Inversion of Control Programming Assignment

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

Sunday 1st October, 2017

Escuela de Ciencias de la Computación e Informática

Universidad de Costa Rica.

Contents

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 of 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.

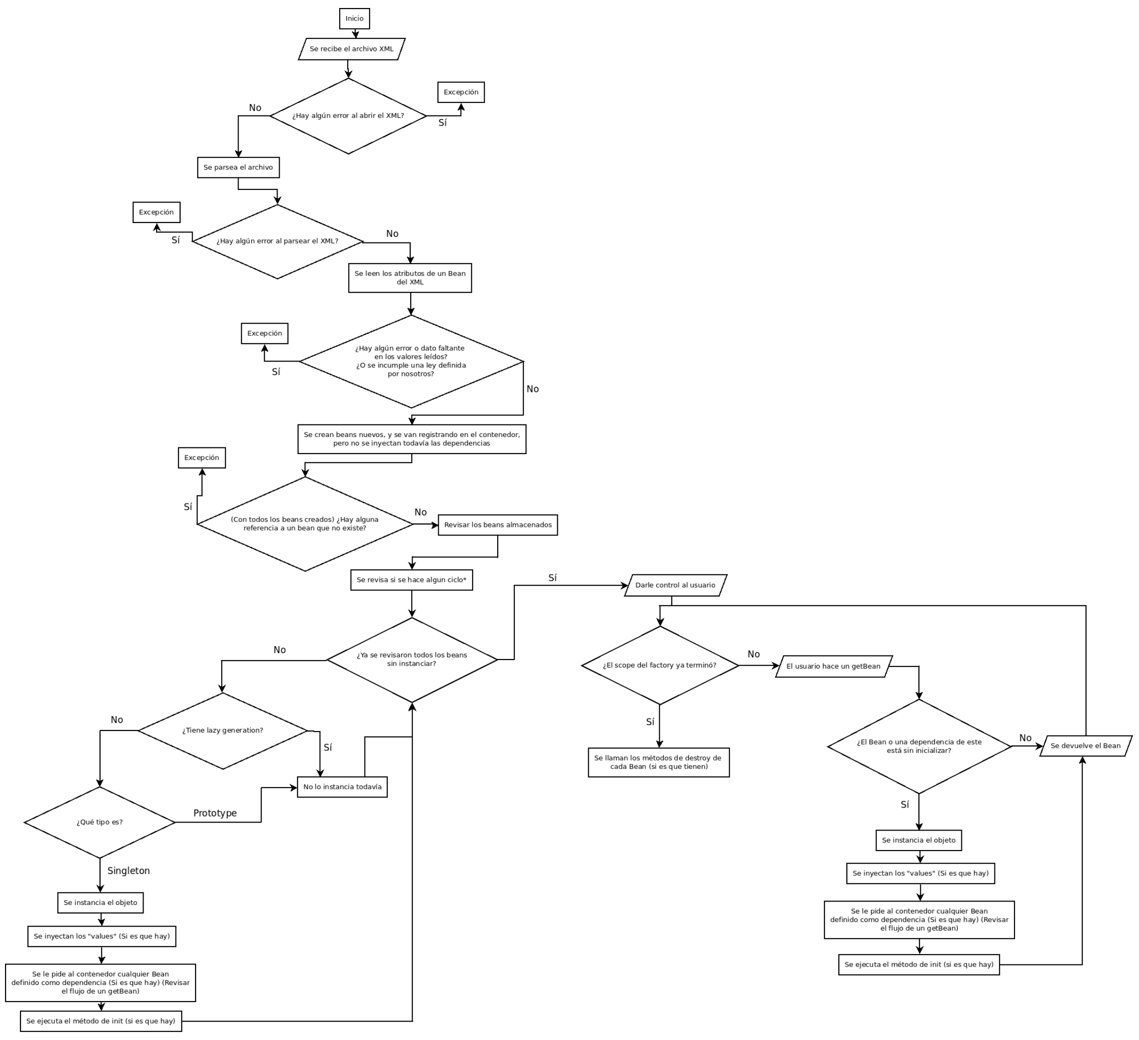
The process of autowiring can be executed in different ways, depending on the configuration. In the case that it is set to "byName", it searches for a bean in the container, that has the same id as the name specified, in order to wire it. In the case that it is set to "byType", a bean with the respective 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 exception 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.

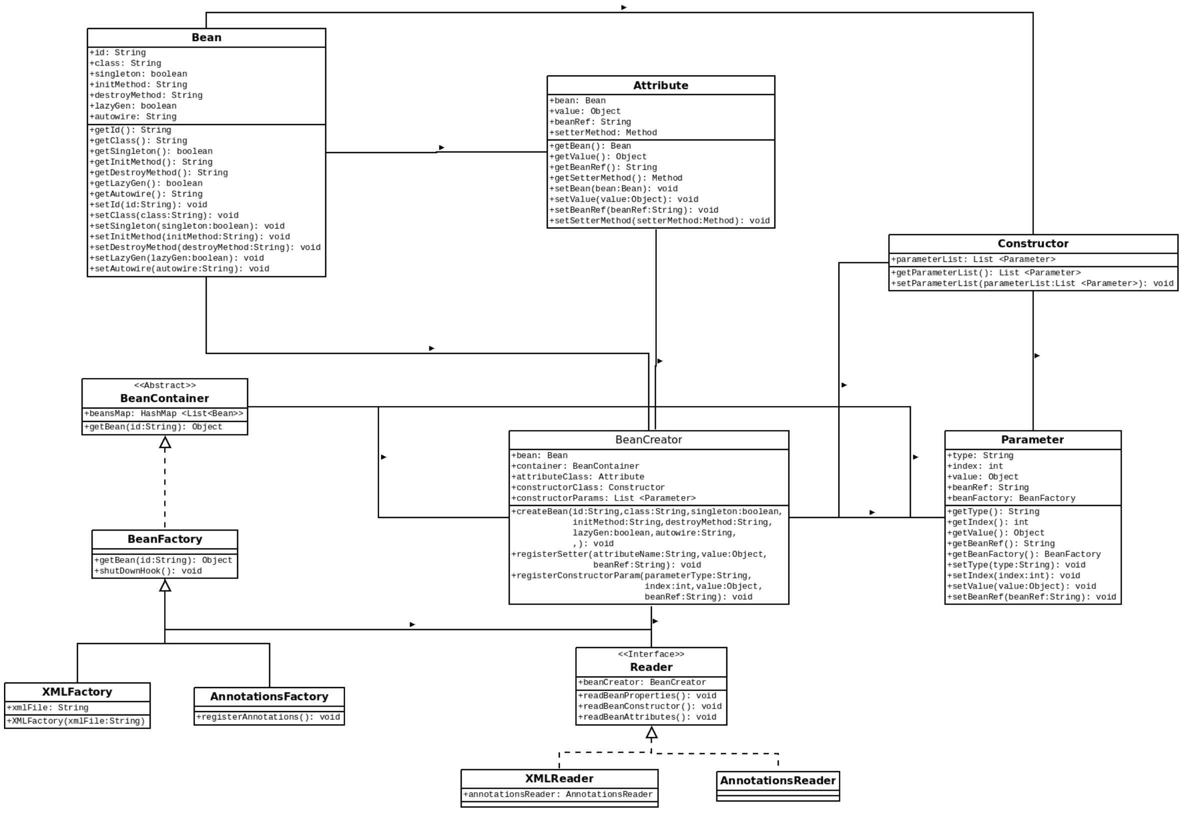
Finally, the user can also call the shutDownHook method of the BeanFactory, which will iterate through all the beans in the HashMap and destroy all the instances for each bean.

3.2 Flow Diagram



Inicio
Se recibe el archivo XML
¿Hay algún error al abrir el XML?
No
Sí
Excepción
Se parsea el archivo
¿Hay algún error al parsear el XML?
Sí
Excepción
No
Se leen los atributos de un Bean del XML
¿Hay algún error o dato faltante en los valores leídos? ¿O se incumple una ley definida por nosotros?
Sí
Excepción
No
Se crean beans nuevos, y se van registrando en el contenedor, pero no se inyectan todavía las dependencias
(Con todos los beans creados) ¿Hay alguna referencia a un bean que no existe?
Sí
Excepción
No
Revisar los beans almacenados
Se revisa si se hace algun ciclo*
¿Ya se revisaron todos los beans sin instanciar?
No
¿Tiene lazy generation?
Sí
No lo instancia todavía
No
¿Qué tipo es?
Prototype
Singleton
Se instancia el objeto
Se inyectan los "values" (Si es que hay)
Se le pide al contenedor cualquier Bean definido como dependencia (Si es que hay) (Revisar el flujo de un getBean)
Se ejecuta el método de init (si es que hay)
Sí
Darle control al usuario
¿El scope del factory ya terminó?
Sí
Se llaman los métodos de destroy de cada Bean (si es que tienen)
No
El usuario hace un getBean
¿El Bean o una dependencia de este está sin inicializar?
No
Se devuelve el Bean
Sí
Se instancia el objeto
Se inyectan los "values" (Si es que hay)
Se le pide al contenedor cualquier Bean definido como dependencia (Si es que hay) (Revisar el flujo de un getBean)
Se ejecuta el método de init (si es que hay)

