

Dependency Injection

CSCI 4448/5448: Object-Oriented Analysis & Design

Lecture 36

Acknowledgement & Materials Copyright

- I'd like to start by acknowledging Dr. Ken Anderson
- Ken is a Professor and the Chair of the Department of Computer Science
- Ken taught OOAD on several occasions, and has graciously allowed me to use his copyrighted material for this instance of the class
- Although I will modify the materials to update and personalize this class, the original materials this class is based on are all copyrighted © Kenneth M. Anderson; the materials are used with his consent; and this use in no way challenges his copyright

Goals of the Lecture

- Define Dependency Injection and Inversion of Control
- Show Fowler's DI Example
- Look at an example in Spring
- Look at a Plain Old Java version

What is Dependency Injection?

- Dependency Injection is
 - a technique for assembling applications
 - from a set of concrete classes
 - that implement generic interfaces
 - without the concrete classes knowing about each other
- This allows you to create loosely coupled systems as the code you write only ever references the generic interfaces that hide the concrete classes
- Dependency Injection is discussed in a famous blog post by Martin Fowler (he's back!)
- <http://martinfowler.com/articles/injection.html>

So what's Inversion of Control?

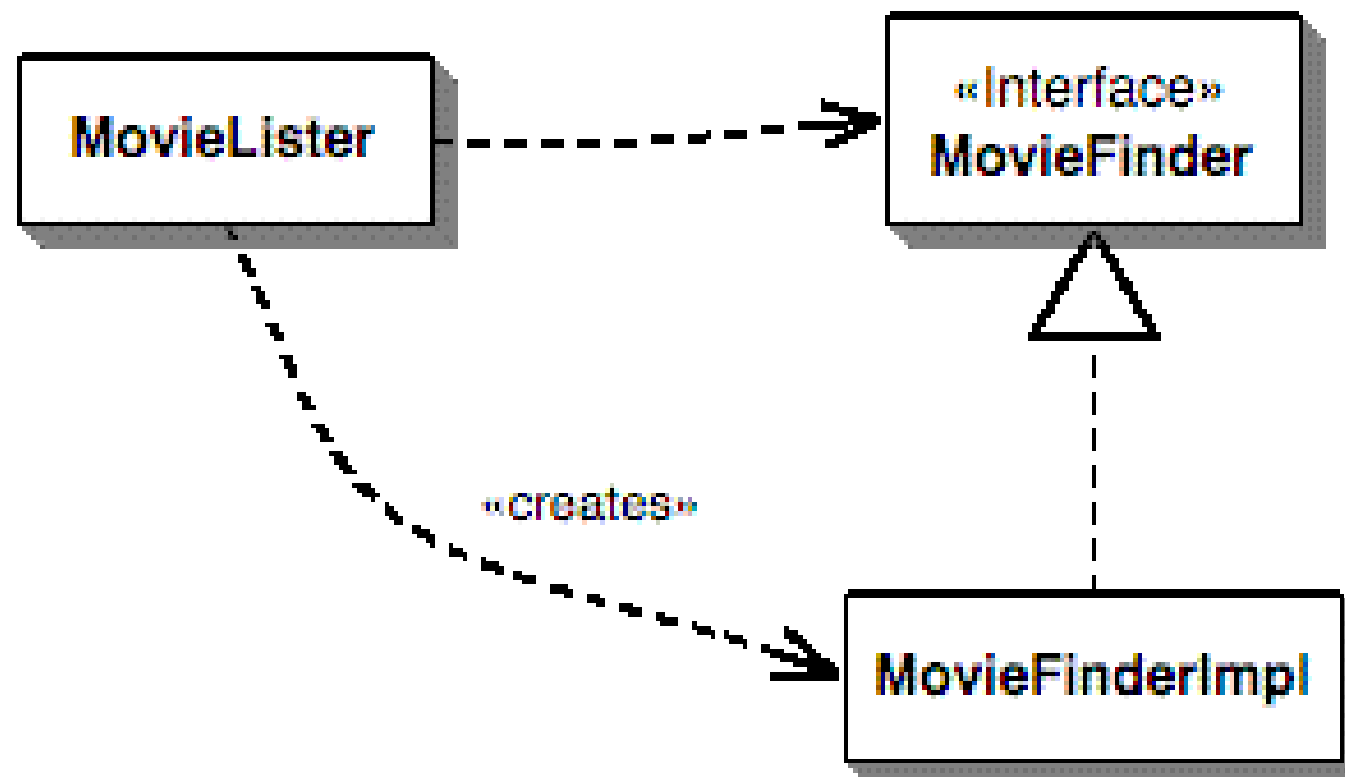
- You hear Inversion of Control with Dependency Injection often...
- **Inversion of Control** is a principle in software engineering by which the control of objects or portions of a program is transferred to a container or framework
- Fowler argued that Inversion of Control was common in using **any** frameworks
- Any time the main control of the program was “inverted” away from your code to the framework code
- He cites an example of inverting control in a UI framework – where instead of the UIs being controlled by the application program, the UI framework would contain the interactions and your program instead would provide event handlers for the various fields on the screen

Fowler's Definition of Dependency Injection

- He used the term **Dependency Injection** specifically for an approach that a container uses to ensure that any user of a plugin follows some convention that allows a separate assembler module to inject the implementation into the lister
- Dependency injection is a pattern which implements Inversion of Control, where the control being inverted is the setting of object dependencies
- The act of connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves

Fowler's Dependency Injection Example

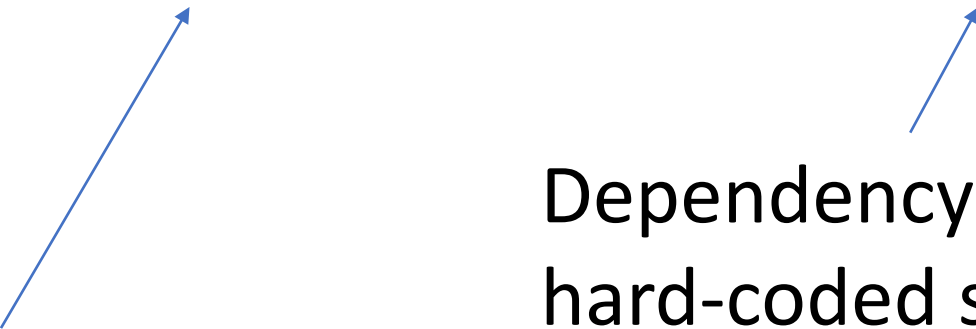
- A MovieLister class is able to list movies with certain characteristics after being provided a database of movies by an instance of MovieFinder
- MovieFinder is an interface; TabDelimitedMovieFinder is a concrete class that can read in a movie database that is stored in a tab-delimited text file



