Politecnico di Milano
Dipartimento di Elettronica, Informazione e Bioingegneria

Software Engineering 2
Code Inspection

Supervisor:
Prof. Elisabetta Di Nitto

Students:
Alberto Gasparin
Vito Matarazzo

Year 2015-2016

Contents

Revision History

Date	Reason for changes	Version
January 3, 2016		v1.0

1 Assigned Classes

Class:GlassFishInjectionProvider

```
appserver/web/jsf-connector/src/main/java/org/glassfish
/faces/integration/GlassFishInjectionProvider.java
```

2 Functional role of the assigned classes

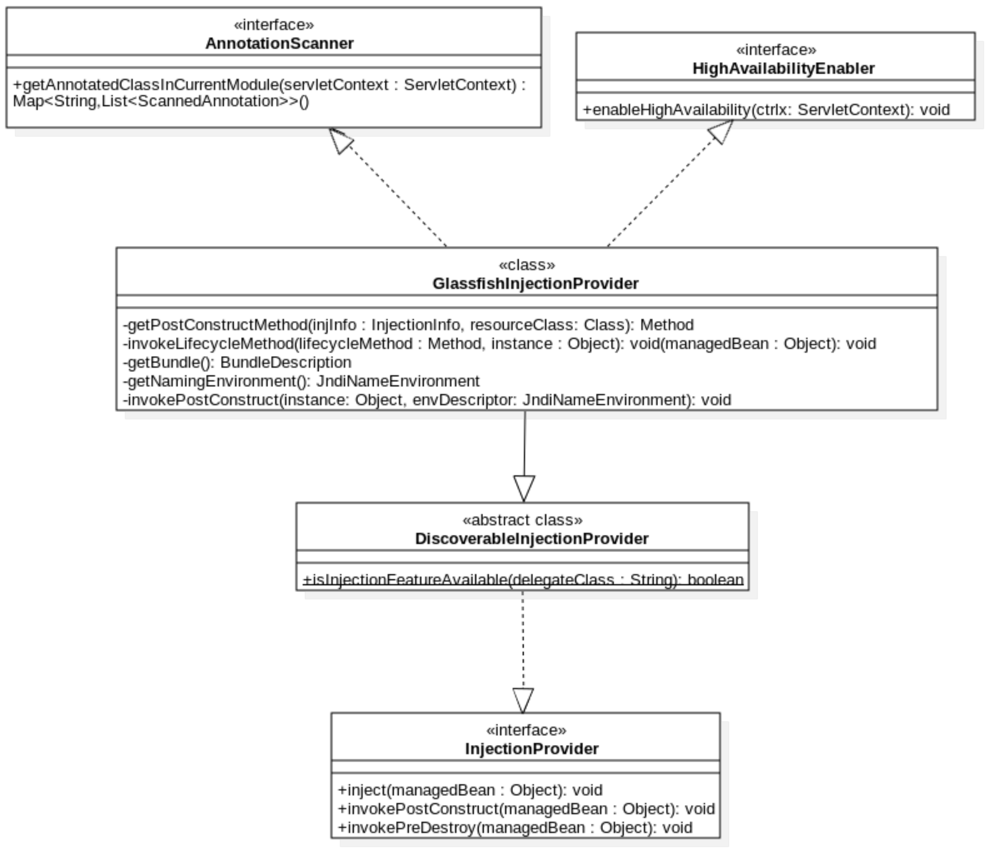
«interface»
AnnotationScanner
+getAnnotatedClassInCurrentModule(servletContext : ServletContext) : Map<String,List<ScannedAnnotation>>()
«interface»
HighAvailabilityEnabler
+enableHighAvailability(ctrlx: ServletContext): void
«class»
GlassfishInjectionProvider
-getPostConstructMethod(injInfo : InjectionInfo, resourceClass: Class): Method
-invokeLifecycleMethod(lifecycleMethod : Method, instance : Object): void(managedBean : Object): void
-getBundle(): BundleDescription
-getNamingEnvironment(): JndiNameEnvironment
-invokePostConstruct(instance: Object, envDescriptor: JndiNameEnvironment): void
«abstract class»
DiscoverableInjectionProvider
+isInjectionFeatureAvailable(delegateClass : String): boolean
«interface»
InjectionProvider
+inject(managedBean : Object): void
+invokePostConstruct(managedBean : Object): void
+invokePreDestroy(managedBean : Object): void

