Dependency Injection and Spring

INF5750/9750 - Lecture 2 (Part II)

Problem area

- Large software contains huge number of classes that work together
- How to wire classes together?
  - With a kind of loose coupling, so even if there are changes in one place, you don’t have to make changes in a lot of source files
- How to make it easy to change, test and maintain your code?

Example: The StudentSystem

- To improve your skills in Java development, you decide to develop a student system

- You decide to use a file to store student information
- You create a class FileStudentDAO responsible for writing and reading to the file
- You create a class StudentSystem responsible for performing the logic of the system
- You've learned that it's a good thing to program to interfaces

The StudentSystem

Responsible for performing useful operations on students

Service layer

<< interface >>
StudentSystem

- int getNrOfStudents(String subjectCode)

<implements>

DefaultStudentSystem

<depends>

Persistence layer

<< interface >>
StudentDAO

- Collection<Student> getAllStudents()

<implements>

FileStudentDAO

Responsible for adding, deleting and getting students from some storage medium

A file

The DefaultStudentSystem

```
public class DefaultStudentSystem implements StudentSystem
{
    private StudentDAO studentDAO = new FileStudentDAO();

    public int getNrOfStudents( String subjectCode )
    {
        Collection<Student> students = studentDAO.getAllStudents();

        int count = 0;

        for ( Student student : students )
        {
            if ( student.getSubjects.contains( subjectCode ) )
            {
                count++;
            }
        }

        return count;
    }
}
```

The StudentDAO reference is instantiated with a concrete class

Works! Or...?

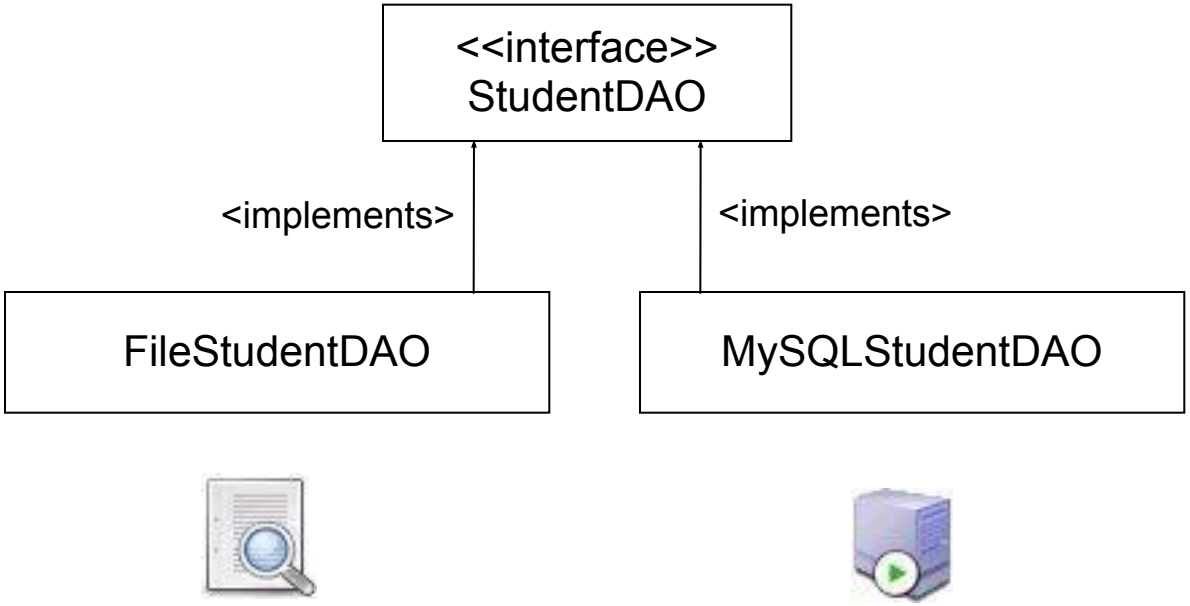
- The system is a big success – University of Oslo wants to adopt it!

You use
a file to store the
student information

University of Oslo uses
a MySQL database to
store their student
information

Works! Or...?

You make a new implementation of the StudentDAO for University of Oslo, a MySQLStudentDAO:



Works! Or...?