Goal: Loose Coupling in Systems

- Our goal (even with the simple system on the previous slide) 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");  
    }  
    ...  
}
```



Dependency on
Concrete Class

Dependency on
hard-coded string

Discussion

- The code on the previous slide has two concrete dependencies
 - a reference to a concrete class that implements MovieFinder
 - a reference to a hard-coded string
- Both references are brittle
 - The name of the movie database cannot change without causing MovieLister to be changed and recompiled
 - The format of the database cannot change without causing MovieLister to be changed to reference the name of the new concrete MovieFinder implementation

Our Target (I)

- For loose-coupling to be achieved, we need code that looks like this

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

- and, furthermore, nowhere in our source code should the strings “TabDelimitedMovieFinder” or “movies.txt” appear... nowhere!

Our Target (II)

- As much as possible, get rid of code with the form
 - `Foo f = new ConcreteFoo();`
- Indeed, for the MovieLister system, we would even 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) {  
        // add code to print list of movies  
        new Main().findWithDirector(args[0]);  
    }  
}
```

- We want this to work even with no explicit call to `setMovieLister()`;

Two types of dependency injection

- In the previous two slides, we've seen (implied) examples of 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")

- **Setter Injection**

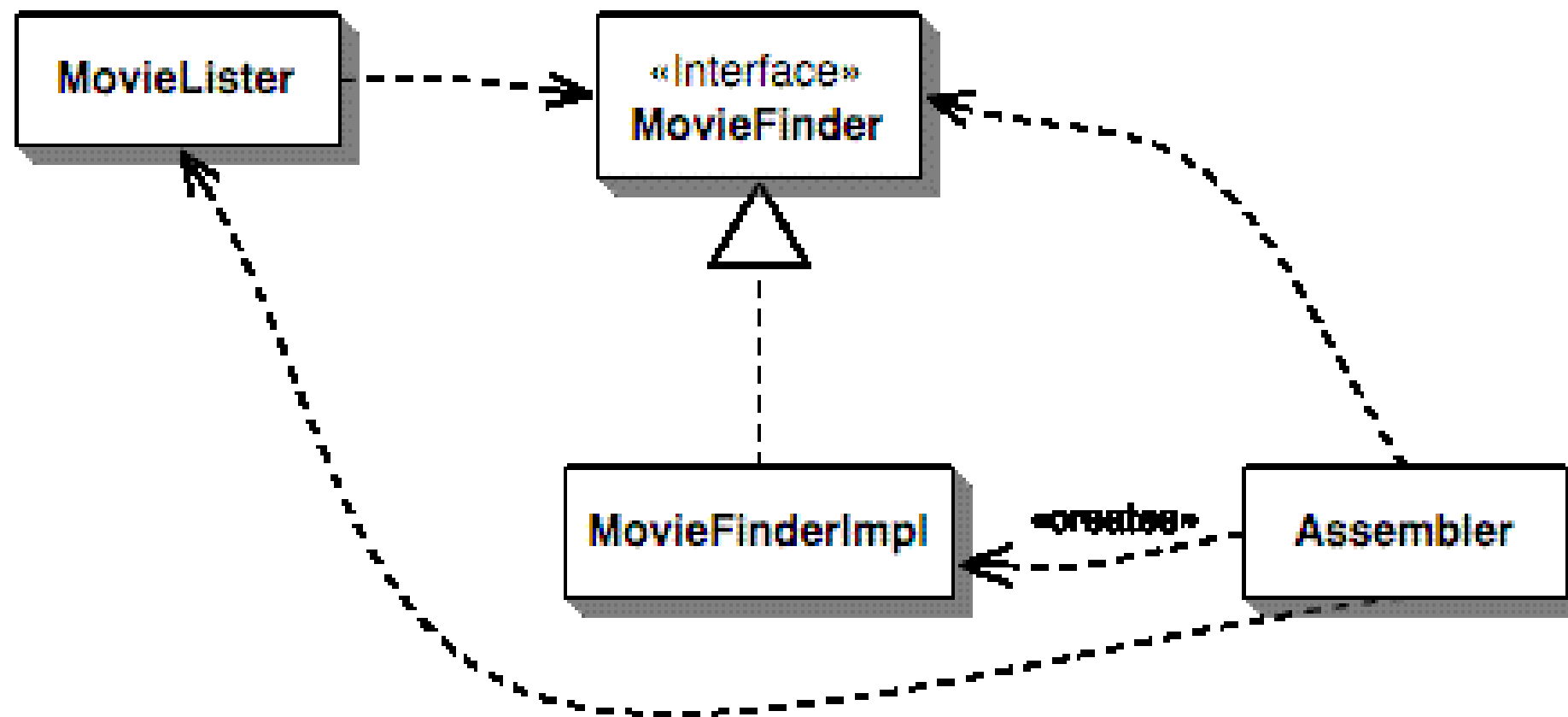
```
public void setMovieLister(MovieLister lister) { this.lister = lister;}
```

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

- There's also **Interface Injection** where an interface provides an injector method to inject dependencies into passed in clients...

Fowler's UML for Dependency Injection

- This is how Fowler shows it in UML...
- An assembler creates (injects) a MovieFinder (Constructor Injection) or provides a MovieLister (Setter Injection)



So, what is dependency injection again?

- One way to think about this is as a “NEEDS-A” relationship
 - Vs. IS-A for inheritance or HAS-A for delegation
- 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
- The “third party” is known as an assembler, and comes from an “Inversion of Control (IoC) container” or a “dependency injection framework”
- There are many such frameworks; one example is part of 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 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

