

# Inversion of Control Programming Assignment

Kevin León Sandoval, B53845  
Josué León Sarkis, B53846  
Elías Calderón Calderón, B51322

Sunday 1<sup>st</sup> October, 2017

Escuela de Ciencias de la Computación e Informática

Universidad de Costa Rica.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Summary . . . . .	3
<b>2</b>	<b>Inversion of Control Container</b>	<b>5</b>
2.1	System description . . . . .	5
2.2	XML Configuration . . . . .	6
2.3	Annotations Configuration . . . . .	10
<b>3</b>	<b>Implementation</b>	<b>14</b>
3.1	Solution description . . . . .	14
3.2	Flow Diagram . . . . .	17
3.3	Class diagram . . . . .	18
<b>A</b>	<b>Source Code</b>	<b>19</b>
	<b>Bibliography</b>	<b>114</b>

# Chapter 1

## Introduction

### 1.1 Summary

*Software engineering* is a Computer Science branch, based on the two concepts software and engineering, and consisting on a set of methods, models, tools, and techniques that facilitate software development.

One of the main goals of software engineers is to reduce this process complexity, which is naturally very high. Software engineering has a lot of design patterns and programming models, that can be used when creating software systems. One of the most used and famous design patterns is *Dependency Injection*. Dependency injection removes the dependencies between objects from their internal composition, and handles them by itself in order to separate the programs in more parts, making a program's objects work independently from others.

In the other hand, we have Inversion of control, this technique consists in, redundantly, inverting the flow of the program execution. This means, that the developer no longer holds the main control of the program, and it is now driven by a framework. Merging this with Dependency injection, gives us a framework that is in charge of creating and controlling the dependencies of each object and injecting them, only when needed, increasing the modularity.

An Inversion of Control container is a very complex program that requires a variety of elaborated algorithms and patterns to successfully perform its job. The term was popularized in 1999 by the computer scientist Stefano

Mazzochi and since then, different frameworks based on this principle have been developed.

This document presents the documentation of the implementation of NAIoCC(*Not Another Inversion of Control Container*). The documentation includes the flow diagram, the class diagram, the description of the solution, and the metadata configuration for both XML and Java annotations.

# Chapter 2

## Inversion of Control Container

### 2.1 System description

The inversion of control supports the following functionalities:

1. Dependency injection:

- a)* Setters.
- b)* Constructor.

2. Scope:

- a)* Singleton.
- b)* Prototype.

3. Lifecycle:

- a)* Initialization.
- b)* Destroy.

4. Autowiring:

- a)* By Name.
- b)* By Type.
- b)* Constructor.

5. Configuration Format:

- a) XML.
- b) Annotations.

6. Extra features:

- a) Lazy Loading.
- b) Stereotype annotations.

## 2.2 XML Configuration

To use the container, the user has to define the beans and the information to create them(*metadata*) in an XML file. The XML file is read by creating an *XML Bean Factory*.

The following concepts are the ones that NAIoCC supports for its XML configuration:

- **Bean** is an abstract object, for the dependency injection, that is created by the bean factory and it is saved in the inversion of control container. It has **id**, **class**, **scope**, **init**, **destroy**, **lazy-generation** and **autowire**.
- **Id** is the unique identification for each bean. It is obligatory. Its value can not be repeated in different beans.
- **Class** specifies the bean's class path. It is obligatory.
- **Scope** specifies the scope of the bean. It is not obligatory. Its value can be **Singleton**, which is the default value, or **Prototype**. Singleton means that just one bean is instantiated and all the requests for that bean use the same instance. **Prototype** consists in creating a new instance of the bean each time it is requested.
- **Init** consists in the initialization method for a bean. It is not obligatory. This method is called when the bean is instantiated, **after** injecting all the dependencies of that bean. In the XML file an init default method for every bean can be defined, if a bean specifies a different init method, the default one is overwritten.

- **Destroy** consists in the destruction method for a bean. It is not obligatory. This method is called when the bean is destroyed. In the XML file a destroy default method for every bean can be defined, if a bean specifies a different destroy method, the default one is overwritten.
- **Lazy-generation** determines if the bean is instantiated when the container is created or when the bean is requested by a user, its effect is only visible in Singletons. It is not obligatory, by default it is set to false. Its value can be the same for different beans. If the value is **true** the bean is instantiated until it is requested, otherwise **false**, the bean is instantiated when the container is created. For Prototype scopes, it has no effect since beans are instantiated when requested.
- **Autowire** specifies the automatic way of wiring the dependencies for all the properties found in a bean's class. It is not obligatory, by default it is set to "none". Its value can be repeated in different beans. The value can be set to "byName", "byType", "constructor" or "none". It has no effect in a specific attribute or constructor, if the respective **attribute** or **constructor** tag is specified.
- **Atomic-autowire** specifies the automatic way of wiring the dependencies for a property in a bean. It is not obligatory, by default it is set to "none". Its value can be repeated in different beans. The value can be set to "byName", "byType", or "none".
- **Attribute** specifies a bean's attribute, so that it is injected through "setter" methods. It is not obligatory. There can be multiple attributes defined in a bean. It has a **name**, and either a **ref** or a **value** (for primitive types), but not both.
- **Name** identifies the name of the attribute to inject. It is obligatory. Its value can not be repeated inside the same bean.
- **Constructor** is used to define constructor injection and specify its parameters. It is not obligatory. It is unique for a bean, therefore there can only be one Constructor tag inside a bean's configuration.
- **Param** identifies a constructor's argument. It is obligatory. It can have **type**, **index** or both. It can also have **value** or **ref**, but not both.
- **Type** identifies the type for a constructor's argument. It is not obligatory.

- **Index** identifies the index in the argument's array for a constructor's argument. It is not obligatory, and its value must be unique inside a bean's constructor.
- **Value** identifies the attribute's value or argument's value. It is not obligatory. Its value is not unique.
- **Ref** makes reference to a declared bean ID for any other bean in the XML. It is not obligatory. The reference is unique but can be used by multiple beans.
- **Annotations Classes** Indicates that there are annotations in some classes. There can be just one annotationsClasses tag in the XML configuration.
- **Class** tag determines a class which contains annotations. It has an attribute called **path**, which specifies the path of the respective class.
- **Path** has the name of the class with annotations. It can be just one per **class** tag.



XML structure:

```
<xml version = "1.0" encoding = "UTF-8"?>
<beans init="defaultInitMethod" destroy="defaultDestroyMethod">
  <bean id = "beanId" class = "package.path.class"
    scope="Singleton/Prototype"
    init="methodName" destroy="methodName"
    lazy-generation="true/false"
    autowire="byName/byType/none">

    <constructor>
      <param type="package.path.class"
        index="numberIndex"
        value="valor"/ref="beanId" />

      <param ref="beanId" atomic-autowire="byName/
        byType" />

      <param type="package.path.class" atomic-
        autowire="byName/byType" />
    </constructor>

    <attribute name="nombreAtr" value="valor"/ref="
      beanId" atomic-autowire="byName/byType" />

    <attribute name="nombreAtr" atomic-autowire="byName
      /byType" />

  </bean>

  <annotationsClasses>

    <class path="package.path.class" />

  </annotationsClasses>

</beans>
```

## 2.3 Annotations Configuration

The annotations configuration can be used alongside XML configuration or making an *Annotations Bean Factory*. The following are the annotations concepts and structure:

- **@Bean:** It indicates to the container that the class with the **@Bean** annotation must be registered as a bean in the container. The bean ID is obligatory, it can be the same name of the class. There can just be one **@Bean** in a class. It goes above the class definition.

*Structure:*

```
@Bean
public class BeanClass {
    ...
}
```

- **@Scope:** It indicates the bean's scope. There can only be one **@Scope** in a class. Its values are **Singleton**, by default, or **Prototype**.

*Structure:*

```
@Bean
@Scope("Singleton")/@Scope("Prototype")
public class BeanClass {
    ...
}
```

- **@Init:** It is the initialization method. There can be just one **@Init** in a class. It determines which method to call when the bean is instantiated.

*Structure:*

```
@Bean
public class BeanClass {
    @Init
    public void initMethod() {
        ...
    }
}
```

- **@Destroy:** It is the destruction method. There can be just one **@Destroy** in a class. It determines which method to call when the bean is destroyed.

*Structure:*

```

@Bean
public class BeanClass {
    @Destroy
    public void destroyMethod() {
    }
}

```

- **@Lazy:** It determines if the bean is instantiated when the container is created or when the bean is requested by a user, in the case that its scope is Singleton. It is not obligatory, by default it is set to false. If @Lazy is present, the bean is instantiated until it is requested, otherwise the bean is instantiated when the container is created. For Prototype scopes, it behaves the same way as normal since beans are instantiated when requested. There can be just one @Lazy in a class.

*Structure:*

```

@Bean
@Lazy
public class BeanClass {
    ...
}

```

- **@ClassAutowire:** It specifies to wire the bean's dependencies automatically, "byName", "byType", "constructor" or "none". The default value is "byName". It can be above the class definition, or above a constructor.

*Structure:*

```

@Bean
@Autowire()
public class BeanClass {
    ...
}

```

- **@Attribute:** It goes above of the attribute, which is going to be a property of the bean and should have an associated setter method. This annotation has an obligatory parameter, to specify the **value** or **ref**(reference) of the attribute. For non-primitive types, it is equivalent to the @Autowired followed by @Qualifier("reference") in the Spring container. There can be multiple @Attribute in a class.

*Structure:*

```

@Bean
public class BeanClass {

```

```

    @Attribute("2")/@Attribute("ref")
    private int classInt;

    public void setClassInt() {
        ...
    }
}

```

- **@AtomicAutowire:** It goes above an attribute or the constructor. It states that the property must be autowired and can be "byName" or "byType". If the type is not indicated, "byType" is assumed firstly and if it doesn't matches with the parameters, "byName" is tried.

*Structure:*

```

@Bean
public class BeanClass {

    @AtomicAutowire()/@AtomicAutowire("byName"/"byType")
    private int classInt;

    @AtomicAutowire()/@AtomicAutowire("byName"/"byType")
    public BeanClass() {}
    public void setClassInt() {
        ...
    }
}

```

- **@Constructor:** It goes above the constructor that will be used in the bean dependency injection. There can be just one @Constructor in the class.

*Structure:*

```

@Bean
public class BeanClass{
    @Constructor
    public BeanClass(. . .){
        . . .
    }
}

```

- **@Parameter:** It should be present after @Constructor. It indicates the value of one of the parameters of the bean's constructor definition. This annotation has an obligatory parameter, to specify the **value** or **ref**(reference).

*Structure:*

```
@Bean
public class BeanClass{
    @Constructor
    @Parameter("val1")/@Parameter("ref")
    public BeanClass(. . .){
        . . .
    }
}
```

# Chapter 3

## Implementation

### 3.1 Solution description

In order to achieve a successful implementation of the Inversion of Control container, NAIoCC's team researched about Java Reflection. Java Reflection is a library of Java, which provides the utilities to get important properties of a class, such as the name, fields, methods, constructors, constructor parameters, annotations and other important components. The team also researched about Spring, in the Spring Documentation, to learn how Spring manages the different configurations and flows to get a general idea of how their IoC work and use it for our implementation.

The team had several meetings, after the research process, to discuss the application domain and define the problem and what was needed. Then we made the design of the implementation, which consisted on a flow diagram and the class diagram. After the design process, we decided how to parse the XML and agreed on using DOM(*Document Object Model*) Parser to read and process the XML configuration file, because it is a very useful and simple tool to do it. DOM Parser processes the main tag as a root of a tree structure, the tags inside as his children, and the properties of the tags as attributes of his children or himself.

In this way, the general solution from a high level perspective consists on reading the beans' configuration, either from the XML file or Java Annotations classes, via the XMLBeanReader or AnnotationsBeanReader, both of them classes that inherit from the abstract class BeanReader. While reading

the configurations, the metadata of each bean is passed to the BeanCreator class, which is in charge of creating each bean with its corresponding basic configuration values and properties. Once the reader finishes reading the properties of a bean, it communicates it to the BeanCreator, for it to pass it to the BeanFactory so that it is added to the container. It is important to highlight that at this point, the beans in the container only hold the metadata, no beans have been instantiated nor autowired.

Once the reader finished reading all the configuration, and with all the beans in the container, the BeanFactory proceeds to check the beans dependencies to detect any cycles and if no cycles were detected, it then instantiates all beans, by iterating through all of them in the beans HashMap and checking important conditions such as their scope, to determine if they should be instantiated now, in the case that it is Singleton and without Lazy Generation, or later when the user requests the bean.

Moreover, in general terms, when a bean is instantiated, a new instance is created and added to the list of instances in the Bean class, which holds the metadata. When creating the new instance, the dependencies are first autowired and then it is created with the specified constructor or the default one if it wasn't specified. Once the new instance is created, its dependencies are then injected, via the setter methods, if indicated as such.

The process of autowiring can be executed in different ways, depending on the configuration. In the case that it is set to "byName", it searches for a bean in the container, that has the same id as the name specified, in order to wire it. In the case that it is set to "byType", a bean with the respective type of the property is searched in the container, and if it finds it, it autowires it. Finally, if set to "constructor", it is similar to "byName" since it finds the beans with the same name of the parameter in the container, to wire it.

The injection of dependencies after the bean is instantiated, consists in calling the setter methods of the attributes specified to inject. It does this by finding the setter method for the attribute, using its name and then invoking the method. However, in the case that the bean's configuration belongs to Java Annotations, the object to inject is searched in the container by its name, specified with the @Qualifier annotation, to inject it afterwards.

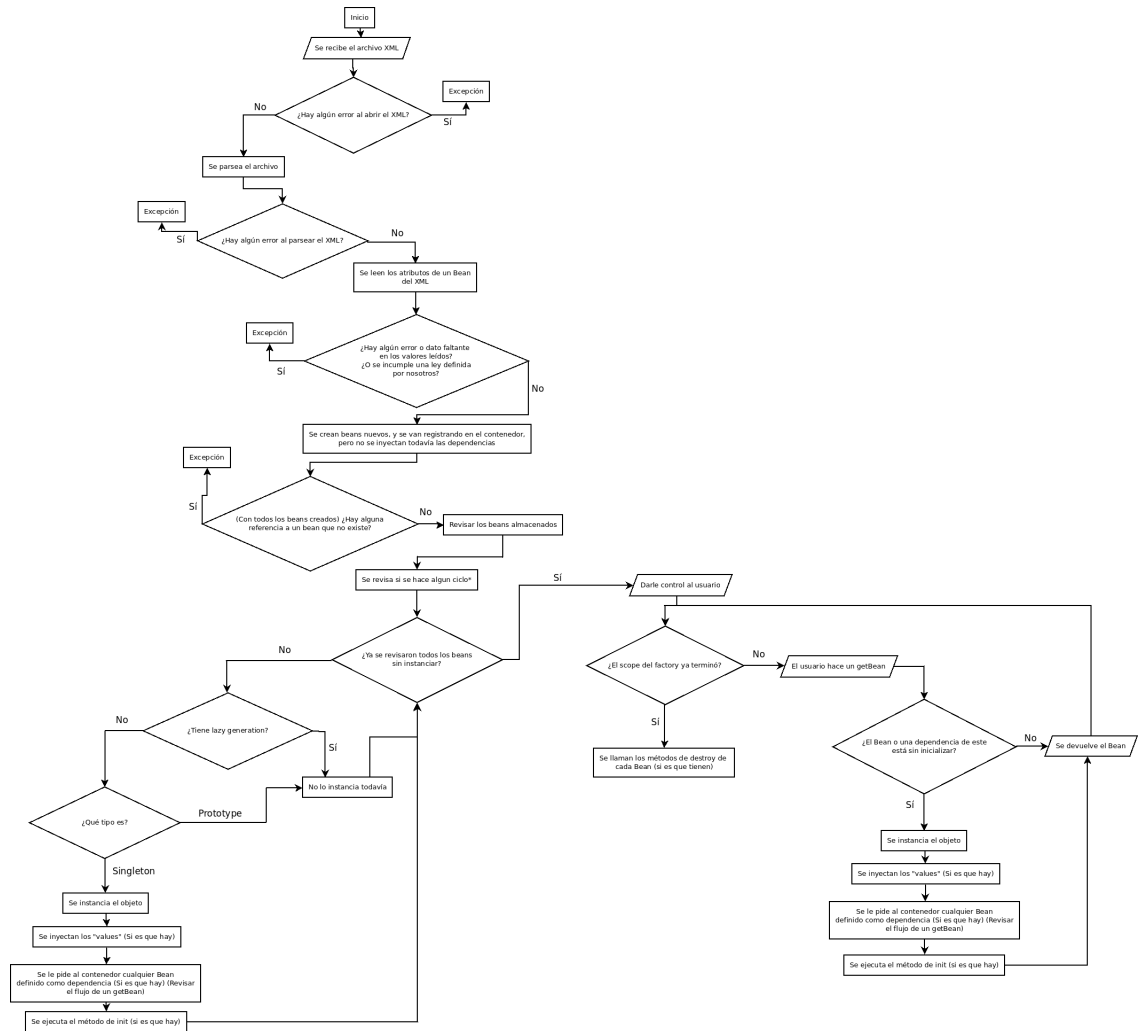
It is important to stand out that many exceptions are controlled, in the various scenarios. For example, if the autowire is set to "byType" and no reference is specified, if it finds more than one bean with that type, an ex-

ception is throwed. Other exceptions include checking that no two beans can have the same id, references that differ on type with the property, unexisting necessary methods to invoke, etc.