Figure 1: The Class diagram related to the GlassFishInjectionProvider class

The class GlassFishInjectionProvider is an extension of DiscoverableInjection-Provider which is an implementation of the InjectionProvider interface. This interface defines an integration point for Java EE vendors. Each vendor needs to provide an implementation of this interface, which will provide the JSF implementation the necessary hooks to perform resource injection. The main operations that any of these implementations must perform are:

- void inject(java.lang.Object managedBean) : which performs the injection of all the required resources (the values preceeded by the @Inject annotation) into the provided object managedBean, without invoking any methods annotated with @PostConstruct, which is instead made by another operation;

- void invokePreDestroy(java.lang.Object managedBean) : which activates any method marked with the @PreDestroy annotation on the provided argument, which is the target managed bean;
- void invokePostConstruct(java.lang.Object managedBean) : which activates any method marked with the @PostConstruct annotation on the target managed bean.

In our case the assigned class represents the implementation of InjectionProvider specific to the GlassFish application servers. The methods we analyzed focus on the postConstruct operation, in particular:

- void invokePostConstruct(Object instance, JndiNameEnvironment envDescriptor) : searches for postConstruct methods in the provided instance's class, using envDescriptor to get injection information about that class, and in its superclasses (until the least derived superclass in the hierarchy) and calls the following two methods;
- Method getPostConstructMethod(InjectionInfo injInfo, Class<? extends Object> resourceClass) : retrieves the postConstruct method specific for the provided resourceClass, searching through all its declared methods, and throws an InjectionException if it is not found;
- void invokeLifecycleMethod(final Method lifecycleMethod, final Object instance) : performs the actual invocation of the postConstruct method contained in lifecycleMethod on the provided instance, wrapping it into a PrivilegedExceptionAction to allow special access to that method.

As shown in Figure 1 the class also implements two more interfaces: Annotation Scanner and HighAvailabilityEnabler. The following method are inherited from the interface described before and will be analysed later in this document:

- Map<String, List<ScannedAnnotation» getAnnotatedClassesInCurrentModule(ServletContext servletContext) : this method is inherited from the AnnotationScanner interface. It builds a Map that has annotations (i.e. a string) as a key, while the value is represented by a list of all classes in the current module that contains that particular annotation. When the method ends it returns the Map to the caller.
- void enableHighAvailability(ServletContext ctx) : this method is inherited from the HighAvailabilityEnabler interface. It tests whether HighAvailability is enabled or not. If it is, the method does nothing, otherwise it overwrites the value of the config parameter EnableAgressiveSessionDirtying setting it to true.

3 List of found issues

Cx is a reference to the Checklist's x^{th} point. This checklist can be found in the Appendix section

3.1 public Map<String, List<ScannedAnnotation» getAnnotatedClassesInCurrentModule(ServletContext servletContext)

- (Line 112) Violation of C1. Variable dc should have a more meaningful name instead of a two-character string:

  ```
  DeploymentContext dc = (DeploymentContext)servletContext.getAttribute
  (Constants.DEPLOYMENT_CONTEXT_ATTRIBUTE);
  ```

- (Line 112 -113) Violation of C52. In

  ```
  DeploymentContext dc = (DeploymentContext)servletContext.getAttribute
  (Constants.DEPLOYMENT_CONTEXT_ATTRIBUTE);
  ```

  method getAttribute might return null. If that happen in the following code a NullPointerException would be thrown when method getTransientAppMetaData is called on the dc variable.

  ```
  Types types = dc.getTransientAppMetaData(Types.class.getName(),
      Types.class);
  ```

  Either an if statement or a try catch is required in order to handle this case.

