Project Course IT
TDA367/DIT211
Workshop: Plugins & Dependency Injection

Adam Waldenberg
adam.waldenberg@gmail.com

April 3, 2011

1 Introduction

In this workshop you will construct a 1984 calendar application (that only shows information for that year) using a custom Swing component and Google Guice [1]. We choose to support one specific year in order to keep the complexity down. Furthermore, you will learn to construct custom Java annotations and be taught how to create a personalized plugin system to use in your project. The constructed plugin system will load your plugins from a specified location, thus extending the functionality of the program. It's presumed that you have a basic handle on how Swing, annotations, *instanceof* and generics work.

This workshop is the last one of the course. It briefly touches each topic discussed but still requires more effort (and time) than previous workshops, meaning you might not complete it during the hours alloted.

1.1 Google Guice

Thanks to Google Guice the need to dynamically allocate objects in your code diminishes as you can instead choose to let Google Guice inject resources into objects. The Guice framework keeps track of allocations and can aid you in the creation of *mock objects* whenever you need to unit test a class or method that needs an external (allocated and injected) resource. Aiding libraries such as Guice are commonly known as dependency injection frameworks.

We will not put much focus on Guice in the assignments, but the framework is explained nevertheless. Only the last assignment in this workshop concerns Google Guice.

1.1.1 Motivation

So, why is using Google Guice a good idea? Firstly, your code becomes dynamic and less *bound* to it's dependencies. With Guice you build injection rules. These injection rules can be changed throughout the execution of the application, giving you the possibility to modify dependencies on-the-fly. Secondly, it makes unit testing easier, as dependencies that normally would mean that a certain class can't be unit tested, can be removed with an injected *mock object*[6].

1.1.2 Using Google Guice

Google Guice is a tool that gives the programmer control over many of aspects concerning the dependencies between classes. Google Guice also supports something called *Aspect-oriented* *programming* [5], something we will not cover in this workshop. As we focus on the mere minimum, it is recommended that you glance at the official API [2] in order to get a broader view.

Imagine we are writing our calendar application and need to be able to get event data for specific days from the Internet (using a http server). To abstract this properly we could imagine something simple such as this:

```
public CalendarApplication {
     public static void main(String[] args) {
        Calendar calendar = new Calendar();
        calendar.print();
     }
}

public class Calendar {
    private CalendarSource source = new CalendarSourceInternet();
    /* Code that fills the calendar. Not important
       in this example. */

    void print() {
        // Code that pretty-prints the calendar.
    }
}

public class CalendarDay {
    /* Methods to fetch and store information about
       this specific day. */
}

public interface CalendarSource {
    CalendarDay getCalendarDay(int day);
}

public class CalendarSourceInternet implements CalendarSource {
    public CalendarDay getCalendarDay(int day) {
        /* Fetches calendar data for the specific day and
           fills it with information. */
    }
}
```

Notice how we mix the interface definition and class call inside the Calendar class in order to get a generic implementation. This is the normal way to do it in Java. Often you have a *SomeName* interface and a *SomeName*Impl class that is based on that interface. Imagine that we would want to add a unit test to test the CalendarDay returned (to check so that the returned data is valid). We would do something like this:

```java
public class CalendarDayTest {
    @Test
    public void testDayValidity() {
        CalendarSource source = new CalendarSourceInternet();
        // Test code here.
    }
}
```

Now consider; what is wrong with this code? What constitutes a *good* unit test? The basic answer is that it should work under *all conditions*, unless you are testing something that fundamentally requires a certain state. As *CalendarSourceInternet* requires a working Internet connection it could potentially fail whenever the test of the *CalendarDay* class is executed. This is not acceptable. It would be acceptable if the test was intended for the *CalendarSourceInternet* class, but this is not the case. As mentioned previously, Guice can help us remedy this problem. Let's start by *guicing up* the simple example above, one class at a time:

```java
@ImplementedBy(CalendarSourceInternet.class)
public interface CalendarSource {
    CalendarDay getCalendarDay(int day);
}
```