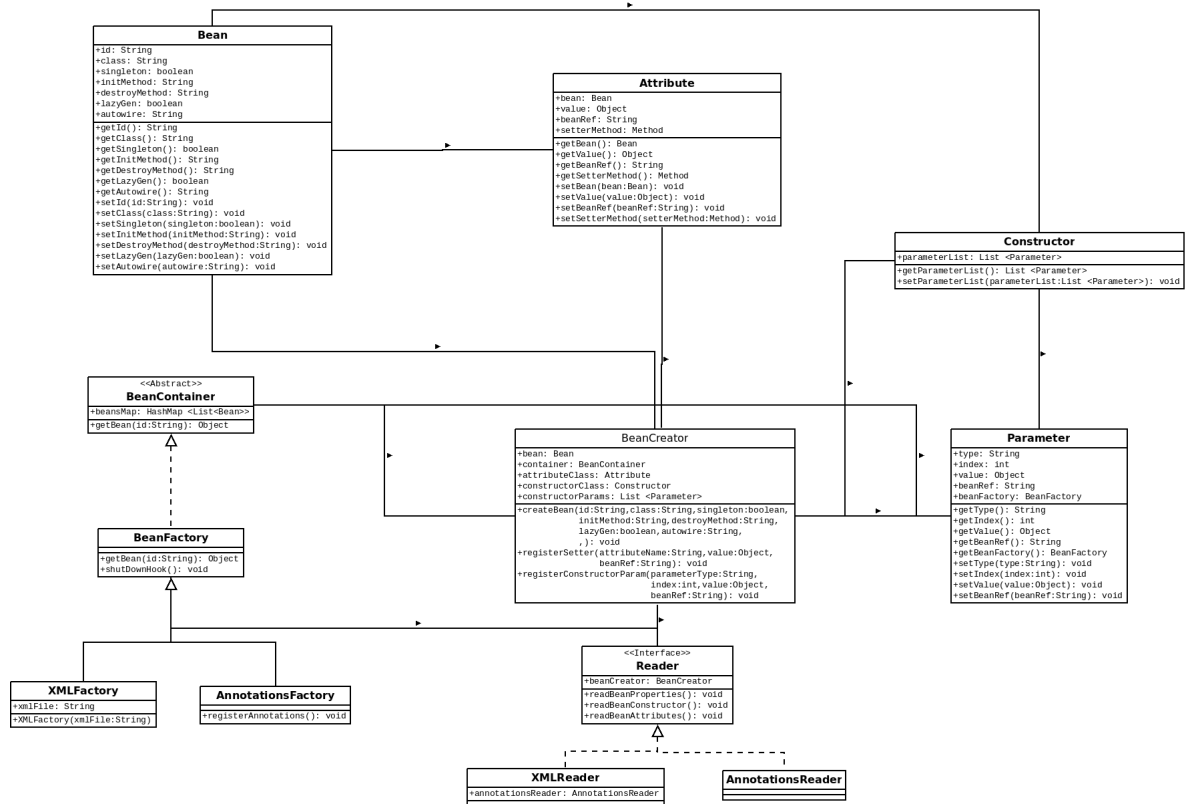
Finally, the user can also call the `shutDownHook` method of the `BeanFactory`, which will iterate through all the beans in the `HashMap` and destroy all the instances for each bean.



## 3.2 Flow Diagram



### 3.3 Class diagram



# Appendix A

## Source Code

### BeanReader

```
1 package com.ci1330.ecci.ucr.ac.cr.readers;
2
3 import com.ci1330.ecci.ucr.ac.cr.bean.AutowiredEnum;
4 import com.ci1330.ecci.ucr.ac.cr.bean.Scope;
5 import com.ci1330.ecci.ucr.ac.cr.exception.
    XmlBeanReaderException;
6 import com.ci1330.ecci.ucr.ac.cr.factory.BeanCreator;
7 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
8
9 /**
10  * @author Elias Calderon, Josue Leon, Kevin Leon
11  * Date: 13/09/2017
12  * The father class for a reader, defines the {@link BeanCreator
13  * } to use and the method, readBeans
14  */
15 public class BeanReader{
16
17     /**
18      * Object used to create the beans
19      */
20     protected BeanCreator beanCreator;
21
22     /**
23      * General constructor that initializes the creator
24      * @param beanFactory the factory that the creator will use
25      */
26     public BeanReader (BeanFactory beanFactory) {
27         this.beanCreator = new BeanCreator(beanFactory);
28     }
29 }
```

```

29  /**
30   * This constructor receives the bean creator
31   * @param beanCreator the creator to use
32   */
33  public BeanReader (BeanCreator beanCreator) {
34      this.beanCreator = beanCreator;
35  }
36
37  /**
38   * Abstract method, that indicates the name of the input to
39   * read
40   * @param inputName the name of the configuration container
41   */
42  public void readBeans (String inputName) {}
43
44  /**
45   * Determines which type of {@link AutowireEnum} is entered,
46   * if not found, throws an exception and exits.
47   * Atomic autowiring, only accepts byName, byType or none.
48   * @param atomic_autowireString the String to match with a
49   * type of {@link AutowireEnum}
50   * @return the respective {@link AutowireEnum}
51   */
52  protected AutowireEnum determineAtomic_Autowire (String
53      atomic_autowireString) {
54
55      final String byNameString = "byname";
56      final String bytypeString = "bytype";
57      final String noneString = "none";
58
59      AutowireEnum atomic_autowire = null;
60
61      switch (atomic_autowireString) {
62          case byNameString:
63              atomic_autowire = AutowireEnum.byName;
64              break;
65          case bytypeString:
66              atomic_autowire = AutowireEnum.byType;
67              break;
68          case noneString:
69              atomic_autowire = AutowireEnum.none;
70              break;
71          default:
72              try {
73                  throw new XmlBeanReaderException("XML_Reader
74                      _Error:_The_value_for_atomic-autowire_"
75                      + atomic_autowireString + "'_was_not_"
76                      + "recognized.");
77              } catch (XmlBeanReaderException e) {

```

```

71         e.printStackTrace();
72         System.exit(1);
73     }
74 }
75
76     return atomic_await;
77 }
78
79 /**
80  * Determines which type of {@link AutowireEnum} is entered,
81  * if not found, throws an exception and exits
82  * @param autowireString the String to match with a type of
83  *   {@link AutowireEnum}
84  * @return the respective {@link AutowireEnum}
85  */
86 protected AutowireEnum determineClass_Autowire (String
87 autowireString) {
88     final String byNameString = "byname";
89     final String bytypeString = "bytype";
90     final String byConstructorString = "constructor";
91     final String noneString = "none";
92
93     AutowireEnum autowire = null;
94
95     //If none of those was specified, the system throws an
96     //exception
97     switch (autowireString) {
98         case byNameString:
99             autowire = AutowireEnum.byName;
100             break;
101         case bytypeString:
102             autowire = AutowireEnum.byType;
103             break;
104         case byConstructorString:
105             autowire = AutowireEnum.constructor;
106             break;
107         case noneString:
108             autowire = AutowireEnum.none;
109             break;
110         default:
111             try {
112                 throw new XmlBeanReaderException("XML_Reader
113                     _Error:_The_value_for_await_" +
114                     autowireString + "'_was_not_recognized.")
115                     ;
116             } catch (XmlBeanReaderException e) {
117                 e.printStackTrace();
118                 System.exit(1);
119             }
120     }

```

```

113     }
114
115     return autowire;
116 }
117
118 /**
119  * Determines which type of {@link Scope} is entered, if not
120  * found, throws an exception and exits
121  * @param scopeString the String to match with a type of {
122  *   @link Scope}
123  * @return the respective {@link Scope}
124  */
125 protected Scope determineScope (String scopeString) {
126     final String singletonString = "singleton";
127     final String prototypeString = "prototype";
128     Scope scope = null;
129
130     //If prototype wasn't specified, the system throws an
131     //exception
132     switch (scopeString) {
133         case prototypeString:
134             scope = Scope.Prototype;
135             break;
136         case singletonString:
137             scope = Scope.Singleton;
138             break;
139         default:
140             try {
141                 throw new XmlBeanReaderException("XMLReader
142                     _Error:_The_value_for_scope_" +
143                     scopeString + "'_was_not_recognized.");
144             } catch (XmlBeanReaderException e) {
145                 e.printStackTrace();
146                 System.exit(1);
147             }
148     }
149
150     return scope;
151 }
152
153 /**
154  * Determines which value of lazy generation is entered, if
155  * not found, throws an exception and exits
156  * @param lazyGenString the String to match with true or
157  *   false
158  * @return a boolean indicating the lazy generation value
159  */
160 protected Boolean determineLazyGen (String lazyGenString) {
161     final String trueString = "true";

```

```

155         final String falseString = "false";
156
157         Boolean lazyGeneration = false;
158
159         //If none of those was specified, the system throws an
            exception
160         switch (lazyGenString) {
161             case trueString:
162                 lazyGeneration = true;
163                 break;
164             case falseString:
165                 lazyGeneration = false;
166                 break;
167             default:
168                 try {
169                     throw new XmlBeanReaderException("XMLReader
                        _Error:_The_value_for_lazy_generation_"
                        + lazyGenString + "'_was_not_recognized."
                        );
170                 } catch (XmlBeanReaderException e) {
171                     e.printStackTrace();
172                     System.exit(1);
173                 }
174             }
175         return lazyGeneration;
176     }
177
178 }

```

### AnnotationsBeanReader

```

1 package com.ci1330.ecci.ucr.ac.cr.readers;
2
3 import com.ci1330.ecci.ucr.ac.cr.annotations.*;
4 import com.ci1330.ecci.ucr.ac.cr.bean.AutowiredEnum;
5 import com.ci1330.ecci.ucr.ac.cr.bean.Stereotype;
6 import com.ci1330.ecci.ucr.ac.cr.exception.
    AnnotationsBeanReaderException;
7 import com.ci1330.ecci.ucr.ac.cr.factory.BeanCreator;
8 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
9
10 import java.lang.annotation.Annotation;
11 import java.lang.reflect.Field;
12 import java.lang.reflect.Method;
13 import java.lang.reflect.Constructor;
14
15 /**
16  * @author Elias Calderon, Josue Leon, Kevin Leon
17  * Date: 13/09/2017
18  *

```

```

19  * The reader is given a String, and then tries to map it with a
    class
20  * and extract the metadata for the BeanCreator
21  */
22  public class AnnotationsBeanReader extends BeanReader {
23
24      private String currID; //The bean ID
25      private Stereotype stereotype; //The type of stereotype
26
27
28      /** Constructor, receives the {@link BeanFactory} that
        created him
29      * @param beanFactory the father {@link BeanFactory}
30      */
31      public AnnotationsBeanReader(BeanFactory beanFactory) {
32          super(beanFactory);
33      }
34
35      /**
36      * Constructor, receives the {@link BeanCreator} that it'll
        use
37      * @param beanCreator the {@link BeanCreator} to use
38      */
39      AnnotationsBeanReader(BeanCreator beanCreator) {
40          super(beanCreator);
41      }
42
43      /**
44      * Receives the name of a class and creates the
        corresponding Class object,
45      * and calls a method to read it
46      * @param inputName the name of the class
47      */
48      @Override
49      public void readBeans(String inputName) {
50          Class reflectClass = null;
51
52          try {
53              reflectClass = Class.forName(inputName);
54          } catch (ClassNotFoundException e) {
55              e.printStackTrace();
56              System.exit(1);
57          }
58
59          //Check if there is more than a stereotype
60          if (reflectClass.isAnnotationPresent(Bea.class) && !
              reflectClass.isAnnotationPresent(Repository.class)
61              && !reflectClass.isAnnotationPresent(Service.class) && !reflectClass.isAnnotationPresent(

```



```

        Controller.class)) {
62
63         this.stereotype = Stereotype.Bean;
64
65     } else if (!reflectClass.isAnnotationPresent(Been.class)
66         && reflectClass.isAnnotationPresent(Repository.class)
67         && !reflectClass.isAnnotationPresent(Service.
68             class) && !reflectClass.isAnnotationPresent(
69             Controller.class)) {
70
71         this.stereotype = Stereotype.Repository;
72
73     } else if (!reflectClass.isAnnotationPresent(Been.class)
74         && !reflectClass.isAnnotationPresent(Repository.class)
75         && reflectClass.isAnnotationPresent(Service.
76             class) && !reflectClass.isAnnotationPresent(
77             Controller.class)){
78
79         this.stereotype = Stereotype.Service;
80
81     } else if (!reflectClass.isAnnotationPresent(Been.class)
82         && !reflectClass.isAnnotationPresent(Repository.class)
83         && !reflectClass.isAnnotationPresent(Service.
84             class) && reflectClass.isAnnotationPresent(
85             Controller.class)){
86
87         this.stereotype = Stereotype.Controller;
88
89     } else {
90         try {
91             throw new AnnotationsBeanReaderException("
92                 Annotations_Reader:_The_'class'_ +
93                 inputName + "_does_not_have_the_Stereotype_
94                 Annotation_or_has_more_than_a_Stereotype");
95         } catch (AnnotationsBeanReaderException e) {
96             e.printStackTrace();
97             System.exit(1);
98         }
99     }
100
101     //Now read the rest of the metadata
102     this.readBeanProperties(reflectClass);
103     this.readBeanConstructor(reflectClass);
104     this.readBeanSetter(reflectClass);
105     this.beanCreator.addBeanToContainer();
106 }

```

```

96  /**
97   * Receives the class and starts to read the annotations, if
    any.
98   * @param beanClass the class to search
99   */
100 private void readBeanProperties (Class beanClass) {
101
102     //Get the bean ID depending of the stereotype
103     switch (this.stereotype){
104         case Bean:
105             Bean bean = (Bean) beanClass.
                getDeclaredAnnotation(Been.class);
106             this.currID = bean.value();
107             break;
108         case Controller:
109             Controller controller = (Controller) beanClass.
                getDeclaredAnnotation( Controller.class);
110             this.currID = controller.value();
111             break;
112         case Repository:
113             Repository repository = (Repository) beanClass.
                getDeclaredAnnotation(Repository.class);
114             this.currID = repository.value();
115             break;
116         default:
117             Service service = (Service) beanClass.
                getDeclaredAnnotation( Service.class);
118             this.currID = service.value();
119             break;
120     }
121
122
123     //The default scope is singleton
124     com.ci1330.ecci.ucr.ac.cr.bean.Scope scope = com.ci1330.
        ecci.ucr.ac.cr.bean.Scope.Singleton;
125     if(beanClass.isAnnotationPresent(Scope.class)){
126         Scope scopeAnnotation = (Scope)(beanClass.
            getAnnotation(Scope.class));
127         scope = super.determineScope(scopeAnnotation.value()
            .toLowerCase());
128     }
129
130     //The default class-autowire is none
131     AutowireEnum autowire = AutowireEnum.none;
132     if(beanClass.isAnnotationPresent(ClassAutowire.class)){
133         ClassAutowire autowireAnnotation = (ClassAutowire)(
            beanClass.getAnnotation(ClassAutowire.class));
134         autowire = super.determineClass_Autowire(
            autowireAnnotation.value().toLowerCase());

```

```

135     }
136
137     //The default lazyGen is false
138     boolean lazyGeneration = false;
139     if(beanClass.isAnnotationPresent(Lazy.class)){
140         lazyGeneration = true;
141     }
142
143     //Searches for init and destroy
144     String initMethod = null;
145     String destroyMethod = null;
146
147     //Travel by every method
148     for(Method method : beanClass.getDeclaredMethods()){
149
150         //If there is @Init
151         if(method.isAnnotationPresent(Init.class)){
152             if (initMethod == null) {
153                 initMethod = method.getName();
154             } else {
155                 try {
156                     throw new AnnotationsBeanReaderException
157                         ("AnnotationsReader_Error: The '@Init'
158                          'in the 'bean ' + this.currID + "
159                          was not recognized. It has more than
160                          a definition");
161                 } catch (AnnotationsBeanReaderException e) {
162                     e.printStackTrace();
163                     System.exit(1);
164                 }
165             }
166         }
167
168         //If there is @Destroy
169         if(method.isAnnotationPresent(Destroy.class)){
170             if(destroyMethod == null) {
171                 destroyMethod = method.getName();
172             } else {
173                 try {
174                     throw new AnnotationsBeanReaderException
175                         ("AnnotationsReader_Error: The '@Destroy'
176                          'in the 'bean ' + this.
177                          currID + "
178                          was not recognized. It has
179                          more than a definition");
180                 } catch (AnnotationsBeanReaderException e) {
181                     e.printStackTrace();
182                     System.exit(1);
183                 }
184             }
185         }
186     }
187 }

```

```

176         }
177
178     }
179     this.beanCreator.createBean(this.currID, beanClass.
        getName(), scope, initMethod, destroyMethod,
        lazyGeneration, autowire);
180 }
181
182 /**
183  * Reads the annotations of a constructor, if any.
184  * @param beanClass the class to search
185  */
186 private void readBeanConstructor (Class beanClass) {
187     boolean constructorAlreadyMatched = false;
188     for (Constructor constructor : beanClass.
        getDeclaredConstructors()) {
189
190         //If there is @Constructor
191         if (constructor.isAnnotationPresent(com.ci1330.ecci.
            ucr.ac.cr.annotations.Constructor.class)) {
192
193             if (constructorAlreadyMatched) {
194                 try {
195                     throw new AnnotationsBeanReaderException
                        ("AnnotationsReaderError:The '
                            @Constructor' in the 'bean' " + this.
                            currID + " was not recognized.The
                            constructor has more than a
                            definition");
196                 } catch (AnnotationsBeanReaderException e) {
197                     e.printStackTrace();
198                     System.exit(1);
199                 }
200             }
201             constructorAlreadyMatched = true;
202
203             //If there is @Parameter
204             if (constructor.isAnnotationPresent(Parameter.
                class)) {
205
206                 //Travel by every annotation in the
                constructor
207                 for (Annotation annotation : constructor.
                    getDeclaredAnnotations()) {
208                     if (annotation.annotationType() ==
                        Parameter.class) {
209
210                         String paramType = ((Parameter)
                            annotation).type();

```

```

211         if (paramType.equals("")) {
212             paramType = null;
213         }
214
215         int index = ((Parameter) annotation)
216             .index();
217
218         String value = ((Parameter)
219             annotation).value();
220         if (value.equals("")) {
221             value = null;
222         }
223
224         String beanRef = ((Parameter)
225             annotation).ref();
226         if (beanRef.equals("")) {
227             beanRef = null;
228         }
229
230         final boolean refTypeCombination =
231             paramType != null & beanRef !=
232             null && value == null;
233         final boolean valueTypeCombination =
234             paramType != null && value !=
235             null && beanRef == null;
236
237         //Check if the combinations are
238         valid
239         if ( refTypeCombination ||
240             valueTypeCombination ) {
241             this.beanCreator.
242                 registerConstructorParameter(
243                     paramType, index, value,
244                     beanRef, AutowireEnum.none);
245         } else {
246             try {
247                 throw new
248                     AnnotationsBeanReaderException
249                     ("AnnotationsReader_
250                     error: The '@Parameter'
251                     was not recognized in the
252                     'bean'" + this.currID +
253                     ". It has an illegal
254                     value, ref and type
255                     combination.");
256             } catch (
257                 AnnotationsBeanReaderException
258                 e) {
259                 e.printStackTrace();

```

```

238         System.exit(1);
239     }
240 }
241
242 }
243 }
244
245 }
246 } // if (constructor.isAnnotationPresent(com.cil330.
    ecci.ucr.ac.cr.annotations.Constructor.class))
247 else if (constructor.isAnnotationPresent(
    AtomicAutowire.class)){
248
249     if (constructorAlreadyMatched) {
250         try {
251             throw new AnnotationsBeanReaderException
                ("AnnotationsReaderError:The'
                @AtomicAutowire'in the'bean'"+
                this.currID + "was not recognized.The
                constructor has more than a
                definition");
252         } catch (AnnotationsBeanReaderException e) {
253             e.printStackTrace();
254             System.exit(1);
255         }
256     }
257     constructorAlreadyMatched = true;
258
259     this.beanCreator.explicitConstructorDefinition(
        constructor);
260
261 }
262 }
263 }
264
265 /**
266  * Reads the annotations of a specific method, if any.
267  * @param beanClass the class to search
268  */
269 private void readBeanSetter (Class beanClass) {
270     //Travel by every field
271     for (Field field : beanClass.getDeclaredFields()) {
272
273         //If there is @Attribute
274         if (field.isAnnotationPresent(Attribute.class)) {
275
276             String value = field.getAnnotation(Attribute.
                class).value();
277             if (value.equals("")) {