- For instance, Fowler's example uses the following 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");  
}
```

- “spring.xml” is a standard-to-Spring XML file containing metadata about our application; it contains information that specifies that MovieLister 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
- (Note: this example is based on an older Spring version, but it still shows the mechanics of Dependency Inversion)

getBean()?

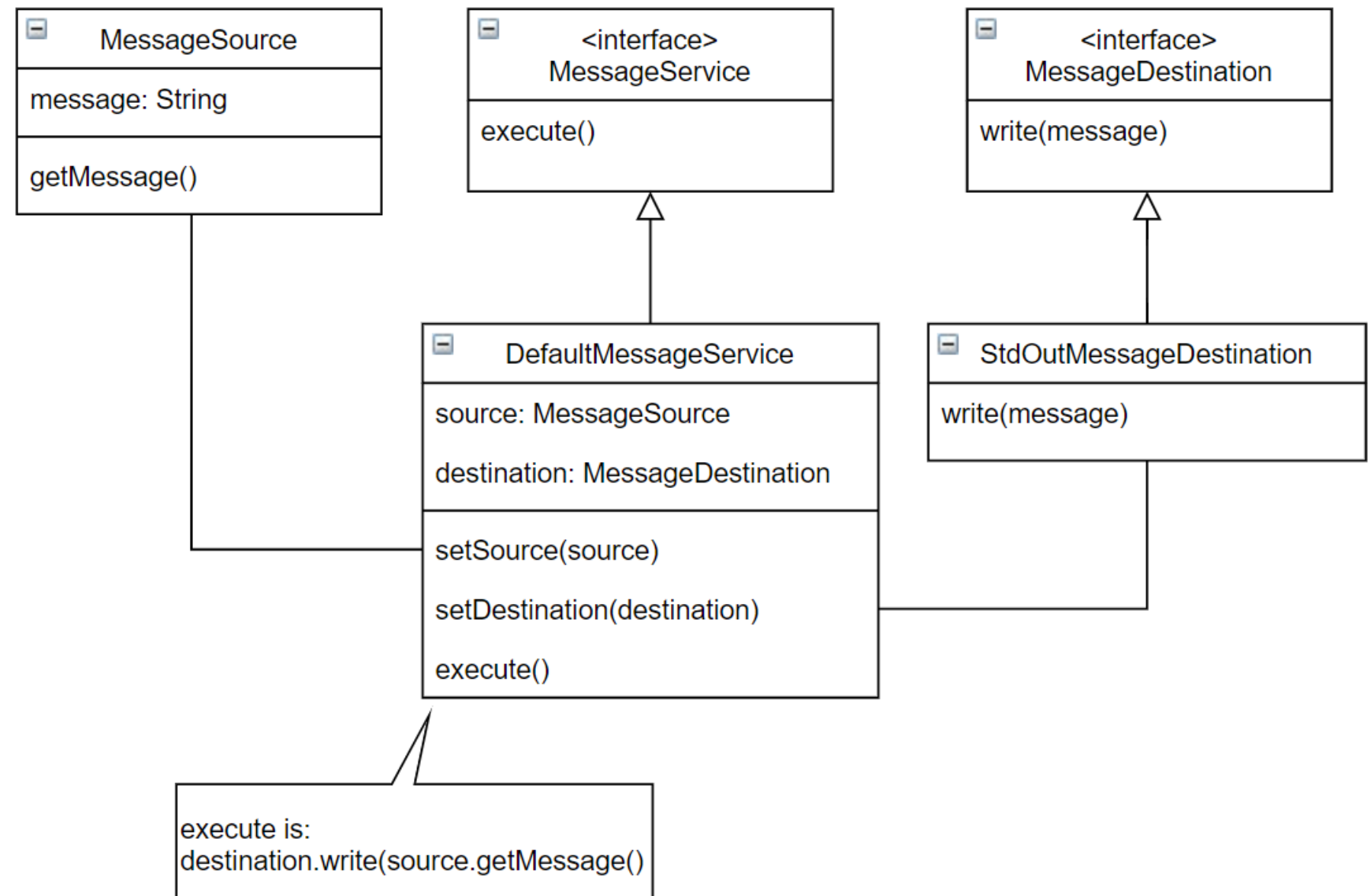
- In Spring, POJOs (plain old java objects) are referred to as “beans”
- This is a reference to J2EE’s notion of a JavaBean which is a Java class that follows certain conventions
 - a property “foo” of type String is accessible via
 - `public String getFoo();`
 - and
 - `public void setFoo(String foo);`
- 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

- I shall now possibly horrify you with a “Hello World” example written using Spring
- I say “horrify” because it will seem horribly complex for a Hello World program
- The complexity is reduced however when you realize that Spring is architected for really large systems
 - 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
- Example is from Apress book “Pro Spring 2.5”
 - Again – Spring is up to 5.3.8, Spring 5 did some re-architecting, but most of the plumbing for DI is still the same...
 - This is primarily for illustration of points, not for teaching Spring, but I will provide more tutorial/reference links if you want to try it out

UML Class Diagram for Hello World Example

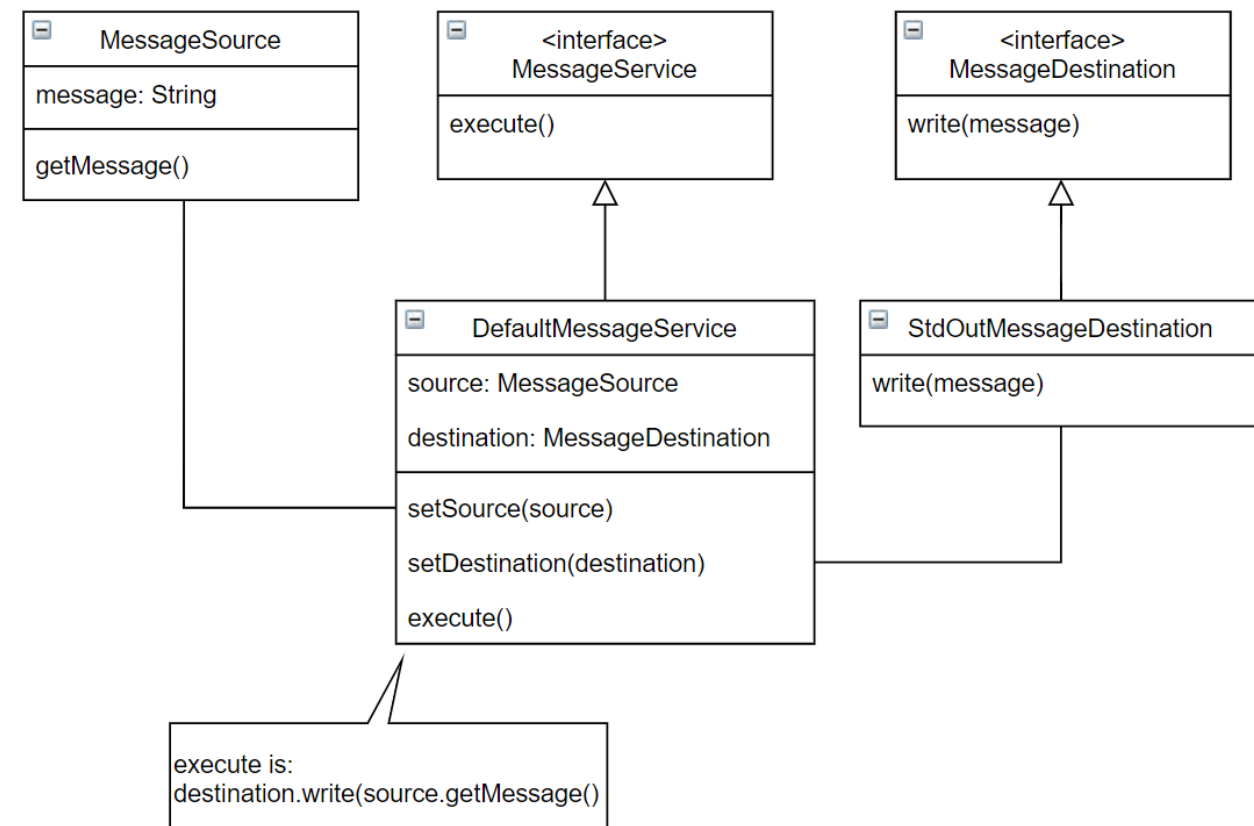
- A MessageService is provided that will let us set a MessageSource for messages and a MessageDestination for messages
- The MessageService provides an execute method that will get the message from the source and have the destination write the message
- The MessageService is what we want to inject



Spring's Hello World (I)

- First, define a MessageSource class