The *@ImplementedBy* annotation will tell Google Guice that *CalendarSourceInternet* is the default implementation for *CalendarSource*, getting rid of the mixing of interfaces and classes everywhere in the code. Next, we need to use this interface inside our code:

```java
public class Calendar {
    @Inject
    private CalendarSource source;
    /* Code that fills the calendar. Not important
       in this example. */

    void print() {
        // Code that pretty-prints the calendar.
    }
}
```

Whenever the *Calendar* class is instantiated by Google Guice, *CalendarSourceInternet* will be injected into *source*. This happens because we previously defined that *CalendarSourceInternet* is the default implementation. In order for this to happen, we also need to make one additional change:

```java
public class CalendarApplication {
    public static void main(String[] args) {
        Injector injector = Guice.createInjector();
        Calendar calendar = injector.getInstance(Calendar.class);
        calendar.print();
    }
}
```

This creates an instance of *Calendar* and injects it into a local variable. Guice also initializes the instance and injects everything we asked for.

We can now rewrite the unit test to use a *mock object* that we define. To be able to use our own mock object and overwrite the injection rule defined with *@ImplementedBy*, we need to use something called Guice Modules, or Linked Bindings [3]:

```
/* Define this somewhere in a package in your test code. Then you
   can use it as a general module that configures mock objects for
   unit tests. */
public class MockModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(CalendarSource.class).to(CalendarMockSource.class);
        //... more mock bindings.
    }
}

public class CalendarMockSource implements CalendarSource {
    public CalendarDay getCalendarDay(int day) {
        /* Some implementation that creates some default
           data in order to return a CalendarDay to the
           test code. */
    }
}

public class CalendarDayTest {
    private Calendar calendar;

    @Before
    public void setup() {
        Injector injector = Guice.createInjector(new MockModule());
        calendar = injector.getInstance(Calendar.class);
    }

    @Test
    public void testDayValidity() {
        // Test code here.
    }
}
```

The *MockModule* class is a Guice plugin that defines to what interfaces different classes are bound to. This time when we create the injector, we pass a Module definition into it. This constructor actually takes a list of Modules, so you can pass as many as you want into it.

With this code, the unit test is now correct and should consistently be able to properly test our code.

Of course, in order for any of this to make sense at all, *CalendarMockSource* has to use some common code for creating a *CalendarDay*. This will in truth be the code that is tested.

1.2 Annotations

Annotations are a powerful feature of Java and were introduced into the language back in JDK 1.5. They let us add *configuration options* to pretty much any part of our code. Most Java programmers use them, but what many of them never do is to actually implement their own annotation classes, something that is also possible. Using annotations, we can add custom meta data to our classes and methods. By using *Java reflection* [7] we can fetch this meta data during run-time in our code.

1.2.1 A crash course in writing your own annotations

We want to build a plugin system for our project. Using annotations, we can control how this plugin system handles our written plugins and how it loads them. Imagine the simplest form of a plugin, where the programmer can only specify the priority of the plugin (when or where in the load chain the plugin gets loaded):

```
@Retention(RetentionPolicy.RUNTIME)
public @interface SimplePlugin {
    String value() default "No description";
    int priority() default 0;
}
```

What happens above is that we define the annotation *@SimplePlugin* with optional description and priority parameters. If the *default* keyword is not specified, the parameter is mandatory and has to be specified whenever the annotation is used. To complement a normal annotation declaration such as the one above there are also certain annotations that you can put above the definition in order to control the behaviour. This is what *@Retention* does in the above example. This requires a thorough explanation, because a lot can be accomplished with this functionality:

