



中国科学技术大学
University of Science and Technology of China

Refactoring HelloWorld Application

SSE USTC Qing Ding
dingqing@ustc.edu.cn

Theme of this Presentation



中国科学技术大学
University of Science and Technology of China

- How a simple *HelloWorld* application can be refactored in order to achieve the agility (and testability)?
 - How can I change a certain part of an application without affecting other parts of the code?
 - How can I wire different parts of the application without writing a lot of glue code myself?
 - How can I test the business logic without being tied up with a particular framework?

Refactoring HelloWorld Application

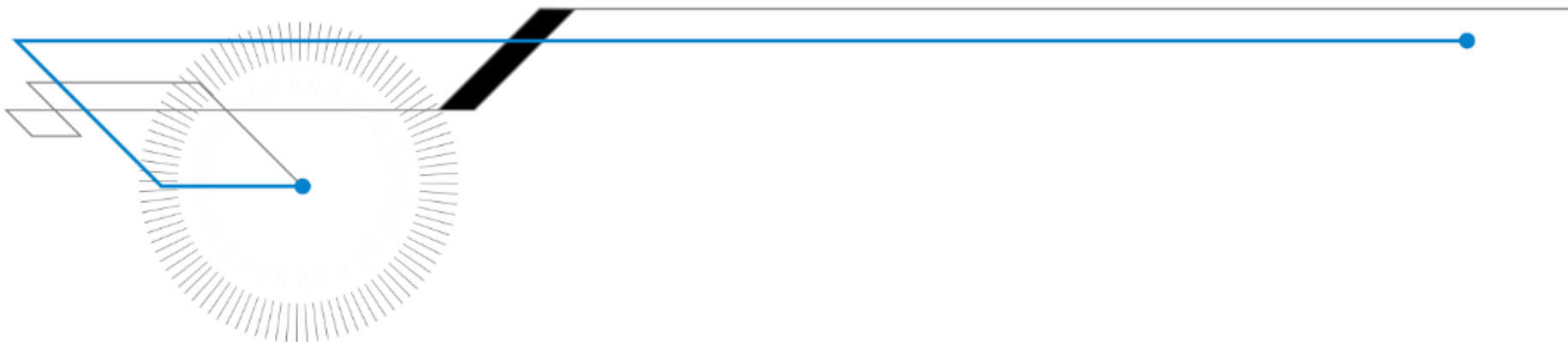


中国科学技术大学
University of Science and Technology of China

1. HelloWorld
2. HelloWorld with command line arguments
3. HelloWorld with decoupling without using Interface
4. HelloWorld with decoupling using Interface
5. HelloWorld with decoupling through Factory
6. HelloWorld using Spring framework as a factory class but not using DI (Dependency Injection)
7. HelloWorld using Spring framework's DI
8. HelloWorld using Spring framework's DI and XML configuration file
9. HelloWorld using Spring framework's DI and XML configuration file with constructor argument



1. HelloWorld Application



HelloWorld



中国科学技术大学
University of Science and Technology of China

```
// This is a good old HelloWorld application we all have written
// the first time we learn Java programming.
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```



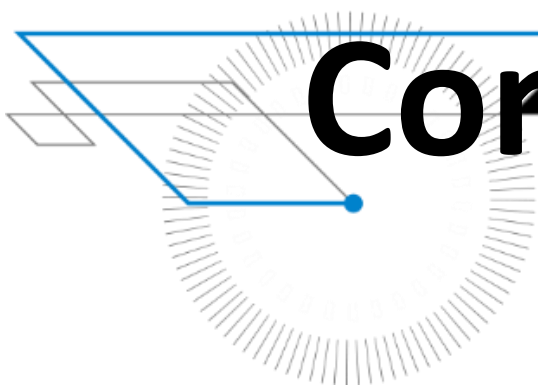
- This code is not extensible. You have change code (and recompile) to handle a situation below.
 - What if I want to change the message?



- Support a simple and flexible mechanism for changing the message



2. HelloWorld Application with Command Line Arguments



HelloWorld With Command Line arguments



中国科学技术大学
University of Science and Technology of China

