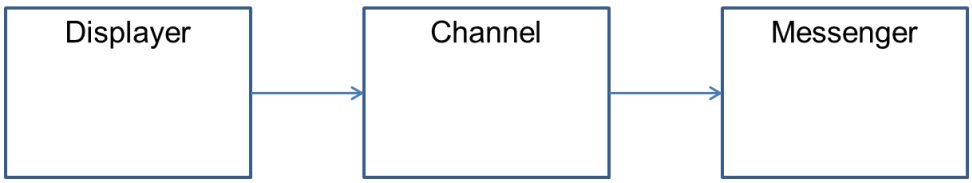
"System Design"

– *Reflection / Annotations / Code Injection / Class Loading [SD 04]*

This Exercise looks in detail into building flexible applications with the help of code injection or dynamic class loading.

1 Introduction

Within this exercise a given application should be extended with the help of reflection, annotation and class loading. In Moodle you will find the classes `Displayer`, `Channel` and `Messenger`. The three classes are connected like this:



Additionally there is a service class `util.Ressources` which is to be used later.

The `Messengers` sole method simply returns messages and looks like this[1]:

```
public class Messenger {

	public String message(){
		return "Hello, World";
	}

}
```

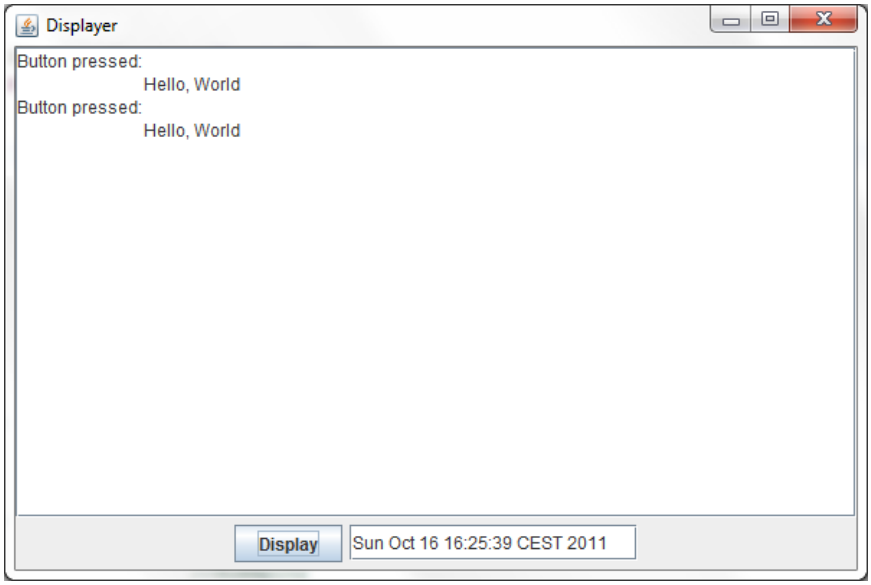
The `Channel` calls the `Messenger` and provides the result of this call:

```
public class Channel{

	public List<String> getMessages() {
		List<String> messages = new ArrayList<>();

		Messenger msgr = new Messenger();
		messages.add(msgr.message());

		return messages;
	}

}
```

Currently the list collects only one message – later the messenger may produce several messages, which need to be collected here.

[1] Please note, that the classes Messenger, Channel, Displayer and the utilities are available in Moodle. You should download these classes and use them as a starting point for the following tasks.

The Displayer displays these messages after a button has been pressed; the graphical user interface looks like this:



The main task within this class is to create the Channel and invoke the Messenger to display the returned results; this is implemented in the following code-snippet:

```
…
public class Displayer extends JFrame{
      …
      public Displayer(){
            …
            channel = new Channel();
            …
            // Button for triggering the Channel
            JButton display = new JButton("Display");
            display.addActionListener(event -> {
                        text.append("Button pressed:\n");
                        List<String> messages = channel.getMessages();
                        if(messages != null && !messages.isEmpty())
                              for(String msg : messages)
                                    text.append("\t" + msg + "\n");
                        else
                              text.append("\tno messages\n");
            });
            …
      }

      public static void main(String[] args){
            new Displayer();
      }
}
```

This design is rather static:

- If a new/additional Messenger comes up the code in the Channel has to be adapted.
- If a new/additional Message-Method comes up the code in the Channel has to be adapted.
- if the returned text of a method changes, the Application has to be restarted to load the changed class

Within this task you should add more flexibility to the application by decoupling the Messenger-class from the Channel. This is done in two different approaches:

- with the help of code-injection (2 – Realizing “Code-Injection”) and
- within a completely self-organizing application. (3 – Realizing “Continuous Deployment”)