- @Retention(policy) controls *when* an annotation will be available. Where *policy* is one of the following:
  - RetentionPolicy.CLASS - Means the declared annotation will be available during compilation. Used by compilation tools such as Ant. *NOTE: This is the default behaviour for a declared annotation that does not set the retention policy.*
  - RetentionPolicy.RUNTIME - Means the declared annotation will be available during run-time (this is exactly what we want for plugins).
  - RetentionPolicy.SOURCE - Means the declared annotation will be available only at source level and will be discarded by the compiler. Used by Java development environments.
- @Target(type[; type...]) controls *where* in the source code an annotation can be used. The *type* arguments have to be one of the following:
  - ElementType.TYPE - Means the annotation can be used on top of declared classes, interfaces, enums and other types. This is what we are interested in.
  - ElementType.FIELD - Means the annotation can be used on top of instance variables.
  - ElementType.METHOD
  - ElementType.PARAMETER
  - ElementType.CONSTRUCTOR
  - ElementType.LOCAL_VARIABLE
  - ElementType.ANNOTATION_TYPE - Means that the annotation can be used on top of another annotation.

    – ElementType.PACKAGE

When *@Target* is omitted (such as in our declaration above), the declared annotation may be used on any program element.

The programmer can now use the defined annotation in his or her project by specifying the *@SimplePlugin* annotation in the code like this:

```
@SimplePlugin
public class LowPriorityPlugin {
    // Plugin with "No description" as description with priority 0.
}

@SimplePlugin(priority = 500)
public class NormalPriorityPlugin {
    // Plugin with "No description" as description with priority 500.
}

@SimplePlugin(value = "High priority plugin", priority = 1000)
public class HighPriorityPlugin {
    /* Plugin with "High priority plugin" as description with
       priority 1000. */
}
```

Once everything is defined, it is up to the programmer to scan plugins and check the parameters of the annotations using Java reflection.

1.3 Java Reflection API

Reflection is a term used to describe the act of applications that can structuraly modify and observe themselves during run-time. In Java this means scanning classes, initiating new instances of classes, fetching method references and pretty much anything you can imagine. For further information of what is possible, refer to the Java Reflection API [7]. We will use reflection to dynamically load plugins and scan them for annotations. In the process you will learn a small part of the reflection API, which should get you started if you want to implement something similar in your own applications.

1.3.1 Using Java to load classes as plugins

Usually, loading classes dynamically is easy and can be done like this:

```
public class ClassLoadingExample {
    public static void main(String[] args) {
        ClassLoader loader = ClassLoadingExample.class.getClassLoader();

        try {
            Class<?> class = loader.loadClass("edu.itproj." +
                                              "plugins.SimplePlugin");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
}
```

The classloader will check if the class is already loaded and only reload it if required. Easy, right? If it only was that simple...

If we would decide to load our plugins in the above manner, we would have no control over where the binaries were fetched from. This would mean that we would be forced to put the plugins inside the applications main class hierarchy or in the classpath. What if we want to load the plugins from a web server or some other service? The solution to this problem is to either use the *URLCloassLoader* class (which has problems under certain operating systems) or extend the *ClassLoader* class with our own implementation, like so:

```
class BetterClassLoader extends ClassLoader {
    private final String location = "file:/home/user/someplugindir/";

    @Override
    public Class<?> findClass(String name) {
        byte[] data = loadClassData(name);
        return defineClass(name, data, 0, data.length);
    }

    private byte[] loadClassData(String name) {
        /* Load binary class data
           (open the class file and read it). */
    }
}
```

As you can see, the location string is hard coded in. A better alternative is to specify the plugin directory using an XML configuration file. However, this is outside the scope of this workshop.

When we use the *BetterClassLoader* class in our project, *findClass(name)* is invoked by *loadClass(name)* if the default classloader fails. Which it should, as our class should be nowhere to be found. Inside *loadClassData(...)* you have endless posibilities. You can load files from a local location or from the Internet. Some people even implement special *plugin packages* that they load with their custom class loaders. Another common practice is to search for plugins in many different locations. It is up to you. Though, during the rest of this workshop we will focus on loading locally stored files from a single location.