- (Line 83) Violation of C14. The definition of the class is longer than 120 characters. It should be truncated, for example before the 'implements' part:

  ```
  public class GlassFishInjectionProvider extends
      DiscoverableInjectionProvider implements AnnotationScanner,
      HighAvailabilityEnabler {
  ```

- (Line 108-112) Violation of C12. Inconsistent convention is used. Before the first statement of the method no blank line is added usually while in this lines it appears.

  ```
  @Override
  public Map<String, List<ScannedAnnotation>>
      getAnnotatedClassesInCurrentModule(ServletContext servletContext)
      throws InjectionProviderException {

          DeploymentContext dc =
              (DeploymentContext)servletContext.getAttribute
          (Constants.DEPLOYMENT_CONTEXT_ATTRIBUTE);
  ```

- (Line 109) Violation of C27. The method is too long. This is due to the error pointed out below.
- (Line 128-162) Violation of C27. The anonymous class below is too big.

  ```
  toAdd = new ScannedAnnotation() {
                  ...
                  };
  ```

3.2 private void invokePostConstruct(Object instance, JndiNameEnvironment envDescriptor)

- Violation of C12. Incosistent convention is used. The blocks of instructions contained in this method (while at line 286, if at line 291 and for at line 304) start with an empty line, while the same blocks in other methods start with the first instruction directly below the opening brace.

- (Line 296-297) Violation of C18. The comment contains an error in the explanation of what the next instruction does, which is adding the PostConstruct methods to the postConstructMethods list, not the preDestroy ones. Moreover, the following line of code simply adds those methods to a list, the invocation is performed only after, in line 306:

```
// Invoke the preDestroy methods starting from
// the least-derived class downward.
postConstructMethods.addFirst(postConstructMethod);
}

              nextClass = nextClass.getSuperclass();
          }

          for (Method postConstructMethod : postConstructMethods) {

              invokeLifecycleMethod(postConstructMethod, instance);
```

- (Line 293-294) Violation of C15. The line is broken without any comma or operators; it should be broken after the opening bracket of the parameters' list, or after the first parameter:

```
                      Method postConstructMethod = getPostConstructMethod
                            (injInfo, nextClass);
```

- (Line 286, 291) Violation of C40. These two lines contain a comparison between an object and "null" with "!=" instead of using the equals method:
Line 286:

```
while ((!Object.class.equals(nextClass)) && (nextClass != null)) {
```

Line 291:

```
if (injInfo.getPostConstructMethodName() != null) {
```

- (Line 286) Violation of C45. The two conditions in the while expression have a useless pair of parenthesis that can be avoided:

```
while ((!Object.class.equals(nextClass)) && (nextClass != null)) {
```

3.3 private Method getPostConstructMethod(InjectionInfo injInfo, Class<? extends Object> resourceClass)

- (Line 324) Violation of C1 and C2. Variable m should have a more meaningful name instead of a one-character name:

```
Method m = injInfo.getPostConstructMethod();
```

- (Line 345-346) Violation of C15. The line is broken without any comma or operators; it should be broken after the opening bracket of the parameters' list, or after a '+' operator:

```
                throw new InjectionException
                    ("InjectionManager exception. PostConstruct method "
                        +
```

- (Line 327-328) Violation of C29. The postConstructMethodName variable should be defined at the beginning of this method instead of the first if block, since it is also used in the second if block by calling again the same method getPostConstructMethodName():

```
String postConstructMethodName =
                injInfo.getPostConstructMethodName();
```

- (Line 326, 344) Violation of C40. These two lines contain a comparison between an object and "null" with "==" instead of using the equals method:

```
if( m == null ) {
```

- (Line 331) Violation of C41. Comment in this line contains a spelling error.

```
// This does not include super-classses
```

- (Line 335-336) Violation of C45. The second condition in the if expression has a useless pair of parenthesis that can be avoided:

```
if( next.getName().equals(postConstructMethodName) &&
                    (next.getParameterTypes().length == 0) ) {
```

- (Line 332) Violation of C52. The invoked method getDeclaredMethods() can throw a SecurityException if a security manager is present and it denies access to the declared methods within the invoked class. This exception is not handled by this method or any other method in the assigned class, so it can cause problems if raised.

```
for(Method next : resourceClass.getDeclaredMethods()) {
```

3.4 private void invokeLifecycleMethod(final Method lifecycleMethod, final Object instance)

- (Line 388-389) Violation of C8. The indentation of the catch block is not composed of 4 spaces as the rest of the class.

```
        } catch( Exception t) {

                String msg = "Exception attempting invoke lifecycle "
                                + " method " + lifecycleMethod;
```

- (Line 388-389) Violation of C15. The line is broken without any comma or operators; it should be broken after '+' operator:

- (Line 388-389) Violation of C42. The error message inserted into the following InjectionException does not explain the cause of that problem:

```
String msg = "Exception attempting invoke lifecycle "
                    + " method " + lifecycleMethod;
```

- (Line 386-399) Violation of C50. The exceptions caught in this method should be properly detailed instead of using the most general class Exception. In particular:

  1. (Line 376) java.security.AccessController.doPrivileged(...) throws a PrivilegedActionException if the specified action's run method throws a checked exception, while unchecked ones propagate through the doPrivileged method;
  2. (Line 380) lifecycleMethod.setAccessible(true) throws a SecurityException if accessibility of the lifecycleMethod object may not be changed;
  3. (Line 382) lifecycleMethod.invoke(instance) throws InvocationTargetException if the underlying method throws any exception.

  Moreover, at line 394 an instanceof check is being performed on the caught exception. A separate catch clause should be created for this exception type:

```
} catch( Exception t) {

                String msg = "Exception attempting invoke lifecycle "
                    + " method " + lifecycleMethod;
                if (LOGGER.isLoggable(Level.FINE)) {
                    LOGGER.log(Level.FINE, msg, t);
                }
                InjectionException ie = new InjectionException(msg);
                Throwable cause = (t instanceof
                    InvocationTargetException) ?
                    t.getCause() : t;
                ie.initCause( cause );
                throw ie;

        }
```

- Violation of C12. The try and catch blocks in this method start with an empty line, while the same blocks in other methods and any other block(if, for, while...) of instructions start with the first instruction directly below the opening brace, so an incosistent convention is used.

3.5 public void enableHighAvailability(ServletContext ctx)

- (Line 436-441) Violation of C41. There is a spelling error in the comment below. The method tests *if* (not with) the HA has been enabled.

```
/**
     * Method to test with HA has been enabled.
     * If so, then set the JSF context param
     * com.sun.faces.enableAgressiveSessionDirtying to true
     * @param ctx
     */
    public void enableHighAvailability(ServletContext ctx)
```

- (Line 453) Violation of C14. The line is longer than 120 characters. It should be truncated before, for example by putting both the parameters of the called method in the next line:

  ```
  config.overrideContextInitParameter(WebConfiguration.
  BooleanWebContextInitParameter.EnableAgressiveSessionDirtying,
  ```

- (Line 442-444) Violation of C12. Inconsistent convention is used. Before an if statement no blank line is added usually while in this lines it appears.

  ```
  boolean enableHA = ((Boolean)enableHAObj).booleanValue();

          if (LOGGER.isLoggable(Level.FINE)) {
                  LOGGER.log(Level.FINE,
                  "isDistributable = {0} enableHA = {1}",
                          new Object[]{isDistributable, enableHA});
      }
  ```

3.6 Overall class problems

- Violation of C27. Possible God class: the assigned class can be considered a God class, because it is very big and overly complex, and manages too many objects. God classes should be split apart to be more object-oriented.
- Violation of C27. The assigned class has a great number of imports, in particular from other classes of the same project; this could indicate a high degree of coupling.
- Violation of C12. There is no blank line separating the import statements from the class documentation which begins at line 79.
- Violation of C23. The javadoc is incomplete:

  1. the assigned class has no explanation on what it does, the documentation only states that it's an InjectionProvider specific to the GlassFish application server;
  2. when a method throws an exception, the @throws clause should explain more in detail what error has caused it, instead of just saying "if an error occurs".
  3. There is no javadoc for the private method of the class:

     ```
     private JndiNameEnvironment getNamingEnvironment()
             throws InjectionException
     private void invokePostConstruct(Object instance,
                                          JndiNameEnvironment
                                              envDescriptor)
        throws InjectionException
     private Method getPostConstructMethod(InjectionInfo injInfo,
                                               Class<? extends
                                                   Object>
                                                   resourceClass)
            throws InjectionException
     private void invokeLifecycleMethod(final Method lifecycleMethod,
                                          final Object instance)
            throws InjectionException
     private BundleDescriptor getBundle()
     ```

   This might be acceptable if the methods would have been well commented, which they are not.