```

```

278         value = null;
279     }
280
281     String ref = field.getAnnotation(Attribute.class)
282         .ref();
283     if (ref.equals("")) {
284         ref = null;
285     }
286
287     //Check if there is value or ref
288     if ((ref == null && value != null) || (ref !=
289         null && value == null)) {
290         this.beanCreator.registerSetter(field.
291             getName(), value, ref, AutowireEnum.none)
292             ;
293     } else {
294         try {
295             throw new AnnotationsBeanReaderException
296                 ("AnnotationsReaderError:The '
297                     @Attribute' was not recognized in the
298                     'bean'"+ this.currID + ". It has an
299                     illegal combination of value and ref
300                     .");
301         } catch (AnnotationsBeanReaderException e) {
302             e.printStackTrace();
303             System.exit(1);
304         }
305     }
306
307     //The reader will only recognize autowire if an
308     //Attribute annotation is not present
309     else if (field.isAnnotationPresent(AtomicAutowire.
310         class)) {
311
312         //It is assumed to be the special annotation
313         //autowiring
314         this.beanCreator.registerSetter(field.getName(),
315             null, null, AutowireEnum.annotation);
316     }
317 }

```

### XmlBeanReader

```

1 package com.ci1330.ecci.ucr.ac.cr.readers;
2
3 import javax.xml.parsers.DocumentBuilderFactory;
4 import javax.xml.parsers.DocumentBuilder;

```

```

5 import javax.xml.parsers.ParserConfigurationException;
6
7 import com.ci1330.ecci.ucr.ac.cr.bean.AutowireEnum;
8 import com.ci1330.ecci.ucr.ac.cr.bean.Scope;
9 import com.ci1330.ecci.ucr.ac.cr.exception.
    XmlBeanReaderException;
10 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
11 import org.w3c.dom.Document;
12 import org.w3c.dom.NodeList;
13 import org.w3c.dom.Node;
14 import org.w3c.dom.Element;
15 import org.xml.sax.SAXException;
16
17 import java.io.File;
18 import java.io.IOException;
19
20 /**
21  * @author Elias Calderon, Josue Leon, Kevin Leon
22  * Date: 13/09/2017
23  * The reader is given a String, and then tries to map it with a
    XML file
24  * and extract the metadata for the BeanCreator
25  */
26 public class XmlBeanReader extends BeanReader {
27
28     /**
29      * The annotations reader is used if in the xml file, a read
        annotations
30      * statement is found.
31      */
32     private String defaultInitMethod; //The init method
33     private String defaultDestroyMethod; //The destroy method
34     private String currID; //The bean ID
35
36     //Init and destroy method tags
37     private final String initTag = "init";
38     private final String destroyTag = "destroy";
39
40     //Bean properties tags
41     private final String idTag = "id";
42     private final String classTag = "class";
43     private final String scopeTag = "scope";
44     private final String autowireTag = "autowire";
45     private final String lazyGenerationTag = "lazy-generation";
46
47     //Constructor tags
48     private final String constructorTag = "constructor";
49     private final String paramTag = "param";
50     private final String typeTag = "type";

```



```

51     private final String indexTag = "index";
52
53     //Constructor tags
54     private final String nameTag = "name";
55
56     //Constructor and Attribute tags
57     private final String valueTag = "value";
58     private final String beanRefTag = "ref";
59     private final String atomic_autowireTag = "atomic-autowire";
60
61
62     /**
63      * Constructor, receives the {@link BeanFactory} that
64      * created him
65      * @param beanFactory the father {@link BeanFactory}
66      */
67     public XmlBeanReader(BeanFactory beanFactory){
68         super(beanFactory);
69     }
70
71     /**
72      * Receives the name of the XML and creates the root
73      * @param inputName the name of the XML file
74      */
75     @Override
76     public void readBeans(String inputName) {
77
78         final String beanTag = "bean";
79
80         File fXmlFile = new File(inputName);
81         DocumentBuilderFactory dbFactory =
82             DocumentBuilderFactory.newInstance();
83
84         DocumentBuilder dBuilder = null;
85         try {
86             dBuilder = dbFactory.newDocumentBuilder();
87         } catch (ParserConfigurationException e) {
88             e.printStackTrace();
89             System.exit(1);
90         }
91
92         Document doc = null;
93         try {
94             doc = dBuilder.parse(fXmlFile);
95         } catch (SAXException | IOException e) {
96             e.printStackTrace();
97             System.exit(1);
98         }

```

```

108     doc.getDocumentElement().normalize();
109     Element rootElement = this.readRoot(doc);
110
111     //Get all the beans in the XML
112     NodeList nodeList = rootElement.getElementsByTagName(
113         beanTag);
114
115     //Travel by every bean
116     for (int index = 0; index < nodeList.getLength(); index
117         ++){
118
119         Node node = nodeList.item(index);
120
121         //Check if it is an Element
122         if ((node.getNodeType() == Node.ELEMENT_NODE)) {
123
124             Element beanElement = (Element) node;
125             this.readBeanProperties(beanElement);
126             this.readBeanConstructor(beanElement);
127             this.readBeanAttributes(beanElement);
128             super.beanCreator.addBeanToContainer();
129
130             } else {
131                 try {
132                     throw new XmlBeanReaderException("XmlReader
133                         _Error:_A_'bean'_was_not_recognized.");
134                 } catch (XmlBeanReaderException e) {
135                     e.printStackTrace();
136                     System.exit(1);
137                 }
138             }
139         }
140     }
141     this.readAnnotationsStatement(rootElement);
142 }
143
144 /**
145  * Starts reading the root of the xml
146  * @param xmlRootFile the root of the file
147  * @return rootElement
148  */
149 private Element readRoot(Document xmlRootFile) {
150     final String beansTag = "beans";
151
152     Element rootElement = xmlRootFile.getDocumentElement();
153     //Check if there is a correct root
154     if (rootElement.getTagName().equals(beansTag)) {
155         //Check if there is an init property in the root
156         if (rootElement.hasAttribute(this.initTag)) {
157             //Check if there is an init method in the root

```

```

144         if (!rootElement.getAttribute(this.initTag).
145             equals("")) {
146             this.defaultInitMethod = rootElement.
147                 getAttribute(this.initTag);
148         }
149         //Check if there is a destroy property in the root
150         if (rootElement.hasAttribute(this.destroyTag)) {
151             //Check if there is a destroy method
152             if (!rootElement.getAttribute(this.destroyTag).
153                 equals("")) {
154                 this.defaultDestroyMethod = rootElement.
155                     getAttribute(this.destroyTag);
156             }
157         }
158     } else {
159         try {
160             throw new XmlBeanReaderException("XmlReader_
161                 Error:_The_root_of_the_XML_document_is_" +
162                 rootElement.getTagName() + "_instead_of_" +
163                 beans'.");
164         } catch (XmlBeanReaderException e) {
165             e.printStackTrace();
166             System.exit(1);
167         }
168     }
169     return rootElement;
170 }
171
172 /**
173  * Reads the properties of a bean from the bean xml node,
174  * any invalid combination or value, throws an exception
175  * and exits the program.
176  * @param beanElement the XML element of a bean
177  */
178 private void readBeanProperties(Element beanElement) {
179
180     final String defaultScope = "singleton";
181     final String defaultAutowire = "none";
182     final String defaultLazyGen = "false";
183
184     //Check if the bean has both ID and class
185     if (beanElement.hasAttribute(this.idTag) && beanElement.
186         hasAttribute(this.classTag)) {
187
188         this.currID = beanElement.getAttribute(this.idTag);
189         String className = beanElement.getAttribute(this.
190             classTag);
191         String initMethod = null;

```

```

183 String destroyMethod = null;
184
185 //Check if there is an init property in the current
    bean
186 if (beanElement.getAttribute(this.initTag)) {
187     //Check if there is an init method
188     if (!beanElement.getAttribute(this.initTag).
        equals("")) {
189         initMethod = beanElement.getAttribute(this.
            initTag);
190     }
191 } else {
192     //If not, put the init method as the default one
193     initMethod = this.defaultInitMethod;
194 }
195
196 //Check if there is an init property in the current
    bean
197 if (beanElement.getAttribute(this.destroyTag)) {
198     //Check if there is a destroy method
199     if (!beanElement.getAttribute(this.destroyTag).
        equals("")) {
200         destroyMethod = beanElement.getAttribute(
            this.destroyTag);
201     }
202 } else {
203     //If not, put the destroy method as the default
        one
204     destroyMethod = this.defaultDestroyMethod;
205 }
206
207 //

```

---

```

208
209 //Check the scope value
210 String scopeString = beanElement.getAttribute(this.
    scopeTag).toLowerCase();
211 if (scopeString.equals("")) {
212     scopeString = defaultScope;
213 }
214
215 //Check the autowire value
216 String autowireString = beanElement.getAttribute(
    this.autowireTag).toLowerCase();
217 if (autowireString.equals("")) {
218     autowireString = defaultAutowire;
219 }
220

```

```

221         //Get the lazy-generation
222         String lazyGenString = beanElement.getAttribute(this
                .lazyGenerationTag).toLowerCase();
223         if (lazyGenString.equals("")) {
224             lazyGenString = defaultLazyGen;
225         }
226
227         //

```

---

```

228
229         AutowireEnum autowire = super.
                determineClass_Autowire(autowireString);
230         Scope scope = super.determineScope(scopeString);
231         boolean lazyGeneration = super.determineLazyGen(
                lazyGenString);
232
233         this.beanCreator.createBean(this.currID, className,
                scope, initMethod, destroyMethod, lazyGeneration,
                autowire);
234
235     } //if (beanElement.hasAttribute("id") && beanElement.
        hasAttribute("class"))
236     else {
237         try {
238             throw new XmlBeanReaderException("XmlReader_
                    error:_ID_and_Class_value_for_all_tags_must_
                    be_entered.");
239         } catch (XmlBeanReaderException e) {
240             e.printStackTrace();
241             System.exit(1);
242         }
243     }
244 }
245
246 /**
247  * Reads the constructor of a bean from the constructor xml
        node, any invalid combination or value, throws an
        exception
248  * and exits the program.
249  *
250  * @param beanElement the XML Element for a bean
251  */
252 private void readBeanConstructor(Element beanElement) {
253
254     //Get all the constructor in the current bean
255     NodeList constructorList = beanElement.
        getElementsByTagName(this.constructorTag);
256

```

```

257 //Check if there is more than a constructor definition
258 if (constructorList.getLength() > 1) {
259     try {
260         throw new XmlBeanReaderException("XmlReader_
                error:_Multiple_constructors_tags_in_bean_" +
                this.currID + ".");
261     } catch (XmlBeanReaderException e) {
262         e.printStackTrace();
263         System.exit(1);
264     }
265 } else if (constructorList.getLength() > 0) {
266
267     Element constructorElement = (Element)
                constructorList.item(0);
268     NodeList constructorArgs = constructorElement.
                getElementsByTagName(this.paramTag);
269
270     //Travel every param
271     for (int index = 0; index < constructorArgs.
                getLength(); index++) {
272         Node parameterNode = constructorArgs.item(index)
                ;
273
274         //Check if it is an Element so we can cast it
275         if (parameterNode.getNodeType() == Node.
                ELEMENT_NODE) {
276             Element parameterElement = (Element)
                parameterNode;
277
278             //Combination of only type and atomic-
                autowire tag
279             final boolean autowireByTypeCombination =
                parameterElement.hasAttribute(this.
                typeTag) && parameterElement.
                hasAttribute(this.atomic-autowireTag)
280                 && !(parameterElement.hasAttribute(
                this.beanRefTag)) && !(
                parameterElement.hasAttribute(
                this.valueTag));
281
282             //Combination of only beanRef and atomic-
                autowire tag
283             final boolean autowireByNameCombination =
                parameterElement.hasAttribute(this.
                beanRefTag) && parameterElement.
                hasAttribute(this.atomic-autowireTag)
284                 && !(parameterElement.hasAttribute(
                this.typeTag)) && !(
                parameterElement.hasAttribute(

```

```

285         this.valueTag));
286
287         //Combination of only type and beanRef
288         final boolean typeRefCombination =
            parameterElement.hasAttribute(this.
                typeTag) && parameterElement.hasAttribute
                (this.beanRefTag)
                && !(
                    parameterElement
                    .
                    hasAttribute
                    (this
                    .
                    valueTag
                    )) &&
                    !(
                        parameterElement
                        .
                        hasAttribute
                        (this
                        .
                        atomic_awaitTag
                        ));
289
290         //Combination of only type and value
291         final boolean typeValueCombination =
            parameterElement.hasAttribute(this.
                typeTag) && parameterElement.
                hasAttribute(this.valueTag)
                && !(parameterElement.hasAttribute(
                    this.beanRefTag)) && !(
                    parameterElement.hasAttribute(
                    this.atomic_awaitTag));
292
293         //Check if any combination matches
294         if (awaitByTypeCombination ||
295             awaitByNameCombination ||
                typeRefCombination ||
                typeValueCombination ) {
296
297             int argIndex = -1;
298             try {
299                 //Tries to get the index if it
                    exists
300                 if ( parameterElement.hasAttribute(
                    this.indexTag)) {
301                     if (!parameterElement.
                        getAttribute(this.indexTag).
                        equals("")) {

```

```

302         argIndex = Integer.parseInt(
            parameterElement.
            getAttribute(this.
            indexTag));
303     } else {
304         throw new
            XmlBeanReaderException("
            XML_Reader_Error: An
            invalid_value_was_entered
            in_index_tag.");
305     }
306 }
307
308 } catch (NumberFormatException |
    XmlBeanReaderException e) {
309     e.printStackTrace();
310     System.exit(1);
311 }
312
313 //If nothing was specified put it to
    null
314 String type = parameterElement.
    getAttribute(this.typeTag);
315 if (type.equals("")) {
316     type = null;
317 }
318
319 //If nothing was specified put it to
    null
320 String value = parameterElement.
    getAttribute(this.valueTag);
321 if (value.equals("")) {
322     value = null;
323 }
324
325 //If nothing was specified put it to
    null
326 String ref = parameterElement.
    getAttribute(this.beanRefTag);
327 if (ref.equals("")) {
328     ref = null;
329 }
330
331 //If nothing was specified put it to
    none
332 String atomic_autowireString =
    parameterElement.getAttribute(this.
    atomic_autowireTag).toLowerCase();
333 if (atomic_autowireString.equals("")) {

```



```

334         atomic_autowireString = "none";
335     }
336     AutowireEnum atomic_autowire = super.
        determineAtomic_Autowire(
            atomic_autowireString);
337
338     this.beanCreator.
        registerConstructorParameter(type,
            argIndex, value, ref, atomic_autowire
        );
339
340     } else {
341         try {
342             throw new XmlBeanReaderException("
                Xml_Reader_error:_A_'param'_has_
                an_invalid_tag_combination,_in_
                bean_" + this.currID + ".");
343         } catch (XmlBeanReaderException e) {
344             e.printStackTrace();
345             System.exit(1);
346         }
347     }
348     } else {
349         try {
350             throw new XmlBeanReaderException("Xml_
                Reader_error:_A_'param'_was_not_
                recognized_in_the_'bean'__" + this.
                currID + ".");
351         } catch (XmlBeanReaderException e) {
352             e.printStackTrace();
353             System.exit(1);
354         }
355     }
356 }
357 }
358 }
359 }
360 }
361
362 /**
363  * Reads an attribute of a bean from the attribute xml node,
364  * any invalid combination or value, throws an exception
365  * and exits the program.
366  *
367  * @param beanElement the XML Element for a bean.
368  */
369 private void readBeanAttributes(Element beanElement) {
370     NodeList attributeList = beanElement.

```

```

371         getElementsByTagName("attribute");
372     for (int index = 0; index < attributeList.getLength();
373         index++) {
374         Node attributeNode = attributeList.item(index);
375
376         //Check if it is an Element
377         if (attributeNode.getNodeType() == Node.ELEMENTNODE
378             ) {
379
380             Element attributeElement = (Element)
381                 attributeNode;
382
383             //Combination of only name and Value tag
384             final boolean nameValueCombination =
385                 attributeElement.hasAttribute(this.nameTag)
386                 && attributeElement.hasAttribute(this.
387                     valueTag)
388                 && !(attributeElement.hasAttribute(this.
389                     beanRefTag));
390
391             //Combination of only name and Ref tag
392             final boolean nameRefCombination =
393                 attributeElement.hasAttribute(this.nameTag)
394                 && attributeElement.hasAttribute(this.
395                     beanRefTag)
396                 && !(attributeElement.hasAttribute(this.
397                     valueTag));
398
399             //Combination of only name and autowire tag
400             final boolean atomicAutowireCombination =
401                 attributeElement.hasAttribute(this.nameTag)
402                 && attributeElement.hasAttribute(this.
403                     atomic_autowireTag)
404                 && !(attributeElement.hasAttribute(this.
405                     beanRefTag)) && !(attributeElement.
406                     hasAttribute(this.valueTag));
407
408             //Check if any combination matches
409             if ( nameValueCombination || nameRefCombination
410                 || atomicAutowireCombination ) {
411
412                 //If the name is empty, throw an exception
413                 String name = attributeElement.getAttribute(
414                     this.nameTag);
415                 if (name.equals("")) {
416                     try {
417                         throw new XmlBeanReaderException("
418                             Xml_Reader_error: _An_ 'attribute'_
419                             has_a_null_name_in_bean_"+this.