The next step is to use the *BetterClassLoader* class:

```
public class ClassLoadingExample {
    public static void main(String[] args) {
        try {
            BetterClassLoader loader = new BetterClassLoader();
            Object plugin = loader.loadClass("SimplePlugin").
                                newInstance();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
}
```

That's all there is to it. The above code loads and creates a new instance of a class located at *"file:/home/user/someplugindir/SimplePlugin.class"* (if *BetterClassLoader.loadClassData()* was correctly implemented).

1.3.2 Using reflection magic to scan plugins for annotations

Next, we need to be able to scan the plugins for meta data. Consider the following piece of code where we implement an annotation scanner that scans for SimplePlugin annotations:

```
public class AnnotationScanner {
    public static SimplePlugin
    getSimplePluginAnnotation(Class<?> c) {
        return c.getAnnotation(SimplePlugin.class);
    }

    /* This method also prints if we found a SimplePlugin
       annotation or something else and prints additional
       informartion. */
    public static Annotation[] getAllAnnotations(Class<?> c) {
        Annotation[] annotations = c.getAnnotations();

        for(Annotation a : annotations) {
            if (SimplePlugin.class.isAssignableFrom(a.getClass())) {
                SimplePlugin sp = (SimplePlugin) a;

                System.out.println("Found @SimplePlugin");
                System.out.println("Value: " + sp.value());
                System.out.println("Priority: " + sp.priority());
            } else {
                System.out.println("Found some other annotation");
            }
        }

        return annotations;
    }
}
```

Above we have one method that just fetches a *@SimplePlugin* annotation and another method that fetches all annotations on a class and prints some information about *@SimplePlugin* annotations that were found. That's it. With this small piece of code you should be able to easily scan run-time classes for defined annotation.

1.4 More on Google Guice

Often when using Google Guice you'll want to inject objects that are defined outside the main application context, inside an external resource such as a jar file with classes. You can of course not use annotations to define injection rules in these cases. To accomplish injection with *the outside world*, you can use Guice Modules and define linkage in the same manner as we did in the previous example. Many times when you need an injected object you also need to do some kind of initialisation on the object, either before or after you instantiate it. To accomplish this you can use the *@Provides* annotation inside a defined Guice module. With this, you can define methods that *provide* different kind of objects to Guice (for you injection rules).

When using Guice, you will normally never have to write custom Singletons. This is because Guice can handle scoping [4] on different objects and has the ability to define Singleton objects using the *@Singleton* annotation. This is actually covered briefly in the last assignment of this workshop.

Again, this workshop only covers the mere basics as we do not have space or time to cover all parts of Guice, but it is definitely something you should look into for your project.

2 Starting a new project

It is time to implement our Calendar application. To simplify everything, there is a code skeleton that you can start from.

Download the code skeleton for this workshop from the course website and study the code and code comments. You should import this skeleton as a project into Eclipse. Most of the work in is already done for you. Your job is to fill in the blanks.

When you run the application now, all you should get is an empty window. Let's get to work and make some magic happen.

Each time you open a file with assignments, every *TODO* inside the file will be numbered according to the order that each *TODO* should be done in. The first assignment will be numbered A*1, A2, A3...*, the second *B1, B2, B3...* and so on. Use the *Tasks* tab in Eclipse to locate everything that needs to be done for each assignment.

If you run into problems during the assignments, there is a troubleshooting section at the end of this document with some common issues that can arise.

2.1 The layout of the calendar application

The application is structured in the following way:

- calendarapp - Main package group.
- calendarapp.ctrl - Controller classes.
- calendarapp.gui - Classes associated with the user-interface.
- calendarapp.plugins - Annotations, classes and interfaces concerning plugins.
- calendarapp.plugins.exported - Plugins classes that will be exported (The actual plugins).
- calendarapp.utils - Utility classes and definitions; classes and definitions that are general and could be used outside the scope of this application.