```
public class HelloWorldWithCommandLine {  
    public static void main(String[] args) {  
        if(args.length > 0) {  
            System.out.println(args[0]);  
        } else {  
            System.out.println("Hello World!");  
        }  
    }  
}
```



- This code externalize the message content and read it in at runtime, from the command line argument
 - You can change the message without changing and recompiling the code



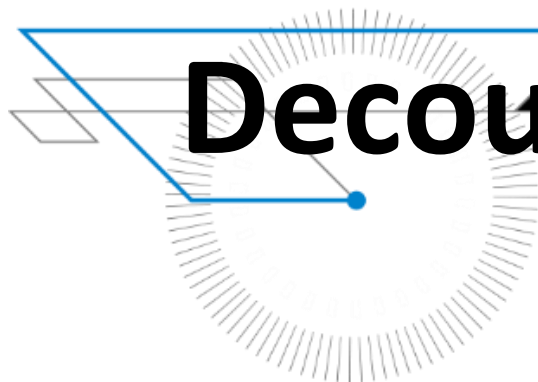
- The code responsible for the rendering message (renderer – the code that does *println*) is also responsible for obtaining the message
 - Changing how the message is obtained means changing the code in the renderer
- The renderer cannot be changed easily
 - What if I want to output the message differently, maybe to stderr instead of stdout, or enclosed in HTML tags rather than as plain text?



- Rendering logic should be in a logically separate code from the rest of the code
- Message provider logic should be in a logically separate code from the rest of the code



3. HelloWorld Application with Decoupling (without using Interface)





- De-couple message provider logic implementation from the rest of the code by creating a separate class

```
public class HelloWorldMessageProvider {  
    public String getMessage() {  
        return "Hello World!";  
    }  
}
```



- De-couple message rendering logic implementation from the rest of the code
- Message rendering logic is given *HelloWorldMessageProvider* object by someone (code is in the next slide) – this is Dependency Injection behavior

```
public class StandardOutMessageRenderer {  
    private HelloWorldMessageProvider messageProvider = null;  
    public void render() {  
        if (messageProvider == null) {  
            throw new RuntimeException("You must set the property  
messageProvider of class:" + StandardOutMessageRenderer.class.getName());  
        }  
        System.out.println(messageProvider.getMessage());  
    }  
}  
// continued
```



```
// continued from previous page
public void setMessageProvider(HelloWorldMessageProvider provider) {
    this.messageProvider = provider;
}
public HelloWorldMessageProvider    getMessageProvider() {
    return this.messageProvider;
}
}
```




- Launcher

```
public class HelloWorldDecoupled {  
    public static void main(String[] args) {  
        StandardOutMessageRenderer mr =  
            new StandardOutMessageRenderer();  
        HelloWorldMessageProvider mp =  
            new HelloWorldMessageProvider();  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
}
```

HelloWorld With Decoupling: Areas Refactored



- Message provider logic and message renderer logic are separated from the rest of the code

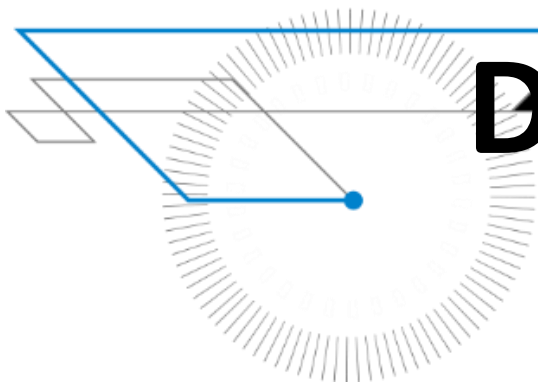


- A particular *MessageRenderer* implementation (*StandardOutMessageRenderer*) and a particular *MessageProvider* implementation (*HelloWorldMessageProvider*) are hard-coded in the main code

- Let these components implement interfaces and the launcher use these interfaces



4. HelloWorld Application with Decoupling Using Interface





- Message provider logic now uses Java interface

```
public interface MessageProvider {  
    public String getMessage();  
}  
  
public class HelloWorldMessageProvider  
implements MessageProvider {  
    public String getMessage() {  
        return "Hello World!";  
    }  
}
```



- Message renderer logic now uses Java interface
- Message rendering logic is given MessageProvider object by someone – this is Dependency Injection behavior

```
public interface MessageRenderer {  
    public void render();  
    public void setMessageProvider(MessageProvider provider);  
    public MessageProvider getMessageProvider();  
}
```



```
public class StandardOutMessageRenderer implements MessageRenderer {  
    // MessageProvider is Java Interface  
    private MessageProvider messageProvider = null;  
    public void render() {  
        if (messageProvider == null) {  
            throw new RuntimeException("You must set the property  
messageProvider of class:" + StandardOutMessageRenderer.class.getName());  
        }  
        System.out.println(messageProvider.getMessage());  
    }  
    // Continued to the next page
```


HelloWorld With Decoupling



中国科学技术大学
University of Science and Technology of China

- Decouple message rendering logic

```
public void setMessageProvider(MessageProvider provider) {  
    this.messageProvider = provider;  
}  
  
public MessageProvider getMessageProvider() {  
    return this.messageProvider;  
}  
}
```



- Launcher