3.3 Class diagram


Bean
+id: String
+class: String
+singleton: boolean
+initMethod: String
+destroyMethod: String
+lazyGen: boolean
+autowire: String
+getId(): String
+getClass(): String
+getSingleton(): boolean
+getInitMethod(): String
+getDestroyMethod(): String
+getLazyGen(): boolean
+getAutowire(): String
+setId(id:String): void
+setClass(class:String): void
+setSingleton(singleton:boolean): void
+setInitMethod(initMethod:String): void
+setDestroyMethod(destroyMethod:String): void
+setLazyGen(lazyGen:boolean): void
+setAutowire(autowire:String): void
Attribute
+bean: Bean
+value: Object
+beanRef: String
+setterMethod: Method
+getBean(): Bean
+getValue(): Object
+getBeanRef(): String
+getSetterMethod(): Method
+setBean(bean:Bean): void
+setValue(value:Object): void
+setBeanRef(beanRef:String): void
+setSetterMethod(setterMethod:Method): void
Constructor
+parameterList: List <Parameter>
+getParameterList(): List <Parameter>
+setParameterList(parameterList:List <Parameter>): void
<<Abstract>>
BeanContainer
+beansMap: HashMap <List<Bean>>
+getBean(id:String): Object
BeanCreator
+bean: Bean
+container: BeanContainer
+attributeClass: Attribute
+constructorClass: Constructor
+constructorParams: List <Parameter>
+createBean(id:String,class:String,singleton:boolean,
initMethod:String,destroyMethod:String,
lazyGen:boolean,autowire:String,
,): void
+registerSetter(attributeName:String,value:Object,
beanRef:String): void
+registerConstructorParam(parameterType:String,
index:int,value:Object,
beanRef:String): void
Parameter
+type: String
+index: int
+value: Object
+beanRef: String
+beanFactory: BeanFactory
+getType(): String
+getIndex(): int
+getValue(): Object
+getBeanRef(): String
+getBeanFactory(): BeanFactory
+setType(type:String): void
+setIndex(index:int): void
+setValue(value:Object): void
+setBeanRef(beanRef:String): void
BeanFactory
+getBean(id:String): Object
+shutDownHook(): void
XMLFactory
+xmlFile: String
+XMLFactory(xmlFile:String)
AnnotationsFactory
+registerAnnotations(): void
<<Interface>>
Reader
+beanCreator: BeanCreator
+readBeanProperties(): void
+readBeanConstructor(): void
+readBeanAttributes(): void
XMLReader
+annotationsReader: AnnotationsReader
AnnotationsReader

Appendix A

Source Code

BeanReader

```java
package com.ci1330.ecci.ucr.ac.cr.readers;

import com.ci1330.ecci.ucr.ac.cr.bean.AutowireEnum;
import com.ci1330.ecci.ucr.ac.cr.bean.Scope;
import com.ci1330.ecci.ucr.ac.cr.exception.
    XmlBeanReaderException;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanCreator;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 * The father class for a reader, defines the {@link BeanCreator
     } to use and the method, readBeans
 */
public class BeanReader{

    /**
     * Object used to create the beans
     */
    protected BeanCreator beanCreator;

    /**
     * General constructor that initializes the creator
     * @param beanFactory the factory that the creator will use
     */
    public BeanReader (BeanFactory beanFactory) {
        this.beanCreator = new BeanCreator(beanFactory);
    }

    /**
     * This constructor receives the bean creator
     * @param beanCreator the creator to use
     */
    public BeanReader (BeanCreator beanCreator) {
        this.beanCreator = beanCreator;
    }

    /**
     * Abstract method, that indicates the name of the input to read
     * @param inputName the name of the configuration container
     */
    public void readBeans (String inputName) {}

    /**
     * Determines which type of {@link AutowireEnum} is entered, if not found, throws an exception and exits.
     * Atomic autowiring, only accepts byName, byType or none.
     * @param atomic_autowireString the String to match with a type of {@link AutowireEnum}
     * @return the respective {@link AutowireEnum}
     */
    protected AutowireEnum determineAtomic_Autowire (String atomic_autowireString) {

        final String byNameString = "byname";
        final String bytypeString = "bytype";
        final String noneString = "none";

        AutowireEnum atomic_autowire = null;

        switch (atomic_autowireString) {
            case byNameString:
                atomic_autowire = AutowireEnum.byName;
                break;
            case bytypeString:
                atomic_autowire = AutowireEnum.byType;
                break;
            case noneString:
                atomic_autowire = AutowireEnum.none;
                break;
            default:
                try {
                    throw new XmlBeanReaderException("XML Reader Error: The value for atomic-autowire '" + atomic_autowireString + "' was not recognized.");
                } catch (XmlBeanReaderException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
        }

        return atomic_autowire;
    }

    /**
     * Determines which type of {@link AutowireEnum} is entered, if not found, throws an exception and exits
     * @param autowireString the String to match with a type of {@link AutowireEnum}
     * @return the respective {@link AutowireEnum}
     */
    protected AutowireEnum determineClass_Autowire (String autowireString) {
        final String byNameString = "byname";
        final String bytypeString = "bytype";
        final String byConstructorString = "constructor";
        final String noneString = "none";

        AutowireEnum autowire = null;

        //If none of those was specified, the system throws an exception
        switch (autowireString) {
            case byNameString:
                autowire = AutowireEnum.byName;
                break;
            case bytypeString:
                autowire = AutowireEnum.byType;
                break;
            case byConstructorString:
                autowire = AutowireEnum.constructor;
                break;
            case noneString:
                autowire = AutowireEnum.none;
                break;
            default:
                try {
                    throw new XmlBeanReaderException("XML␣Reader␣Error:␣The␣value␣for␣autowire␣'" + autowireString + "'␣was␣not␣recognized.");
                } catch (XmlBeanReaderException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
```

```java
        }

        return autowire;
    }

    /**
     * Determines which type of {@link Scope} is entered, if not found, throws an exception and exits
     * @param scopeString the String to match with a type of {@link Scope}
     * @return the respective {@link Scope}
     */
    protected Scope determineScope (String scopeString) {
        final String singletonString = "singleton";
        final String prototypeString = "prototype";
        Scope scope = null;

        //If prototype wasn't specified, the system throws an exception
        switch (scopeString) {
            case prototypeString:
                scope = Scope.Prototype;
                break;
            case singletonString:
                scope = Scope.Singleton;
                break;
            default:
                try {
                    throw new XmlBeanReaderException("XML Reader Error: The value for scope '" + scopeString + "' was not recognized.");
                } catch (XmlBeanReaderException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
        }

        return scope;
    }

    /**
     * Determines which value of lazy generation is entered, if not found, throws an exception and exits
     * @param lazyGenString the String to match with true or false
     * @return a boolean indicating the lazy generation value
     */
    protected Boolean determineLazyGen (String lazyGenString) {
        final String trueString = "true";
        final String falseString = "false";

        Boolean lazyGeneration = false;

        //If none of those was specified, the system throws an
           exception
        switch (lazyGenString) {
            case trueString:
                lazyGeneration = true;
                break;
            case falseString:
                lazyGeneration = false;
                break;
            default:
                try {
                    throw new XmlBeanReaderException("XML Reader
                        Error: The value for lazy generation '"
                        + lazyGenString + "' was not recognized."
                        );
                } catch (XmlBeanReaderException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
        }
        return lazyGeneration;
    }

}
```