```

```

currID + "."");
399     } catch (XmlBeanReaderException e) {
400         e.printStackTrace();
401         System.exit(1);
402     }
403 }
404
405 //If the value is empty put it to null
406 String value = attributeElement.getAttribute
    (this.valueTag);
407 if (value.equals("")) {
408     value = null;
409 }
410
411 //If the value is empty put it to null
412 String beanRef = attributeElement.
    getAttribute(this.beanRefTag);
413 if (beanRef.equals("")) {
414     beanRef = null;
415 }
416
417 //If nothing was specified, put it to none
418 String atomic_awareString =
    attributeElement.getAttribute(this.
        atomic_awareTag).toLowerCase();
419 if (atomic_awareString.equals("")) {
420     atomic_awareString = "none";
421 }
422 AutowireEnum atomic_aware = super.
    determineAtomic_Aware(
        atomic_awareString);
423
424 this.beanCreator.registerSetter(name, value,
    beanRef, atomic_aware);
425
426 } else {
427     try {
428         throw new XmlBeanReaderException("Xml_
            Reader_error:_The_'attribute'_must_
            have_'name'_and_'value'_or_'name'_and_
            _'ref'_in_bean_"+this.currID + ".");
429     } catch (XmlBeanReaderException e) {
430         e.printStackTrace();
431         System.exit(1);
432     }
433 }
434
435 } else {
436     try {

```

```

437         throw new XmlBeanReaderException("XmlReader
           _error:_An_'attribute'_was_not_recognized
           _in_the_'bean'_ " + this.currID + ".");
438     } catch (XmlBeanReaderException e) {
439         e.printStackTrace();
440         System.exit(1);
441     }
442 }
443
444 }
445
446 }
447
448 /**
449  * The method tells the annotationsBeanReader to read a
    specific class. If it has more than one tag, exits
    abnormally.
450  *
451  * @param beanElement the XML Element for a bean
452  */
453 private void readAnnotationsStatement(Element beanElement) {
454
455     final String annotationClassesTag = "annotationsClasses"
456     ;
457     final String pathTag = "path";
458
459     //Get all the annotationsClasses in the root
460     NodeList annotationsList = beanElement.
461         getElementsByTagName(annotationClassesTag);
462
463     //Check if there is more than a annotationsClasses
464     definition
465     if (annotationsList.getLength() > 1) {
466         try {
467             throw new XmlBeanReaderException("XmlReader_
468                 error:_'annotationsClasses'_has_more_than_one_
469                 definition.");
470         } catch (XmlBeanReaderException e) {
471             e.printStackTrace();
472             System.exit(1);
473         }
474     } else if(annotationsList.getLength() > 0){
475
476         //Create a new Annotations reader with the same
477         creator of this factory.
478         AnnotationsBeanReader annotationsBeanReader = new
479             AnnotationsBeanReader(super.beanCreator);
480         Element annotationsElement = (Element)
481             annotationsList.item(0);

```

```

474         NodeList classList = annotationsElement.
            getElementsByTagName( this.classTag );
475
476         //Travel by every class
477         for( int index = 0; index < classList.getLength(); ++
            index ){
478             Node classNode = classList.item(index);
479
480             //Check if it is an Element
481             if ( classNode.getNodeType() == Node.ELEMENTNODE
                ) {
482                 Element classElement = (Element) classNode;
483
484                 //If the annotation has a path
485                 if (!( classElement.getAttribute(pathTag).
                    equals("") )) {
486                     annotationsBeanReader.readBeans(
                        classElement.getAttribute(pathTag));
487
488                 } else {
489
490                     try {
491                         throw new XmlBeanReaderException("
                            XmlReader_error: A class in '
                            annotationClasses' doesn't have a
                            'path'");
492                     } catch (XmlBeanReaderException e) {
493                         e.printStackTrace();
494                         System.exit(1);
495                     }
496
497                 }
498             } else {
499                 try {
500                     throw new XmlBeanReaderException("Xml_
                            Reader_error: A 'class' in '
                            annotationClasses' was not recognized
                            ");
501                 } catch (XmlBeanReaderException e) {
502                     e.printStackTrace();
503                     System.exit(1);
504                 }
505             }
506         }
507     }
508 }
509
510 }

```

## AtomicAutowire

```
1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import com.ci1330.ecci.ucr.ac.cr.bean.AutowireEnum;
4
5 import java.lang.annotation.ElementType;
6 import java.lang.annotation.Retention;
7 import java.lang.annotation.RetentionPolicy;
8 import java.lang.annotation.Target;
9
10 /**
11  * @author Elias Calderon, Josue Leon, Kevin Leon
12  * Date: 17/09/2017
13  */
14 @Retention(RetentionPolicy.RUNTIME)
15 @Target({ElementType.FIELD, ElementType.CONSTRUCTOR})
16 public @interface AtomicAutowire {
17
18 }
```

## Attribute

```
1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 17/09/2017
11  */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.FIELD)
14 public @interface Attribute {
15     String value() default "";
16     String ref() default "";
17 }
```

## Bean

```
1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
```

```

9  * @author Elias Calderon , Josue Leon , Kevin Leon
10 * Date: 17/09/2017
11 */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.TYPE)
14 public @interface Bean {
15     String value();
16 }

```

### ClassAutowire

```

1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon , Josue Leon , Kevin Leon
10 * Date: 17/09/2017
11 */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.TYPE)
14 public @interface ClassAutowire {
15     String value() default "byname";
16 }

```

### Constructor

```

1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3
4 import java.lang.annotation.ElementType;
5 import java.lang.annotation.Retention;
6 import java.lang.annotation.RetentionPolicy;
7 import java.lang.annotation.Target;
8
9 /**
10 * @author Elias Calderon , Josue Leon , Kevin Leon
11 * Date: 17/09/2017
12 */
13 @Retention(RetentionPolicy.RUNTIME)
14 @Target(ElementType.CONSTRUCTOR)
15 public @interface Constructor {
16 }

```

### Controller

```

1 package com.ci1330.ecci.ucr.ac.cr.annotations;

```

```

2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 17/09/2017
11  */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.TYPE)
14 public @interface Controller {
15     String value();
16 }

```

### Destroy

```

1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 17/09/2017
11  */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.METHOD)
14 public @interface Destroy {
15 }

```

### Init

```

1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 17/09/2017
11  */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.METHOD)
14 public @interface Init {

```



```
15 }
```

### Lazy

```
1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 17/09/2017
11  */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.TYPE)
14 public @interface Lazy {
15 }
```

### Parameter

```
1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 17/09/2017
11  */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.CONSTRUCTOR)
14 public @interface Parameter {
15     String type();
16     int index() default -1;
17     String value() default "";
18     String ref() default "";
19 }
```

### Scope

```
1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
```

```

7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10 * Date: 17/09/2017
11 */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.TYPE)
14 public @interface Scope {
15     String value() default "singleton";
16 }

```

### Service

```

1 package com.ci1330.ecci.ucr.ac.cr.annotations;
2
3 import java.lang.annotation.ElementType;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.RetentionPolicy;
6 import java.lang.annotation.Target;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10 * Date: 17/09/2017
11 */
12 @Retention(RetentionPolicy.RUNTIME)
13 @Target(ElementType.TYPE)
14 public @interface Service {
15     String value();
16 }

```

### AnnotationsFactory

```

1 package com.ci1330.ecci.ucr.ac.cr.factory;
2
3 import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
4 import com.ci1330.ecci.ucr.ac.cr.readers.AnnotationsBeanReader;
5
6 import java.util.HashMap;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10 * Date: 13/09/2017
11 *
12 * AnnotationsFactory class which inherits from BeanFactory
13 * and registers the Annotations classes from which the
14 * configuration
15 * must be read and tells the reader to parse it.
16 */
17 public class AnnotationsFactory extends BeanFactory{

```

```

18     private AnnotationsBeanReader annotationsBeanReader;    //
        Instance of the annotations reader
19
20     /**
21     * Constructor of the class , it initializes the super-class
        attributes and
22     * also the annotations bean reader.
23     */
24     public AnnotationsFactory() {
25         super();
26         annotationsBeanReader = new AnnotationsBeanReader(this);
27     }
28
29     /**
30     * Constructor of the class , it initializes the super-class
        attributes and
31     * also the annotations bean reader. It receives the path of
        a class which
32     * holds annotations configurations for the reader to parse
        it.
33     * @param classConfig the name of the class to use
34     */
35     public AnnotationsFactory(String classConfig) {
36         super();
37         annotationsBeanReader = new AnnotationsBeanReader(this);
38         this.registerConfig(classConfig);
39     }
40
41     /**
42     * Allows the user to register more configurations
43     * classes later , indicating their path.
44     * @param classConfig the name of the class to use
45     */
46     public void registerConfig(String classConfig){
47         annotationsBeanReader.readBeans(classConfig);
48         super.initContainer();
49     }
50
51     /**
52     * Return a bean instance from the super class.
53     * @param id the beanId
54     * @return the bean instance
55     */
56     @Override
57     public Object getBean(String id) {
58         return super.getBean(id);
59     }
60
61     /**

```

```

62     * Adds a bean to the container
63     * @param bean the {@link Bean} class
64     */
65     @Override
66     public void addBean(Bea bean) {
67         super.addBean(bean);
68     }
69
70     /**
71     * Calls the super method for shutDownHook
72     */
73     @Override
74     public void shutDownHook() {
75         super.shutDownHook();
76     }
77
78     //

```

---

```

79     // Standard Setters and Getters section
80     //

```

---

```

81
82     @Override
83     public HashMap<String, Bea> getBeansMap() {
84         return super.getBeansMap();
85     }
86
87     @Override
88     public void setBeansMap(HashMap<String, Bea> beansMap) {
89         super.setBeansMap(beansMap);
90     }
91
92     public AnnotationsBeanReader getAnnotationsBeanReader() {
93         return annotationsBeanReader;
94     }
95
96     public void setAnnotationsBeanReader(AnnotationsBeanReader
97         annotationsBeanReader) {
98         this.annotationsBeanReader = annotationsBeanReader;
99     }

```

### BeanConstructorModule

```

1 package com.ci1330.ecci.ucr.ac.cr.factory;
2
3 import com.ci1330.ecci.ucr.ac.cr.bea.Bea;
4 import com.ci1330.ecci.ucr.ac.cr.bea.BeaParameter;

```

```

5 import com.ci1330.ecci.ucr.ac.cr.exception.
   BeanConstructorConflictException;
6 import com.ci1330.ecci.ucr.ac.cr.exception.
   BeanConstructorNotFoundException;
7
8 import java.lang.reflect.Constructor;
9
10 /**
11  * @author Elias Calderon, Josue Leon, Kevin Leon
12  * Date: 28/09/2017
13  */
14 public class BeanConstructorModule {
15
16     /**
17      * Checks if all parameters have an index assigned. If at
18      * least one doesn't, it returns false.
19      * @return true if all indexes were assigned, false if not
20      */
21     private static boolean checkParametersIndexes(Beans bean){
22         boolean allIndexesAssigned = true;
23         int paramListIndex = 0;
24         BeanParameter beanParameter;
25         while(paramListIndex < bean.getBeanConstructor().
26             getBeanParameterList().size()
27             && allIndexesAssigned){
28             beanParameter = bean.getBeanConstructor().
29                 getBeanParameterList().get(paramListIndex);
30             if(beanParameter.getIndex() == -1){
31                 allIndexesAssigned = false;
32             }
33             paramListIndex++;
34         }
35         return allIndexesAssigned;
36     }
37
38     /**
39      * Searches for the Class types of each parameter in the
40      * bean's constructor.
41      * @param bean the bean to use
42      * @return an array containing the Class types of each
43      * parameter
44      */
45     private static Class[] obtainParametersClassArray(Beans bean)
46     {
47         String parameterClass = null;
48         String beanParameterType;
49         Class param = null;
50         Class[] parametersClassArray = new Class[bean.
51             getBeanConstructor().getBeanParameterList().size()];

```

```

45     for (BeanParameter p : bean.getBeanConstructor().
46         getBeanParameterList()) {
47         beanParameterType = p.getExplicitTypeName();
48         if (beanParameterType == null) {
49             beanParameterType = p.getBeanFactory().findBean(
50                 p.getBeanRef()).getClass().toString();
51         }
52         switch (beanParameterType) {
53             case "int":
54                 param = int.class;
55                 break;
56             case "byte":
57                 param = byte.class;
58                 break;
59             case "short":
60                 param = short.class;
61                 break;
62             case "long":
63                 param = long.class;
64                 break;
65             case "float":
66                 param = float.class;
67                 break;
68             case "double":
69                 param = double.class;
70                 break;
71             case "boolean":
72                 param = boolean.class;
73                 break;
74             case "char":
75                 param = char.class;
76                 break;
77             default:
78                 parameterClass = p.getExplicitTypeName();
79                 try {
80                     param = Class.forName(parameterClass);
81                 } catch (ClassNotFoundException e) {
82                     e.printStackTrace();
83                 }
84                 break;
85         }
86         parametersClassArray[p.getIndex()] = param;
87     }
88     return parametersClassArray;
89 }
90 /**
91  * Compares the bean's constructor parameter type with the

```

```

    bean's class constructor parameter
92  * type. If they match, it assigns the respective index to
    the bean's constructor parameter.
93  * The switch is needed for primitive types checking and
    casting.
94  * @param beanParameter bean's constructor parameter
95  * @param beanClassConstructorParameter bean's class
    constructor parameter
96  * @param paramIndex current index of the bean's class
    constructor parameter
97  * @return True if parameters matched
98  */
99  private static boolean setBeanParameterIndex(BeanParameter
    beanParameter, Class beanClassConstructorParameter, int
    paramIndex){
100     boolean parametersMatched = false;
101     String beanParameterType = beanParameter.
        getExplicitTypeName();
102     if(beanParameterType == null){
103         beanParameterType = beanParameter.getBeanFactory().
            findBean(beanParameter.getBeanRef()).getClass().
            toString();
104     }
105     switch (beanParameterType) {
106         case "int":
107
108             if (beanClassConstructorParameter.toString().
                equals("int")) {
109                 parametersMatched = true;
110                 beanParameter.setIndex(paramIndex);
111             }
112             break;
113         case "java.lang.Integer":
114
115             if (beanClassConstructorParameter.toString().
                equals("int")) {
116                 parametersMatched = true;
117                 beanParameter.setIndex(paramIndex);
118             }
119             break;
120         case "byte":
121             if (beanClassConstructorParameter.toString().
                equals("byte")) {
122                 parametersMatched = true;
123                 beanParameter.setIndex(paramIndex);
124             }
125             break;
126         case "java.lang.Byte":
127             if (beanClassConstructorParameter.toString().

```

```

128         equals("byte")) {
129             parametersMatched = true;
130             beanParameter.setIndex(paramIndex);
131         }
132         break;
133     case "short":
134         if (beanClassConstructorParameter.toString().
135             equals("short")) {
136             parametersMatched = true;
137             beanParameter.setIndex(paramIndex);
138         }
139         break;
140     case "java.lang.Short":
141         if (beanClassConstructorParameter.toString().
142             equals("short")) {
143             parametersMatched = true;
144             beanParameter.setIndex(paramIndex);
145         }
146         break;
147     case "long":
148         if (beanClassConstructorParameter.toString().
149             equals("long")) {
150             parametersMatched = true;
151             beanParameter.setIndex(paramIndex);
152         }
153         break;
154     case "java.lang.Long":
155         if (beanClassConstructorParameter.toString().
156             equals("long")) {
157             parametersMatched = true;
158             beanParameter.setIndex(paramIndex);
159         }
160         break;
161     case "float":
162         if (beanClassConstructorParameter.toString().
163             equals("float")) {
164             parametersMatched = true;
165             beanParameter.setIndex(paramIndex);
166         }
167         break;
168     case "java.lang.Float":
169         if (beanClassConstructorParameter.toString().
170             equals("float")) {
171             parametersMatched = true;
172             beanParameter.setIndex(paramIndex);
173         }
174         break;
175     case "double":
176         if (beanClassConstructorParameter.toString().

```



```

170         equals("double")) {
171             parametersMatched = true;
172             beanParameter.setIndex(paramIndex);
173         }
174         break;
175     case "java.lang.Double":
176         if (beanClassConstructorParameter.toString().
177             equals("double")) {
178             parametersMatched = true;
179             beanParameter.setIndex(paramIndex);
180         }
181         break;
182     case "boolean":
183         if (beanClassConstructorParameter.toString().
184             equals("boolean")) {
185             parametersMatched = true;
186             beanParameter.setIndex(paramIndex);
187         }
188         break;
189     case "java.lang.Boolean":
190         if (beanClassConstructorParameter.toString().
191             equals("boolean")) {
192             parametersMatched = true;
193             beanParameter.setIndex(paramIndex);
194         }
195         break;
196     case "char":
197         if (beanClassConstructorParameter.toString().
198             equals("char")) {
199             parametersMatched = true;
200             beanParameter.setIndex(paramIndex);
201         }
202         break;
203     case "java.lang.Character":
204         if (beanClassConstructorParameter.toString().
205             equals("char")) {
206             parametersMatched = true;
207             beanParameter.setIndex(paramIndex);
208         }
209         break;
210     default:
211         if (beanParameter.getExplicitTypeName() == null
212             &&
213             beanParameter.getBeanFactory().findBean(
214                 beanParameter.getBeanRef()).