```
public class HelloWorldDecoupled {  
    public static void main(String[] args) {  
        MessageRenderer mr = new StandardOutMessageRenderer();  
        MessageProvider mp = new HelloWorldMessageProvider();  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
}
```



- Message rendering logic can change without affecting messaging provider logic
- Message provider logic can change without affecting message rendering logic



- Using different implementation of either the *MessageRenderer* or *MessageProvider* interfaces means a change to the business logic code (launcher in this example)

- Create a simple factory class that reads the implementation class names from a properties file and instantiate them during runtime on behalf of the application



5. HelloWorld Application with Decoupling through Factory class

A decorative graphic on the left side of the slide, consisting of a circular pattern of lines and a blue line with a dot, resembling a stylized '5' or a clock face.

HelloWorld With Factory Class



中国科学技术大学
University of Science and Technology of China

```
public class MessageSupportFactory {  
    private static MessageSupportFactory instance = null;  
    private Properties props = null;  
    private MessageRenderer renderer = null;  
    private MessageProvider provider = null;  
    private MessageSupportFactory() {  
        props = new Properties();  
        try {  
            props.load(new FileInputStream("msf.properties"));  
            // get the implementation classes  
            String rendererClass = props.getProperty("renderer.class");  
            String providerClass = props.getProperty("provider.class");  
            renderer = (MessageRenderer) Class.forName(rendererClass).newInstance();  
            provider = (MessageProvider) Class.forName(providerClass).newInstance();  
        } catch (Exception ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

HelloWorld With Factory Class



中国科学技术大学
University of Science and Technology of China

```
static {  
    instance = new MessageSupportFactory();  
}  
public static MessageSupportFactory getInstance() {  
    return instance;  
}  
public MessageRenderer getMessageRenderer() {  
    return renderer;  
}  
public MessageProvider getMessageProvider() {  
    return provider;  
}  
}
```


HelloWorld With Factory Class



中国科学技术大学
University of Science and Technology of China

```
public class HelloWorldDecoupledWithFactory {  
    public static void main(String[] args) {  
        MessageRenderer mr =  
            MessageSupportFactory.getInstance().getMessageRenderer();  
        MessageProvider mp =  
            MessageSupportFactory.getInstance().getMessageProvider();  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
}
```



```
# msf.properties  
renderer.class=StandardOutMessageRenderer  
provider.class=HelloWorldMessageProvider
```



- Message retrieval implementation and Message renderer implementation can be replaced simply by changing the properties file
 - No change is required in the business logic code



- You still have to write a lot of glue code yourself to assemble the application together
 - You have to write *MessageSupportFactory* class
 - You still have to inject an instance of *MessageProvider* into the implementation of *MessageRenderer* manually

HelloWorld With Factory: Areas for Further Refactoring



University of Science and Technology of China

- We can use Spring framework to handles the problems mentioned in previous slide
- Replace *MessageSupportFactory* class with Spring framework's *DefaultListableBeanFactory* class
 - You can think of *DefaultListableBeanFactory* class as a more generic version of *MessageSupportFactory* class



6. HelloWorld Application with Spring Framework (but not using DI feature yet)

HelloWorld With Spring Framework



中国科学技术大学
University of Science and Technology of China

```
public class HelloWorldSpring {  
    public static void main(String[] args) throws Exception {  
        // Get the bean factory - the code of getBeanFactory() is in the next  
        // slide  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer)  
        factory.getBean("renderer");  
        MessageProvider mp = (MessageProvider) factory.getBean("provider");  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
    // Continued in the next page
```

HelloWorld With Spring Framework (No need to understand this code)



中国科学院大学
University of Science and Technology of China

```
// You write your own getBeanFactory() method using Spring framework's
// DefaultListableBeanFactory class.
private static BeanFactory getBeanFactory() throws Exception {
    // get the bean factory
    DefaultListableBeanFactory factory = new DefaultListableBeanFactory();
    // create a definition reader
    PropertiesBeanDefinitionReader rdr = new
PropertiesBeanDefinitionReader(factory);
    // load the configuration options
    Properties props = new Properties();
    props.load(new FileInputStream("beans.properties"));
    rdr.registerBeanDefinitions(props);
    return factory;
}
}
```




- Removed the need of your own glue code (*MessageSupportFactory*)
- Gained a much more robust factory implementation with better error handling and fully de-coupled configuration mechanism



- The startup code must have knowledge of the MessageRenderer's dependencies and must obtain dependencies and pass them to the MessageRenderer
 - Spring acts as no more than a sophisticated factory class creating and supplying instances of classes as needed in this case
- You are providing your own getBeanFactory() method using low-level API's of Spring framework