AnnotationsBeanReader

```java
package com.ci1330.ecci.ucr.ac.cr.readers;

import com.ci1330.ecci.ucr.ac.cr.annotations.*;
import com.ci1330.ecci.ucr.ac.cr.bean.AutowireEnum;
import com.ci1330.ecci.ucr.ac.cr.bean.Stereotype;
import com.ci1330.ecci.ucr.ac.cr.exception.
    AnnotationsBeanReaderException;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanCreator;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;

import java.lang.annotation.Annotation;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * The reader is given a String, and then tries to map it with a class
 * and extract the metadata for the BeanCreator
 */
public class AnnotationsBeanReader extends BeanReader {

    private String currID; //The bean ID
    private Stereotype stereotype; //The type of stereotype


    /** Constructor, receives the {@link BeanFactory} that created him
     * @param beanFactory the father {@link BeanFactory}
     */
    public AnnotationsBeanReader(BeanFactory beanFactory) {
        super(beanFactory);
    }

    /**
     * Constructor, receives the {@link BeanCreator} that it'll use
     * @param beanCreator the {@link BeanCreator} to use
     */
    AnnotationsBeanReader(BeanCreator beanCreator) {
        super(beanCreator);
    }

    /**
     * Receives the name of a class and creates the corresponding Class object,
     * and calls a method to read it
     * @param inputName the name of the class
     */
    @Override
    public void readBeans(String inputName) {
        Class reflectClass = null;

        try {
            reflectClass = Class.forName(inputName);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }

        //Check if there is more than a stereotype
        if (reflectClass.isAnnotationPresent(Bean.class) && !reflectClass.isAnnotationPresent(Repository.class)
                && !reflectClass.isAnnotationPresent(Service.class) && !reflectClass.isAnnotationPresent(
                Controller.class)) {

            this.stereotype = Stereotype.Bean;

        } else if(!reflectClass.isAnnotationPresent(Bean.class) && reflectClass.isAnnotationPresent(Repository.class)
                && !reflectClass.isAnnotationPresent(Service.class) && !reflectClass.isAnnotationPresent(Controller.class)) {

            this.stereotype = Stereotype.Repository;

        } else if(!reflectClass.isAnnotationPresent(Bean.class) && !reflectClass.isAnnotationPresent(Repository.class)
                && reflectClass.isAnnotationPresent(Service.class) && !reflectClass.isAnnotationPresent(Controller.class)){

            this.stereotype = Stereotype.Service;

        } else if(!reflectClass.isAnnotationPresent(Bean.class) && !reflectClass.isAnnotationPresent(Repository.class)
                && !reflectClass.isAnnotationPresent(Service.class) && reflectClass.isAnnotationPresent(Controller.class)){

            this.stereotype = Stereotype.Controller;

        } else {
            try {
                throw new AnnotationsBeanReaderException("Annotations Reader error: The 'class' " + inputName + " does not have the Stereotype Annotation or has more than a Stereotype");
            } catch (AnnotationsBeanReaderException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }

        //Now read the rest of the metadata
        this.readBeanProperties(reflectClass);
        this.readBeanConstructor(reflectClass);
        this.readBeanSetter(reflectClass);
        this.beanCreator.addBeanToContainer();
    }

    /**
     * Receives the class and starts to read the annotations, if
         any.
     * @param beanClass the class to search
     */
    private void readBeanProperties (Class beanClass) {

        //Get the bean ID depending of the stereotype
        switch (this.stereotype){
            case Bean:
                Bean bean  = (Bean) beanClass.
                    getDeclaredAnnotation(Bean.class);
                this.currID = bean.value();
                break;
            case Controller:
                Controller controller = (Controller) beanClass.
                    getDeclaredAnnotation(Controller.class);
                this.currID = controller.value();
                break;
            case Repository:
                Repository repository = (Repository) beanClass.
                    getDeclaredAnnotation(Repository.class);
                this.currID = repository.value();
                break;
            default:
                Service service = (Service) beanClass.
                    getDeclaredAnnotation(Service.class);
                this.currID = service.value();
                break;
        }


        //The default scope is singleton
        com.ci1330.ecci.ucr.ac.cr.bean.Scope scope = com.ci1330.
            ecci.ucr.ac.cr.bean.Scope.Singleton;
        if(beanClass.isAnnotationPresent(Scope.class)){
            Scope scopeAnnotation = (Scope)(beanClass.
                getAnnotation(Scope.class));
            scope = super.determineScope(scopeAnnotation.value()
                .toLowerCase());
        }

        //The default class-autowire is none
        AutowireEnum autowire = AutowireEnum.none;
        if(beanClass.isAnnotationPresent(ClassAutowire.class)){
            ClassAutowire autowireAnnotation = (ClassAutowire)(
                beanClass.getAnnotation(ClassAutowire.class));
            autowire = super.determineClass_Autowire(
                autowireAnnotation.value().toLowerCase());
        }

        //The default lazyGen is false
        boolean lazyGeneration = false;
        if(beanClass.isAnnotationPresent(Lazy.class)){
            lazyGeneration = true;
        }

        //Searches for init and destroy
        String initMethod = null;
        String destroyMethod = null;

        //Travel by every method
        for(Method method : beanClass.getDeclaredMethods()){

            //If there is @Init
            if(method.isAnnotationPresent(Init.class)){
                if (initMethod == null) {
                    initMethod = method.getName();
                } else {
                    try {
                        throw new AnnotationsBeanReaderException
                            ("AnnotationsReader error: The '@Init
                            ' in the 'bean' "+ this.currID + "
                            was not recognized. It has more than
                            a definition");
                    } catch (AnnotationsBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
            }

            //If there is @Destroy
            if(method.isAnnotationPresent(Destroy.class)){
                if(destroyMethod == null) {
                    destroyMethod = method.getName();
                } else {
                    try {
                        throw new AnnotationsBeanReaderException
                            ("AnnotationsReader error: The '
                            @Destroy' in the 'bean' "+ this.
                            currID + " was not recognized. It has
                             more than a definition");
                    } catch (AnnotationsBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
```

```java
            }

        }
        this.beanCreator.createBean(this.currID, beanClass.getName(), scope, initMethod, destroyMethod, lazyGeneration, autowire);
    }

    /**
     * Reads the annotations of a constructor, if any.
     * @param beanClass the class to search
     */
    private void readBeanConstructor (Class beanClass) {
        boolean constructorAlreadyMatched = false;
        for (Constructor constructor : beanClass.getDeclaredConstructors()) {

            //If there is @Constructor
            if (constructor.isAnnotationPresent(com.ci1330.ecci.ucr.ac.cr.annotations.Constructor.class)) {

                if (constructorAlreadyMatched) {
                    try {
                        throw new AnnotationsBeanReaderException("Annotations Reader error: The '@Constructor' in the 'bean' " + this.currID + " was not recognized. The constructor has more than a definition");
                    } catch (AnnotationsBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
                constructorAlreadyMatched = true;

                //If there is @Parameter
                if (constructor.isAnnotationPresent(Parameter.class)) {

                    //Travel by every annotation in the constructor
                    for (Annotation annotation : constructor.getDeclaredAnnotations()) {
                        if (annotation.annotationType() == Parameter.class) {

                            String paramType = ((Parameter) annotation).type();
```

```java
if (paramType.equals("")) {
    paramType = null;
}

int index = ((Parameter) annotation)
    .index();

String value = ((Parameter)
    annotation).value();
if (value.equals("")) {
    value = null;
}

String beanRef = ((Parameter)
    annotation).ref();
if (beanRef.equals("")) {
    beanRef = null;
}

final boolean refTypeCombination =
    paramType != null & beanRef !=
    null && value == null;
final boolean valueTypeCombination =
     paramType != null && value !=
    null && beanRef == null;

//Check if the combinations are
    valid
if ( refTypeCombination ||
    valueTypeCombination ) {
    this.beanCreator.
        registerConstructorParameter(
        paramType, index, value,
        beanRef, AutowireEnum.none);
} else {
    try {
        throw new
            AnnotationsBeanReaderException
            ("Annotations␣Reader␣
            error:␣The␣'@Parameter'␣
            was␣not␣recognized␣in␣the
            ␣'bean'␣" + this.currID +
             ".␣It␣has␣an␣illegal␣
            value,␣ref␣and␣type␣
            combination.");
    } catch (
        AnnotationsBeanReaderException
         e) {
        e.printStackTrace();
```

```
                                    System.exit(1);
                                }
                            }

                        }
                    }

                }
            } // if (constructor.isAnnotationPresent(com.ci1330.
                ecci.ucr.ac.cr.annotations.Constructor.class))
            else if (constructor.isAnnotationPresent(
                AtomicAutowire.class)){

                if (constructorAlreadyMatched) {
                    try {
                        throw new AnnotationsBeanReaderException
                            ("Annotations␣Reader␣error:␣The␣'
                            @AtomicAutowire'␣in␣the␣'bean'␣" +
                            this.currID + "␣was␣not␣recognized.␣
                            The␣constructor␣has␣more␣than␣a␣
                            definition");
                    } catch (AnnotationsBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
                constructorAlreadyMatched = true;

                this.beanCreator.explicitConstructorDefinition(
                    constructor);

            }
        }
    }

    /**
     * Reads the annotations of a specific method, if any.
     * @param beanClass the class to search
     */
    private void readBeanSetter (Class beanClass) {
        //Travel by every field
        for (Field field : beanClass.getDeclaredFields()) {

            //If there is @Attribute
            if (field.isAnnotationPresent(Attribute.class)) {

                String value = field.getAnnotation(Attribute.
                    class).value();
                if (value.equals("")) {
```

```java
                    value = null;
                }

                String ref = field.getAnnotation(Attribute.class).ref();
                if (ref.equals("")) {
                    ref = null;
                }

                //Check if there is value or ref
                if ((ref == null && value != null) || (ref != null && value == null)) {
                    this.beanCreator.registerSetter(field.getName(), value, ref, AutowireEnum.none);
                } else {
                    try {
                        throw new AnnotationsBeanReaderException("Annotations␣Reader␣error:␣The␣'@Attribute'␣was␣not␣recognized␣in␣the␣'bean'␣"+ this.currID + ".␣It␣has␣an␣illegal␣combination␣of␣value␣and␣ref.");
                    } catch (AnnotationsBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
            }
            //The reader will only recognize autowire if an Attribute annotation is not present
            else if (field.isAnnotationPresent(AtomicAutowire.class)) {

                //It is assumed to be the special annotation autowiring
                this.beanCreator.registerSetter(field.getName(), null, null, AutowireEnum.annotation);
            }
        }
    }

}
```

XmlBeanReader

```java
package com.ci1330.ecci.ucr.ac.cr.readers;

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.ParserConfigurationException;

import com.ci1330.ecci.ucr.ac.cr.bean.AutowireEnum;
import com.ci1330.ecci.ucr.ac.cr.bean.Scope;
import com.ci1330.ecci.ucr.ac.cr.exception.XmlBeanReaderException;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.w3c.dom.Node;
import org.w3c.dom.Element;
import org.xml.sax.SAXException;

import java.io.File;
import java.io.IOException;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 * The reader is given a String, and then tries to map it with a XML file
 * and extract the metadata for the BeanCreator
 */
public class XmlBeanReader extends BeanReader {

    /**
     * The annotations reader is used if in the xml file, a read annotations
     * statement is found.
     */
    private String defaultInitMethod; //The init method
    private String defaultDestroyMethod; //The destroy method
    private String currID; //The bean ID

    //Init and destroy method tags
    private final String initTag = "init";
    private final String destroyTag = "destroy";

    //Bean properties tags
    private final String idTag = "id";
    private final String classTag = "class";
    private final String scopeTag = "scope";
    private final String autowireTag = "autowire";
    private final String lazyGenerationTag = "lazy-generation";

    //Constructor tags
    private final String constructorTag = "constructor";
    private final String paramTag = "param";
    private final String typeTag = "type";
    private final String indexTag = "index";

    //Constructor tags
    private final String nameTag = "name";

    //Constructor and Attribute tags
    private final String valueTag = "value";
    private final String beanRefTag = "ref";
    private final String atomic_autowireTag = "atomic-autowire";


    /**
     * Constructor, receives the {@link BeanFactory} that
         created him
     * @param beanFactory the father {@link BeanFactory}
     */
    public XmlBeanReader(BeanFactory beanFactory){
        super(beanFactory);
    }

    /**
     * Receives the name of the XML and creates the root
     * @param inputName the name of the XML file
     */
    @Override
    public void readBeans(String inputName) {

        final String beanTag = "bean";

        File fXmlFile = new File(inputName);
        DocumentBuilderFactory dbFactory =
            DocumentBuilderFactory.newInstance();

        DocumentBuilder dBuilder = null;
        try {
            dBuilder = dbFactory.newDocumentBuilder();
        } catch (ParserConfigurationException e) {
            e.printStackTrace();
            System.exit(1);
        }

        Document doc = null;
        try {
            doc = dBuilder.parse(fXmlFile);
        } catch (SAXException | IOException e) {
            e.printStackTrace();
            System.exit(1);
        }

