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.

Say , class A has a dependency to class B if class uses class B as a variable.(reference)

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.

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.

Types of Dependancy Injection :

- Constructor Injection
- Setter Injection

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.

Example : without DI

Create a Java project "**com.cg.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

Using dependency injection with annotations

As of Spring 2.5 it is possible to configure the dependency injection via annotations.

Create a new Java project " **com.cg.spring.di.model.annotations.first**" and include the minimal required spring jars into your classpath.

Copy your model class from the **com.cg.spring.di.model.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.

NOTE :

@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);
    }
}
```

Main program :

```
package main;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
```

```
import
org.springframework.context.support.ClassPathXmlApplicationConte
xt;

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.

DI with XML

```
<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.ClassPathXmlApplicationConte
xt;

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();
    }
}
```