```
public class MessageSource {  
    private String message;  
    public MessageSource(String message) {  
        this.message = message;  
    }  
    public String getMessage() {  
        return message;  
    }  
}
```

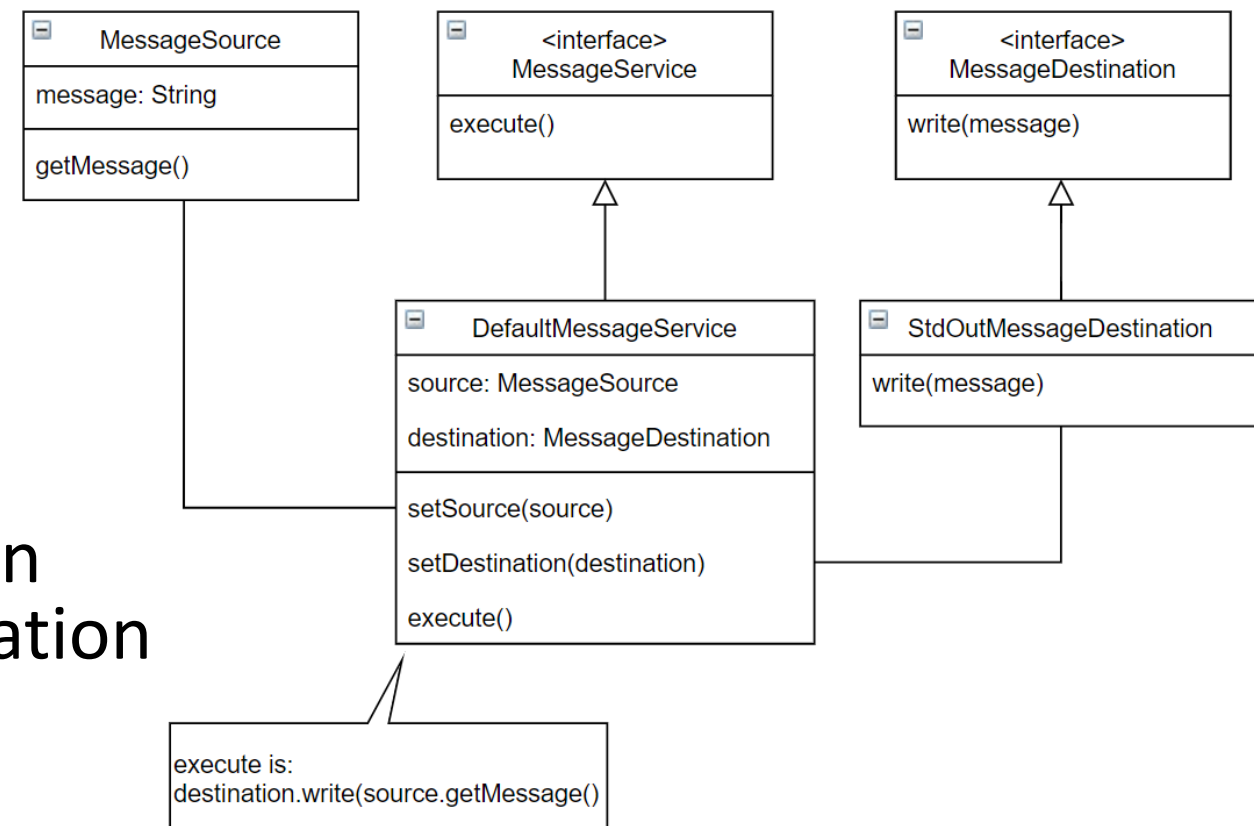


Spring's Hello World (II)

- Second, define a MessageDestination interface and a concrete implementation

```
public interface MessageDestination {  
    public void write(String message);  
}
```

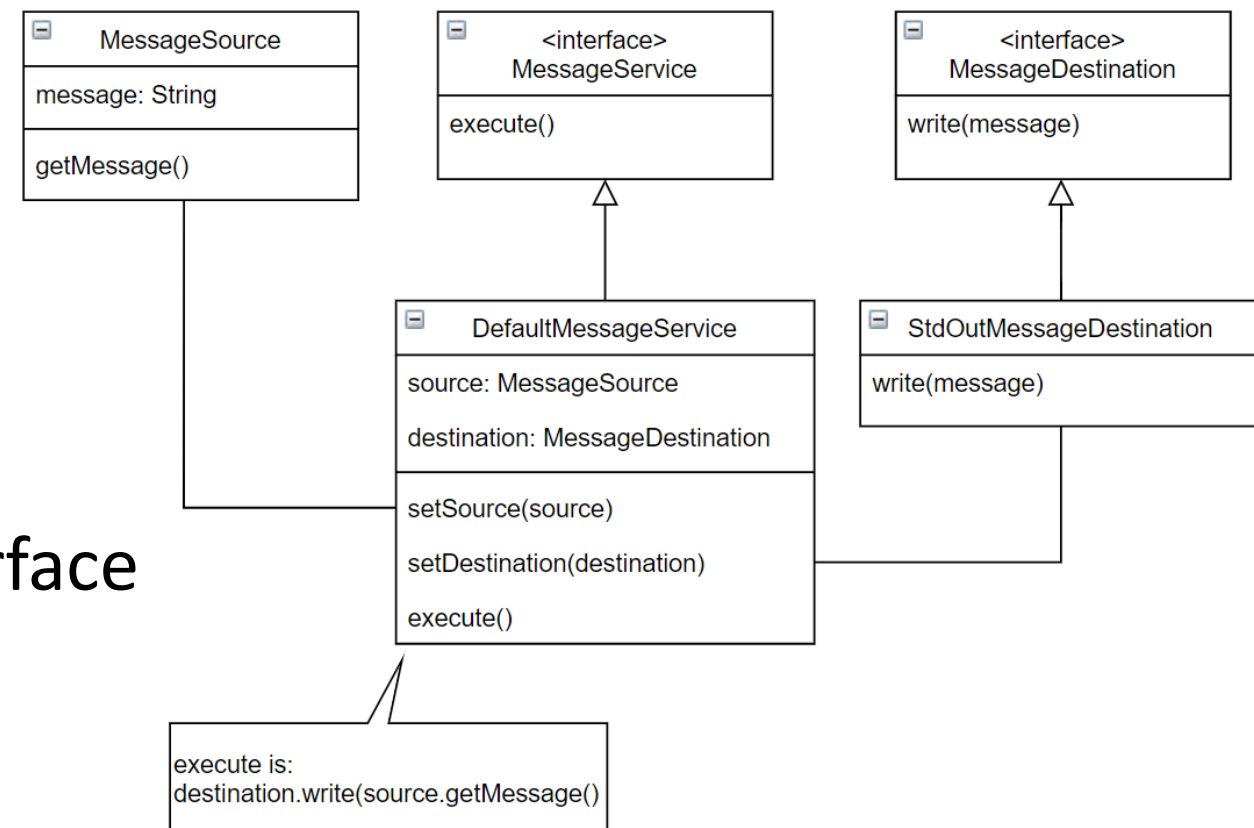
```
public class StdoutMessageDestination implements MessageDestination {  
    public void write(String message) {  
        System.out.println(message);  
    }  
}
```



Spring's Hello World (III)

- Third, define a MessageService interface