```

```java
        doc.getDocumentElement().normalize();
        Element rootElement = this.readRoot(doc);

        //Get all the beans in the XML
        NodeList nodeList = rootElement.getElementsByTagName(beanTag);

        //Travel by every bean
        for (int index = 0; index < nodeList.getLength(); index++) {

            Node node = nodeList.item(index);

            //Check if it is an Element
            if ((node.getNodeType() == Node.ELEMENT_NODE)) {

                Element beanElement = (Element) node;
                this.readBeanProperties(beanElement);
                this.readBeanConstructor(beanElement);
                this.readBeanAttributes(beanElement);
                super.beanCreator.addBeanToContainer();

            } else {
                try {
                    throw new XmlBeanReaderException("Xml␣Reader␣Error:␣A␣'bean'␣was␣not␣recognized.");
                } catch (XmlBeanReaderException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
            }
        }
        this.readAnnotationsStatement(rootElement);
    }

    /**
     * Starts reading the root of the xml
     * @param xmlRootFile the root of the file
     * @return rootElement
     */
    private Element readRoot(Document xmlRootFile) {
        final String beansTag = "beans";

        Element rootElement = xmlRootFile.getDocumentElement();
        //Check if there is a correct root
        if (rootElement.getTagName().equals(beansTag)) {
            //Check if there is an init property in the root
            if (rootElement.hasAttribute(this.initTag)) {
                //Check if there is an init method in the root
                if (!rootElement.getAttribute(this.initTag).equals("")) {
                    this.defaultInitMethod = rootElement.getAttribute(this.initTag);
                }
            }
            //Check if there is a destroy property in the root
            if (rootElement.hasAttribute(this.destroyTag)) {
                //Check if there is a destroy method
                if (!rootElement.getAttribute(this.destroyTag).equals("")) {
                    this.defaultDestroyMethod = rootElement.getAttribute(this.destroyTag);
                }
            }
        } else {
            try {
                throw new XmlBeanReaderException("Xml␣Reader␣Error:␣The␣root␣of␣the␣XML␣document␣is␣" + rootElement.getTagName() + "␣instead␣of␣'beans'.");
            } catch (XmlBeanReaderException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }
        return rootElement;
    }

    /**
     * Reads the properties of a bean from the bean xml node, any invalid combination or value, throws an exception
     * and exits the program.
     * @param beanElement the XML element of a bean
     */
    private void readBeanProperties(Element beanElement) {

        final String defaultScope = "singleton";
        final String defaultAutowire = "none";
        final String defaultLazyGen = "false";

        //Check if the bean has both ID and class
        if (beanElement.hasAttribute(this.idTag) && beanElement.hasAttribute(this.classTag)) {

            this.currID = beanElement.getAttribute(this.idTag);
            String className = beanElement.getAttribute(this.classTag);
            String initMethod = null;
            String destroyMethod = null;

            //Check if there is an init property in the current bean
            if (beanElement.hasAttribute(this.initTag)) {
                //Check if there is an init method
                if (!beanElement.getAttribute(this.initTag).equals("")) {
                    initMethod = beanElement.getAttribute(this.initTag);
                }
            } else {
                //If not, put the init method as the default one
                initMethod = this.defaultInitMethod;
            }

            //Check if there is an init property in the current bean
            if (beanElement.hasAttribute(this.destroyTag)) {
                //Check if there is a destroy method
                if (!beanElement.getAttribute(this.destroyTag).equals("")) {
                    destroyMethod = beanElement.getAttribute(this.destroyTag);
                }
            } else {
                //If not, put the destroy method as the default one
                destroyMethod = this.defaultDestroyMethod;
            }

            // ____________________________________________________________________

            //Check the scope value
            String scopeString = beanElement.getAttribute(this.scopeTag).toLowerCase();
            if (scopeString.equals("")) {
                scopeString = defaultScope;
            }

            //Check the autowire value
            String autowireString = beanElement.getAttribute(this.autowireTag).toLowerCase();
            if (autowireString.equals("")) {
                autowireString = defaultAutowire;
            }

```

```
            //Get the lazy-generation
            String lazyGenString = beanElement.getAttribute(this.lazyGenerationTag).toLowerCase();
            if (lazyGenString.equals("")) {
                lazyGenString = defaultLazyGen;
            }

            //----------------------------------------------------------------

            AutowireEnum autowire = super.determineClass_Autowire(autowireString);
            Scope scope = super.determineScope(scopeString);
            boolean lazyGeneration = super.determineLazyGen(lazyGenString);

            this.beanCreator.createBean(this.currID, className, scope, initMethod, destroyMethod, lazyGeneration, autowire);

        } //if (beanElement.hasAttribute("id") && beanElement.hasAttribute("class"))
        else {
            try {
                throw new XmlBeanReaderException("Xml␣Reader␣error:␣ID␣and␣Class␣value␣for␣all␣tags␣must␣be␣entered.");
            } catch (XmlBeanReaderException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }
    }

    /**
     * Reads the constructor of a bean from the constructor xml node, any invalid combination or value, throws an exception
     * and exits the program.
     *
     * @param beanElement the XML Element for a bean
     */
    private void readBeanConstructor(Element beanElement) {

        //Get all the constructor in the current bean
        NodeList constructorList = beanElement.getElementsByTagName(this.constructorTag);