```

```

211         getBeanClass().equals(
212             beanClassConstructorParameter)){
213     parametersMatched = true;
214     beanParameter.setIndex(paramIndex);
215     } else try {
216         if (Class.forName(beanParameter.
217             getExplicitTypeName()).equals(
218                 beanClassConstructorParameter)){
219             parametersMatched = true;
220             beanParameter.setIndex(paramIndex);
221         }
222     } catch (ClassNotFoundException e) {
223         e.printStackTrace();
224     }
225     break;
226 }
227 return parametersMatched;
228 }
229
230 /**
231  * Sets the constructor method and parameters to a bean for
232  * it to be ready
233  * to be autowired and injected. If indexes are not
234  * specified, the method checks
235  * all constructors of the bean's class to match one.
236  * @param bean the bean to use
237  */
238 public static void registerConstructor(Bean bean) {
239     Constructor matchedConstructor = null;
240
241     if (!checkParametersIndexes(bean)) { //Checks if at least
242         one parameter doesn't have an index assigned
243         int totalParametersOneType = 0;
244         int totalParametersMatched = 0;
245         int constructorMatches = 0;
246         int paramIndex = 0;
247         boolean twoMatchesForOneParam = false;
248
249         Constructor[] classConstructors = bean.getBeanClass
250             ().getDeclaredConstructors();
251         Class[] classConstructorParameters;
252
253         for (Constructor classConstructor :
254             classConstructors) { // Iterates through
255             all constructors in the bean's class
256
257             classConstructorParameters = classConstructor.
258                 getParameterTypes();

```

```

249         if (classConstructorParameters.length == bean.
            getBeanConstructor().getBeanParameterList().
            size()) { // Checks if the current class
                constructor has same amount of parameters
250
251
252         for (BeanParameter beanParameter : bean.
            getBeanConstructor().getBeanParameterList
            ()) { // Iterates through all the
                declared parameters in the configuration
253
254             for (Class parameter :
                classConstructorParameters) {
                //
                // Iterates through all the parameters
                // of the current class constructor
255             if (setBeanParameterIndex(
                beanParameter, parameter,
                paramIndex)) { // Compares the
                parameters and assigns an index
                to the bean's constructor
                parameter if they matched
                totalParametersOneType++;
                totalParametersMatched++;
256             }
257             paramIndex++;
258         }
259     }
260
261     paramIndex = 0;
262     if (totalParametersOneType > 1) {
263         twoMatchesForOneParam = true;
264     }
265     totalParametersOneType = 0;
266 }
267
268 if (totalParametersMatched == bean.
269     getBeanConstructor().getBeanParameterList
    ().size() && !twoMatchesForOneParam) {

```

```

270         constructorMatches++;
271         matchedConstructor = classConstructor;
272     }
273     totalParametersMatched = 0;
274 }
275 twoMatchesForOneParam = false;
276 }
277 if (constructorMatches == 0) {
278     try {
279         throw new BeanConstructorNotFoundException("
                Bean_creation_error:_constructor_not_
                found_for_the_specified_parameters_in_
                bean:_"+ bean.getId() + "._");
280     } catch (BeanConstructorNotFoundException e) {
281         e.printStackTrace();
282         System.exit(1);
283     }
284 }
285 if (constructorMatches > 1) {
286     try {
287         throw new BeanConstructorConflictException("
                Bean_creation_error:_there_are_multiple_
                constructors_for_the_specified_parameters
                in_bean:_"+ bean.getId() +
288                 "._Couldn't identify which one is_
                intended_to_be_called_(same_
                parameter_quantity_and_types).");
289     } catch (BeanConstructorConflictException e) {
290         e.printStackTrace();
291         System.exit(1);
292     }
293 }
294 }
295 else{ // All parameters specified in the configuration
296     // have indexes assigned.
297     try {
298         matchedConstructor = bean.getBeanClass().
299             getConstructor(obtainParametersClassArray(
300                 bean));
301     } catch (NoSuchMethodException e) {
302         System.err.println("Bean_creation_error:_
303             constructor_not_found_for_the_specified_
304             parameters_in_bean:_"+ bean.getId() + "._");
305         e.printStackTrace();
306         System.exit(1);
307     }
308 }
309 bean.getBeanConstructor().setConstructorMethod(
310     matchedConstructor); // sets the Constructor to

```

```

305         }
306     }

```

## BeanCreator

```

1  package com.ci1330.ecci.ucr.ac.cr.factory;
2
3  import com.ci1330.ecci.ucr.ac.cr.bean.*;
4  import com.ci1330.ecci.ucr.ac.cr.exception.*;
5
6  import java.lang.reflect.Constructor;
7  import java.lang.reflect.Field;
8  import java.lang.reflect.Method;
9  import java.lang.reflect.Modifier;
10
11 import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
12
13 /**
14  * @author Elias Calderon, Josue Leon, Kevin Leon
15  * Date: 13/09/2017
16  *
17  * Module in charge of receiving each bean's metadata
18  * from the reader and creating the bean, with all
19  * the properties it needs for it to be instantiated
20  * later.
21  */
22 public class BeanCreator {
23
24     // Classes needed to create the bean
25     private Bean bean;
26     private BeanFactory beanFactory;
27     private BeanAttribute attributeClass;
28     private BeanConstructor beanConstructorTemp;
29
30     /**
31      * Constructor of the class which receives the beanFactory
32      * and
33      * assigns it for later use.
34      * @param beanFactory the factory to add beans
35      */
36     public BeanCreator(BeanFactory beanFactory) {
37         this.beanFactory = beanFactory;
38     }
39
40     /**
41      * Method which receives the basic IoC properties for the
42      * bean
43      * and creates it.
44      * @param id the bean's ID

```

```

43      * @param beanClass the bean's class
44      * @param scope the bean's scope
45      * @param initMethodName the bean's init method name
46      * @param destroyMethodName the bean's destroy method name
47      * @param lazyGen the bean's lazy generation value
48      * @param autowireEnum the bean's autowire mode
49      */
50      public void createBean(String id, String beanClass, Scope
        scope, String initMethodName, String destroyMethodName,
        boolean lazyGen, AutowireEnum autowireEnum) {
51          try {
52              if (this.beanFactory.containsBean(id)) {
53                  throw new RepeatedIdException("Creation_error:_
                    Bean_id_" + id + "_is_repeated.");
54              }
55          } catch (RepeatedIdException r) {
56              r.printStackTrace();
57              System.exit(1);
58          }
59
60          bean = new Bean(this.beanFactory);
61          bean.setId(id);
62          try {
63              bean.setBeanClass(Class.forName(beanClass));
64              // Sets the beans type
65          } catch (ClassNotFoundException e) {
66              System.err.println("Creation_error:_bean_class_not_
                    found_for_bean:_ " + id + ".");
67              e.printStackTrace();
68              System.exit(1);
69          }
70          bean.setBeanScope(scope);
71          Method initMethod = null;
72          Method destroyMethod = null;
73          Method[] beanMethods = this.bean.getBeanClass().
            getDeclaredMethods();
74          for (Method method : beanMethods) {
75              if (Modifier.isPrivate(method.getModifiers())){
76                  method.setAccessible(true);
77              }
78              if (initMethodName != null && method.getName().
                contains(initMethodName)) { //Finds the
                initialization and destruction methods for the
                bean
79                  if (method.getParameterCount() == 0) {
80                      initMethod = method;
81                  }
82              }
            if (destroyMethodName != null && method.getName().

```

```

83         contains(destroyMethodName)) {
84             if (method.getParameterCount() == 0) {
85                 destroyMethod = method;
86             }
87         }
88     bean.setInitMethod(initMethod);
89     bean.setDestroyMethod(destroyMethod);
90     bean.setLazyGen(lazyGen);
91     bean.setAutowireEnum(autowireEnum);
92     this.beanConstructorTemp = new BeanConstructor(null);
93     // Creates a temporary constructor to receive the
94     // parameters of the bean
95 }
96
97 /**
98  * Method that returns an object after
99  * casting the string value to its real type.
100  * @param stringValue a string that contains the value
101  * @return object with respective type
102  */
103 private Object obtainValueType(String stringValue) {
104     boolean parsed = false;
105     Object value = null;
106     try {
107         value = Integer.valueOf(stringValue); // It
108         // tries to cast the string to the stated types and
109         // if not proceeds to the next one
110         parsed = true;
111     } catch (NumberFormatException e) {
112         //No es un int.
113     }
114     if (!parsed) {
115         try {
116             value = Byte.valueOf(stringValue);
117             parsed = true;
118         } catch (NumberFormatException e) {
119             //No es un byte.
120         }
121     }
122     if (!parsed) {
123         try {
124             value = Short.valueOf(stringValue);
125             parsed = true;
126         } catch (NumberFormatException e) {
127             //No es un byte.
128         }
129     }
130     if (!parsed) {

```

```

127         try {
128             value = Long.valueOf(stringValue);
129             parsed = true;
130         } catch (NumberFormatException e) {
131             //No es un byte.
132         }
133     }
134     if (!parsed) {
135         try {
136             value = Float.valueOf(stringValue);
137             parsed = true;
138         } catch (NumberFormatException e) {
139             //No es un byte.
140         }
141     }
142     if (!parsed) {
143         try {
144             value = Double.valueOf(stringValue);
145             parsed = true;
146         } catch (NumberFormatException e) {
147             //No es un byte.
148         }
149     }
150     if (!parsed) {
151         if ((stringValue.toLowerCase()).equals("true")) {
152             value = true;
153             parsed = true;
154         } else if ((stringValue.toLowerCase()).equals("false
155             ")) {
156             value = false;
157             parsed = true;
158         }
159     }
160     if (stringValue.length() == 1 && !parsed) {
161         try {
162             value = stringValue.charAt(0);
163             parsed = true;
164         } catch (Exception e) {
165             //No es un char.
166         }
167     }
168     if (!parsed) {
169         value = stringValue;
170     }
171     return value;
172 }
173 /**
174  *

```



```

175     * Method to register an attribute of the bean and find
176     * its setter method to be used later when injecting
177     * the bean's dependencies
178     * @param attributeName the name of the attribute to
        register
179     * @param stringValue a string with the attribute's value
180     * @param beanRef a string with the attribute's bean
        reference
181     * @param atomic_autowire the atomic autowiring mode
182     */
183     public void registerSetter(String attributeName, String
        stringValue, String beanRef, AutowireEnum atomic_autowire
    ){
184         try {
185             if (this.beanFactory.containsBean(beanRef)) {
186                 throw new RepeatedIdException("Creation_error:_
                    Bean_attribute_with_reference_to:_ " + beanRef
                    + "_is_repeated.");
187             }
188         } catch (RepeatedIdException r) {
189             r.printStackTrace();
190             System.exit(1);
191         }
192
193         Object value = null;
194         if(stringValue != null){
195             value = this.obtainValueType(stringValue);
196         }
197
198         Method setterMethod = null;
199         Method[] beanMethods = this.bean.getBeanClass().
            getDeclaredMethods();
200         Class beanRefType = null;
201
202         for(Method method: beanMethods){
203             if (Modifier.isPrivate(method.getModifiers())){
204                 method.setAccessible(true);
205             }
206
207             // Checks if the method is the respective setter for
                this attribute
208             if (method.getName().startsWith("set") && method.
                getName().toLowerCase().contains(attributeName.
                toLowerCase())){
209                 if (method.getParameterCount() == 1){
210                     setterMethod = method;
211                 }
212             }
213         }

```

```

214
215         if(setterMethod == null){
216             try {
217                 throw new SetterMethodNotFoundException("
                Creation_error:_Bean_attribute's_setter_
                method_not_found_for_attribute:" +
                attributeName + ".");
218             } catch (SetterMethodNotFoundException e) {
219                 e.printStackTrace();
220                 System.exit(1);
221             }
222         }
223
224         //If the value is null, the user is using beans, so
225         search for the type that the attribute should have
226         if(value == null) {
227             Field[] beanFields = this.bean.getBeanClass().
228                 getDeclaredFields();
229             for(Field field: beanFields){
230                 if(Modifier.isPrivate(field.getModifiers())){
231                     field.setAccessible(true);
232                 }
233                 if(field.getName().equals(attributeName)) {
234                     beanRefType = field.getType();
235                 }
236             }
237
238             //If the user specified autowire byName at atomic level,
239             the beanRef is the same as the attributeName
240             if (beanRef == null && atomic_autowire == AutowireEnum.
241                 byName) {
242                 beanRef = attributeName;
243             }
244             BeanAttribute beanAttribute = new BeanAttribute(beanRef,
245                 beanRefType, this.beanFactory, value,
246                 atomic_autowire, setterMethod);
247             bean.appendAttribute(beanAttribute);
248         }
249
250         /**
251         * Method which registers a parameter of the bean's
252         * constructor
253         * @param paramType the name of the parameter's type
254         * @param stringValue a string with the parameter's value
255         * @param beanRef a string with the parameter's bean
256         * reference
257         * @param atomic_autowire the atomic autowiring mode
258         */

```

```

252 public void registerConstructorParameter(String paramType,
    int index, String stringValue, String beanRef,
    AutowireEnum atomic_autowire){
253     Object value = null;
254     if(stringValue != null){
255         value = this.obtainValueType(stringValue);
256     }
257     if(value == null && beanRef == null && paramType == null
    ){
258         try {
259             throw new InvalidPropertyException("Bean_
                creation_error:_parameter's_type,_reference_
                or_value_is_invalid_for_a_declared_bean_
                parameter.");
260         } catch (InvalidPropertyException e) {
261             e.printStackTrace();
262             System.exit(1);
263         }
264     }
265
266     Class beanRefClass = null;
267
268     //If the value is null, the user is using beans, so
269     //search for the type that the parameter should have
270     //But because this is a constructor parameter, only
271     //search for it if we have at least the type
272     if (value == null && paramType != null) {
273         try {
274             beanRefClass = Class.forName(paramType);
275         } catch (ClassNotFoundException e) {
276             e.printStackTrace();
277             System.exit(1);
278         }
279     }
280
281     BeanParameter beanConstructorParam = new BeanParameter(
        beanRef, beanRefClass, this.beanFactory, value,
        atomic_autowire, index, paramType);
282     this.beanConstructorTemp.append(beanConstructorParam);
283
284     /**
285     * There is a special case in which an AtomicAutowire
286     * annotation is found above a constructor
287     * In this case, the constructor is already known, but the
288     * parameters need to be set later.
289     * So the Reader sends the constructor explicitly, and it is
290     * added to the current bean.

```

```

288     *
289     * The method addBeanToContainer won't interfere in this
        definition , because if the user didn't
290     * specify another constructor elsewhere , the
        beanConstructorTemp won't be added to the current bean ,
291     * leaving the explicit definition untouched.
292     */
293     public void explicitConstructorDefinition (Constructor
        constructorMethod) {
294         this.bean.setBeanConstructor(new BeanConstructor(
            constructorMethod));
295     }
296
297     /**
298     * Adds the bean to the container and resets all its
        attributes
299     * for the creator to be ready to read another bean's data
300     */
301     public void addBeanToContainer(){
302         //If there were no parameters specified for the
        constructor , it is assumed the user didn't
303         //indicate to use constructor injection
304         if (this.beanConstructorTemp.getBeanParameterList().size
            () > 0) {
305             this.bean.setBeanConstructor(this.
                beanConstructorTemp);
306         }
307         this.beanFactory.addBean(this.bean);
308         bean = null;
309         attributeClass = null;
310         beanConstructorTemp = null;
311     }
312
313     //


---


314     // Standard Setters and Getters section
315     //


---


316
317     public Bean getBean() {
318         return bean;
319     }
320
321     public void setBean(Bean bean) {
322         this.bean = bean;
323     }
324

```

```

325     public BeanFactory getBeanFactory() {
326         return beanFactory;
327     }
328
329     public void setBeanFactory(BeanFactory beanFactory) {
330         this.beanFactory = beanFactory;
331     }
332
333     public BeanAttribute getAttributeClass() {
334         return attributeClass;
335     }
336
337     public void setAttributeClass(BeanAttribute attributeClass)
338     {
339         this.attributeClass = attributeClass;
340     }
341 }

```

### BeanFactory

```

1  package com.ci1330.ecci.ucr.ac.cr.factory;
2
3  import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
4  import com.ci1330.ecci.ucr.ac.cr.bean.BeanAttribute;
5  import com.ci1330.ecci.ucr.ac.cr.bean.BeanParameter;
6  import com.ci1330.ecci.ucr.ac.cr.bean.Scope;
7  import com.ci1330.ecci.ucr.ac.cr.exception.
    BeanTypeConflictException;
8  import com.ci1330.ecci.ucr.ac.cr.exception.IdNotFoundException;
9
10 import java.util.ArrayList;
11 import java.util.HashMap;
12 import java.util.List;
13 import java.util.Map;
14
15 /**
16  * @author Elias Calderon, Josue Leon, Kevin Leon
17  * Date: 13/09/2017
18  *
19  * BeanFactory parent class which has the container and manages
20  * the control flow of NAIoCC. User's request for beans via an
21  * instance of this class.
22  */
23 public abstract class BeanFactory {
24
25     protected HashMap<String,Bean> beansMap; // The container in
        which beans are stored. A map with beans' id as key and
        the respective Bean as value
26

```

```

27     private boolean nonFatalCycle;
28
29     /**
30      * Constructor of the class , initializes the container
31      */
32     public BeanFactory(){
33         beansMap = new HashMap<>();
34     }
35
36     /**
37      * Adds a bean to the container before initializing it.
38      * @param bean the bean to add
39      */
40     public void addBean(Bean bean){
41         this.beansMap.put(bean.getId() , bean);
42     }
43
44     /**
45      * Returns the instance of the bean, already injected. If it
46      * is singleton it
47      * returns the only instance , otherwise creates a new one (
48      * prototype).
49      * @param id the bean's id
50      * @return the requested bean's instance
51      */
52     public Object getBean(String id) {
53         try {
54
55             if (!this.beansMap.containsKey(id)) {
56                 throw new IdNotFoundException("Exception_Error:_
57                 The_id:_"+ id + "_does_not_exist.");
58             }
59
60             Bean currBean = this.beansMap.get(id);
61             if (currBean.getBeanScope() == Scope.Prototype ||
62                 currBean.getInstance() == null) {
63
64                 currBean.createNewInstance(); // Adds the new
65                 instance to the bean's list
66                 currBean.injectDependencies();
67                 currBean.initialize();
68             }
69
70             return currBean.getInstance(); // Returns the last
71             instance of the bean's list
72         } catch (IdNotFoundException e) {
73             e.printStackTrace();
74         }
75     }

```

```

70         System.exit(1);
71     }
72
73     return null;
74 }
75
76 /**
77  * Iterates through all beans and checks their scope to
78  * initialize and inject its dependencies.
79  */
80 protected void initContainer(){
81     for(HashMap.Entry<String,Bean> beanEntry: beansMap.
82         entrySet()) { // Iterates through the container to
83         // autowire dependencies
84         Bean currBean = beanEntry.getValue();
85         currBean.autowire(); // Autowires
86         // the bean, if indicated as such
87         currBean.checkBeanProperties(); // Checks
88         // there are no conflicts in its properties
89     }
90
91     cycleDetection(); // Checks if there a cycles
92     // between the dependencies of the beans
93
94     for(HashMap.Entry<String,Bean> beanEntry: beansMap.
95         entrySet()){ // Iterates through the container to
96         // initialize beans
97         Bean currBean = beanEntry.getValue();
98
99         if(currBean.getBeanScope() == Scope.Singleton && !
100            currBean.isLazyGen() // Instantiates the bean
101            // only if it is Singleton, without lazy generation
102            // and haven't been initialized
103            && currBean.getInstance() == null){
104             currBean.createNewInstance();
105             currBean.injectDependencies();
106             currBean.initialize();
107         }
108     }
109 }
110
111 /**
112  * Finds a bean by its type for autowiring purposes. If
113  * there's no bean
114  * with this type in the container or if there are more than
115  * one, it returns null.
116  * @param beanType the bean's type

```

```

106     * @return the Bean with the type requested , null if not
107         found
108     */
109     public Bean findBean(Class beanType) throws
110         BeanTypeConflictException {
111         Bean bean = null;
112
113         for (HashMap.Entry<String, Bean> beanEntry: beansMap.
114             entrySet()) { //Iterates through the container
115
116             if (beanEntry.getValue().getBeanClass().equals(
117                 beanType)) { //Checks if it is of the
118                 requested type
119                 if (bean == null) {
120                     bean = beanEntry.getValue();
121                 } else {
122                     throw new BeanTypeConflictException("
123                         Injection by type error: two or more
124                         beans share the same type.");
125                 }
126             }
127         }
128
129         return bean;
130     }
131
132     /**
133     * Finds a bean by its name. If there's no bean
134     * with this name in the container, it returns null.
135     * @param beanId the bean's id
136     * @return The bean with the corresponding id, null if it
137     *         wasn't found
138     */
139     public Bean findBean(String beanId){
140         Bean bean = null;
141         if (this.beansMap.containsKey(beanId)){
142             bean = this.beansMap.get(beanId);
143         }
144         return bean;
145     }
146
147     /**
148     * Checks if the specified bean is in the container.
149     * @param beanId the bean's id
150     * @return true if the bean is in the container, false
151     *         otherwise
152     */
153     public boolean containsBean(String beanId){