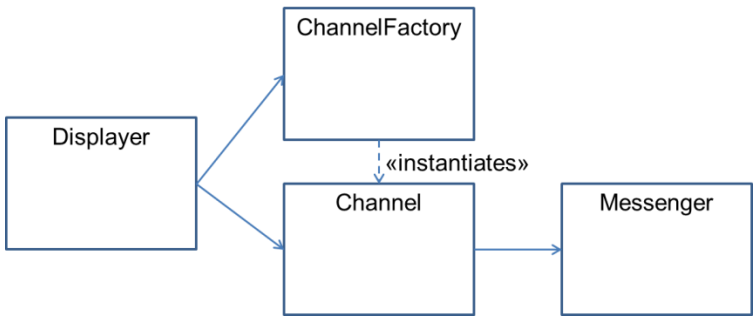
2 Realizing “Code-Injection”

Within this task you extend the application by the ability to configure an application with a properties file. This properties-file allows defining, by which class an annotated, but uninitialized variable should be instantiated. This common mechanism of code-injection is for example heavily used for defining beans in the Java Enterprise Edition.

Start with the initial `Displayer`-, `Channel`- and `Messenger`-Class and refactor them continuously. (For this example you don’t need the `util.Ressources`-Class).

2.1 Creating a Factory

Create a `ChannelFactory` which provides an instance method for creating a Channel instance. The Channel-Class and the messenger-Class stay unchanged so far. Use this factory in the Displayer-Class to get a new instance of the Channel. The appropriate class diagram would look like this.

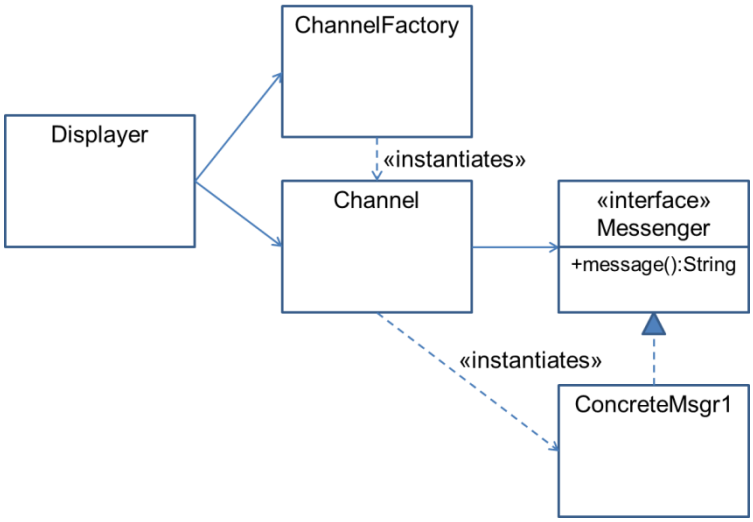


2.2 Extracting an Interface

Make the Messenger-Class an interface which provides the method `message():String`. Create an implementing class which is used for instantiating the variable.

Hint: Use Refactoring in Eclipse. First “Rename” Messenger to ConcreteMessenger1, then “Extract Interface” named “Messenger”.

The appropriate class diagram would look like this.



That is, the Channel would define the messenger like this:

```
    private Messenger msgr = new ConcreteMessenger1();
```

2.3 Reading a Properties-File

The properties-file should give the name of the actual class used for instantiation (e.g. `ConcreteMessenger1`).

Property-files in Java are simple text files with the following format:

```
#comment
key1:property1
key2:property2
…
```

For our application the key should be the string “msgr” and as property the name of the desired class, that is:

```
#Define the messenger's properties
msgr: app.ConcreteMessenger1
```

This property-file should be read by the factory-class upon the creation of the Channel-object. Initially these properties should only be read, without any further effect. (For debugging it would make sense to print the value of the property you have read).

Properties can be loaded with the following Java-Code (Exception-Handling is omitted):

```
Properties properties = new Properties();
BufferedInputStream stream;
stream = new BufferedInputStream(new FileInputStream("aPath"));
properties.load(stream);
stream.close();
String propString = properties.getProperty("aKey");
```

2.4 Creating an Object out of the Property

Use the retrieved property (which should be a name of an implementing Messenger-class) to create a new Messenger-object. To do so first load the class with the help of `Class.forName(classname)` and then call `newInstance()`. For debugging, print this object – otherwise you don't need it now.