- Won't work! The FileStudentDAO is hard-coded into the StudentSystem:

```
public class DefaultStudentSystem implements StudentSystem
{
    private StudentDAO studentDAO = new FileStudentDAO();
    ...
```

- The DefaultStudentSystem implementation is responsible for obtaining a StudentDAO
- Dependent both on the StudentDAO interface and the implementation

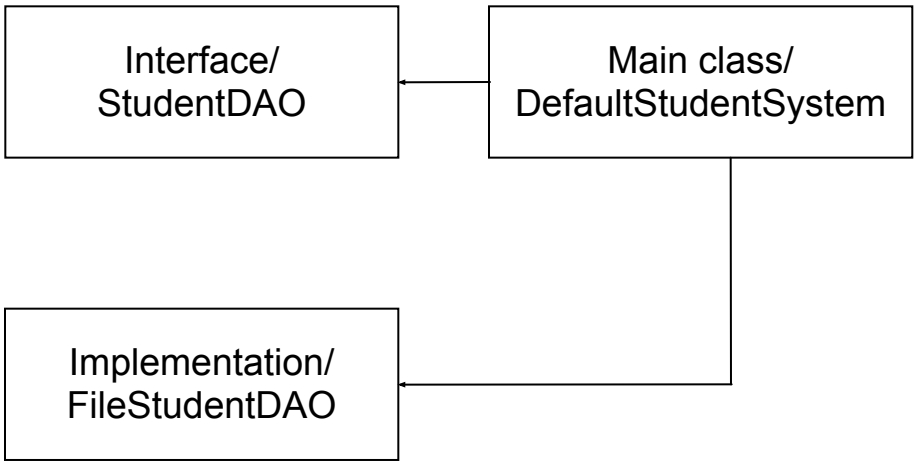
Works! Or...?

- How to deploy the StudentSystem at different locations?
- Develop various versions for each location?
  - Time consuming
  - Confusing and error-prone
  - Requires more efforts for versioning

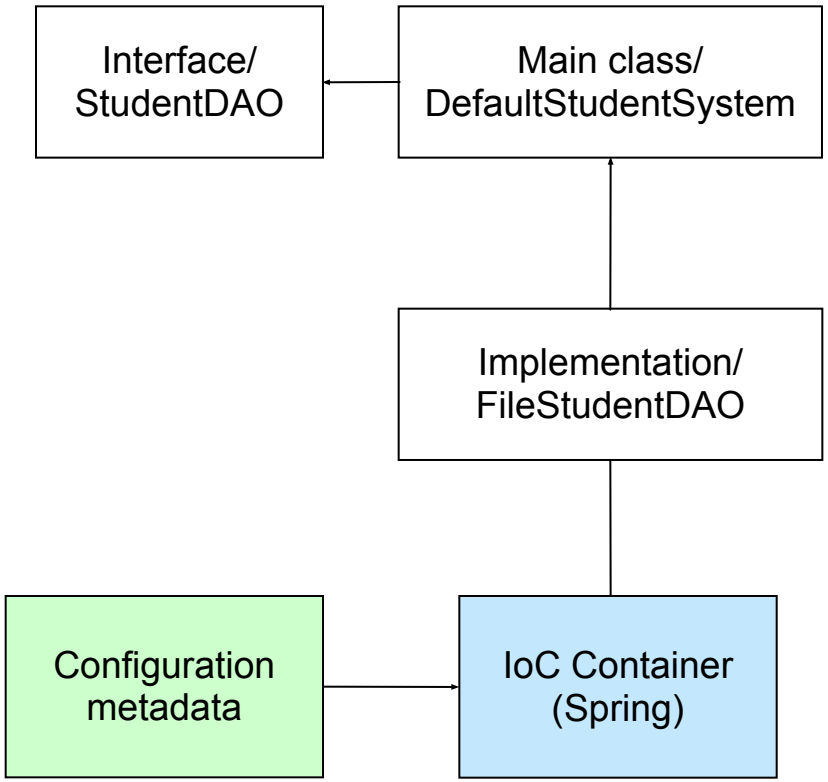
- Use *Dependency Injection!*
  - More specific term derived from the term *Inversion of Control*

Dependency Injection

Traditional



Using Dependency Injection



Dependency Injection

- Objects define their dependencies only through constructor arguments or setter-methods
  - Enables loose coupling