```

```java
        //Check if there is more than a constructor definition
        if (constructorList.getLength() > 1) {
            try {
                throw new XmlBeanReaderException("Xml Reader error: Multiple constructors tags in bean " + this.currID + ".");
            } catch (XmlBeanReaderException e) {
                e.printStackTrace();
                System.exit(1);
            }
        } else if (constructorList.getLength() > 0) {

            Element constructorElement = (Element) constructorList.item(0);
            NodeList constructorArgs = constructorElement.getElementsByTagName(this.paramTag);

            //Travel every param
            for (int index = 0; index < constructorArgs.getLength(); index++) {
                Node parameterNode = constructorArgs.item(index);

                //Check if it is an Element so we can cast it
                if (parameterNode.getNodeType() == Node.ELEMENT_NODE) {
                    Element parameterElement = (Element) parameterNode;

                    //Combination of only type and atomic-autowire tag
                    final boolean autowireByTypeCombination = parameterElement.hasAttribute(this.typeTag)  && parameterElement.hasAttribute(this.atomic_autowireTag)
                            && !(parameterElement.hasAttribute(this.beanRefTag)) && !(parameterElement.hasAttribute(this.valueTag));

                    //Combination of only beanRef and atomic-autowire tag
                    final boolean autowireByNameCombination = parameterElement.hasAttribute(this.beanRefTag)  && parameterElement.hasAttribute(this.atomic_autowireTag)
                            && !(parameterElement.hasAttribute(this.typeTag)) && !(parameterElement.hasAttribute(
                    this.valueTag));

        //Combination of only type and beanRef
        final boolean typeRefCombination =
            parameterElement.hasAttribute(this.
            typeTag) && parameterElement.hasAttribute
            (this.beanRefTag)
                                            && !(
                                                parameterElement
                                                .
                                                hasAttribute
                                                (this
                                                .
                                                valueTag
                                                )) &&
                                                 !(
                                                parameterElement
                                                .
                                                hasAttribute
                                                (this
                                                .
                                                atomic_autowireTag
                                                ));

        //Combination of only type and value
        final boolean typeValueCombination =
            parameterElement.hasAttribute(this.
            typeTag) &&  parameterElement.
            hasAttribute(this.valueTag)
                && !(parameterElement.hasAttribute(
                    this.beanRefTag)) && !(
                    parameterElement.hasAttribute(
                    this.atomic_autowireTag));

        //Check if any combination matches
        if ( autowireByTypeCombination ||
            autowireByNameCombination ||
            typeRefCombination ||
            typeValueCombination ) {

             int argIndex = -1;
             try {
                 //Tries to get the index if it
                     exists
                 if ( parameterElement.hasAttribute(
                     this.indexTag)) {
                      if (!parameterElement.
                          getAttribute(this.indexTag).
                          equals("")) {
```

```
                                    argIndex = Integer.parseInt(
                                        parameterElement.
                                        getAttribute(this.
                                        indexTag));
                                } else {
                                    throw new
                                        XmlBeanReaderException("
                                        XML Reader Error: An 
                                        invalid value was entered
                                         in index tag.");
                                }
                            }

                        } catch (NumberFormatException |
                            XmlBeanReaderException e) {
                            e.printStackTrace();
                            System.exit(1);
                        }

                        //If nothing was specified put it to
                            null
                        String type = parameterElement.
                            getAttribute(this.typeTag);
                        if (type.equals("")) {
                            type = null;
                        }

                        //If nothing was specified put it to
                            null
                        String value = parameterElement.
                            getAttribute(this.valueTag);
                        if (value.equals("")) {
                            value = null;
                        }

                        //If nothing was specified put it to
                            null
                        String ref = parameterElement.
                            getAttribute(this.beanRefTag);
                        if (ref.equals("")) {
                            ref = null;
                        }

                        //If nothing was specified put it to
                            none
                        String atomic_autowireString =
                            parameterElement.getAttribute(this.
                            atomic_autowireTag).toLowerCase();
                        if (atomic_autowireString.equals("")) {
```

```java
                            atomic_autowireString = "none";
                        }
                        AutowireEnum atomic_autowire = super.determineAtomic_Autowire(atomic_autowireString);

                        this.beanCreator.registerConstructorParameter(type, argIndex, value, ref, atomic_autowire);

                    } else {
                        try {
                            throw new XmlBeanReaderException("Xml Reader error: A 'param' has an invalid tag combination, in bean " + this.currID + ".");
                        } catch (XmlBeanReaderException e) {
                            e.printStackTrace();
                            System.exit(1);
                        }
                    }
                } else {
                    try {
                        throw new XmlBeanReaderException("Xml Reader error: A 'param' was not recognized in the 'bean' " + this.currID + ".");
                    } catch (XmlBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }

            }
        }

    }

    /**
     * Reads an attribute of a bean from the attribute xml node, any invalid combination or value, throws an exception
     * and exits the program.
     *
     * @param beanElement the XML Element for a bean.
     */
    private void readBeanAttributes(Element beanElement) {

        NodeList attributeList = beanElement.
```

```
            getElementsByTagName("attribute");
        for (int index = 0; index < attributeList.getLength();
            index++) {
            Node attributeNode = attributeList.item(index);

            //Check if it is an Element
            if (attributeNode.getNodeType() == Node.ELEMENT_NODE
                ) {

                Element attributeElement = (Element)
                    attributeNode;

                //Combination of only name and Value tag
                final boolean nameValueCombination =
                    attributeElement.hasAttribute(this.nameTag)
                    && attributeElement.hasAttribute(this.
                    valueTag)
                        && !(attributeElement.hasAttribute(this.
                            beanRefTag));

                //Combination of only name and Ref tag
                final boolean nameRefCombination =
                    attributeElement.hasAttribute(this.nameTag)
                    && attributeElement.hasAttribute(this.
                    beanRefTag)
                        && !(attributeElement.hasAttribute(this.
                            valueTag));

                //Combination of only name and autowire tag
                final boolean atomicAutowireCombination =
                    attributeElement.hasAttribute(this.nameTag)
                    && attributeElement.hasAttribute(this.
                    atomic_autowireTag)
                        && !(attributeElement.hasAttribute(this.
                            beanRefTag)) && !(attributeElement.
                            hasAttribute(this.valueTag));

                //Check if any combination matches
                if ( nameValueCombination || nameRefCombination
                    || atomicAutowireCombination ) {

                    //If the name is empty, throw an exception
                    String name = attributeElement.getAttribute(
                        this.nameTag);
                    if (name.equals("")) {
                        try {
                            throw new XmlBeanReaderException("
                                Xml Reader error: An 'attribute' 
                                has a null name in bean "+this.
                                currID + ".");
                        } catch (XmlBeanReaderException e) {
                            e.printStackTrace();
                            System.exit(1);
                        }
                    }

                    //If the value is empty put it to null
                    String value = attributeElement.getAttribute
                        (this.valueTag);
                    if (value.equals("")) {
                        value = null;
                    }

                    //If the value is empty put it to null
                    String beanRef = attributeElement.
                        getAttribute(this.beanRefTag);
                    if (beanRef.equals("")) {
                        beanRef = null;
                    }

                    //If nothing was specified, put it to none
                    String atomic_autowireString =
                        attributeElement.getAttribute(this.
                        atomic_autowireTag).toLowerCase();
                    if (atomic_autowireString.equals("")) {
                        atomic_autowireString = "none";
                    }
                    AutowireEnum atomic_autowire = super.
                        determineAtomic_Autowire(
                        atomic_autowireString);

                    this.beanCreator.registerSetter(name, value,
                         beanRef, atomic_autowire);

                } else {
                    try {
                        throw new XmlBeanReaderException("Xml
                            Reader error: The 'attribute' must
                            have 'name' and 'value' or 'name' and
                            'ref' in bean "+this.currID + ".");
                    } catch (XmlBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }

            } else {
                try {
```

```
                    throw new XmlBeanReaderException("Xml␣Reader
                        ␣error:␣An␣'attribute'␣was␣not␣recognized
                        ␣in␣the␣'bean'␣" + this.currID +".");
                } catch (XmlBeanReaderException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
            }

        }

    }

    /**
     * The method tells the annotationsBeanReader to read a
        specific class. If it has more than one tag, exits
        abnormally.
     *
     * @param beanElement the XML Element for a bean
     */
    private void readAnnotationsStatement(Element beanElement) {

        final String annotationClassesTag = "annotationsClasses"
            ;
        final String pathTag = "path";

        //Get all the annotationsClasses in the root
        NodeList annotationsList = beanElement.
            getElementsByTagName(annotationClassesTag);

        //Check if there is more than a annotationClasses
            definition
        if (annotationsList.getLength() > 1) {
            try {
                throw new XmlBeanReaderException("Xml␣Reader␣
                    error:␣'annotationClasses'␣has␣more␣than␣one␣
                    definition.");
            } catch (XmlBeanReaderException e) {
                e.printStackTrace();
                System.exit(1);
            }
        } else if(annotationsList.getLength() > 0){

            //Create a new Annotations reader with the same
                creator of this factory.
            AnnotationsBeanReader annotationsBeanReader = new
                AnnotationsBeanReader(super.beanCreator);
            Element annotationsElement = (Element)
                annotationsList.item(0);
```

```java
            NodeList classList = annotationsElement.getElementsByTagName(this.classTag);

            //Travel by every class
            for(int index = 0; index < classList.getLength(); ++index){
                Node classNode = classList.item(index);

                //Check if it is an Element
                if (classNode.getNodeType() == Node.ELEMENT_NODE) {
                    Element classElement = (Element) classNode;

                    //If the annotation has a path
                    if (!(classElement.getAttribute(pathTag).equals(""))) {
                        annotationsBeanReader.readBeans(classElement.getAttribute(pathTag));

                    } else {

                        try {
                            throw new XmlBeanReaderException("Xml Reader error: A class in 'annotationClasses' doesn't have a 'path'");
                        } catch (XmlBeanReaderException e) {
                            e.printStackTrace();
                            System.exit(1);
                        }

                    }
                } else {
                    try {
                        throw new XmlBeanReaderException("Xml Reader error: A 'class' in 'annotationClasses' was not recognized");
                    } catch (XmlBeanReaderException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
            }
        }
    }

}
```

AtomicAutowire

```java
package com.ci1330.ecci.ucr.ac.cr.annotations;

import com.ci1330.ecci.ucr.ac.cr.bean.AutowireEnum;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.CONSTRUCTOR})
public @interface AtomicAutowire {

}
```

Attribute

```java
package com.ci1330.ecci.ucr.ac.cr.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Attribute {
    String value() default "";
    String ref() default "";
}
```

Bean

```java
package com.ci1330.ecci.ucr.ac.cr.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
```

```
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Bean {
    String value();
}
```

ClassAutowire

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

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface ClassAutowire {
    String value() default "byname";
}
```

Constructor

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


import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CONSTRUCTOR)
public @interface Constructor {
}
```

Controller

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

```java

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Controller {
    String value();
}
```

Destroy

```java
package com.ci1330.ecci.ucr.ac.cr.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Destroy {
}
```

Init

```java
package com.ci1330.ecci.ucr.ac.cr.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Init {
}
```

Lazy

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

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Lazy {
}
```

Parameter

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

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.CONSTRUCTOR)
public @interface Parameter {
    String type();
    int index() default -1;
    String value() default "";
    String ref() default "";
}
```

Scope

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

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

```java

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Scope {
    String value() default "singleton";
}
```

Service

```java
package com.ci1330.ecci.ucr.ac.cr.annotations;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 17/09/2017
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Service {
    String value();
}
```

AnnotationsFactory

```java
package com.ci1330.ecci.ucr.ac.cr.factory;

import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
import com.ci1330.ecci.ucr.ac.cr.readers.AnnotationsBeanReader;

import java.util.HashMap;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * AnnotationsFactory class which inherits from BeanFactory
 * and registers the Annotations classes from which the
   configuration
 * must be read and tells the reader to parse it.
 */
public class AnnotationsFactory extends BeanFactory{

    private AnnotationsBeanReader annotationsBeanReader;    //
       Instance of the annotations reader

    /**
     * Constructor of the class, it initializes the super-class
        attributes and
     * also the annotations bean reader.
     */
    public AnnotationsFactory() {
        super();
        annotationsBeanReader = new AnnotationsBeanReader(this);
    }

    /**
     * Constructor of the class, it initializes the super-class
        attributes and
     * also the annotations bean reader. It receives the path of
         a class which
     * holds annotations configurations for the reader to parse
        it.
     * @param classConfig the name of the class to use
     */
    public AnnotationsFactory(String classConfig) {
        super();
        annotationsBeanReader = new AnnotationsBeanReader(this);
        this.registerConfig(classConfig);
    }

    /**
     * Allows the user to register more configurations
     * classes later, indicating their path.
     * @param classConfig the name of the class to use
     */
    public void registerConfig(String classConfig){
        annotationsBeanReader.readBeans(classConfig);
        super.initContainer();
    }

    /**
     * Return a bean instance from the super class.
     * @param id the beanId
     * @return the bean instance
     */
    @Override
    public Object getBean(String id) {
        return super.getBean(id);
    }

    /**
```

```
     * Adds a bean to the container
     * @param bean the {@link Bean} class
     */
    @Override
    public void addBean(Bean bean) {
        super.addBean(bean);
    }

    /**
     * Calls the super method for shutDownHook
     */
    @Override
    public void shutDownHook() {
        super.shutDownHook();
    }

    // ------------------------------------------------------------------------
    // Standard Setters and Getters section
    // ------------------------------------------------------------------------

    @Override
    public HashMap<String, Bean> getBeansMap() {
        return super.getBeansMap();
    }

    @Override
    public void setBeansMap(HashMap<String, Bean> beansMap) {
        super.setBeansMap(beansMap);
    }

    public AnnotationsBeanReader getAnnotationsBeanReader() {
        return annotationsBeanReader;
    }

    public void setAnnotationsBeanReader(AnnotationsBeanReader
        annotationsBeanReader) {
        this.annotationsBeanReader = annotationsBeanReader;
    }
}
```