```



```

146         return this.beansMap.containsKey(beanId);
147     }
148
149     /**
150     * Destroys all beans' instances of the container.
151     */
152     public void shutdownHook() {
153         for (HashMap.Entry<String, Bean> beanEntry: beansMap.
154             entrySet()) {
155             beanEntry.getValue().destroyAllInstances();
156         }
157     }
158
159     /**
160     * Iterates all the references of all the beans and checks
161     * if there is a cycle
162     */
163     private void cycleDetection() {
164         HashMap<String, List<String>> setterReferences = new
165             HashMap<>();
166         HashMap<String, List<String>> constructorReferences =
167             new HashMap<>();
168
169         for (Map.Entry<String, Bean> currEntry : this.beansMap.
170             entrySet()) {
171             Bean currBean = currEntry.getValue();
172
173             this.insertConstructorReferences(currBean,
174                 constructorReferences);
175             this.insertSetterReferences(currBean,
176                 setterReferences);
177         }
178
179         //Checks if any of those maps has a cycle
180         this.thereIsCycle(constructorReferences, true);
181         this.thereIsCycle(setterReferences, false);
182     }
183
184     /**
185     * Registers the constructor references for a bean
186     * @param currBean the bean to search
187     * @param constructorReferences a list of the references
188     */
189     private void insertConstructorReferences(Bean currBean,
190         HashMap<String, List<String>> constructorReferences) {
191         List<String> referencesList = new ArrayList<>(); //If
192             there is no dependency the list will be empty
193
194         //If the bean has a constructor

```

```

186         if (currBean.getBeanConstructor() != null) {
187
188             //Iterate every parameter that has a reference and
189             //put it on the map
190             for (BeanParameter currBeanParameter : currBean.
191                 getBeanConstructor().getBeanParameterList()) {
192
193                 String currReference = currBeanParameter.
194                     getBeanRef();
195                 //If the parameter has a beanRef, append it
196                 if (currReference != null) {
197                     referencesList.add(currReference);
198                 }
199             }
200         }
201         constructorReferences.put(currBean.getId(),
202             referencesList);
203     }
204
205     /**
206     * Registers the setter references for a bean
207     * @param currBean the bean to search
208     * @param setterReferences a list of the references
209     */
210     private void insertSetterReferences(Been currBean, HashMap<
211         String, List<String>> setterReferences) {
212         List<String> referenceList = new ArrayList<>(); //If
213             there is no dependency the list will be empty
214
215         //Iterate every attribute that has a reference and put
216         //it on the map
217         for (BeanAttribute currBeanAttribute : currBean.
218             getBeanAttributeList()) {
219
220             String currReference = currBeanAttribute.getBeanRef
221                 ();
222             //If the parameter has a beanRef, append it
223             if (currReference != null) {
224                 referenceList.add(currReference);
225             }
226         }
227         setterReferences.put(currBean.getId(), referenceList);
228     }
229
230     /**
231     * For every entry in the map, checks the cycles, if there

```

```

226         is an invalid one, the program exits.
227     * @param referenceMap all the references for all beans
228     * @param isConstructorInjection indicates if it is checking
229         constructor injection or not.
230     */
231     private void thereIsCycle(HashMap< String , List<String> >
        referenceMap , boolean isConstructorInjection) {
232         List<String> cycleLessReferences = new ArrayList<>(); //
233             References that were already confirmed as cycle-less
234         List<String> currentTrail = new ArrayList<>(); //The
235             reference trail
236
237         for (String beanEntry : referenceMap.keySet()) {
238             this.nonFatalCycle = false;
239             if (checkCycle(beanEntry, referenceMap, currentTrail
240                 , cycleLessReferences , isConstructorInjection)) {
241                 System.err.println("CYCLE_DETECTED: A reference
242                     or chain of references of " + beanEntry + "
243                     causes an invalid cycle.");
244                 System.exit(1);
245             } else if (this.nonFatalCycle) {
246                 System.err.println("CYCLE_DETECTED_(WARNING):
247                     The cycle is not fatal! But keep track of the
248                     cycles ...");
249             }
250         }
251     }
252
253     /**
254     * Recursively check if a chain of references causes a cycle
255     .
256     * @param reference The reference to check
257     * @param referenceMap Map of all references
258     * @param currentTrail The current trail of the recursive
259         call
260     * @param cycleLessReferences The trail of cycle less
261         references , so we don't repeat searches
262     * @param isConstructorInjection indicates if it's checking
263         constructor injection.
264     * @return true if there was a cycle , false if not.
265     */
266     private boolean checkCycle(String reference , HashMap< String
        , List<String> > referenceMap ,
267         List<String> currentTrail , List<
        String> cycleLessReferences ,
268         boolean isConstructorInjection)
269     {
270         boolean cycleDetected = false;

```

```

257
258     if ( cycleLessReferences.contains(reference) ||
259         referenceMap.get(reference).isEmpty() ) {
260         //If the reference was already checked or doesn't
261         //have associated references, there is no cycle
262         cycleDetected = false;
263     } else if (currentTrail.contains(reference)) {
264         //If the dependency was already in the trail, there
265         //is a cycle
266         //But in setter injection, only a pure prototype
267         //cycle causes trouble
268         System.err.println("CYCLE_DETECTED: _Checking_if_the_
269         cycle_is_fatal...");
270         this.nonFatalCycle = true;
271         if (isConstructorInjection) {
272             cycleDetected = true;
273         } else {
274             cycleDetected = checkIfInvalid(currentTrail,
275             reference);
276         }
277     } else {
278         //If the reference has associated references and is
279         //not in the trail
280         //For every associated reference check if it causes
281         //a cycle
282
283         currentTrail.add(reference); //Add the current
284         //dependency to the trail
285
286         String associatedReference;
287         List<String> associatedReferences = referenceMap.get
288         (reference);
289
290         for (int index = 0; index < associatedReferences.
291             size() && !cycleDetected; index++) {
292
293             associatedReference = associatedReferences.get(
294                 index);
295             cycleDetected = checkCycle(associatedReference,
296                 referenceMap, currentTrail,
297                 cycleLessReferences,
298                 isConstructorInjection);
299         }
300
301         currentTrail.remove(reference); //Remove the current
302         //dependency to the trail
303     }
304 }

```

```

291
292         if (!cycleDetected) {
293             //If the reference doesn't cause a cycle, register
                it as cycle-less
294             cycleLessReferences.add(reference);
295         }
296
297         return cycleDetected;
298     }
299
300     /**
301     * Checks if the cycle has only prototypes
302     * @param trail the trail of the recursive call
303     * @return true if illegal cycle, false if not.
304     */
305     private boolean checkIfInvalid (List<String> trail, String
        reference) {
306         int prototypeCount = 0;
307         int referenceCount = 0;
308
309         String dependency;
310
311         //Start from the reference that causes the cycle
312         for (int index = trail.indexOf(reference); index < trail
            .size(); index++) {
313             dependency = trail.get(index);
314             Bean currBean = this.findBean(dependency);
315
316             if (currBean.getBeanScope() == Scope.Prototype) {
317                 prototypeCount++;
318             }
319
320             referenceCount++;
321         }
322
323         return prototypeCount == referenceCount;
324     }
325
326     //


---


327     // Standard Setters and Getters section
328     //


---


329
330     public HashMap<String, Bean> getBeansMap() {
331         return this.beansMap;
332     }

```

```

333
334     public void setBeansMap(HashMap<String , Bean> beansMap) {
335         this.beansMap = beansMap;
336     }
337
338 }

```

### XMLFactory

```

1 package com.ci1330.ecci.ucr.ac.cr.factory;
2
3
4 import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
5 import com.ci1330.ecci.ucr.ac.cr.readers.XmlBeanReader;
6
7 import java.util.HashMap;
8
9 /**
10  * @author Elias Calderon, Josue Leon, Kevin Leon
11  * Date: 13/09/2017
12  *
13  * XMLFactory class which inherits from BeanFactory
14  * and registers the XML file from which the configuration
15  * must be read and tells the reader to parse it.
16  */
17 public class XMLFactory extends BeanFactory{
18
19     private XmlBeanReader xmlBeanReader;    // Instance of the
20                                             XML configuration reader
21
22     private String xmlFile;    //Path of the XML file which
23                                 holds the configuration
24
25     /**
26      * Constructor of the class, it initializes the super-class
27      * attributes and
28      * also the XML bean reader and the file.
29      * @param xmlFile the name of the file
30      */
31     public XMLFactory(String xmlFile){
32         super();
33         this.xmlFile = xmlFile;
34         this.xmlBeanReader = new XmlBeanReader(this);
35         this.registerConfig();
36         super.initContainer();
37     }
38
39     //Tells the reader to start parsing
40     private void registerConfig(){
41         this.xmlBeanReader.readBeans(this.getXmlFile());
42     }
43 }

```

```

39     }
40
41
42     /**
43      * Return a bean instance from the super class.
44      * @param id the beanId
45      * @return the bean instance
46      */
47     @Override
48     public Object getBean(String id) {
49         return super.getBean(id);
50     }
51
52     /**
53      * Adds a bean to the container
54      * @param bean the {@link Bean} class
55      */
56     @Override
57     public void addBean(Bean bean) {
58         super.addBean(bean);
59     }
60
61     /**
62      * Calls the super method for shutdownHook
63      */
64     @Override
65     public void shutdownHook() {
66         super.shutdownHook();
67     }
68
69     //
70
71     // Standard Setters and Getters section
72
73     public String getXmlFile() {
74         return xmlFile;
75     }
76
77     public XmlBeanReader getXmlBeanReader() {
78         return xmlBeanReader;
79     }
80
81     public void setXmlBeanReader(XmlBeanReader xmlBeanReader) {
82         this.xmlBeanReader = xmlBeanReader;
83     }

```

---



---

```

84
85     @Override
86     public HashMap<String , Bean> getBeansMap() {
87         return super.getBeansMap();
88     }
89
90     @Override
91     public void setBeansMap(HashMap<String , Bean> beansMap) {
92         super.setBeansMap(beansMap);
93     }
94 }

```

### AutowireEnum

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Enumeration for NAI OCC Container.
8  * Used for the different values of the AutowireEnum property.
9  */
10 public enum AutowireEnum {
11
12     byType,
13     byName,
14     constructor,
15     none,
16     annotation;
17
18 }

```

### Bean

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 import com.ci1330.ecci.ucr.ac.cr.factory.BeanConstructorModule;
4 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
5
6 import java.lang.reflect.InvocationTargetException;
7 import java.lang.reflect.Method;
8 import java.util.ArrayList;
9 import java.util.List;
10 import java.util.Stack;
11
12 /**
13  * @author Elias Calderon, Josue Leon, Kevin Leon
14  * Date: 11/09/2017
15  *

```



```

16 * Bean class for NAI OCC Container.
17 * Contains the Metadata of a Bean, manages the creation and
    destruction,
18 * manages all the instances (if prototype), and the autowiring.
19 */
20 public class Bean {
21
22     private String id; //Uniquely identifies the bean
23     private Class beanClass; //Used for different Java
        Reflection methods.
24     private boolean lazyGen; //Flag used to indicate if the bean
        is lazy generated
25     private AutowireEnum autowireEnum; //Indicates the type of
        autowiring the Bean uses.
26     private Scope beanScope; //Indicates the scope of the Bean.
27
28     private BeanFactory beanFactory;
29
30     private Method initMethod; //Initialization method, called
        after the injection of dependencies.
31     private Method destroyMethod; //Destroy method, called when
        the container is going to be destroyed.
32
33     private BeanConstructor beanConstructor; //Class used for
        constructor-injections
34     private List<BeanAttribute> beanAttributeList; //List of
        classes that are used for setter-injection
35
36     /*The stack is used for keeping track of the different
        instances of a bean.
37     The top bean instance is considered as the current one.*/
38     private Stack<Object> beanInstanceStack;
39
40     /**
41     * Constructor of the class, initializes the Instances Stack
        and the BeanAttribute List.
42     */
43     public Bean (BeanFactory beanFactory) {
44         this.beanInstanceStack = new Stack<>();
45         this.beanAttributeList = new ArrayList<>();
46         this.beanFactory = beanFactory;
47     }
48
49     /**
50     * Initializes an instance of a bean, and appends the new
        instance to end of
51     * the beanInstanceStack.
52     */
53     public void createNewInstance() {

```

```

54         if (this.beanScope == Scope.Singleton && this.
55             beanInstanceStack.size() > 0) {
56             System.err.println("Invalid initialization: The_
57                 Singleton_Bean_has_already_been_initialized.");
58             System.exit(1);
59         }
60         Object currInstance = this.newInstance();
61         this.beanInstanceStack.push(currInstance);
62     }
63     /**
64      * Autowires all the properties of the bean
65      */
66     public void autowire () {
67         //Autowire by constructor or, Atomic-autowire all
68         //parameters and register the constructor
69         if (this.beanConstructor != null) {
70             List<BeanParameter> beanParameterList = this.
71                 beanConstructor.getBeanParameterList();
72             if (beanParameterList.size() > 0) {
73                 //If the parameter list has parameters, they are
74                 //autowired (if necessary) and the constructor
75                 //is registered
76                 for (BeanParameter beanParameter :
77                     beanParameterList) {
78                     beanParameter.autowireProperty();
79                 }
80                 BeanConstructorModule.registerConstructor(this);
81             } else {
82                 //If there are no paramters, but the constructor
83                 //isn't null, it's because the user indicated
84                 //autowire by constructor to a single
85                 //constructor
86                 BeanAutowireModule.autowireSingleConstructor(
87                     this.beanConstructor, this.beanFactory, this.
88                     id);
89             }
90         }
91         //Atomic-autowire all attributes
92         for (BeanAttribute beanAttribute : this.
93             beanAttributeList) {
94             beanAttribute.autowireProperty();
95         }
96         //Class autowiring
97         BeanAutowireModule.autowireBean(this);
98     }

```

```

91
92  /**
93   * Checks if all the properties of the bean are correct
94   */
95  public void checkBeanProperties() {
96      for (BeanAttribute beanAttribute : this.
97           beanAttributeList) {
98          beanAttribute.checkProperty();
99      }
100
101      if (this.beanConstructor != null) {
102          for (BeanParameter beanParameter : this.
103              beanConstructor.getBeanParameterList()) {
104              beanParameter.checkProperty();
105          }
106      }
107  }
108
109  /**
110   * Creates an instance, by injecting the constructor, if any
111   *
112   * If there is no specified constructor, it uses the default
113   * one.
114   * @return The new bean instance
115   */
116  private Object newInstance() {
117      Object currInstance = null;
118      if (this.beanConstructor == null) {
119          try {
120              currInstance = this.beanClass.newInstance();
121          } catch (InstantiationException e) {
122              System.err.println("Instantiation_Error:_There_
123                               _was_an_exception_trying_to_instantiate_the_
124                               _bean_" + this.beanClass.toString() + ".");
125              e.printStackTrace();
126              System.exit(1);
127          } catch (IllegalAccessException e) {
128              System.err.println("Instantiation_Error:_There_
129                               _was_an_exception_trying_to_access_the_
130                               _instance_bean_" + this.beanClass.toString() +
131                               ".");
132              e.printStackTrace();
133              System.exit(1);
134          }
135      }
136      else {
137          currInstance = this.beanConstructor.newInstance();
138      }
139  }

```

```

131         return currInstance;
132     }
133
134     /**
135     * Make all the setter-injections by iterating the attribute
136     * list.
137     * It pops the top of the stack, makes all the injections,
138     * and then
139     * it pushes back to the stack.
140     */
141     public void injectDependencies () {
142         Object currInstance = this.getInstance();
143         for (BeanAttribute currBeanAttribute : this.
144             beanAttributeList) {
145             currBeanAttribute.injectDependency(currInstance);
146         }
147     }
148
149     /**
150     * Calls the initialization method for the current bean
151     * instance, if any.
152     */
153     public void initialize () {
154         if (this.initMethod != null) {
155             Object currInstance = this.getInstance();
156             try {
157                 this.initMethod.invoke(currInstance);
158             } catch (IllegalAccessException e) {
159                 System.err.println("Initialize_Error:_There_was_
160                     an_exception_trying_to_access_the_init_method
161                     .");
162                 e.printStackTrace();
163                 System.exit(1);
164             } catch (InvocationTargetException e) {
165                 System.err.println("Initialize_Error:_There_was_
166                     an_exception_trying_to_invoke_the_init_method
167                     .");
168                 e.printStackTrace();
169                 System.exit(1);
170             }
171         }
172     }
173
174     /**
175     * Calls the destruction method for all the beans instances,
176     * if any, and leaves
177     * the stack empty.