- Dependencies are *injected* into objects by a container (like Spring)
- Inversion of Control... (IoC)

- Two major types of dependency injection
  - Setter injection
  - Constructor injection

- Constructor injection for all mandatory collaborators and setter injection for all other properties

Spring Configuration

- *Bean*: A class that is managed by a Spring IoC container
- *Constructor based DI*: Dependency should be part of Constructor arguments
- *Setter based DI*: Provide a public set-method for the dependency reference

```
public class DefaultStudentSystem implements StudentSystem
{
   private StudentDAO studentDAO;

   public void setStudentDAO( StudentDAO studentDAO )
   {
       this.studentDAO = studentDAO;
   }

   public int getNrOfStudents( String subjectCode )
   {
      List students = studentDAO.getAllStudents();
      // method logic goes here...
   }
}
```

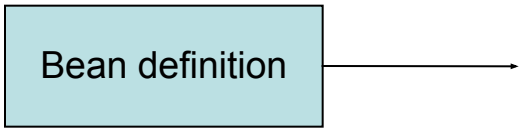
```
public class DefaultStudentSystem implements StudentSystem
{
   private StudentDAO studentDAO;

   public DefaultStudentSystem(StudentDAO studentDAO)
   {
      this.studentDAO = studentDAO;
   }

   public int getNrOfStudents( String subjectCode )
   {
      List students = studentDAO.getAllStudents();
      // method logic goes here...
   }
}
```

Dependency Injection with Spring

- Configuration: How to instantiate, configure, and assemble the objects in your application
  - The Spring container accepts many configuration formats
  - XML based configuration and annotations most common, Java properties or programmatically




```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN“
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

   <bean id=”studentDAO"
      class="no.uio.inf5750.impl.FileStudentDAO“/>

</beans>
```

Dependency Injection with Spring

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN“
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

   <bean id=”studentDAO"
      class="no.uio.inf5750.impl.FileStudentDAO“/>

   <bean id=”studentSystem"
      class="no.uio.inf5750.impl.DefaultStudentSystem">
      <property name=”studentDAO”>
         <ref bean=”studentDAO"/>
      </property>
   </bean>

</beans>
```

Bean identifier, must be unique

Package-qualified class name, normally the implementation class

Refers to the studentStore attribute in the Java class

Refers to the studentStore bean

StudentStore injected into StudentRegister!
(Also known as Reference Injection)

Dependency Injection with Spring

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN“
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

   <bean id=”studentDAO"
      class="no.uio.inf5750.impl.MySQLStudentDAO“/>

   <bean id=”studentSystem" class="no.uio.inf5750.impl.DefaultStudentSystem">
      <constructor-arg ref=”studentDAO"/>
   </bean>

</beans>
```

Change to use the MySQL implementation instead of the File implementation

University of Oslo can now use the system by altering one configuration line without changing the compiled code!

… OR for Setter DI

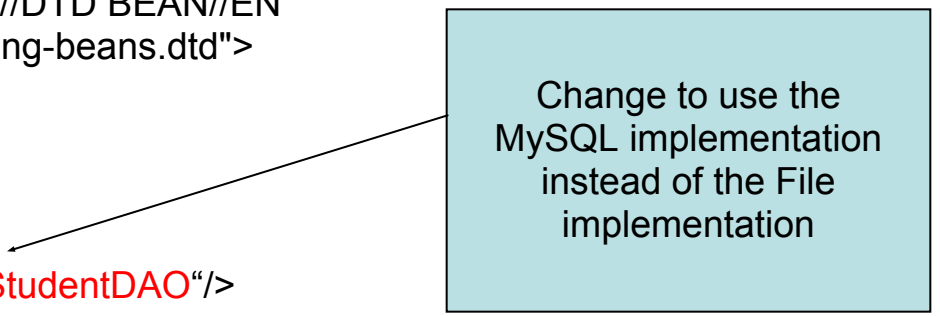
```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN“
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <bean id=”studentDAO"
        class="no.uio.inf5750.impl.MySQLStudentDAO“/>

    <bean id=”studentSystem"
        class="no.uio.inf5750.impl.DefaultStudentSystem">
        <property name=”studentDAO”>
            <ref bean=”studentDAO"/>
        </property>
    </bean>

</beans>
```



University of Oslo can now use the system by altering one configuration line without changing the compiled code!