```
public interface MessageService {  
    public void execute();  
}
```



Spring's Hello World (IV)

- Fourth, define a concrete implementation of MessageService

```
public class DefaultMessageService  
    implements MessageService {
```

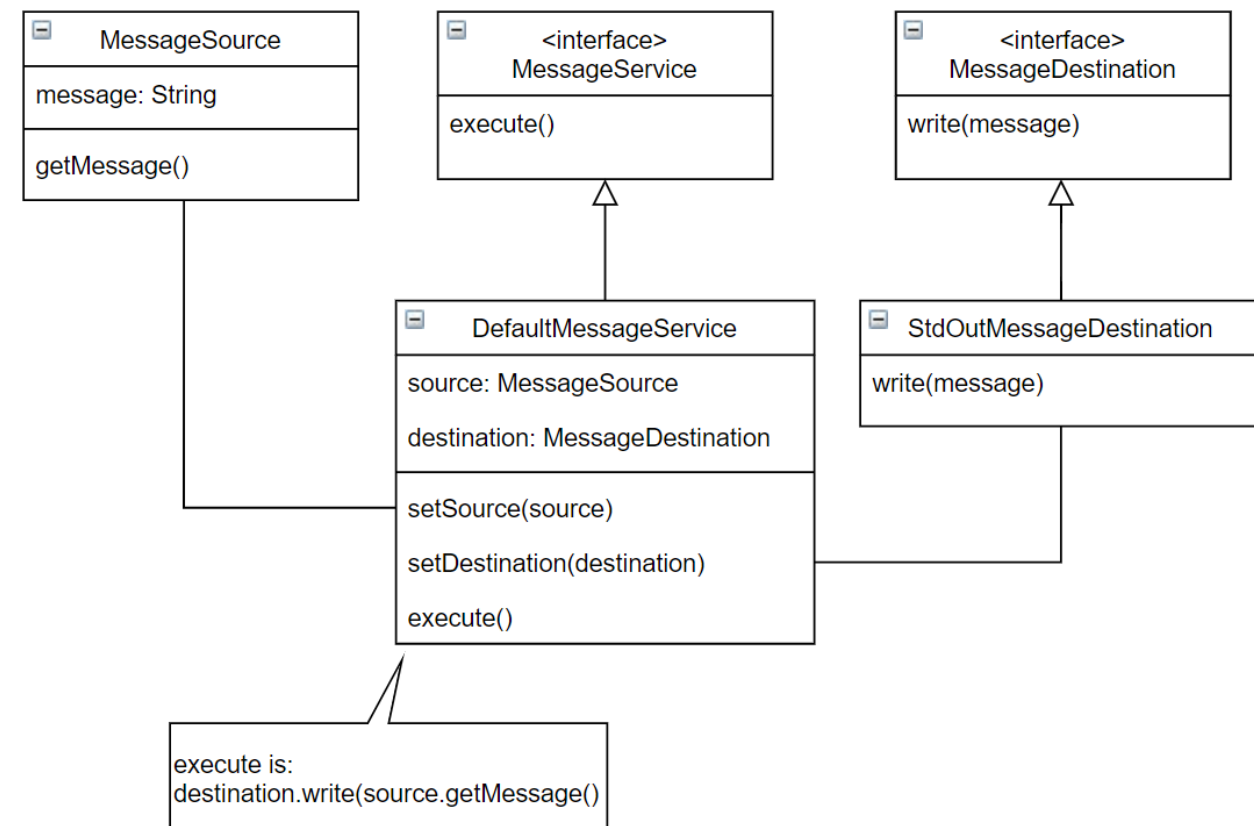
```
    private MessageSource source;  
    private MessageDestination destination;
```

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

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

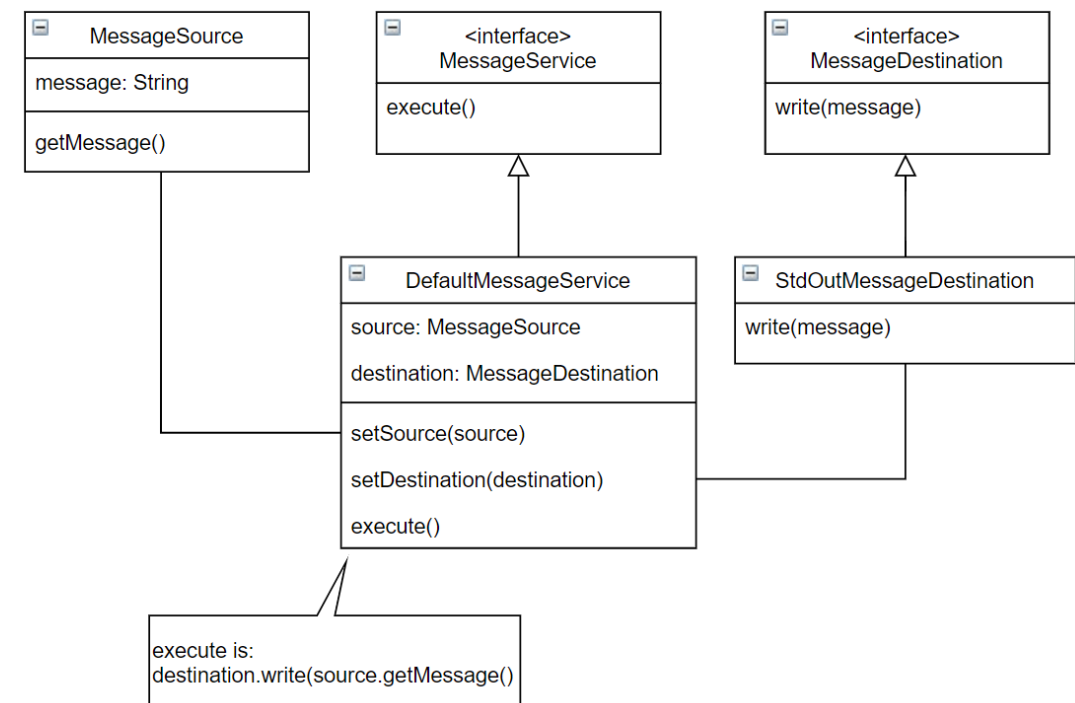
```
    public void execute() {  
        destination.write(source.getMessage());  
    }
```