BeanConstructorModule

```
package com.ci1330.ecci.ucr.ac.cr.factory;

import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
import com.ci1330.ecci.ucr.ac.cr.bean.BeanParameter;
```

```java
import com.ci1330.ecci.ucr.ac.cr.exception.
    BeanConstructorConflictException;
import com.ci1330.ecci.ucr.ac.cr.exception.
    BeanConstructorNotFoundException;

import java.lang.reflect.Constructor;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 28/09/2017
 */
public class BeanConstructorModule {

    /**
     * Checks if all parameters have an index assigned. If at
         least one doesn't, it returns false.
     * @return true if all indexes were assigned, false if not
     */
    private static boolean checkParametersIndexes(Bean bean){
        boolean allIndexesAssigned = true;
        int paramListIndex = 0;
        BeanParameter beanParameter;
        while(paramListIndex < bean.getBeanConstructor().
            getBeanParameterList().size()
                && allIndexesAssigned){
            beanParameter = bean.getBeanConstructor().
                getBeanParameterList().get(paramListIndex);
            if(beanParameter.getIndex() == -1){
                allIndexesAssigned = false;
            }
            paramListIndex++;
        }
        return allIndexesAssigned;
    }

    /**
     * Searches for the Class types of each parameter in the
         bean's constructor.
     * @param bean the bean to use
     * @return an array containing the Class types of each
         parameter
     */
    private static Class[] obtainParametersClassArray(Bean bean)
        {
        String parameterClass = null;
        String beanParameterType;
        Class param = null;
        Class[] parametersClassArray = new Class[bean.
            getBeanConstructor().getBeanParameterList().size()];
        for (BeanParameter p : bean.getBeanConstructor().
            getBeanParameterList()) {
            beanParameterType = p.getExplicitTypeName();
            if(beanParameterType == null){
                beanParameterType = p.getBeanFactory().findBean(
                    p.getBeanRef()).getClass().toString();
            }
            switch (beanParameterType) {
                case "int":
                    param = int.class;
                    break;
                case "byte":
                    param = byte.class;
                    break;
                case "short":
                    param = short.class;
                    break;
                case "long":
                    param = long.class;
                    break;
                case "float":
                    param = float.class;
                    break;
                case "double":
                    param = double.class;
                    break;
                case "boolean":
                    param = boolean.class;
                    break;
                case "char":
                    param = char.class;
                    break;
                default:
                    parameterClass = p.getExplicitTypeName();
                    try {
                        param = Class.forName(parameterClass);
                    } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                    }
                    break;
            }

            parametersClassArray[p.getIndex()] = param;
        }
        return parametersClassArray;
    }

    /**
     * Compares the bean's constructor parameter type with the
```

```
        bean's class constructor parameter
     * type. If they match, it assigns the respective index to
        the bean's constructor parameter.
     * The switch is needed for primitive types checking and
        casting.
     * @param beanParameter bean's constructor parameter
     * @param beanClassConstructorParameter bean's class
        constructor parameter
     * @param paramIndex current index of the bean's class
        constructor parameter
     * @return True if parameters matched
     */
    private static boolean setBeanParameterIndex(BeanParameter
       beanParameter, Class beanClassConstructorParameter, int
       paramIndex){
        boolean parametersMatched = false;
        String beanParameterType = beanParameter.
           getExplicitTypeName();
        if(beanParameterType == null){
            beanParameterType = beanParameter.getBeanFactory().
               findBean(beanParameter.getBeanRef()).getClass().
               toString();
        }
        switch (beanParameterType) {
            case "int":

                if (beanClassConstructorParameter.toString().
                   equals("int")) {
                    parametersMatched = true;
                    beanParameter.setIndex(paramIndex);
                }
                break;
            case "java.lang.Integer":

                if (beanClassConstructorParameter.toString().
                   equals("int")) {
                    parametersMatched = true;
                    beanParameter.setIndex(paramIndex);
                }
                break;
            case "byte":
                if (beanClassConstructorParameter.toString().
                   equals("byte")) {
                    parametersMatched = true;
                    beanParameter.setIndex(paramIndex);
                }
                break;
            case "java.lang.Byte":
                if (beanClassConstructorParameter.toString().
                    equals("byte")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "short":
                if (beanClassConstructorParameter.toString().
                    equals("short")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "java.lang.Short":
                if (beanClassConstructorParameter.toString().
                    equals("short")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "long":
                if (beanClassConstructorParameter.toString().
                    equals("long")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "java.lang.Long":
                if (beanClassConstructorParameter.toString().
                    equals("long")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "float":
                if (beanClassConstructorParameter.toString().
                    equals("float")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "java.lang.Float":
                if (beanClassConstructorParameter.toString().
                    equals("float")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "double":
                if (beanClassConstructorParameter.toString().
                    equals("double")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "java.lang.Double":
                if (beanClassConstructorParameter.toString().
                    equals("double")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "boolean":

                if (beanClassConstructorParameter.toString().
                    equals("boolean")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);

                }
                break;
            case "java.lang.Boolean":

                if (beanClassConstructorParameter.toString().
                    equals("boolean")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "char":
                if (beanClassConstructorParameter.toString().
                    equals("char")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            case "java.lang.Character":
                if (beanClassConstructorParameter.toString().
                    equals("char")) {
                     parametersMatched = true;
                     beanParameter.setIndex(paramIndex);
                }
                break;
            default:

                if(beanParameter.getExplicitTypeName() == null
                   &&
                        beanParameter.getBeanFactory().findBean(
                            beanParameter.getBeanRef()).
```

```
                            getBeanClass().equals(
                            beanClassConstructorParameter)){
                    parametersMatched = true;
                    beanParameter.setIndex(paramIndex);
                    } else try {
                        if(Class.forName(beanParameter.
                            getExplicitTypeName()).equals(
                            beanClassConstructorParameter)){
                            parametersMatched = true;
                            beanParameter.setIndex(paramIndex);
                        }
                    } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                    }
                    break;
            }
            return parametersMatched;
    }

    /**
     * Sets the constructor method and parameters to a bean for
         it to be ready
     * to be autowired and injected. If indexes are not
         specified, the method checks
     * all constructors of the bean's class to match one.
     * @param bean the bean to use
     */
    public static void registerConstructor(Bean bean) {
        Constructor matchedConstructor = null;

        if(!checkParametersIndexes(bean)) { //Checks if at least
             one parameter doesn't have an index assigned
            int totalParametersOneType = 0;
            int totalParametersMatched = 0;
            int constructorMatches = 0;
            int paramIndex = 0;
            boolean twoMatchesForOneParam = false;

            Constructor[] classConstructors = bean.getBeanClass
                ().getDeclaredConstructors();
            Class[] classConstructorParameters;

            for (Constructor classConstructor :
                classConstructors) {        // Iterates through
                all constructors in the bean's class

                classConstructorParameters = classConstructor.
                    getParameterTypes();

                    if (classConstructorParameters.length == bean.
                        getBeanConstructor().getBeanParameterList().
                        size()) { // Checks if the current class
                        constructor has same amount of parameters



                        for (BeanParameter beanParameter : bean.
                            getBeanConstructor().getBeanParameterList
                            ()) {       // Iterates through all the
                            declared parameters in the configuration

                            for (Class parameter :
                                classConstructorParameters) {
                                                                        //
                                 Iterates through all the parameters
                                of the current class constructor
                                if(setBeanParameterIndex(
                                    beanParameter, parameter,
                                    paramIndex)){ // Compares the
                                    parameters and assigns an index
                                    to the bean's constructor
                                    parameter if they matched
                                    totalParametersOneType++;
                                    totalParametersMatched++;
                                }
                                paramIndex++;
                            }

                            paramIndex = 0;
                            if (totalParametersOneType > 1) {
                                twoMatchesForOneParam = true;
                            }
                            totalParametersOneType = 0;
                        }

                        if (totalParametersMatched == bean.
                            getBeanConstructor().getBeanParameterList
                            ().size() && !twoMatchesForOneParam) {
```

```
                    constructorMatches++;
                    matchedConstructor = classConstructor;
                }
                totalParametersMatched = 0;
            }
            twoMatchesForOneParam = false;
        }
        if (constructorMatches == 0) {
            try {
                throw new BeanConstructorNotFoundException("
                    Bean creation error: constructor not 
                    found for the specified parameters in 
                    bean: " + bean.getId() + ".");
            } catch (BeanConstructorNotFoundException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }
        if (constructorMatches > 1) {
            try {
                throw new BeanConstructorConflictException("
                    Bean creation error: there are multiple 
                    constructors for the specified parameters
                     in bean: " + bean.getId() +
                        ". Couldn't identify which one is 
                            intended to be called (same 
                            parameter quantity and types).");
            } catch (BeanConstructorConflictException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }
    }
    else{   // All parameters specified in the configuration
         have indexes assigned.
        try {
            matchedConstructor = bean.getBeanClass().
                getConstructor(obtainParametersClassArray(
                bean));
        } catch (NoSuchMethodException e) {
            System.err.println("Bean creation error: 
                constructor not found for the specified 
                parameters in bean: " + bean.getId() + ".");
            e.printStackTrace();
            System.exit(1);
        }
    }
    bean.getBeanConstructor().setConstructorMethod(
        matchedConstructor);      // sets the Constructor to
```

```
            the bean, ready to be autowired and injected
    }
}
```