```

```

171     */
172     public void destroyAllInstances() {
173         Object currInstance;
174         while (!this.beanInstanceStack.empty()) {
175             currInstance = this.beanInstanceStack.pop();
176             if (this.destroyMethod != null) {
177                 try {
178
179                     this.destroyMethod.invoke(currInstance);
180
181                 } catch (IllegalAccessException e) {
182                     System.err.println("Destruction_Error:_There
                                     _was_an_exception_trying_to_access_the_
                                     destroyAllInstances_method.");
183                     e.printStackTrace();
184                     System.exit(1);
185                 } catch (InvocationTargetException e) {
186                     System.err.println("Destruction_Error:_There
                                     _was_an_exception_trying_to_invoke_the_
                                     destroyAllInstances_method.");
187                     e.printStackTrace();
188                     System.exit(1);
189                 }
190             }
191         }
192     }
193
194     /**
195     * Peeks the top of the stack.
196     * @return Returns the current bean.
197     */
198     public Object getInstance () {
199         if (this.beanInstanceStack.empty()) {
200             return null;
201         } else {
202             return this.beanInstanceStack.peek();
203         }
204     }
205
206     /**
207     * Appends an attribute to the end of the attribute list.
208     * @param beanAttributeToAppend bean attribte to append
209     */
210     public void appendAttribute (BeanAttribute
211                                beanAttributeToAppend) {
212         this.beanAttributeList.add(beanAttributeToAppend);
213     }
214

```

```

215 //


---


216 // Standard Setters and Getters section
217 //


---


218
219 public String getId() {
220     return id;
221 }
222
223 public void setId(String id) {
224     this.id = id;
225 }
226
227 public void setBeanClass(Class beanClass) {
228     this.beanClass = beanClass;
229 }
230
231 public Class getBeanClass () {
232     return beanClass;
233 }
234
235 public boolean isLazyGen() {
236     return lazyGen;
237 }
238
239 public void setLazyGen(boolean lazyGen) {
240     this.lazyGen = lazyGen;
241 }
242
243 public AutowireEnum getAutowireEnum() {
244     return autowireEnum;
245 }
246
247 public void setAutowireEnum(AutowireEnum autowireEnum) {
248     this.autowireEnum = autowireEnum;
249 }
250
251 public void setBeanScope(Scope beanScope) {
252     this.beanScope = beanScope;
253 }
254
255 public Scope getBeanScope() {
256     return beanScope;
257 }
258
259 public BeanFactory getBeanFactory() {

```

```

260         return beanFactory;
261     }
262
263     public void setBeanFactory(BeanFactory beanFactory) {
264         this.beanFactory = beanFactory;
265     }
266
267     public void setInitMethod(Method initMethod) {
268         this.initMethod = initMethod;
269     }
270
271     public void setDestroyMethod(Method destroyMethod) {
272         this.destroyMethod = destroyMethod;
273     }
274
275     public void setBeanConstructor(BeanConstructor
        beanConstructor) {
276         this.beanConstructor = beanConstructor;
277     }
278
279     public List<BeanAttribute> getBeanAttributeList () {
280         return beanAttributeList;
281     }
282
283     public BeanConstructor getBeanConstructor() {
284         return beanConstructor;
285     }
286 }

```

### BeanAttribute

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
4
5 import java.lang.reflect.InvocationTargetException;
6 import java.lang.reflect.Method;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 14/09/2017
11  *
12  * BeanAttribute class for NAI OCC Container.
13  * Contains the Metadata of an attribute and manages the setter
14  * injection of a
15  * dependency for a Bean.
16  */
17 public class BeanAttribute extends BeanProperty {

```

```

18     private Method setterMethod; //Used for invoking the
        respective class setter
19
20     /**
21      * Constructor of the class , initializes the class and super
        -class attributes.
22      * @param beanRef init value for the super's beanRef
        attribute
23      * @param beanFactory init value for the super's beanFactory
        attribute
24      * @param value init value for the super's value attribute
25      * @param setterMethod init value for the bean's setter
        method
26      */
27     public BeanAttribute(String beanRef, Class beanRefClass,
        BeanFactory beanFactory, Object value, AutowireEnum
        atomic_autowire, Method setterMethod) {
28         super(beanRef, beanRefClass, beanFactory, value,
        atomic_autowire);
29         this.setterMethod = setterMethod;
30     }
31
32     /**
33      * Receives an object , an injects a dependency to the object
        .
34      * The dependency is fetched by using the super.getInstance
        method
35      * @param objectToInject The bean instance without
        injections
36      */
37     void injectDependency(Object objectToInject) {
38         Object dependency = super.getInstance();
39         try {
40             this.setterMethod.invoke(objectToInject , dependency)
                ;
41
42         } catch (IllegalAccessException e) {
43             System.err.println("Setter_Error:_There_was_an_
                exception_trying_to_access_the_setter_method_for
                :\n"
44                 + "\t" + this.setterMethod.toString() + ".")
                ;
45             e.printStackTrace();
46             System.exit(1);
47         } catch (InvocationTargetException e) {
48             System.err.println("Setter_Error:_There_was_an_
                exception_trying_to_invoke_the_setter_method_for
                :\n"
49                 + "\t" + this.setterMethod.toString() + ".")

```



```

50         ;
51         e.printStackTrace();
52         System.exit(1);
53     }
54 }
55
56 //

```

---

```

57 // Standard Setters and Getters section
58 //

```

---

```

59
60 public void setSetterMethod(Method setterMethod) {
61     this.setterMethod = setterMethod;
62 }
63
64 }

```

### BeanAutowireModule

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 import com.ci1330.ecci.ucr.ac.cr.exception.BeanAutowireException
4     ;
5 import com.ci1330.ecci.ucr.ac.cr.exception.
6     BeanTypeConflictException;
7 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
8 import com.thoughtworks.paranamer.AdaptiveParanamer;
9 import com.thoughtworks.paranamer.Paranamer;
10
11 import java.lang.reflect.*;
12 import java.util.ArrayList;
13 import java.util.List;
14
15 /**
16  * @author Elias Calderon, Josue Leon, Kevin Leon
17  * Date: 13/09/2017
18  *
19  * The class is in charge of autowiring (at class level) a bean.
20  * It also has the capability to autowire a single constructor.
21  */
22 public class BeanAutowireModule {
23
24     /**
25      * Determines which type of autowiring needs to be done
26      * @param bean the bean to autowire
27      */

```

```

26     static void autowireBean (Bean bean) {
27         switch (bean.getAutowireEnum()) {
28             case byName:
29                 autowireByName(bean);
30                 break;
31             case byType:
32                 autowireByType(bean);
33                 break;
34             case constructor:
35                 autowireConstructor(bean);
36                 break;
37             case none:
38                 break;
39             default:
40                 try {
41                     throw new BeanAutowireException("Autowire_
Module_Error:_Unexpected_value_recieved_
while_trying_to_autowire_the_bean_" +
bean.getId());
42                 } catch (BeanAutowireException e) {
43                     e.printStackTrace();
44                     System.exit(1);
45                 }
46             }
47     }
48
49     /**
50     * Iterates all the fields of a class. For every field it
51     * searches that the field name matches a beanId
52     * in the container, if found, creates a {@link
53     * BeanAttribute} for the field.
54     * @param bean the bean to use
55     */
56     private static void autowireByName (Bean bean) {
57         Class currInstanceClass = bean.getBeanClass();
58         BeanFactory beanFactory = bean.getBeanFactory();
59         List<BeanAttribute> registeredAttributes = bean.
60             getBeanAttributeList();
61
62         Method currAttributeSetter;
63         String currAttributeName;
64         Class currAttributeType;
65
66         //For every field of the class
67         for (Field currAttribute: currInstanceClass.
68             getDeclaredFields()) {
69             //If the field is private, make it accessible
70             if (Modifier.isPrivate(currAttribute.getModifiers()))
71                 {

```

```

67         currAttribute.setAccessible(true);
68     }
69
70     currAttributeName = currAttribute.getName();
71     currAttributeType = currAttribute.getType();
72
73     if(beanFactory.findBean(currAttributeName) != null){
74         currAttributeSetter = findSetter(
75             currAttributeName, currAttributeType, bean);
76
77         //If the attribute isn't already registered in
78         //the Bean (the user didn't overwrite the
79         //autowiring for the
80         //attribute), put it in the bean.
81         if (!attributeIsAlreadyRegistered(
82             registeredAttributes, currAttributeName)) {
83             BeanAttribute beanAttribute = new
84                 BeanAttribute(currAttributeName,
85                     currAttributeType, beanFactory, null,
86                     AutowireEnum.none, currAttributeSetter);
87             bean.appendAttribute(beanAttribute);
88         }
89     }
90 }
91
92 /**
93  * Finds the setter method for an attribute
94  * @param attributeName the name of the attribute used
95  * @param attributeClass the type of the attribute
96  * @param bean used to recover the class of the bean
97  * @return the setter method
98  */
99 private static Method findSetter (String attributeName,
100     Class attributeClass, Bean bean) {
101     Method[] beanMethods;
102     Class[] methodParameterTypes;
103
104     //Search every method in the bean
105     beanMethods = bean.getBeanClass().getDeclaredMethods();
106     for (Method method : beanMethods) {
107         //If private, make it accessible
108         if (Modifier.isPrivate(method.getModifiers())){
109             method.setAccessible(true);
110         }
111
112         //Check if it has set at the start and contains the
113         //name of the attribute
114         if (method.getName().startsWith("set") && method.

```

```

107         getName().toLowerCase().contains(attributeName.
108         toLowerCase())) {
109
110             methodParameterTypes = method.getParameterTypes
111             ();
112             //Check the parameters are valid
113             if (method.getParameterCount() == 1 &&
114                 methodParameterTypes[0].equals(attributeClass
115                 )) {
116                 return method;
117             }
118         }
119     }
120
121     try {
122         throw new BeanAutowireException("Autowire Module_
123         Error:_The_field_" + attributeName + "_of_" + bean
124         .getBeanClass().toString() +
125         "_matches_with_automwiring,_but_no_setter_
126         method_was_found_for_it.");
127     } catch (BeanAutowireException e) {
128         e.printStackTrace();
129         System.exit(1);
130     }
131     return null; //Keep the compiler happy
132 }
133
134 /**
135  * Iterates the list of {@link BeanAttribute} and returns
136  * true if the bean reference is already found,
137  * and false if not.
138  * @param registeredAttributes The list of BeanAttributes
139  * @param beanRef the bean reference to search
140  * @return the result of the search
141  */
142 private static boolean attributeIsAlreadyRegistered (List<
143     BeanAttribute> registeredAttributes, String beanRef) {
144     for (BeanAttribute registeredAttribute :
145         registeredAttributes) {
146         if (registeredAttribute.getBeanRef().equals(beanRef)
147         ) {
148             return true;
149         }
150     }
151     return false;
152 }
153

```

```

144  /**
145   * Iterates every field in the bean class and tries to
      search for a bean that match in type with
146   * the field. If there are multiple definitions of beans
      with that type, an exception is thrown.
147   * @param bean the bean to autowire
148   */
149  private static void autowireByType (Bean bean) {
150
151      Class currInstanceClass = bean.getBeanClass();
152      BeanFactory beanFactory = bean.getBeanFactory();
153      List<BeanAttribute> registeredAttributes = bean.
          getBeanAttributeList();
154
155      Method currAttributeSetter;
156      String currAttributeName;
157      Class currAttributeClass;
158
159      Bean typeLikeBean = null;
160
161      //Iterates all the fields
162      for (Field currAttribute: currInstanceClass.
          getDeclaredFields()) {
163          //If the field is private, make it accesible
164          if (Modifier.isPrivate(currAttribute.getModifiers()))
165              {
166                  currAttribute.setAccessible(true);
167              }
168
169          currAttributeClass = currAttribute.getType();
170
171          //If there are multiple beans with that type, exit
            abnormally.
172          try {
173              typeLikeBean = beanFactory.findBean(
                  currAttributeClass);
174          } catch (BeanTypeConflictException e) {
175              e.printStackTrace();
176              System.exit(1);
177          }
178
179          //If the bean was found
180          if (typeLikeBean != null){
181
182              currAttributeName = currAttribute.getName();
183              currAttributeSetter = findSetter(
                  currAttributeName, currAttributeClass, bean);
184
185              //And it wasn't already in the container,

```

```

185         register it
186         if (!attributeIsAlreadyRegistered(
187             registeredAttributes, currAttributeName)) {
188             BeanAttribute beanAttribute = new
189                 BeanAttribute(typeLikeBean.getId(),
190                     currAttributeClass, beanFactory, null,
191                     AutowireEnum.none, currAttributeSetter);
192             bean.appendAttribute(beanAttribute);
193         }
194     }
195 }
196
197 /**
198  * Iterates all constructors. For every constructor,
199  * searches that its parameters' names, match with a bean's
200  * id.
201  * If they all match, the constructor is selected. If there
202  * is more than one matched constructor, an exception is
203  * thrown.
204  * @param bean the bean to autowire
205  */
206 private static void autowireConstructor (Bean bean) {
207     if (bean.getBeanConstructor() == null) { //If the user
208         already defined the constructor explicitly this
209         process is omitted
210         Constructor[] classConstructors = bean.getBeanClass
211             ().getDeclaredConstructors();
212         BeanFactory beanFactory = bean.getBeanFactory();
213
214         Parameter[] constructorParameters;
215         String[] parameterNames;
216         Constructor matchedConstructor = null;
217         boolean allParamsMatched, allParamsClassesMatched;
218         Paranamer paranamer = new AdaptiveParanamer(); //
219             Utility to recover parameter names
220
221         List<BeanParameter> beanParameterList = new
222             ArrayList<>();
223         for (Constructor classConstructor :
224             classConstructors) {
225             //If it has parameters
226             if (classConstructor.getParameterCount() > 0) {
227                 allParamsMatched = true;
228
229                 parameterNames = paranamer.
230                     lookupParameterNames(classConstructor);
231                 constructorParameters = classConstructor.

```

```

218         getParameters();
219         //Look if the names match
220         for (String parameter : parameterNames) {
221             if (beanFactory.findBean(parameter) ==
222                 null) {
223                 allParamsMatched = false;
224                 break;
225             }
226         }
227         //If they all matched
228         if (allParamsMatched) {
229             //And the constructor didn't match
230             already
231             if (matchedConstructor == null) {
232                 //Check that the types also match
233                 allParamsClassesMatched =
234                     checkParametersTypes(beanFactory,
235                         constructorParameters,
236                         parameterNames, beanParameterList
237                         );
238
239                 if (allParamsClassesMatched) {
240                     matchedConstructor =
241                         classConstructor;
242                 } else {
243                     try {
244                         throw new
245                             BeanAutowireException("
246                                 AutowireModuleError:_
247                                 parameter_types_mismatch_
248                                 for_auowiring_by_
249                                 constructor._Bean:_ " +
250                                 bean.getId());
251                     } catch (BeanAutowireException e
252                         ) {
253                         e.printStackTrace();
254                         System.exit(1);
255                     }
256                 }
257             } else {
258                 //If it did, exit abnormally
259                 try {
260                     throw new BeanAutowireException(
261                         "AutowireModuleError:_there
262                         _are_multiple_constructors_
263                         that_match_in_their_
264                         parameters_names,_in_

```

```

248         autowiring_by_constructor._
249         Bean:_" + bean.getId());
250     } catch (BeanAutowireException e) {
251         e.printStackTrace();
252         System.exit(1);
253     }
254 }
255 }
256 }
257
258     if (matchedConstructor != null) {
259         BeanConstructor beanConstructor = new
260             BeanConstructor(matchedConstructor);
261         beanConstructor.setBeanParameterList(
262             beanParameterList);
263         bean.setBeanConstructor(beanConstructor);
264     }
265 }
266
267 /**
268  * For a specific constructor searches that its parameters'
269  * names, match with a bean's id.
270  * If they all match, the BeanParameters are created. If the
271  * parameters didn't match, exits abnormally.
272  * @param beanConstructor the constructor to match
273  * @param beanFactory the factory to use
274  * @param beanId the id of the bean (used for throwing the
275  * error)
276  */
277 public static void autowireSingleConstructor (
278     BeanConstructor beanConstructor, BeanFactory beanFactory,
279     String beanId) {
280     Constructor classConstructor = beanConstructor.
281         getConstructorMethod();
282
283     //If it has parameters
284     if (classConstructor.getParameterCount() > 0) {
285         Boolean allParamsMatched = true;
286         Paranamer paranamer = new AdaptiveParanamer();
287
288         String[] parameterNames = paranamer.
289             lookupParameterNames(classConstructor);
290         Parameter[] constructorParameters = classConstructor
291             .getParameters();

```



```

285         List<BeanParameter> beanParameterList =
                beanConstructor.getBeanParameterList();
286
287         //Look if the names match
288         for (String parameter : parameterNames) {
289             if (beanFactory.findBean(parameter) == null) {
290                 allParamsMatched = false;
291                 break;
292             }
293         }
294
295         //If they don't match, exit abnormally.
296         if (!allParamsMatched) {
297             try {
298                 throw new BeanAutowireException("Autowire_
                    Module_Error:_One_or_more_constructor_
                    parameters_names_does_not_match_with_a_
                    bean,_in_autowiring_a_single_constructor_
                    _For_bean:_\" + beanId);
299             } catch (BeanAutowireException e) {
300                 e.printStackTrace();
301                 System.exit(1);
302             }
303         } else {
304             Boolean allParamsClassesMatched =
                checkParametersTypes(beanFactory,
                constructorParameters, parameterNames,
                beanParameterList);
305
306             //If the types didn't match, exit abnormally.
307             if (!allParamsClassesMatched) {
308                 try {
309                     throw new BeanAutowireException("
                        Autowire_Module_Error:_parameter_
                        types_mismatch_for_autowiring_by_
                        constructor._For_bean:_\" + beanId);
310                 } catch (BeanAutowireException e) {
311                     e.printStackTrace();
312                     System.exit(1);
313                 }
314             }
315         }
316     }
317 }
318
319 /**
320  * Checks that the types of the parameters, match the types
321  * of the beans.

```

```

322     * @param beanFactory the factory to use
323     * @param constructorParameters the array of parameters
324     * @param constructorParameterNames the array of parameter
        names
325     * @param beanParameterList the list of bean parameters, in
        which we will start to append if a parameter matches
326     * @return true if they all match, false if they don't.
327     */
328     private static boolean checkParametersTypes(BeanFactory
        beanFactory, Parameter[] constructorParameters, String[]
        constructorParameterNames, List<BeanParameter>
        beanParameterList) {
329         boolean allParamsClassesMatched = true;
330         int parameterIndex = 0;
331
332         Class currBeanClass;
333
334         //For every parameter
335         for (Parameter constructorParameter :
            constructorParameters) {
336             currBeanClass = beanFactory.findBean(
                constructorParameterNames[parameterIndex]).
                getBeanClass();
337             //If the type of the bean is the same as the type of
                the parameter
338             if ( currBeanClass == constructorParameter.getType()
                ) {
339                 //Append it
340                 beanParameterList.add(new BeanParameter(
                    constructorParameterNames[parameterIndex],
                    currBeanClass, beanFactory, null,
                    AutowireEnum.none, parameterIndex,
                    constructorParameter.getType().toString()));
341             } else {
342                 allParamsClassesMatched = false;
343                 break;
344             }
345
346             parameterIndex++;
347         }
348
349         return allParamsClassesMatched;
350     }
351 }

```

#### BeanConstructor

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 import java.lang.reflect.Constructor;

```

```

4 import java.lang.reflect.InvocationTargetException;
5 import java.util.ArrayList;
6 import java.util.List;
7
8 /**
9  * @author Elias Calderon, Josue Leon, Kevin Leon
10  * Date: 15/09/2017
11  *
12  * Bean Constructor for NAI OCC Container.
13  * Contains the Metadata of a Bean's constructor, manages the
14  * constructor injection.
15 */
16 public class BeanConstructor {
17     private Constructor constructorMethod;
18     private List<BeanParameter> beanParameterList;
19
20     /**
21      * Constructor of the class, initializes the Parameter list
22      * and sets the constructor method value.
23      * @param constructorMethod init value for the construction
24      * method.
25      */
26     public BeanConstructor (Constructor constructorMethod) {
27         this.constructorMethod = constructorMethod;
28         this.beanParameterList = new ArrayList<>();
29     }
30
31     /**
32      * Creates a new instance of a bean, with constructor
33      * injection.
34      * @return The injected bean instance.
35      */
36     public Object newInstance() {
37         Object[] parameterInstances = new Object[ this.
38             beanParameterList.size() ];
39         Object beanInstance = null;
40         for (BeanParameter currBeanParameter : this.
41             beanParameterList) {
42             parameterInstances[currBeanParameter.getIndex()] =
43                 currBeanParameter.getInstance();
44         }
45         try {
46             beanInstance = this.constructorMethod.newInstance(
47                 parameterInstances);
48         } catch (InstantiationException e) {
49             System.err.println("Construction Error: There was an
50                 exception trying to instantiate a bean with the
51                 constructor method for:\n"

```