Advantages of DI

- Flexibility
  - Easier to swap implementations of dependencies
  - The system can be re-configured without changing the compiled code
- Reusability
  - Dependencies can be injected into components by need
- Testability
  - Dependencies can be mocked
- Maintainability
  - Improves “single responsibility” in components
  - Cleaner code

Bean properties

- Bean properties can be values defined inline as well (not only references to other beans)

- Spring's XML-based configuration supports
  - Straight values (primitives, Strings...)
  - Collections (Lists, Sets, Maps)
  - Properties

Example: Straight values

```
public class DefaultStudentSystem
    implements StudentSystem
{
    private int maxNrOfStudentsPerCourse;

    public void setMaxNrOfStudentsPerCourse( int nr )
    {
        this.maxNrOfStudentsPerCourse = nr;
    }
```

Java bean

```
<bean id="studentSystem"
    class="no.uio.inf5750.impl.DefaultStudentSystem">
    <property name="maxNrOfStudentsPerCourse">
        <value>100</value>
    </property>
</bean>
```

Spring XML configuration file (beans.xml)

Example: Collections

```
public class DefaultStudentSystem
    implements StudentSystem
{
    private Map subjects;

    public void setSubjects( Map subjects )
    {
        this.subjects = subjects;
    }
```

Java bean

```
<bean id="studentSystem"
    class="no.uio.inf5750.impl.DefaultStudentSystem">
    <property name="subjects">
        <entry>
            <key><value>INF5750</value></key>
            <value>Open Source Software Development</value>
        </entry>
        <entry>
            <key><value>INF5760</value></key>
            <value>Health Information Systems</value>
        </entry>
    </property>
</bean>
```

Spring XML configuration file (beans.xml)

Summary

Class where dependencies are being injected

```
public class DefaultStudentSystem
    implements StudentSystem
{
    private StudentDAO studentDAO;

    @Required
    public void setStudentDAO( StudentDAO studentDAO )
    {
        this.studentDAO = studentDAO;
    }
```

Configuration

The implementation to inject

FileStudentDAO

IoC Container (Spring)

```
<bean id="studentDAO"
class="no.uio.inf5750.impl.FileStudentDAO"/>

<bean id="studentSystem"
class="no.uio.inf5750.impl.DefaultStudentSystem">
    <property name="studentDAO">
        <ref bean="studentDAO"/>
    </property>
</bean>
```

<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanFactoryPostProcessor"/>

IoC using Annotations

- Annotation: Meta-tag applied to classes, methods, props
  - Affects the way tools and frameworks treat source code
  - Typically used for configuration
- @Component
  - Defines a class as a Spring container-managed component
- @Autowired
  - Tells Spring to inject a component
- Classpath scanning for components

  <context:component-scan base-package="org.example"/>
- Autowiring modes:
  - By type: injects bean with same class type as property (default)
  - By name: injects bean with same name as property

IoC using Annotations

```
<context:component-scan base-package="no.uio.inf5750"/>
```

Spring config. Enables auto-wiring and detection of components

```
@Component
public class DefaultStudentDao implements StudentDao
{
    // Implementatation omitted
}
```

Class marked as @Component. Will be detected by container.

```
@Component
public class DefaultStudentSystem implements StudentSystem
{
    @Autowired
    private StudentDao studentDao;

    public void saveStudent( Student student )
    {
        studentDao.save( student );
    }
}
```

StudentDao will be autowired and is ready to use

Annotations or XML config?

- Annotations:
  - More concise configuration
  - Faster development

- XML:
  - Gives overview of all beans and dependencies in the system
  - Keeps the source code unaware of the container
  - All wiring is explicit (multiple implementations easier to manage)

Resources

- Spring reference documentation

  www.springframework.org -> Documentation -> Reference manual

- http://www.springbyexample.org/examples/intro-to-ioc.html
- http://kohari.org/2007/08/15/defending-dependency-injection/
- http://blog.springsource.org/2007/07/11/setter-injection-versus-constructor-injection-and-the-use-of-required/
- http://en.wikipedia.org/wiki/Loose_coupling