```
}
```



Spring's Hello World (V)

- 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;
```

Spring Init Code

```
public class DISpringHelloWorld {
    public static void main(String[] args) {
        DefaultListableBeanFactory bf = new DefaultListableBeanFactory();
        BeanDefinitionReader reader = new PropertiesBeanDefinitionReader(bf);
        reader.loadBeanDefinitions(
            new FileSystemResource(
                new File("hello.properties")));
        MessageService service = (MessageService) bf.getBean("service");
        service.execute();
    }
}
```

Where the magic happens

Spring's Hello World (VI)

- I say “magic” on the previous slide, because with that 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`
- 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 (VII)

- How does the magic happen?
- With the hello.properties file

```
source.(class)=MessageSource
source.$0="Hello Spring"
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

XML Configuration for the same example

- The use of property files were deprecated; instead, configuration metadata was 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/
    http://www.springframework.org/schema/lang http://www.springframework.org/schema/"
  <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 (VIII)

- 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("hello.xml")));

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

For examples of DI in the current Spring version, see:

<https://docs.spring.io/spring-framework/docs/current/reference/html/core.html#beans-factory-collaborators>

Return to the Zoo

- Using Spring and Dependency Injection, it is possible to create a version of the Zoo program that would only reference “Animal” and not any of its subclasses (Dog, Cat, Hippo, etc.)
- To do this in Spring, we make use of its ability to specify collection classes in its configuration XML files (see next slide)
- The main routine is simply a variant on what we’ve seen before
 - we will load a “zoo.xml” configuration file
 - retrieve the “zoo” bean
 - and invoke its “exerciseAnimals()” method

Zoo XML

- Here, we define that there is a bean called “zoo” and it takes a parameter to its constructor that is a list of beans, in this case beans that reference the Animal subclasses below
- Here we define instances of animal subclasses; this is where the subclass names are referenced (nowhere else)

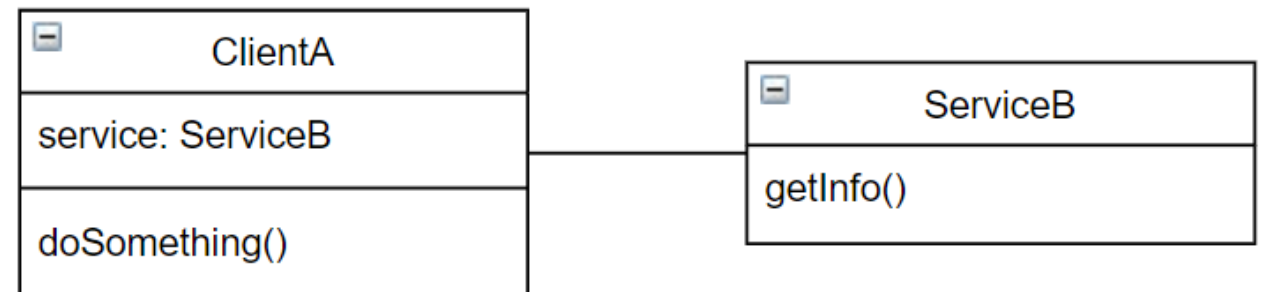
```
<bean id="zoo" class="Zoo">
  <constructor-arg index="0">
    <list>
      <ref local="bat" />
      <ref local="cat" />
      <ref local="dog" />
      <ref local="elephant" />
      <ref local="hippo" />
      <ref local="lion" />
      <ref local="rhino" />
      <ref local="tiger" />
      <ref local="wolf" />
    </list>
  </constructor-arg>
</bean>
```

```
<bean id="bat" class="Bat" />
<bean id="cat" class="Cat" />
<bean id="dog" class="Dog" />
<bean id="elephant" class="Elephant" />
<bean id="hippo" class="Hippo" />
<bean id="lion" class="Lion" />
<bean id="rhino" class="Rhino" />
<bean id="tiger" class="Tiger" />
<bean id="wolf" class="Wolf" />
```

A plain Java example

- In typical Java OO code, you have a class dependency to get things done, like here where ClientA is using ServiceB

```
public class ClientA {  
    ServiceB service;  
    public void doSomething() {  
        String info = service.getInfo();  
    }  
}  
  
public class ServiceB {  
    public String getInfo() {  
        return "ServiceB's Info";  
    }  
}
```

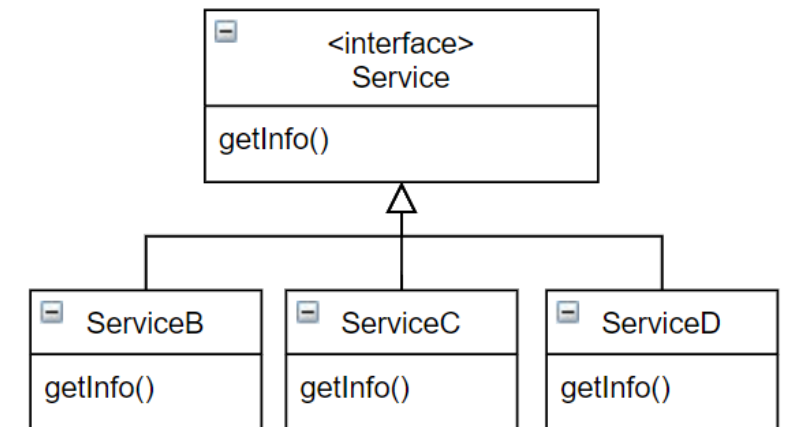
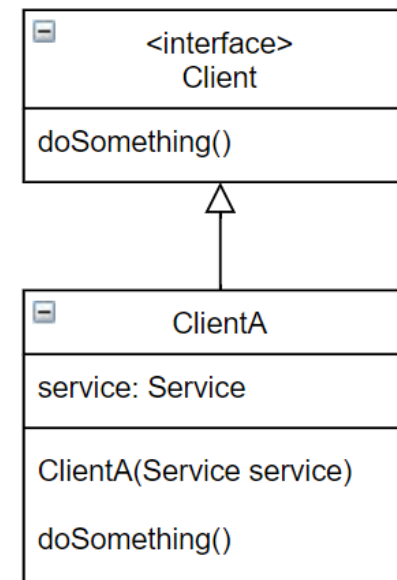


Refactoring for Dependency Injection