BeanCreator

```
package com.ci1330.ecci.ucr.ac.cr.factory;

import com.ci1330.ecci.ucr.ac.cr.bean.*;
import com.ci1330.ecci.ucr.ac.cr.exception.*;

import java.lang.reflect.Constructor;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;

import com.ci1330.ecci.ucr.ac.cr.bean.Bean;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Module in charge of receiving each bean's metadata
 * from the reader and creating the bean, with all
 * the properties it needs for it to be instantiated
 * later.
 */
public class BeanCreator {

    // Classes needed to create the bean
    private Bean bean;
    private BeanFactory beanFactory;
    private BeanAttribute attributeClass;
    private BeanConstructor beanConstructorTemp;

    /**
     * Constructor of the class which receives the beanFactory
        and
     * assigns it for later use.
     * @param beanFactory the factory to add beans
     */
    public BeanCreator(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    /**
     * Method which receives the basic IoC properties for the
        bean
     * and creates it.
     * @param id the bean's ID
     * @param beanClass the bean's class
     * @param scope the bean's scope
     * @param initMethodName the bean's init method name
     * @param destroyMethodName the bean's destroy method name
     * @param lazyGen the bean's lazy generation value
     * @param autowireEnum the bean's autowire mode
     */
    public void createBean(String id, String beanClass, Scope scope, String initMethodName, String destroyMethodName, boolean lazyGen, AutowireEnum autowireEnum) {
        try {
            if (this.beanFactory.containsBean(id)) {
                throw new RepeatedIdException("Creation error: Bean id " + id + " is repeated.");
            }
        } catch (RepeatedIdException r) {
            r.printStackTrace();
            System.exit(1);
        }

        bean = new Bean(this.beanFactory);
        bean.setId(id);
        try {
            bean.setBeanClass(Class.forName(beanClass)); // Sets the beans type
        } catch (ClassNotFoundException e) {
            System.err.println("Creation error: bean class not found for bean: " + id + ".");
            e.printStackTrace();
            System.exit(1);
        }
        bean.setBeanScope(scope);
        Method initMethod = null;
        Method destroyMethod = null;
        Method[] beanMethods = this.bean.getBeanClass().getDeclaredMethods();
        for (Method method : beanMethods) {
            if(Modifier.isPrivate(method.getModifiers())){
                method.setAccessible(true);
            }
            if (initMethodName != null && method.getName().contains(initMethodName)) {       //Finds the initialization and destruction methods for the bean
                if (method.getParameterCount() == 0) {
                    initMethod = method;
                }
            }
            if (destroyMethodName != null && method.getName().
                contains(destroyMethodName)) {
                if (method.getParameterCount() == 0) {
                    destroyMethod = method;
                }
            }
        }
        bean.setInitMethod(initMethod);
        bean.setDestroyMethod(destroyMethod);
        bean.setLazyGen(lazyGen);
        bean.setAutowireEnum(autowireEnum);
        this.beanConstructorTemp = new BeanConstructor(null);
            // Creates a temporary constructor to receive the
            parameters of the bean
    }

    /**
     * Method that returns an object after
     * casting the string value to its real type.
     * @param stringValue a string that contains the value
     * @return object with respective type
     */
    private Object obtainValueType(String stringValue) {
        boolean parsed = false;
        Object value = null;
        try {
            value = Integer.valueOf(stringValue);        // It
                tries to cast the string to the stated types and
                if not proceeds to the next one
            parsed = true;
        } catch (NumberFormatException e) {
            //No es un int.
        }
        if (!parsed) {
            try {
                value = Byte.valueOf(stringValue);
                parsed = true;
            } catch (NumberFormatException e) {
                //No es un byte.
            }
        }
        if (!parsed) {
            try {
                value = Short.valueOf(stringValue);
                parsed = true;
            } catch (NumberFormatException e) {
                //No es un byte.
            }
        }
        if (!parsed) {
            try {
                value = Long.valueOf(stringValue);
                parsed = true;
            } catch (NumberFormatException e) {
                //No es un byte.
            }
        }
        if (!parsed) {
            try {
                value = Float.valueOf(stringValue);
                parsed = true;
            } catch (NumberFormatException e) {
                //No es un byte.
            }
        }
        if (!parsed) {
            try {
                value = Double.valueOf(stringValue);
                parsed = true;
            } catch (NumberFormatException e) {
                //No es un byte.
            }
        }
        if (!parsed) {
            if ((stringValue.toLowerCase()).equals("true")) {
                value = true;
                parsed = true;
            } else if ((stringValue.toLowerCase()).equals("false
               ")) {
                value = false;
                parsed = true;
            }
        }
        if (stringValue.length() == 1 && !parsed) {
            try {
                value = stringValue.charAt(0);
                parsed = true;
            } catch (Exception e) {
                //No es un char.
            }
        }
        if (!parsed) {
            value = stringValue;
        }
        return value;
    }

    /**
     *
     * Method to register an attribute of the bean and find
     * its setter method to be used later when injecting
     * the bean's dependencies
     * @param attributeName the name of the attribute to
       register
     * @param stringValue a string with the attribute's value
     * @param beanRef a string with the attribute's bean
       reference
     * @param atomic_autowire the atomic autowiring mode
     */
    public void registerSetter(String attributeName, String
       stringValue, String beanRef, AutowireEnum atomic_autowire
       ){
        try {
            if (this.beanFactory.containsBean(beanRef)) {
                throw new RepeatedIdException("Creation error: 
                    Bean attribute with reference to: " + beanRef
                     + " is repeated.");
            }
        } catch (RepeatedIdException r) {
            r.printStackTrace();
            System.exit(1);
        }

        Object value = null;
        if(stringValue != null){
            value = this.obtainValueType(stringValue);
        }

        Method setterMethod = null;
        Method[] beanMethods = this.bean.getBeanClass().
            getDeclaredMethods();
        Class beanRefType = null;

        for(Method method: beanMethods){
            if(Modifier.isPrivate(method.getModifiers())){
                method.setAccessible(true);
            }

            // Checks if the method is the respective setter for
                this attribute
            if(method.getName().startsWith("set") && method.
                getName().toLowerCase().contains(attributeName.
                toLowerCase())){
                if(method.getParameterCount() == 1){
                    setterMethod = method;
                }
            }
        }

        if(setterMethod == null){
            try {
                throw new SetterMethodNotFoundException("
                    Creation error: Bean attribute's setter 
                    method not found for attribute: " +
                    attributeName + ".");
            } catch (SetterMethodNotFoundException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }

        //If the value is null, the user is using beans, so
            search for the type that the attribute should have
        if(value == null) {
            Field[] beanFields = this.bean.getBeanClass().
                getDeclaredFields();
            for(Field field: beanFields){
                if(Modifier.isPrivate(field.getModifiers())){
                    field.setAccessible(true);
                }
                if(field.getName().equals(attributeName)) {
                    beanRefType = field.getType();
                }
            }
        }

        //If the user specified autowire byName at atomic level,
             the beanRef is the same as the attributeName
        if (beanRef == null && atomic_autowire == AutowireEnum.
            byName) {
            beanRef = attributeName;
        }
        BeanAttribute beanAttribute = new BeanAttribute(beanRef,
             beanRefType, this.beanFactory, value,
            atomic_autowire, setterMethod);
        bean.appendAttribute(beanAttribute);
    }

    /**
     * Method which registers a parameter of the bean's
         constructor
     * @param paramType the name of the parameter's type
     * @param stringValue a string with the parameter's value
     * @param beanRef a string with the parameter's bean
         reference
     * @param atomic_autowire the atomic autowiring mode
     */
```

```java
    public void registerConstructorParameter(String paramType,
        int index, String stringValue, String beanRef,
        AutowireEnum atomic_autowire){
        Object value = null;
        if(stringValue != null){
            value = this.obtainValueType(stringValue);
        }
        if(value == null && beanRef == null && paramType == null
            ){
            try {
                throw new InvalidPropertyException("Bean␣
                    creation␣error:␣parameter's␣type,␣reference␣
                    or␣value␣is␣invalid␣for␣a␣declared␣bean␣
                    parameter.");
            } catch (InvalidPropertyException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }

        Class beanRefClass = null;

        //If the value is null, the user is using beans, so
            search for the type that the parameter should have
        //But because this is a constructor parameter, only
            search for it if we have at least the type
        if (value == null && paramType != null) {
            try {
                beanRefClass = Class.forName(paramType);
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
                System.exit(1);
            }
        }

        BeanParameter beanConstructorParam = new BeanParameter(
            beanRef, beanRefClass, this.beanFactory, value,
            atomic_autowire, index, paramType);
        this.beanConstructorTemp.append(beanConstructorParam);

    }

    /**
     * There is a special case in which an AtomicAutowire
        annotation is found above a constructor
     * In this case, the constructor is already known, but the
        parameters need to be set later.
     * So the Reader sends the constructor explicitly, and it is
         added to the current bean.
     *
     * The method addBeanToContainer won't interfere in this
         definition, because if the user didn't
     * specify another constructor elsewhere, the
         beanConstructorTemp won't be added to the current bean,
     * leaving the explicit definition untouched.
     */
    public void explicitConstructorDefinition (Constructor
        constructorMethod) {
        this.bean.setBeanConstructor(new BeanConstructor(
            constructorMethod));
    }

    /**
     * Adds the bean to the container and resets all its
         attributes
     * for the creator to be ready to read another bean's data
     */
    public void addBeanToContainer(){
        //If there were no parameters specified for the
            constructor, it is assumed the user didn't
        //indicate to use constructor injection
        if (this.beanConstructorTemp.getBeanParameterList().size
            () > 0) {
            this.bean.setBeanConstructor(this.
                beanConstructorTemp);
        }
        this.beanFactory.addBean(this.bean);
        bean = null;
        attributeClass = null;
        beanConstructorTemp = null;
    }

    //
        -----------------------------------------------------------------------
    // Standard Setters and Getters section
    //
        -----------------------------------------------------------------------

    public Bean getBean() {
        return bean;
    }

    public void setBean(Bean bean) {
        this.bean = bean;
    }

    public BeanFactory getBeanFactory() {
        return beanFactory;
    }

    public void setBeanFactory(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    public BeanAttribute getAttributeClass() {
        return attributeClass;
    }

    public void setAttributeClass(BeanAttribute attributeClass)
        {
        this.attributeClass = attributeClass;
    }

}
```