```

43         + "\t" + this.constructorMethod.toString()
44         + ".");
45     e.printStackTrace();
46     System.exit(1);
47 } catch (IllegalAccessException e) {
48     System.err.println("Construction_Error: There was an
49         _exception trying to access the constructor _
50         method for:\n"
51         + "\t" + this.constructorMethod.toString()
52         + ".");
53     e.printStackTrace();
54     System.exit(1);
55 } catch (InvocationTargetException e) {
56     System.err.println("Construction_Error: There was an
57         _exception trying to invoke the constructor _
58         method for:\n"
59         + "\t" + this.constructorMethod.toString() +
60         "\n"
61         + "_with_" + parameterInstances[0].getClass
62         () + ".");
63     e.printStackTrace();
64     System.exit(1);
65 }
66
67     return beanInstance;
68 }
69
70 /**
71  * Appends a bean parameter to the list.
72  * @param beanParameter
73  */
74 public void append(BeanParameter beanParameter){
75     this.beanParameterList.add(beanParameter);
76 }
77
78 //
79
80 // Standard Setters and Getters section
81 //

```

---

```

77
78     public void setConstructorMethod(Constructor
          constructorMethod) {
79         this.constructorMethod = constructorMethod;
80     }
81
82     public Constructor getConstructorMethod() {
83         return constructorMethod;
84     }
85
86     public void setBeanParameterList(List<BeanParameter>
          beanParameterList) {
87         this.beanParameterList = beanParameterList;
88     }
89
90     public List<BeanParameter> getBeanParameterList() {
91         return beanParameterList;
92     }
93
94 }

```

### BeanParameter

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
4
5 /**
6  * @author Elias Calderon, Josue Leon, Kevin Leon
7  * Date: 15/09/2017
8  *
9  * BeanParameter class for NAI OCC Container.
10 * Contains the Metadata of a Bean's constructor's parameter.
11 */
12 public class BeanParameter extends BeanProperty {
13
14     private int index; //The position of the parameter in the
          constructor.
15     private String explicitTypeName;
16
17     /**
18      * Constructor of the class, initializes the class and super
          -class attributes.
19      * @param beanRef init value for the super's beanRef
          attribute
20      * @param beanFactory init value for the super's beanFactory
          attribute
21      * @param value init value for the super's value attribute
22      * @param index init value for the parameter's index.
23      */

```

```

24     public BeanParameter(String beanRef, Class beanRefClass,
25                           BeanFactory beanFactory, Object value, AutowireEnum
26                           atomic_autowire, int index, String explicitTypeName) {
27         super(beanRef, beanRefClass, beanFactory, value,
28               atomic_autowire);
29         this.index = index;
30         this.explicitTypeName = explicitTypeName;
31     }
32     //

```

---

```

31     // Standard Setters and Getters section
32     //

```

---

```

33
34     public void setIndex(int index) {
35         this.index = index;
36     }
37
38     public int getIndex() {
39         return index;
40     }
41
42     public String getExplicitTypeName() {
43         return explicitTypeName;
44     }
45
46     public void setExplicitTypeName(String explicitTypeName) {
47         this.explicitTypeName = explicitTypeName;
48     }
49
50 }

```

## BeanProperty

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 import com.ci1330.ecci.ucr.ac.cr.exception.
4     BeanAtomicAutowireException;
5 import com.ci1330.ecci.ucr.ac.cr.exception.BeanPropertyException
6     ;
7 import com.ci1330.ecci.ucr.ac.cr.exception.
8     BeanTypeConflictException;
9 import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
10
11 /**
12  * @author Elias Calderon, Josue Leon, Kevin Leon
13  * Date: 13/09/2017

```

```

11  *
12  * BeanParameter class for NAI OCC Container.
13  * Contains the Metadata of a Bean's property. Manages the
14  * fetching from the factory, if
15  * the property references a Bean.
16  */
17  public abstract class BeanProperty {
18
19      private String beanRef; //The beanId that references a bean
20      private Class beanRefType;
21      private BeanFactory beanFactory;
22      private Object value; //The explicit value, specified by the
23      //end-user.
24      private AutowireEnum atomic-autowire; //Specifies the atomic
25      //autowiring for the property
26
27      /**
28       * Constructor of the class, initializes the class
29       * attributes.
30       * @param beanRef init value for the property's beanRef
31       * attribute
32       * @param beanFactory init value for the property's
33       * beanFactory attribute
34       * @param value init value for the property's value
35       * attribute
36       */
37      BeanProperty(String beanRef, Class beanRefType, BeanFactory
38          beanFactory, Object value, AutowireEnum atomic-autowire)
39      {
40          this.beanRef = beanRef;
41          this.beanRefType = beanRefType;
42          this.beanFactory = beanFactory;
43          this.value = value;
44          this.atomic-autowire = atomic-autowire;
45      }
46
47      /**
48       * The bean instance can either be an explicit value, or be
49       * fetched from the BeanFactory
50       * @return instance of the value or instance sent by the
51       * container for the reference
52       */
53      Object getInstance () {
54          if (this.value == null) {
55              return this.beanFactory.getBean(this.beanRef);
56          } else {
57              return this.value;
58          }
59      }
60  }

```

```

49
50  /**
51   * According to the value of /atomic-autowire, autowires the
52   * property
53   */
54  void autowireProperty () {
55      switch (this.atomic-autowire) {
56          case byName:
57              //This case is mostly for parameter autowiring,
58              //in which the type is known until the
59              //container
60              //is fully created
61              this.autowireByName();
62              break;
63          case byType:
64              //This case is for both parameters and
65              //attributes.
66              //It searches the container for a bean that
67              //matches with its type, if found stores its ID
68              //in the container
69              this.autowireByType();
70              break;
71          case annotation:
72              //This case is exclusive for attributes that
73              //were autowired using an annotation.
74              //It first tries to autowire byType, if it fails
75              // , tries to autowire byName
76              this.autowireByAnnotation();
77              break;
78      }
79  }
80
81  /**
82   * Searches for a bean with the name of beanRef, if not
83   * found, throws an exception
84   * If found, puts beanRefType (if null) to the type of the
85   * recovered bean
86   */
87  private void autowireByName () {
88      if (this.beanFactory.findBean(this.beanRef) == null) {
89          try {
90              throw new BeanAtomicAutowireException("Bean_
91                  Atomic-Autowire_Error:_At_atomic-autowiring_
92                  byName_no_bean_was_found_for_the_beanId_" +
93                  this.beanRef);
94          } catch (BeanAtomicAutowireException e) {
95              e.printStackTrace();
96              System.exit(1);
97          }
98      }
99  }

```



```

85         } else if (this.beanRefType == null) {
86             //If the beanRefType was null, put it as the type of
            the recovered bean
87             this.beanRefType = this.beanFactory.findBean(this.
                beanRef).getBeanClass();
88         }
89     }
90
91     /**
92     * Searches for a bean with the type of beanRefType, if not
        found, throws an exception
93     * If found, stores the recovered bean's ID
94     */
95     private void autowireByType () {
96         try {
97             if (this.beanFactory.findBean(this.beanRefType) ==
                null) {
98
99                 try {
100                     throw new BeanAtomicAutowireException("Bean_
                        Atomic-Autowire_Error:_At_atomic-
                        autowiring_byType_no_bean_was_found_for_
                        the_type_" + this.beanRefType);
101                 } catch (BeanAtomicAutowireException e) {
102                     e.printStackTrace();
103                     System.exit(1);
104                 }
105
106                 } else {
107                     //If a bean exists, store the property's ID
108                     this.beanRef = this.beanFactory.findBean(this.
                        beanRefType).getId();
109                 }
110             } catch (BeanTypeConflictException e) {
111                 e.printStackTrace();
112                 System.exit(1);
113             }
114         }
115
116     /**
117     * Special autowire for annotation, first tries to do
        autowire byType, if it fails, it does autowire byName
118     */
119     private void autowireByAnnotation () {
120         try {
121             //Sees if there exists a bean with that type (
                autowire byType)
122             if (this.beanFactory.findBean(this.beanRefType) ==
                null) {

```

```

123
124         //Sees if there exists a bean with that
            reference (autowire byName)
125         if (this.beanFactory.getBean(this.beanRef) ==
            null) {
126             try {
127                 throw new BeanAtomicAutowireException("
                    BeanAtomicAutowireError: _At_ atomic
                    _autowiring_byName_no_bean_was_found_
                    for_the_beanId_" + this.beanRef);
128             } catch (BeanAtomicAutowireException e) {
129                 e.printStackTrace();
130                 System.exit(1);
131             }
132         }
133         //We don't have to do anything if the beanRef is
            valid, because the type is already assign
134         //And the checkProperty method will check that
            everything matches.
135
136     } else {
137         this.beanRef = this.beanFactory.getBean(this.
            beanRefType).getId();
138     }
139     } catch (BeanTypeConflictException e) {
140         e.printStackTrace();
141         System.exit(1);
142     }
143 }
144
145 /**
146  * Checks if the metadata of a bean is correct, if not,
            throws an exception.
147  */
148 public void checkProperty() {
149     if (value == null) {
150         boolean thereIsProblem = true;
151         //If the reference is not null and exists a
            reference for it in the container
152         if (this.beanRef != null && this.beanFactory.
            findBean(this.beanRef) != null) {
153
154             //If the type is not null and the bean returned
                by the factory matches with the declared type
155             if (this.beanRefType != null && this.beanRefType
                == this.beanFactory.getBean(this.beanRef).
                    getBeanClass()) {
156
157                 thereIsProblem = false;

```

```

158         }
159     }
160 }
161
162     if (thereIsProblem) {
163         try {
164             throw new BeanPropertyException("Bean_
                Property_Error:_There_was_an_unexpected_
                exception_with_the_reference_and_type_of_
                the_property_" + this.beanRef);
165         } catch (BeanPropertyException e) {
166             e.printStackTrace();
167             System.exit(1);
168         }
169     }
170 }
171 }
172
173 //

```

---

```

174 // Standard Setters and Getters section
175 //

```

---

```

176
177 public void setBeanRef(String beanRef) {
178     this.beanRef = beanRef;
179 }
180
181 public void setBeanFactory(BeanFactory beanFactory) {
182     this.beanFactory = beanFactory;
183 }
184
185 public BeanFactory getBeanFactory() {
186     return beanFactory;
187 }
188
189 public void setValue(Object value) {
190     this.value = value;
191 }
192
193 public String getBeanRef() {
194     return this.beanRef;
195 }
196
197 public Class getBeanRefType() {
198     return this.beanRefType;
199 }

```

```

200
201     public Object getValue() {
202         return value;
203     }
204 }

```

### Scope

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Enumeration for NAI OCC Container.
8  * Used for the different values of the Scope property.
9  */
10 public enum Scope {
11     Singleton,
12     Prototype;
13 }

```

### Stereotype

```

1 package com.ci1330.ecci.ucr.ac.cr.bean;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Enumeration for NAI OCC Container.
8  * Used for the different values of the Stereotype Annotations.
9  */
10 public enum Stereotype {
11     Bean,
12     Controller,
13     Service,
14     Repository;
15 }

```

### AnnotationsBeanReaderException

```

1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Indicates an exception in {@link com.ci1330.ecci.ucr.ac.cr.
    readers.AnnotationsBeanReader}

```

```

8  */
9  public class AnnotationsBeanReaderException extends Exception {
10     public AnnotationsBeanReaderException() {
11     }
12
13     public AnnotationsBeanReaderException(String message) {
14         super(message);
15     }
16 }

```

### BeanAtomicAutowireException

```

1  package com.ci1330.ecci.ucr.ac.cr.exception;
2
3  /**
4   * @author Elias Calderon, Josue Leon, Kevin Leon
5   * Date: 13/09/2017
6   *
7   * Indicates an exception in {@link com.ci1330.ecci.ucr.ac.cr.
8   *   bean.BeanProperty} while trying
9   * to atomicly autowire.
10  */
11 public class BeanAtomicAutowireException extends Exception {
12
13     public BeanAtomicAutowireException() {
14         super();
15     }
16
17     public BeanAtomicAutowireException(String message) {
18         super(message);
19     }
20 }

```

### BeanAutowireException

```

1  package com.ci1330.ecci.ucr.ac.cr.exception;
2
3  /**
4   * @author Elias Calderon, Josue Leon, Kevin Leon
5   * Date: 13/09/2017
6   *
7   * Indicates an exception in the {@link com.ci1330.ecci.ucr.ac.
8   *   cr.bean.BeanAutowireModule}
9   */
10 public class BeanAutowireException extends Exception{
11
12     public BeanAutowireException() {
13         super();
14     }
15 }

```

```

15     public BeanAutowireException(String message) {
16         super(message);
17     }
18 }

```

### BeanConstructorConflictException

```

1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Indicates an exception when there is a conflict in {@link com
8  * .ci1330.ecci.ucr.ac.cr.factory.BeanConstructorModule}
9  */
10 public class BeanConstructorConflictException extends Exception{
11
12     public BeanConstructorConflictException() {
13         super();
14     }
15
16     public BeanConstructorConflictException(String message) {
17         super(message);
18     }
19 }

```

### BeanConstructorNotFoundException

```

1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Indicates an exception when the {@link com.ci1330.ecci.ucr.ac
8  * .cr.factory.BeanConstructorModule} can not
9  * find a Constructor.
10 */
11 public class BeanConstructorNotFoundException extends Exception
12 {
13
14     public BeanConstructorNotFoundException() {
15         super();
16     }
17
18     public BeanConstructorNotFoundException(String message) {
19         super(message);
20     }
21 }

```

19  
20 }

### BeanPropertyException

```
1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Indicates an exception when something goes wrong in {@link
8  *   com.ci1330.ecci.ucr.ac.cr.bean.BeanProperty}
9  */
10 public class BeanPropertyException extends Exception {
11
12     public BeanPropertyException() {
13         super();
14     }
15
16     public BeanPropertyException(String message) {
17         super(message);
18     }
19 }
```

### BeanTypeConflictException

```
1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Indicates an exception when trying to invoke findBean method
8  *   of {@link com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory}
9  *   and there are two beans of the same type.
10 */
11 public class BeanTypeConflictException extends Exception {
12
13     public BeanTypeConflictException() {
14         super();
15     }
16
17     public BeanTypeConflictException(String message) {
18         super(message);
19     }
20 }
```

### IdNotFoundException

```
1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Throws an exception if someone tries to recover a bean from {
8  *   @link com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory}
9  * and the id does not exist.
10 */
11 public class IdNotFoundException extends Exception {
12     public IdNotFoundException() {
13     }
14     public IdNotFoundException(String message) {
15         super(message);
16     }
17 }
```

### InvalidPropertyException

```
1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
5  * Date: 13/09/2017
6  *
7  * Indicates an exception when {@link com.ci1330.ecci.ucr.ac.cr.
8  *   factory.BeanCreator}
9  * receives invalid property information.
10 */
11 public class InvalidPropertyException extends Exception {
12     public InvalidPropertyException() {
13         super();
14     }
15
16     public InvalidPropertyException(String message) {
17         super(message);
18     }
19
20 }
```

### RepeatedIdException

```
1 package com.ci1330.ecci.ucr.ac.cr.exception;
2
3 /**
4  * @author Elias Calderon, Josue Leon, Kevin Leon
```



```

5  * Date: 13/09/2017
6  *
7  * Indicates an exception when {@link com.ci1330.ecci.ucr.ac.cr.
   *   factory.BeanCreator} receives a repeated bean id.
8  */
9  public class RepeatedIdException extends Exception{
10
11     public RepeatedIdException(){
12     }
13
14     public RepeatedIdException(String message){
15         super(message);
16     }
17
18 }

```

#### SetterMethodNotFoundException

```

1  package com.ci1330.ecci.ucr.ac.cr.exception;
2
3  /**
4   * @author Elias Calderon, Josue Leon, Kevin Leon
5   * Date: 13/09/2017
6   *
7   * Indicates an exception when the {@link com.ci1330.ecci.ucr.ac
   *   .cr.factory.BeanCreator} does not find the setter method
8   * of an attribute.
9   */
10 public class SetterMethodNotFoundException extends Exception{
11
12     public SetterMethodNotFoundException() {
13         super();
14     }
15
16     public SetterMethodNotFoundException(String message) {
17         super(message);
18     }
19 }

```

#### XmlBeanReaderException

```

1  package com.ci1330.ecci.ucr.ac.cr.exception;
2
3  /**
4   * @author Elias Calderon, Josue Leon, Kevin Leon
5   * Date: 13/09/2017
6   *
7   * Indicates an exception when {@link com.ci1330.ecci.ucr.ac.cr.
   *   readers.XmlBeanReader} finds an error
8   */

```

```
9 public class XmlBeanReaderException extends Exception {  
10     public XmlBeanReaderException() {  
11     }  
12  
13     public XmlBeanReaderException(String message) {  
14         super(message);  
15     }  
16 }
```

# Bibliography

- [1] Inversion of Control History. <http://picocontainer.com/inversion-of-control-history.html>. [Online; accessed 29-September-2017].
- [2] Java dom parser. [https://www.tutorialspoint.com/java\\_xml/java\\_dom\\_parser.htm](https://www.tutorialspoint.com/java_xml/java_dom_parser.htm). [Online; accessed 26-August-2017].
- [3] Paranamer. <https://github.com/paul-hammant/paranamer>. [Online; accessed 23-September-2017].
- [4] Spring Tutorial. <https://www.tutorialspoint.com/spring/index.htm>. [Online; accessed 25-August-2017].
- [5] Core Technologies. <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans>, 2017. [Online; accessed 20-August-2017].
- [6] Derek Banas. Java Reflection Tutorial. <https://www.youtube.com/watch?v=agnblS47F18>, 2012. [Online; accessed 18-August-2017].
- [7] Andrew Binstock. Excellent Explanation of Dependency Injection (Inversion of Control). <https://www.javaworld.com/article/2071914/excellent-explanation-of-dependency-injection--inversion-of-control-.html>, 2008. [Online; accessed 29-September-2017].
- [8] Java Brains. Spring Framework. <https://www.youtube.com/watch?v=GB8k2-Egfv0&list=PLC97BDEFDCDD169D7>, 2011. [Online; accessed 15-August-2017].
- [9] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. <https://martinfowler.com/articles/injection.html>, 2004. [Online; accessed 29-September-2017].

- [10] Telusko Learnings. Annotations in Java. <https://www.youtube.com/watch?v=rWlHQnvrZcw>, 2016. [Online; accessed 10-September-2017].
- [11] BASE Logic. Creating and processing custom Java annotations. <https://www.youtube.com/watch?v=J2GohD6r8Co>, 2017. [Online; accessed 10-September-2017].