**Consider the following knight class.**

public com.springinaction.knights;

public class DamselRescuingKnight implements Knight

{

private RescueDamselQuest quest;

public DamselRescuingKnight()

{

this.quest=new RescueDamselQuest();

//Now, this makes DamselRescuingKnight tightly coupled to RescueDamselQuest

}

public void embarkOnQuest()

{

quest.embark();

}

}

Now, as you can see, DamselRescuingKnight creates in own quest ( search for something) , A RescueDamselQuest in the constructor. This makes a DamselRescuingKnight tightly coupled to a DamselRescuingKnight and severely limits the knight’s quest embarking repertoire. (a set of skills) If a Damsel(a young unmarried woman) needs rescuing, the knight’s there. But if a dragon needs slaying or a round table needs rounding, this knight’s going to have to sit it out.  
  
  
What’s more, it will be terribly difficult to write a unit test for DamselRescuingKnight. Because, in such a test, you’s like to be able to assert that the quest’s embark method, is called when the knight’s embarkOnQuest() method is called. But, there is no clear way to accomplish it here. Unfortunately, DamselRescuingKnight will remain uninterested.

Now, the problem with tightly coupled code is that it is difficult to test, difficult to reuse and difficult to understand and it typically exhibits “whack-a-mole” bug behaviour. (fixing one bug results in the creation of one or more new bugs). On the other hand, a certain amount of coupling is necessary-completely coupled code does not do anything. In order to do anything useful, classes need to know about each other somehow. Coupling is necessary but should be carefully managed.

With **Dependency Injection,** objects are given their dependencies at compile time by some third party that coordinates each subject in the system. Objects are not expected to create or obtain their dependencies. Rather, dependencies are injected into the objects that need them.

Let’s look at the following class:

**public com.springinaction.knights;**

**public class BraveKnight implements Knight**

**{**

**private Quest quest;**

**public BraveKnight(Quest quest)**

**{**

**this.quest=quest;**

**}**

**public void embarkOnQuest()**

**{**

**quest.embark();**

**}**

**}**

Now, as you can see, BraveKnight, unlike DamselRescuingKnight, does not create his own quest. Instead, he’s given a quest at construction time as a constructor argument. This is a type of Dependency injection known as constructor injection.

What’s more, the quest he’s given is typed as quest, an interface that all quests implement. So, BraveKnight could embark on a RescueDamselQuest, a SlayDragonQuest, a MakeRoundTableRounderQuest, or any other quest implementation he’s given.

(this is also one approach offered by good coding practice: program for interfaces, not implementations)

The point is that BraveKnight is not coupled to any specific implementation of Quest. It does not matter to him **(BraveKnight)** what kind of quest he’s asked to embark on, as long as it implements the Quest interface. That’s the key benefit of DI- loose coupling. If an object only knows about their dependencies **by their interface(not by their implementation or how they’re instantiated).** Then the dependency can be swapped out with the different implementation **without the depending object knowing the difference.**One of the most common ways a dependency is swapped out is with a mock implementation during testing. The DamselRescuingKnight could not be tested due to tight coupling. But you can easily test BraveKnight by giving it a mock implementation of Quest, as shown below.

**package com.springinaction.knights;**

**import static.org.mockito.Mockito.\*;**

**import org.junit.Test;**

**public class BraveKnightTest**

**{**

**@Test**

**public void knightShouldEmbarkOnQuest()**

**{**

**Quest mockQuest=mock(Quest.class);**

**BraveKnight knight=new BraveKnight(mockQuest);**

**knight.embarkOnQuest();**

**verify(mockQuest,times(1)).embark();**

**}**

**}**

Here, you use a mock object framework known as Mockito to create a mock implementation of the Quest interface. With the mock object in hand, you create a new instance Of BraveKnight, injecting the mock quest via the constructor. After calling the embarkOnQuest() method, you ask Mockito to verify that the mock Quest’s embark() method was called exactly once.

**verify(mockQuest,times(1)).embark();**

**Injecting A Quest Into A Knight:**

Now that the BraveKnight is written in such a way that you can give a knight any quest you want, how can you specify which Quest to give him? Suppose, for instance, that you’d like for the BraveKnight to embark on a quest to slay a dragon. A quest is created below:

**package com.springinaction.knights;**

**import java.io.PrintStream;**

**public class SlayDragonQuest implements Quest**

**{**

**private PrintStream stream;**

**public SlayDragonQuest implements Quest**

**{**

**private PrintStream stream;**

**public SlayDragonQuest(PrintStream stream)**

**{**

**this.stream=stream;**

**}**

**}**

**public void embark()**

**{**

**stream.println("Embarking on quest to slay the dragon");**

**}**

**}**

Now, we want to inject it in BraveKnight runtime:  
  
The BraveKnight’s definition remains same:

**public com.springinaction.knights;**

**public class BraveKnight implements Knight**

**{**

**private Quest quest;**

**public BraveKnight(Quest quest)**

**{**

**this.quest=quest;**

**}**

**public void embarkOnQuest()**

**{**

**quest.embark();**

**}**

**}**

Now, how to inject it?   
  
**Injecting A SlayDragonQuest into A BraveKnight With Spring:**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns="http://www.w3.org/2001/XMLSchema-instance"

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

http://www.springframework.org/schema/beans/spring-beans.xsd">

</beans>

<bean id="knight" class="com.springincation.knights.BraveKnight">

<constructor-arg ref="quest"/>

<!--refer with the id attached with the class-->

</bean>

<bean id="quest" class="com.springincation.knights.SlayDragonQuest">

<constructor-arg value="#{T(System).out}"/>

</bean>

</beans>

Here, **BraveKnight** and **SlayDragonQuest** are declared as beans in Spring. In the case of BraveKnight bean, it’s constructed, passing a reference to the SlayingDragonQuest bean as a constructor argument. Meanwhile, the SlayingDragonQuest bean declaration uses the spring expression language to pass System.out (which is a PrintStream) to SlayDragon’s constructor.

Now, note that, How, System.out is passed as the argument of the constructor of the SlayingDragonQuest class

Now, advantage of Spring is It offers Java Based Configuration as well (as an alternative to xml configuration)

package com.springinaction.knights.config;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

//external packages' needed classes are imported

import com.springinaction.knights.BraveKnight;

import com.springinaction.knights.Knight;

import com.springinaction.knights.Quest;

import com.springinaction.knights.SlayDragonQuest;

@Configuration

public class KnightConfig

{

@Bean

public Knight knight()

{

return new BraveKnight(quest());

//since, quest function returns a class object which(the class) implements Quest interface

}

@Bean

public Quest quest()

{

return new slayDragonQuest(System.out);

}

}

Whether you use XML based or java based configuration, the benefits of DI are same.