Software Design and Architecture

Dependency Injection

Design principles

High-level principles

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution Principle
- **I**nterface Segregation
- **Dependency Inversion**

Low-level principles

- Encapsulate what varies
- **Program to interfaces, not implementations**
- Favor composition over inheritance
- Strive for loose coupling

Problem

How can we wire interfaces together without creating a dependency on their concrete implementations?

A challenge of the "**program to interfaces, not implementations** " design principle

Problem

How can we wire interfaces together without creating a dependency on their concrete implementations?

Idea

- Would like to reduce (eliminate) coupling between concrete classes
- Would like to be able to substitute different implementations without recompiling
  e.g., be able to test and deploy the same binary even though some objects may vary

Solution

Separate objects from their assemblers (also called **inversion of control**)

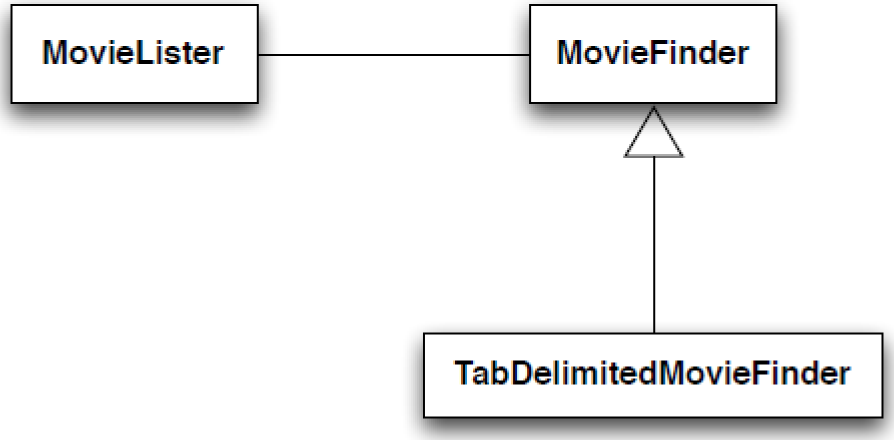
Dependency Injection

A technique for assembling applications from a set of concrete classes (implementing generic interfaces) without the concrete classes knowing about each other

Dependency Injection

Allows us to create loosely coupled systems as your code only ever references the generic interfaces that hide the concrete classes

An Example



A MovieLister class is able to list movies with certain characteristics after being provided a database of movies by an instance of MovieFinder

MovieFinder is an interface; TabDelimitedMovieFinder is a concrete class that can read in a movie database that is stored in a tab-delimited text file

An Example

Our goal is to avoid having our code depend on concrete classes

In other words, we do **NOT** want to see something like this

```
public class MovieLister {
      private MovieFinder finder;
      public MovieLister() {
      this.finder = new TabDelimitedMovieFinder("movies.txt"); }
 …}
```

TabDelimitedMovieFinder – **dependency on concrete class**

movies.txt – **dependency on hard-coded string**

An Example

```
    public class MovieLister {
        private MovieFinder finder;
        public MovieLister() {
        this.finder = new TabDelimitedMovieFinder("movies.txt"); }
     ...}
```

TabDelimitedMovieFinder – **dependency on concrete class**

movies.txt – **dependency on hard-coded string**

Problem with these two concrete dependencies

(1) The name of the movie database cannot be changed without causing MovieLister to be changed and recompiled

(2) The format of the database cannot be changed without causing MovieLister to be changed to reference the name of the new concrete MovieFinder implementation

An Example

Our Target: loose-coupling Code

```
public class MovieLister {
    private MovieFinder finder;
    public MovieLister(MoveFinder finder) {
        this.finder = finder;
    }
    …
}
```

and, furthermore, nowhere in our source code should the strings "TabDelimitedMovieFinder" or "movies.txt" appear… **nowhere!**

An Example

Our Target: loose-coupling Code

We would also like to see code like this

```
public class Main {
    private MovieLister lister;
    public void setMovieLister(MovieLister lister) { this.lister = lister;}
    public List<Movie> findWithDirector(String director) {
        return lister.findMoviesWithDirector(director);
    }
    public static void main(String[] args) {
        new Main().findWithDirector(args[0]); // add code to print list of movies
}
}
```

Two types of dependency injection

Constructor Injection

```
public MovieLister(MovieFinder finder) {
        this.finder = finder;
}
```

Setter Injection

```
public void setMovieLister(MovieLister lister) {
        this.lister = lister;}
```

Two types of dependency injection

Constructor Injection

```
public MovieLister(MovieFinder finder) {
        this.finder = finder;
}
```

The MovieLister class indicates its dependency via its constructor ("I need a MovieFinder");

Two types of dependency injection

Setter Injection

```
        public void setMovieLister(MovieLister lister) {
                this.lister = lister;}
```

The Main class indicated its dependency via a setter ("I need a MovieLister")

What is dependency injection?

The idea here is that classes in an application indicate their dependencies in very abstract ways

- MovieLister NEEDS-A MovieFinder
- Main NEEDS-A MovieLister

and then **a third party injects (or inserts) a class** that will meet that dependency at run-time

What is dependency injection?