2.2 Assignment A: Creating the calendar view

First, open the file *calendarapp.gui.CalendarMonth* and inspect the source code. The *CalendarMonth* class is used by the class *calendarapp.gui.Calendar* when making a view of all the months in the year 1984. Open the file under *calendarapp.gui.Calendar* and inspect the source code. Most of the class is already coded for you. Do every *TODO* note for this assignment and see what happens. Your job is to create a grid of widgets for displaying calendar days. If you get it right, you should see a calendar view next time you start the application.

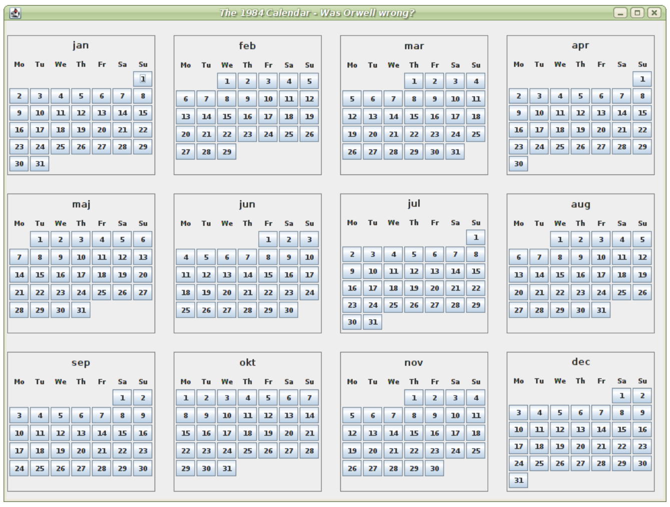

Figure 1: This is how the application should look after implementing the calendar view (Assignment A).

2.3 Assignment B: Creating annotations for our plugin types

We will need two different plugins for the project. Calendar plugins and GUI plugins. The calendar plugins define different holidays and events in the calendar view. The idea is to give days some special text and colour whenever they are associated with a certain plugin. The GUI plugins should instead have the ability to add Swing elements into the main calendar window.

As our project supports two types of plugins it also needs two different plugin annotations:

- CalendarPlugin - This annotation should define a *description* parameter describing the type of day or event defined inside a certain plugin. Alternatively, you can use the normal value parameter for this, just like the example annotations. The annotation should also define a *priority* parameter. This can later be used to prioritize in what order calendar definitions are applied. For example, if we define a plugin that specifies Sundays as red days while also defining another plugin that specifies Easter, it is important that the Easter holiday takes precedence (*Påskdagen* in Sweden). The prioritization is taken care of for you, using two Comparator classes that are already defined inside the calendar application.

- GUIPlugin - This annotation should define the same parameters as the annotation discussed previously. In addition, this annotation should also take an area parameter which should be defined as an enumeration that defines the two areas *GUIPLUGIN_NORTH* and *GUIPLUGIN_SOUTH*. The main GUI will have two Swing containers where plugins will be appended, this is how we specify into which one the plugin should be loaded.

The files *calendarapp.plugins.CalendarPlugin* and *calendarapp.plugins.GUIPlugin* are already created for you. What you need to do now is to define them properly. You will also have to modify the enum definition in *calendarapp.plugins.PluginArea*.

Once you have defined *PluginArea*, open the classes *calendarapp.gui.MainWindow* and *calendarapp.gui.PluginPanel*. You are supposed to implement the usage of *PluginArea* in these two classes. Refer to the *TODO* comments for further information. Take care to only do the comments related to this assignment.

2.4 Assignment C: Creating a general plugin interface and some plugins

Next, we will start on the plugin handling. The interface *calendarapp.plugins.Pluggable* should describe a general plugin. This plugin interface will later be implemented in the application and annotated with the annotations you previously defined. The idea is to use generics in *calendarapp.plugins.Pluggable.* Classes implementing this interface can then specify what kind of type they want to return back from their method(s). This way we don't need to return or handle general Object classes in our plugins. Open *calendarapp.plugins.Pluggable* and complete it.