4. The method getAnnotatedClassesInCurrentModule is not described in the javadoc.

- Violation of C26. Methods in the assigned class are not grouped by functionality, but by accessibility: first the public methods, then the private ones, except for the last method (enableHighAvaialbility) which is again a public one.

4 Other problems

- The following methods miss the @Override annotation above the method signature. This annotation is used to instruct compiler that method annotated with @Override is an overridden method from super class or interface. It's use prevent the programmer to make mistakes such as misspelling the method name or not matching the parameters (in this way the method would be overloaded and the compiler will not display an error. The @Override annotation prevents that by throwing a compilation error). Moreover, it makes the code more understanable.

```
public void inject(Object managedBean) throws
    InjectionProviderException
public void invokePreDestroy(Object managedBean)
public void invokePostConstruct(Object managedBean)
public void enableHighAvailability(ServletContext ctx)
```

- The following portion of code does not respect OOP principal: "Program to an Interface, not to an Concrete Implementation". Creating a rigid model that can only use LinkedList is not a best practice and it limits flexibility.

```
    private void invokePostConstruct(Object instance,
        JndiNameEnvironment envDescriptor) throws InjectionException
        {
        LinkedList<Method> postConstructMethods = new
            LinkedList<Method>();
        ...
}
```

- The anonymous class that follows is too articulate. A named class would be better.

```
toAdd = new ScannedAnnotation(){
...
}
```

- Useless import: import org.apache.catalina.core.StandardContext is never used, so can be removed.

- The four private attributes of the class could be made final; they are only initialized in the constructor.

```
private ComponentEnvManager compEnvManager;
private InjectionManager injectionManager;
private InvocationManager invokeMgr;
private JCDIService jcdiService;
```

- Line 147 and 148 contain 2 confusing ternaries, because "if" expressions with an "else" clause should not contain the negation of a predicate in the "if" clause, like if (x != y) ..; else ..; this should be rephrased as if (x == y)..; else ..; to make the code easier to read.

```
 int result = str != null ? str.hashCode() : 0;
 result = 31 * result + (obj != null ? obj.hashCode() : 0);
```

- Line 124 contains an assignment inside one of the operands of the comparison; this can make code more complicated and harder to read.

```
 if (null == (classesWithThisAnnotation =
     classesByAnnotation.get(curAnnotationName))) {
```

5 Appendix

5.1 Checklist

Naming Conventions

1. All class names, interface names, method names, class variables, method variables, and constants used should have meaningful names and do what the name suggests.
2. If one-character variables are used, they are used only for temporary "throw-away" variables, such as those used in for loops.
3. Class names are nouns, in mixed case, with the first letter of each word in capitalized. Examples: `class Raster`; `class ImageSprite`;
4. Interface names should be capitalized like classes.
5. Method names should be verbs, with the first letter of each addition word capitalized. Examples: `getBackground()`; `computeTemperature()`.
6. Class variables, also called attributes, are mixed case, but might begin with an underscore ('`_`') followed by a lowercase first letter. All the remaining words in the variable name have their first letter capitalized. Examples: `_windowHeight`, `timeSeriesData`.
7. Constants are declared using all uppercase with words separated by an underscore. Examples: `MIN_WIDTH`; `MAX_HEIGHT`.

Indention

8. Three or four spaces are used for indentation and done so consistently.
9. No tabs are used to indent.

Braces

10. Consistent bracing style is used, either the preferred "Allman" style (first brace goes underneath the opening block) or the "Kernighan and Ritchie" style (first brace is on the same line of the instruction that opens the new block).
11. All `if`, `while`, `do-while`, `try-catch`, and `for` statements that have only one statement to execute are surrounded by curly braces. Example: avoid this:

    ```
            if ( condition )
                    doThis();
    ```

    instead do this:

    ```
        if ( condition )
        {
            doThis();
        }
    ```

File Organization

12. Blank lines and optional comments are used to separate sections (beginning comments, package/import statements, class/interface declarations which include class variable/attributes declarations, constructors, and methods).
13. Where practical, line length does not exceed 80 characters.
14. When line length must exceed 80 characters, it does NOT exceed 120 characters.

Wrapping Lines

15. Line break occurs after a comma or an operator.
16. Higher-level breaks are used.
17. A new statement is aligned with the beginning of the expression at the same level as the previous line.