The “third party” is known as a “Inversion of Control container” or a “dependency injection framework”

- There are many such frameworks; one example is **Spring** which has been around in some form since October 2002

The Basic Idea

Take

- a set of components (concrete classes + interfaces)

Add

- a set of configuration metadata

Provide that to

- a dependency injection framework

And finish with

- a small set of bootstrap code that gets access to an **IoC (Inverse of Control) container**, retrieves the first object from that container by supplying the name of a generic interface, and invokes a method to kick things off

Example: Spring-specific code

Spring-specific code to create an instance of MovieLister

```
public void testWithSpring() throws Exception {
    ApplicationContext ctx = new
    FileSystemXmlApplicationContext("spring.xml");
    MovieLister lister = (MovieLister) ctx.getBean("MovieLister");
    Movie[] movies = lister.moviesDirectedBy("Terry Gilliam");
}
```

Example: Spring-specific code

"spring.xml" is a standard-to-Spring XML file containing metadata about our application; it contains information that specifies that MoveLister needs a TabDelimitedMovieFinder and that the database is in a file called "movies.txt"

Spring then ensures that TabDelimitedMovieFinder is created using "movies.txt" and inserted into MovieLister when **ctx.getBean()** is invoked

What is getBean()?

In Spring, POJOs (plain old java objects) are referred to as “beans”

- This is a reference to J2EE’s notion of a JavaBean
  - which is a Java class that follows certain conventions
    - property “foo” of type String is accessible via

      public String getFoo();
    - and

      public void setFoo(String foo);

What is getBean()?

Once you have specified what objects your application has in a Spring configuration file, you pull instances of those objects out of the Spring container via the getBean method

Spring’s Hello World example

It’s horribly complex for a Hello World program

- The complexity is reduced however when you realize that Spring is architected for really large systems
- and the “complexity tax” imposed by the framework pays off when you are dealing with large numbers of objects that need to be composed together
- the “complexity tax” pays dividends when you are able to add a new type of object to a Spring system by adding a new .class file to your classpath and updating one configuration file

Spring's Hello World example

First, define a MessageSource class

```
public class MessageSource {

  private String message;

  public MessageSource(String message) {
    this.message = message;
  }

  public String getMessage() {
    return message;
  }

}
```

Spring’s Hello World example

Second, define a MessageDestination interface and a concrete implementation

```
public interface MessageDestination {

  public void write(String message);

}



public class StdoutMessageDestination implements MessageDestination {

  public void write(String message) {
    System.out.println(message);
  }

}
```

Spring's Hello World example

Third, define a MessageService interface

```
public interface MessageService {

  public void execute();

}
```

Spring’s Hello World example

Fourth, define a concrete implementation of MessageService

```
public class DefaultMessageService implements MessageService {

  private MessageSource source;
  private MessageDestination destination;

  public void setSource(MessageSource source) {
    this.source = source;
  }

  public void setDestination(MessageDestination destination) {
    this.destination = destination;
  }

  public void execute() {
    destination.write(source.getMessage());
  }
}
```

Spring's Hello World example

Fifth, create a main program that gets a Spring container, retrieves a MessageService bean, and invokes the service

```
import org.springframework.beans.factory.support.BeanDefinitionReader;
import org.springframework.beans.factory.support.DefaultListableBeanFactory;
import org.springframework.beans.factory.support.PropertiesBeanDefinitionReader;
import org.springframework.core.io.FileSystemResource;

import java.io.File;

public class DISpringHelloWorld {

  public static void main(String[] args) {
    DefaultListableBeanFactory bf = new DefaultListableBeanFactory();//Spring init code
    BeanDefinitionReader reader = new PropertiesBeanDefinitionReader(bf);
    reader.loadBeanDefinitions(
      new FileSystemResource(
        new File("C:/Users/boonjv/Desktop/hello/hello.properties")));

    MessageService service = (MessageService) bf.getBean("service");// where the magic happens
    service.execute();
  }

}
```

Spring’s Hello World example

“**magic**” on the previous slide is because with the call to getBean(), the following things happen automatically

- an instance of MessageSource is created and configured with a message
- an instance of StdoutMessageDestination is created
- an instance of MessageService is created
- the previous two instances (message source, message destination) are plugged into MessageService

Spring’s Hello World example

In short, you got back an instance of MessageService without having to create any objects; and, the object you got back was ready for use

- you just had to invoke “execute()” on it

Spring's Hello World example

How does the magic happen?

- With the hello.properties file

```
source.(class)=MessageSource
source.$0=The semester is almost over!!@!!
destination.(class)=StdoutMessageDestination
service.(class)=DefaultMessageService
service.source(ref)=source
service.destination(ref)=destination
```

- It defines three beans: source, destination, and service
- $0 refers to a constructor argument; (class) sets the concrete class of the bean; (ref) references a bean defined elsewhere
- With this information, the "**service**" bean can be created and configured

Spring’s Hello World example

XML Configuration Files

The use of property files are now deprecated; instead, configuration metadata is stored in XML files; Here’s an XML file equivalent to hello.properties:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:lang="http://www.springframework.org/schema/lang"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.5.xsd">

  <bean id="source" class="MessageSource">
    <constructor-arg index="0" value="Hello XML Spring" />
  </bean>

  <bean id="destination" class="StdoutMessageDestination" />

  <bean id="service" class="DefaultMessageService">
    <property name="source" ref="source" />
    <property name="destination" ref="destination" />
  </bean>

</beans>
```

Spring's Hello World example

To use hello.xml, the main program is simplified to:

```
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

import java.io.File;

public class DIXMLSpringHelloWorld {

 public static void main(String[] args) {
  XmlBeanFactory bf =
   new XmlBeanFactory(
    new FileSystemResource(
     new File("C:/Users/boonjv/Desktop/hello/hello.xml")));

  MessageService service = (MessageService) bf.getBean("service");
  service.execute();
 }

}
```