Once you complete the *Pluggable* interface, open *calendarapp.plugins.exported.EasterPlugin* and do the *TODO's* inside. When completed, you should have a correctly defined plugin. If you look inside *EasterPlugin*, you can also see that it implements the *Coloured* interface as well. Using interfaces in this manner, you can patch together different kinds of plugins. When the plugin is loaded you can check which interfaces it implements. With this technique, you will be able to know how to handle plugins with different interfaces patched on. It's a good complement to the annotations.

Once that is done, there are two other *CalendarPlugins* that you need to complete, namely *calendarapp.plugins.exported.SundayPlugin* and *calendarapp.plugins.exported.BigBrotherPlugin.* Take care of the priority with *SundayPlugin*, as this plugin will overlap with *EasterPlugin.*

Before we can start on the next step, you also need to create a GUIPlugin. If you look at the calendar, it looks very dull. We need to add some flare to it! Your next plugin should add a JLabel with big, centered text to the upper plugin area (as you might remember, our interface has two areas for externally added widgets). This time you will have to write more of the plugin on your own. Open *calendarapp.plugins.exported.TitlePlugin* and follow the instructions in the *TODO* comments.

2.5 Assignment D: Preparing a simple plugin environment

It's time to use what you have learned and eventually construct classes that dynamically load the plugins we created. We will load the plugins from a plugin directory instead of the the normal application hierarchy or classpath. Locate the location of your home directory. On Linux you can check the location of your home directory in a terminal:

- Open a fresh terminal.
- Usually, your current working directory should initially be your home directory, but we can write *'cd ~'* and press ENTER just to make sure we are in the right place.
- Write *'pwd'* and press ENTER. The terminal should output the full path to your home directory. Save this string. You will need it when writing the plugin loader.

Using the command *'ln'*, we can create soft links in this directory that point to any class file that we want to put there. This way, we won't have to copy the *'.class'* files to the plugins directory whenever we recompile. Next, if you are under Linux, do the following:

- Create a *plugins* directory in your home directory.
- Open a terminal and *cd* into the created plugins directory.
- Create a link to *EasterPlugin.class* by writing *'ln -s <full absolute or relative path to the compiled EasterPlugin.class>'*
- Create similar links for *SundayPlugin.class*, *TitlePlugin.class* and *BigBrotherPlugin.class.*

- Now, make sure that the links are valid by writing for example *'javap TitlePlugin'* in the terminal. Do the same check for *EasterPlugin*, *SundayPlugin* and *BigBrotherPlugin*. Notice that javap takes the class name with *'.class'* omitted.

Under Windows you can just copy the class-files/plugins to some location by hand. Just make sure that you copy them again each time you recompile modified plugins. Alternatively, if you are using Vista or Windows 7, you can use the *mklink* command to create similar symbolic links. Note that you also need administrative rights to be able to create symbolic links with the given command.

2.6 Assignment E: Creating the classes needed for loading plugins

Now that we have everything in place, it's time to actually implement the plugin loading. The class *calendarapp.plugins.PluginLoader* is the first file we will edit. When completed, *PluginLoader* will load the plugins from the location specified by the static variable *PluginLoader.pluginPath*. The class has two exposed methods for loading plugins; *loadCalendarPlugins()* and *loadGUIPlugins()*, which return a *List<Pluggable<?>>* of loaded plugins. These methods are called by *calendarapp.gui.Calendar* and *calendarapp.gui.CalendarMonth*. So, whenever these classes call those two methods, they scan the returned plugins and handle them differently depending on different criteria.

Inspect the source code for *calendarapp.plugins.PluginLoader* thoroughly. Parts of the private methods *getPluginClasses()* and *getPluginsFromPluginClasses* are missing. Fill in the blanks.

Next up is *calendarapp.plugins.PluginClassLoader*. This class should look very familiar, as it is based on the previous example of *BetterClassLoader*. The method *findClass(name)* receives an actual class name from the application, that is; a name without the *.class* suffix. If you look at the static variable *exportedPackage*, you can see that it is used to prepend the class name with the package in which the plugin was compiled. This is needed in order for Java to be able to create a valid definition of the class when *ClassLoader.defineClass(...)* is called.