```
public interface Client {  
    void doSomething();  
}
```

```
public interface Service {  
    String getInfo();  
}
```

```
public class ServiceB implements Service {  
    @Override  
    public String getInfo() {  
        return "ServiceB's Info";  
    }  
}
```



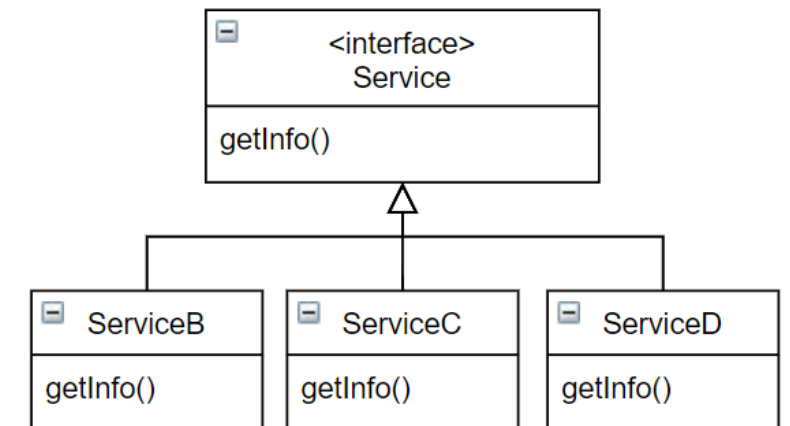
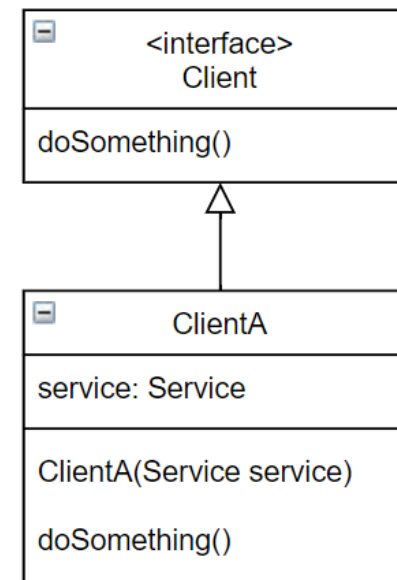
<https://www.codejava.net/coding/what-is-dependency-injection-with-java-code-example>

Alternate Services

- We may have alternative services to consider

```
public class ServiceC implements Service {  
    @Override  
    public String getInfo() {  
        return "ServiceC's Info";  
    }  
}
```

```
public class ServiceD implements Service {  
    @Override  
    public String getInfo() {  
        return "ServiceD's Info";  
    }  
}
```



<https://www.codejava.net/coding/what-is-dependency-injection-with-java-code-example>

Constructor Injection

- Instead of a concrete service, the Service implementation is “injected” via constructor
 - ClientA is not dependent on a specific Service implementation

public class ClientA implements Client {

Service service;

```
public ClientA(Service service) {  
    this.service = service;  
}
```

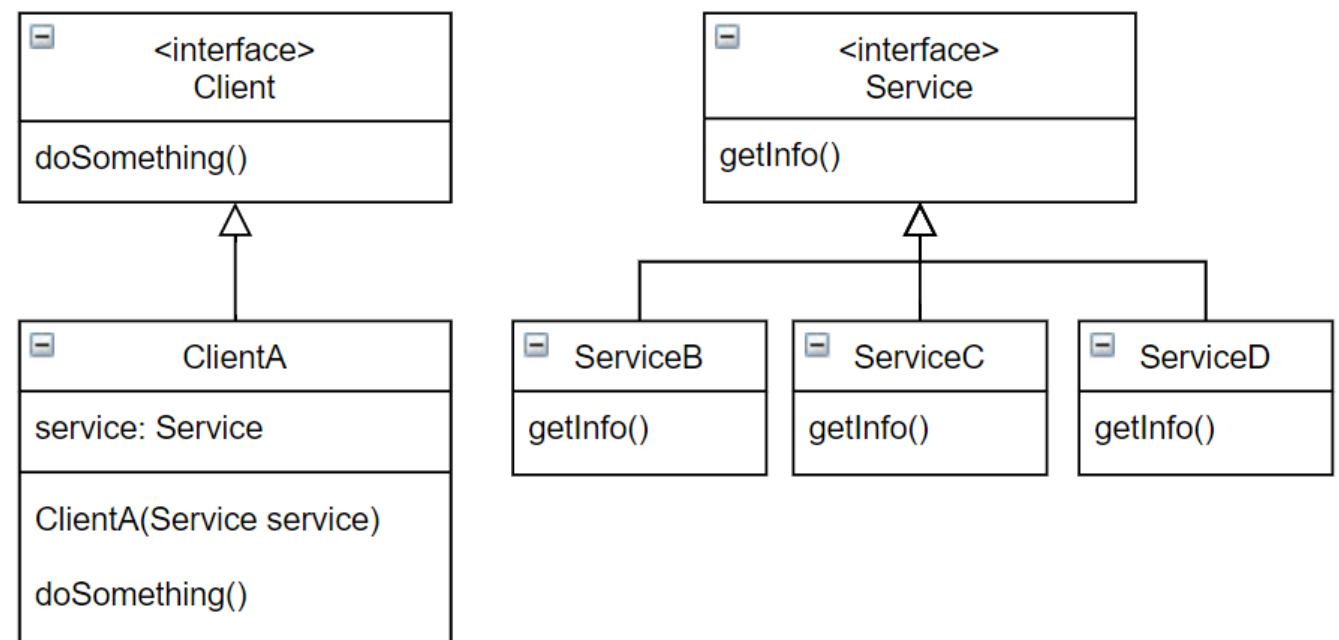
@Override