2.5 Define an Annotation

Define a Field-Annotation called `MessengerContext` for tagging the field which should be injected later. The annotation should be like this:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface MessengerContext {
}
```

In the Channel-Class annotate the `msgr`-field with the new annotation, that is:

```
public class Channel{

      @MessengerContext
      private Messenger msgr = new ConcreteMessenger1();
…
```

In the **`ChannelFactory`**, take the previously created instance of the **`Channel`** and retrieve all fields annotated with `MessengerContext` Make them accessible and simply print them – in our example this should return the one field defined above.

2.6 Do the Code Injection

After having prepared everything, the final code-injection is simple:

- Remove the instantiation in the Channel class; i.e. simply write

```
public class Channel{

      @MessengerContext
      private Messenger msgr;
…
```

- Go through all retrieved, annotated fields and set the Messenger-object you created in section 2.4 to the field using the `set`-Method. If the field was private (as in our case) make it accessible first.

Now you're done! Your application can now be configured by simply adding a new class and entering the name of the class in the properties field. Test it by creating a second concrete Messenger-Class (which gives a different message) and change the entry in the property file for this class – the original application should stay untouched.

3 Realizing “Continuous Deployment”

Within this task you extend the application by the ability – in the end – to add and change Messenger-Classes during runtime.

This behavior realizes the ability of an Application-Server, who can add new functionality during runtime (“hot deployment”) and is a fundamental technique for decoupling parts of your system.

Start again with the initial `Displayer-`, `Channel-` and `Messenger-`Class and refactor them continuously. Do the development in the following steps. Step 1 to 5 are to be done in the `Channel-`Class, step 6 in the `Displayer-`Class. The `Messenger-`Class has to be changed only for testing the application.

3.1 Using Reflection

Get rid of the predefined Messenger-method `message()`. Use reflection to determine all the methods of the Messenger class who are able to produce a message: these are the methods with no parameters and String as a return-type.

Invoke these methods, collect their results and display these results.

Test this behavior by adding or removing methods from the messenger class.

3.2 Using more Reflection

Get rid of the predefined Messenger-class. Have a look at all classes in your class-directory and again, use reflection to determine all the methods of all classes who are able to produce a message: these are the methods with no parameters and String as a return-type. This time also make sure you only select non-static methods.

Invoke these methods, collect their results and display these results. For collecting the classes you may use the already implemented static method `String[] getClassNames("app")` in the class `util.Ressources`, which returns all available classes in your app-directory. If you want to access classes in the default directory you may leave out the parameter.

A found class can be loaded by name with the help of the Method `Class.forName(className)`.

Test this behavior by adding or removing methods from the messenger class and adding additional Messenger classes.

3.3 Using Annotations

Not all methods found in the previous extension are meant for displaying, e.g. some toSting()-Methods should not be used. Therefore, introduce a simple Marker-Annotation called `Displayable`, which can be tagged to the message-methods you wish to be displayed. The Annotation should look something like this:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Displayable {
}
```

The annotation should be used like in the following example. Your displayer should now look for the tagged methods and only try to display these. (Therefore, only the first message of the example should be executed):

```
public class Messenger {

	@Displayable
	public String message(){
		return "Hello, World";
	}

	public String message2(){
		return "Hi, there";
	}
}
```

Again, test this by adding and removing some annotations and adding some new, annotated classes during runtime.

3.4 Dynamic Class Loading

This is the most difficult part of the task!

Currently all Messenger classes are searched and loaded when found. If a new class is found the messages will be displayed too, but if an existing class has changed, the old code will still be used. To change this, add a dynamic class loading policy:

- Access the file for the corresponding class – this can be done with `Ressources.getFile(class_name):File`.
- Remember for each loaded class the time the class was last changed – this can be done best with a `Map<String, Long>` which remembers the last date last modified (as `long`) for a classname (as `String`). You can access the last modification time with the method `lastModified` of `File`.
- Check – when collecting the messages again – if the file of the loaded class has changed (that is if the modification date you have saved is different to the files current modification date); if so, reload the file.

- For reloading you need a new class-loader, as the “old one” remembers, that he had already loaded the class and therefore refuses to load again. A class loader based on the script may look like this. (Instead of supplying the bin-folder-url by hand, this is retrieved with the `util.Ressources` form the environment.)

```
public class ClassReloader extends URLClassLoader{

      public ClassReloader() {
            super(classPath(), null);
      }

      private static URL[] classPath(){
            URL url = Ressources.getBaseDir();
            URL[] urls = {url};
            return urls;
      }
}
```

- The class loader can be called like this:
```
ClassLoader loader = new ClassReloader();
Class<?> cls = Class.forName("classname", true, loader);
```
- Be aware, that a class always exists in its own context – i.e. the classes use their own version of `String`, etc.

3.5 Continuous Dynamic Class Loading

This is the simplest part of the task and only changes the Displayer-class

Instead of updating by pressing the display button the Displayer should check regularly. Therefore, use a process for continuous checking and get rid of the display button. To achieve this, you, may use the `TimerProcess` of the Displayer.