Your objective in this class is to implement the missing part in *loadClassData()*, which is supposed to read the class as binary data and return a byte array.

2.7 Assignment F: Loading the GUI plugins

Now we have come so far that we can start loading GUI plugins into the plugin panels. As the plugin panels are the widgets that will hold the plugins, they are the logical destination for the plugin handling code as well. Open *calendarapp.gui.PluginPanel* and complete the code.

Take care so that you properly check that the loaded plugin actually belongs to the area definition defined by the plugin panel. If you fail to do so, the widget might be added twice, because there are two plugin panels in the view.

Depending on how you defined *Calendar.getConstraints()*, you might have to edit the constraints again in order to get the plugged in widgets on top of the calendar. *TIP: Using 'constraints.gridy = 1' in getConstraints() might work.*

If everything went according to plan, you should now have a window with a label at the top.

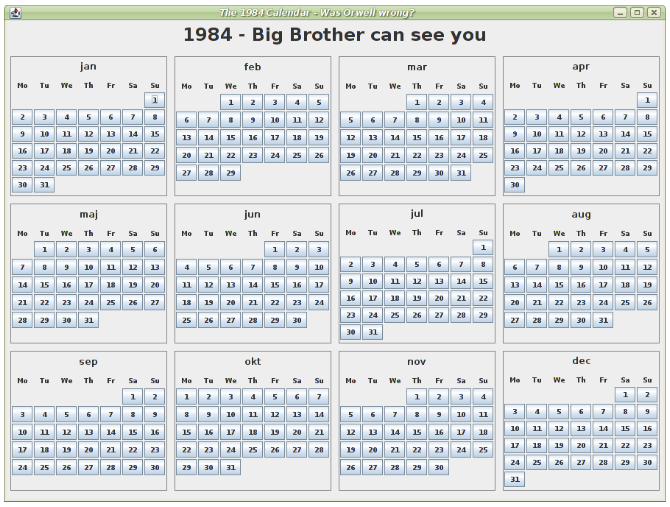
The 1984 Calendar - Was Orwell wrong?
1984 - Big Brother can see you
jan
feb
mar
apr
maj
jun
jul
aug
sep
okt
nov
dec

Figure 2: This is how the application should look once the plugin panels handle GUI plugins correctly (Assignment F).

2.8 Assignment G: Loading the Calendar plugins

The last step in the plugin handling is to load the plugins annotated with the *@CalendarPlugin* annotation. The buttons for calendar days are added in the private method *calendarapp.gui.CalendarMonth.addDay()* by creating instances of *calendarapp.gui.CalendarDayButton*. In addition to the button string, the constructor of *CalendarDayButton* takes a list of loaded plugins and reads the configuration of each button using the plugins. So the actual plugin handling is supposed to be located inside *CalendarDayButton*.

Open *CalendarDayButton* and get working.

When you completed the assignment, start the application and click on the coloured buttons. You should get a window with annotation texts. If you got it right, you should see the description annotations from both days when clicking on the button representing Easter.

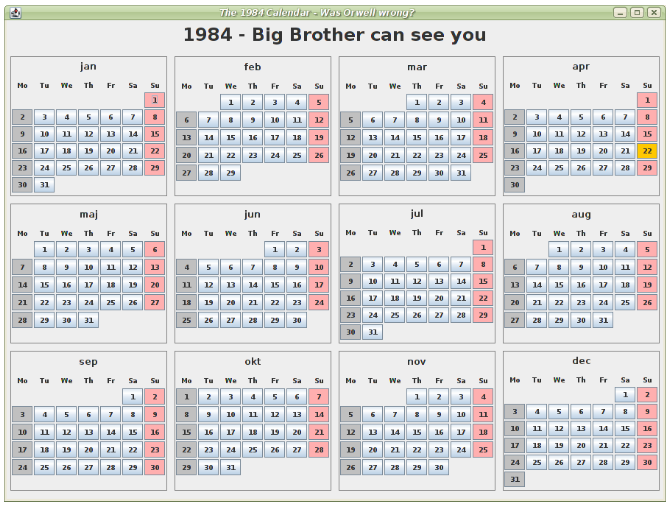
The 1984 Calendar - Was Orwell wrong?
1984 - Big Brother can see you
jan
feb
mar
apr
maj
jun
jul
aug
sep
okt
nov
dec

