. Create a main class which reads the configuration file and starts the application.

**package** main;

**import** org.springframework.beans.factory.BeanFactory;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**import** testbean.MySpringBeanWithDependency;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

ApplicationContext context = **new** ClassPathXmlApplicationContext("META-INF/beans.xml");

BeanFactory factory = context;

MySpringBeanWithDependency test = (MySpringBeanWithDependency) factory

.getBean("mySpringBeanWithDependency");

test.run();

}

}

**7. Summary**

This tutorial gave you an applied approach for using dependency injection with the Spring framework. Both possible way, with annotations and with XML were discussed.