```
public void doSomething() {  
    String info = service.getInfo();  
}
```

```
}
```

- Here we’re creating an instance of the service, injecting it into ClientA

```
Service service = new ServiceB();  
Client client = new ClientA(service);  
client.doSomething();
```



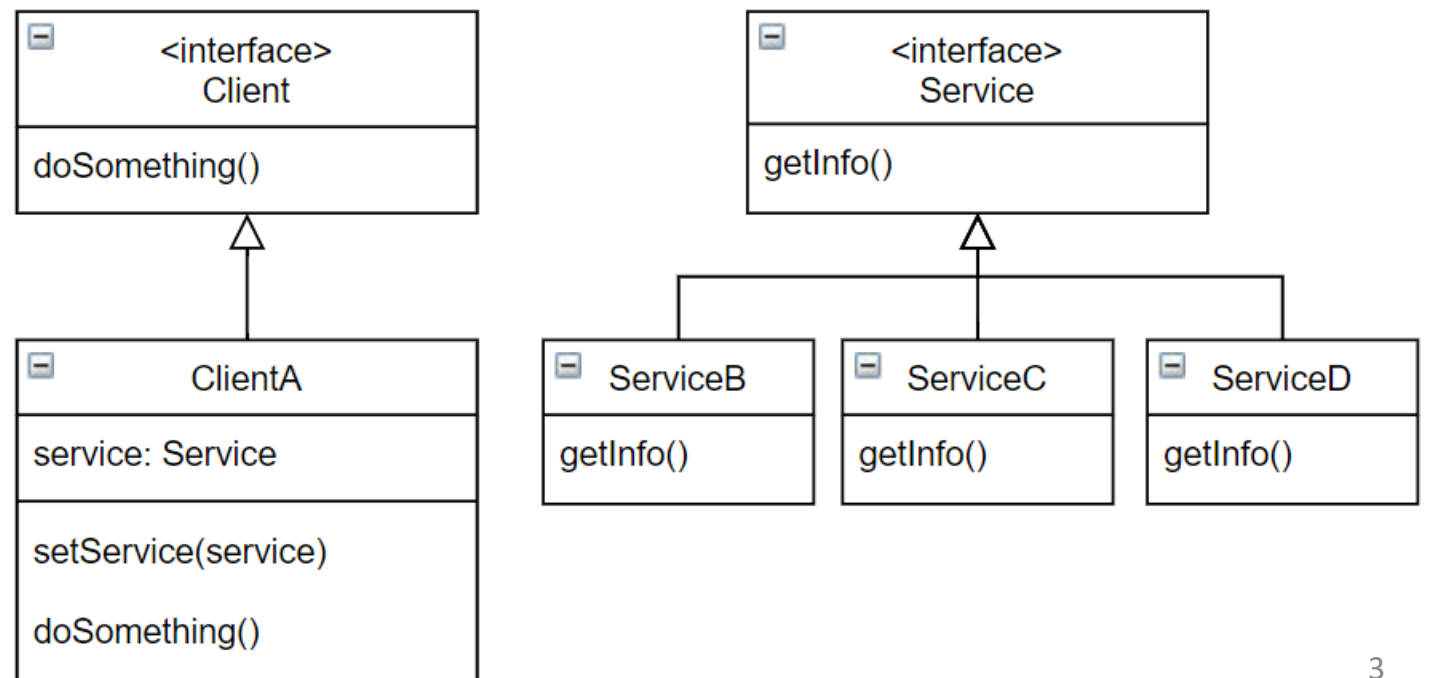
Setter Injection

- Setter Injection is used to pass a depending object to a dependent one – if we modify ClientA to this:

```
public void setService(Service service) {  
    this.service = service;  
}
```

- A Service Implementation might look like this:

```
((ClientA) client).setService(new ServiceC());  
client.doSomething();
```



Why would you use DI?

- Adding Dependency Injection can add complexity to a solution to gain the Inversion of Control, injecting elements from an external source into your code
- When would you need to provide for this high level of support for external variation in providing alternative implementations to classes or attributes?
- One typical example is testing, where you might want to inject test code for select unit tests
 - A great student submission on using DI in unit testing
 - <https://openclassrooms.com/en/courses/5684146-create-web-applications-efficiently-with-the-spring-boot-mvc-framework/6157116-make-services-unit-testable-using-dependency-injection>
- Another discussion on StackOverflow shows an example of variations in providing a Logger to code
 - from a single concrete Logger, to using a logging interface, to adding a Logger factory, to an external XML file defining the Logger function, to a DI framework that injects a certain Logger
 - Which would you use?
 - <https://stackoverflow.com/questions/14301389/why-does-one-use-dependency-injection>

Summary

- This represents a barebones introduction to dependency injection frameworks and patterns
 - Spring's functionality for this has changed over time
 - <https://www.vogella.com/tutorials/SpringDependencyInjection/article.html> shows an example of doing DI with annotations or with XML
 - But it's also easy to do some basic Dependency Injection-like code in plain Java
- You've seen the core feature of dependency injection frameworks
 - The ability to remove the names of concrete classes out of your source code while having those classes automatically instantiated and injected into your system based on configuration metadata
- Another good and current Spring example at
 - <https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring>
- Google Guice vs. Spring
 - <https://www.baeldung.com/guice-spring-dependency-injection>

Next Steps

- **Project 6 due Wed 11/17 at 8 PM**
 - Project 6 includes Zoom or in-person code demonstration sign-ups
 - Sign up for a Project 6 demo slot here:
<https://docs.google.com/document/d/1tsO0Tf5EIEs3elwQgh3n1RyiRY3dduhfuZ0YlhEVz2g/edit?usp=sharing>
- **Graduate Pecha Kucha due Mon 11/29 at 10 AM**
 - **You must sign up for a Pecha Kucha class presentation slot here:**
<https://docs.google.com/document/d/1EGvA3ZnKVhheJqdRUp7obqG7n3sLQPqCtV8mxwJhB0U/edit?usp=sharing>
- **Project 7 due Wed 12/8 at 8 PM**
 - Includes report, code, and recorded demonstration
- **Graduate Final Research Presentation due Wed 12/8 at 8 PM**
 - Details on assignments in Canvas Files/Class Files
- **New Quiz due Wed 11/17**
 - Last quiz will open next weekend, will be due 12/1
- **Coming soon...**
 - Reflection
 - Architecture
 - APIs
 - Anti-/Other Patterns
 - Graduate Pecha Kuchas
 - Wrapping up
- Posting extra bonus points from coding exercise shortly
- New Piazza topic this week for your comments for Participation Grade
- Piazza article posts now available for extra bonus points (if you're not getting points in class)
- Find a class staff member if you need anything!