Figure 3: This is how the application should look if you annotated your plugins right and added a correct implementation into *CalendarDayButton* (Assignment G). Notice how the colour of the 22'nd of April is orange. This is because Easter takes precedence over the normal Sunday plugin.

2.9 Assignment H: A quick example of using Google Guice

Before you start, you need to download the archive *guice-3.0.zip* from the Guice website [1]. Inside the archive you'll find *guice-3.0.jar*, which you can add to your project.

You might have noticed that the calendar application has a factory for returning controller singletons. There is only one controller defined in this application, but the idea is that one should be able to call the factory and return any controller defined, whenever needed.

Delete *calendarapp.ctrl.ControllerFactory* from the project. We will replace the factory with Guice injection. Open *calendarapp.ctrl.CalendarDayButtonAction* and add a *@Singleton* annotation to the class.

The singletons created by Guice are handled *per injector*. This means that if you create a new injector you will get a new instance of the singleton also. There are several ways to solve this problem; something you'll have to figure out yourself.

You can check that the singleton works properly by doing a *System.out.println())* inside the constructor of *CalendarDayButtonAction*. If you only get one printout, everything is correct.

3 Conclusion

Many parts of the source code provided to you in this workshop are far from perfect. They work fine in this small example but will probably have to be restructured for a bigger project.

As you have seen, using a dependency injection framework such as Google Guice can reduce code complexity in your projects. It also adds improved flexibility to your code. The use of annotations in this workshop is just one example of how you can take advantage of the power that annotations give you. The custom class loading you have implemented can be made dynamic to react to changes in the filesystem. That is, if someone was to copy an updated version of an already loaded plugin into the plugin directory, it's possible to listen to changes and reload it.

Now, go forth and use what you have learned here within your project.

4 Troubleshooting

Question: Whenever I try to set the GridBagConstraints inside *calendarapp.plugins.PluginClassLoader.getConstraints()* the component is still not expanding. What am I doing wrong.
Answer: Usually this means that you forgot to set a weight. In a toolkit such as Swing (and many other), a weight specifies how *hungry* a widget is for space.

Question: What do I do if I get the runtime exception NoClassFound (wrong name)?
Answer: This means that java can't define the class you are loading. This is often because the package name or class name does not match the package or class name of the compiled class. Refer to *calendarapp.plugins.PluginClassLoader.exportedPackage* and make sure it points to the same package as the compiled class.

Question: I get a 'bad magic number' exception. Now what?
Answer: In Java classes there is a *0xCAFEBABE* magic number. This is used to verify that the class is actually a valid Java class. If you get this error you probably didn't implement *calendarapp.plugins.PluginClassLoader.loadClassData()* correctly.

References

[1] Google Inc., *Google Guice*,
`http://code.google.com/p/google-guice`

[2] Google Inc., *Google Guice 3.0 Core API*,
`http://google-guice.googlecode.com/svn/trunk/javadoc/index.html`

[3] Google Inc., *Google Guice, Linked Bindings*,
`http://code.google.com/p/google-guice/wiki/LinkedBindings`

[4] Google Inc., *Google Guice, Scopes*,
`http://code.google.com/p/google-guice/wiki/Scopes`

[5] Wikipedia, *Aspect-oriented programming*,
`http://en.wikipedia.org/wiki/Aspect-oriented_programming`

[6] Wikipedia, *Mock Object*,
`http://en.wikipedia.org/wiki/Mock_object`

[7] Oracle, *Trail: The Reflection API*,
`http://download.oracle.com/javase/tutorial/reflect`