Comments

18. Comments are used to adequately explain what the class, interface, methods, and blocks of code are doing.
19. Commented out code contains a reason for being commented out and a date it can be removed from the source file if determined it is no longer needed.

Java Source Files

20. Each Java source file contains a single public class or interface.
21. The public class is the first class or interface in the file.
22. Check that the external program interfaces are implemented consistently with what is described in the javadoc.
23. Check that the javadoc is complete (i.e., it covers all classes and files part of the set of classes assigned to you).

Package and Import Statements

24. If any package statements are needed, they should be the first non-comment statements. Import statements follow.

Class and Interface Declarations

25. The class or interface declarations shall be in the following order:
    (a) class/interface documentation comment;
    (b) class or interface statement;
    (c) class/interface implementation comment, if necessary;
    (d) class (static) variables;
        i. first public class variables;
        ii. next protected class variables;
        iii. next package level (no access modifier);
        iv. last private class variables.
    (e) instance variables;
        i. first public instance variables;
        ii. next protected instance variables;
        iii. next package level (no access modifier);
        iv. last private instance variables.
    (f) constructors;
    (g) methods.
26. Methods are grouped by functionality rather than by scope or accessibility.
27. Check that the code is free of duplicates, long methods, big classes, breaking encapsulation, as well as if coupling and cohesion are adequate.

Initialization and Declarations

28. Check that variables and class members are of the correct type. Check that they have the right visibility (public/private/protected).
29. Check that variables are declared in the proper scope.
30. Check that constructors are called when a new object is desired.
31. Check that all object references are initialized before use.
32. Variables are initialized where they are declared, unless dependent upon a computation.
33. Declarations appear at the beginning of blocks (A block is any code surrounded by curly braces '`{`' and '`}`'). The exception is a variable can be declared in a `for` loop.

Method Calls

34. Check that parameters are presented in the correct order.
35. Check that the correct method is being called, or should it be a different method with a similar name.
36. Check that method returned values are used properly.

Arrays

37. Check that there are no off-by-one errors in array indexing (that is, all required array elements are correctly accessed through the index).
38. Check that all array (or other collection) indexes have been prevented from going out-of-bounds.
39. Check that constructors are called when a new array item is desired.

Object Comparison

40. Check that all objects (including Strings) are compared with `equals` and not with ==.

Output Format

41. Check that displayed output is free of spelling and grammatical errors.
42. Check that error messages are comprehensive and provide guidance as to how to correct the problem.
43. Check that the output is formatted correctly in terms of line stepping and spacing.

Computation, Comparisons and Assignments

44. Check that the implementation avoids "brutish programming": (see `http://users.csc.calpoly.edu/~jdalbey/SWE/CodeSmells/bonehead.html`).
45. Check order of computation/evaluation, operator precedence and parenthesizing.
46. Check the liberal use of parenthesis is used to avoid operator precedence problems.
47. Check that all denominators of a division are prevented from being zero.
48. Check that integer arithmetic, especially division, are used appropriately to avoid causing unexpected truncation/rounding.
49. Check that the comparison and Boolean operators are correct.
50. Check throw-catch expressions, and check that the error condition is actually legitimate.
51. Check that the code is free of any implicit type conversions.

Exceptions

52. Check that the relevant exceptions are caught.
53. Check that the appropriate action are taken for each catch block.

Flow of Control

54. In a `switch` statement, check that all cases are addressed by break or return.
55. Check that all switch statements have a default branch.
56. Check that all loops are correctly formed, with the appropriate initialization, increment and termination expressions.

Files

57. Check that all files are properly declared and opened.
58. Check that all files are closed properly, even in the case of an error.
59. Check that EOF conditions are detected and handled correctly.
60. Check that all file exceptions are caught and dealt with accordingly.

5.2 Software Tool used

- TexMaker (http://www.xm1math.net/texmaker/): LaTeXeditor, used to redact this document.
- StarUML (http://staruml.io/) : used to build Sequence diagrams.
- SonarQube 5.0.1 (http://http://www.sonarqube.org/): for inspection of code quality.
- PMD 5.4.1 (http://pmd.github.io/) is a static Java source code analyzer.

5.3 Hours of work

- Alberto Gasparin $\sim$ 10 h
- Vito Matarazzo $\sim$ 10 h