- Use Dependency Injection (DI) of the Spring Framework
 - Glue the application together externally using the BeanFactory configuration



7. HelloWorld Application using Spring Framework & Dependency Injection (DI)



#Message renderer

renderer.class=StandardOutMessageRenderer

Ask Spring to assign provider bean to the MessageProvider
property

of the Message renderer bean (instead of you doing it
manually)

renderer.messageProvider(ref)=provider

#Message provider

provider.class=HelloWorldMessageProvider

HelloWorld using Spring Framework



中国科学技术大学
University of Science and Technology of China

```
public class HelloWorldSpringWithDI {  
    public static void main(String[] args) throws Exception {  
        // get the bean factory  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer)  
        factory.getBean("renderer");  
        // Note that you don't have to manually inject message  
        // provider to message renderer anymore.  
        mr.render();  
    }  
    // Continued in the next page
```



```
private static BeanFactory getBeanFactory() throws Exception {  
    // get the bean factory  
    DefaultListableBeanFactory factory = new  
    DefaultListableBeanFactory();  
    // create a definition reader  
    PropertiesBeanDefinitionReader rdr = new  
    PropertiesBeanDefinitionReader(factory);  
    // load the configuration options  
    Properties props = new Properties();  
    props.load(new FileInputStream("beans.properties"));  
    rdr.registerBeanDefinitions(props);  
    return factory;  
}
```



- The `main()` method now just obtains the *MessageRenderer* bean and calls *render()*
 - It does not have to obtain `MessageProvider` bean and set the `MessageProvider` property of the `MessageRenderer` bean itself.
 - This “wiring” is performed through Spring framework's Dependency Injection. 这种连接是通过Spring框架的依赖项注入来执行的

A Few Things to Observe



中国科学技术大学
University of Science and Technology of China

- Note that we did not have to make any changes to the classes that are being wired together
- These classes have no reference to Spring framework whatsoever and completely oblivious to Spring framework's existence
 - No need to implement Spring framework's interfaces
 - No need to extend Spring framework's classes
- These classes are genuine POJO's which can be tested without dependent on Spring framework



8. HelloWorld Application with Spring Framework & Dependency Injection (DI) using XML Configuration File



- Dependencies of beans are specified in an XML file
 - XML based bean configuration is more popular than properties file based configuration



```
<beans>
  <bean id="renderer"
    class="StandardOutMessageRenderer">
    <property name="messageProvider">
      <ref local="provider"/>
    </property>
  </bean>
  <bean id="provider"
    class="HelloWorldMessageProvider"/>
</beans>
```

Spring DI with XML Configuration File



中国科学技术大学
University of Science and Technology of China

```
public class HelloWorldSpringWithDIXMLFile {

    public static void main(String[] args) throws Exception {

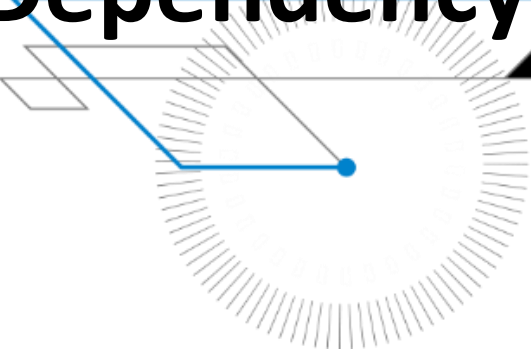
        // get the bean factory
        BeanFactory factory = getBeanFactory();
        MessageRenderer mr =
            (MessageRenderer) factory.getBean("renderer");
        mr.render();
    }

    private static BeanFactory getBeanFactory() throws Exception {
        // get the bean factory
        BeanFactory factory = new XmlBeanFactory(new FileSystemResc
            "beans.xml"));

        return factory;
    }
}
```



9. HelloWorld Application with Spring Framework & Dependency Injection (DI) using XML Configuration File with Constructor argument

A decorative graphic on the left side of the slide, consisting of a circular gear-like pattern with a blue line and a dot, and some geometric shapes.

Spring DI with XML Configuration File: via Constructor



University of Science and Technology of China

```
<beans>
  <bean id="renderer" class="StandardOutMessageRenderer">
    <property name="messageProvider"> set方法注入
      <ref local="provider"/>
    </property>
  </bean>
  <bean id="provider" class="ConfigurableMessageProvider">
    <constructor-arg> 构造器方法注入
      <value>This is a configurable message</value>
    </constructor-arg>
  </bean>
</beans>
```



```
public class ConfigurableMessageProvider implements
    MessageProvider {

    private String message;

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

    public String getMessage() {
        return message;
    }

}
```