BeanFactory

```
package com.ci1330.ecci.ucr.ac.cr.factory;

import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
import com.ci1330.ecci.ucr.ac.cr.bean.BeanAttribute;
import com.ci1330.ecci.ucr.ac.cr.bean.BeanParameter;
import com.ci1330.ecci.ucr.ac.cr.bean.Scope;
import com.ci1330.ecci.ucr.ac.cr.exception.
    BeanTypeConflictException;
import com.ci1330.ecci.ucr.ac.cr.exception.IdNotFoundException;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * BeanFactory parent class which has the container and manages
 * the control flow of NAIoCC. User's request for beans via an
 * instance of this class.
 */
public abstract class BeanFactory {

    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

```

```java
    private boolean nonFatalCycle;

    /**
     * Constructor of the class, initializes the container
     */
    public BeanFactory(){
        beansMap = new HashMap<>();
    }

    /**
     * Adds a bean to the container before initializing it.
     * @param bean the bean to add
     */
    public void addBean(Bean bean){
        this.beansMap.put(bean.getId(), bean);
    }

    /**
     * Returns the instance of the bean, already injected. If it
         is singleton it
     * returns the only instance, otherwise creates a new one (
        prototype).
     * @param id the bean's id
     * @return the requested bean's instance
     */
    public Object getBean(String id) {
        try {

            if (!this.beansMap.containsKey(id)) {
                throw new IdNotFoundException("Exception error: 
                    The id: " + id + " does not exist.");
            }

            Bean currBean = this.beansMap.get(id);
            if (currBean.getBeanScope() == Scope.Prototype ||
                currBean.getInstance() == null) {

                currBean.createNewInstance(); // Adds the new
                    instance to the bean's list
                currBean.injectDependencies();
                currBean.initialize();

            }

            return currBean.getInstance(); // Returns the last
                instance of the bean's list

        } catch (IdNotFoundException e) {
            e.printStackTrace();
            System.exit(1);
        }

        return null;
    }

    /**
     * Iterates through all beans and checks their scope to
        initialize and inject its dependencies.
     */
    protected void initContainer(){
        for(HashMap.Entry<String,Bean> beanEntry: beansMap.
            entrySet()) {     // Iterates through the container to
             autowire dependencies
            Bean currBean = beanEntry.getValue();
            currBean.autowire();                        // Autowires
                the bean, if indicated as such
            currBean.checkBeanProperties();          // Checks
               there are no conflicts in its properties
        }

        cycleDetection();         // Checks if there a cycles
            between the dependencies of the beans

        for(HashMap.Entry<String,Bean> beanEntry: beansMap.
            entrySet()){      // Iterates through the container to
             initialize beans
            Bean currBean = beanEntry.getValue();

            if(currBean.getBeanScope() == Scope.Singleton && !
                currBean.isLazyGen()   // Instantiates the bean
                only if it is Singleton, without lazy generation
                and haven't been initialized
                    && currBean.getInstance() == null){
                currBean.createNewInstance();
                currBean.injectDependencies();
                currBean.initialize();
            }

        }

    }

    /**
     * Finds a bean by its type for autowiring purposes. If
        there's no bean
     * with this type in the container or if there are more than
         one, it returns null.
     * @param beanType the bean's type
     * @return the Bean with the type requested, null if not
        found
     */
    public Bean findBean(Class beanType) throws
        BeanTypeConflictException {
        Bean bean = null;

        for(HashMap.Entry<String,Bean> beanEntry: beansMap.
            entrySet()){      //Iterates through the container

            if(beanEntry.getValue().getBeanClass().equals(
                beanType)){        //Checks if it is of the
                requested type
                if (bean == null) {
                    bean = beanEntry.getValue();
                } else {
                    throw new BeanTypeConflictException("
                        Injection␣by␣type␣error:␣two␣or␣more␣
                        beans␣share␣the␣same␣type.");
                }
            }

        }

        return bean;
    }

    /**
     * Finds a bean by its name. If there's no bean
     * with this name in the container, it returns null.
     * @param beanId the bean's id
     * @return The bean with the corresponding id, null if it
        wasn't found
     */
    public Bean findBean(String beanId){
        Bean bean = null;
        if(this.beansMap.containsKey(beanId)){
            bean = this.beansMap.get(beanId);
        }
        return bean;
    }

    /**
     * Checks if the specified bean is in the container.
     * @param beanId the bean's id
     * @return true if the bean is in the container, false
        otherwise
     */
    public boolean containsBean(String beanId){
```

```java
        return this.beansMap.containsKey(beanId);
    }

    /**
     * Destroys all beans' instances of the container.
     */
    public void shutDownHook(){
        for(HashMap.Entry<String,Bean> beanEntry: beansMap.
            entrySet()){
            beanEntry.getValue().destroyAllInstances();
        }
    }

    /**
     * Iterates all the references of all the beans and checks
        if there is a cycle
     */
    private void cycleDetection() {
        HashMap< String, List<String> > setterReferences = new
            HashMap<>();
        HashMap< String, List<String> > constructorReferences =
            new HashMap<>();

        for (Map.Entry<String, Bean> currEntry : this.beansMap.
            entrySet()) {
            Bean currBean = currEntry.getValue();

            this.insertConstructorReferences(currBean,
                constructorReferences);
            this.insertSetterReferences(currBean,
                setterReferences);
        }

        //Checks if any of those maps has a cycle
        this.thereIsCycle(constructorReferences, true);
        this.thereIsCycle(setterReferences, false);
    }

    /**
     * Registers the constructor references for a bean
     * @param currBean the bean to search
     * @param constructorReferences a list of the references
     */
    private void insertConstructorReferences(Bean currBean,
        HashMap< String, List<String> > constructorReferences) {
        List<String> referencesList = new ArrayList<>(); //If
            there is no dependency the list will be empty

        //If the bean has a constructor
        if (currBean.getBeanConstructor() != null) {

            //Iterate every parameter that has a reference and put it on the map
            for (BeanParameter currBeanParameter : currBean.getBeanConstructor().getBeanParameterList()) {

                String currReference = currBeanParameter.getBeanRef();
                //If the parameter has a beanRef, append it
                if (currReference != null) {
                    referencesList.add(currReference);
                }
            }

        }

        constructorReferences.put(currBean.getId(), referencesList);
    }

    /**
     * Registers the setter references for a bean
     * @param currBean the bean to search
     * @param setterReferences a list of the references
     */
    private void insertSetterReferences(Bean currBean, HashMap<String, List<String> > setterReferences) {
        List<String> referenceList = new ArrayList<>(); //If there is no dependency the list will be empty

        //Iterate every attribute that has a reference and put it on the map
        for (BeanAttribute currBeanAttribute : currBean.getBeanAttributeList()) {

            String currReference = currBeanAttribute.getBeanRef();
            //If the parameter has a beanRef, append it
            if (currReference != null) {
                referenceList.add(currReference);
            }
        }

        setterReferences.put(currBean.getId(), referenceList);
    }

    /**
     * For every entry in the map, checks the cycles, if there
```

```java
         is an invalid one, the program exits.
     * @param referenceMap all the references for all beans
     * @param isConstructorInjection indicates if it is checking constructor injection or not.
     */
    private void thereIsCycle(HashMap< String, List<String> > referenceMap, boolean isConstructorInjection) {
        List<String> cycleLessReferences = new ArrayList<>(); // References that were already confirmed as cycle-less
        List<String> currentTrail = new ArrayList<>(); //The reference trail

        for (String beanEntry : referenceMap.keySet()) {
            this.nonFatalCycle = false;
            if (checkCycle(beanEntry, referenceMap, currentTrail, cycleLessReferences, isConstructorInjection)) {
                System.err.println("CYCLE DETECTED: A reference or chain of references of " + beanEntry + " causes an invalid cycle.");
                System.exit(1);
            } else if (this.nonFatalCycle) {
                System.err.println("CYCLE DETECTED (WARNING): The cycle is not fatal! But keep track of the cycles...");
            }
        }

    }

    /**
     * Recursively check if a chain of references causes a cycle.
     * @param reference The reference to check
     * @param referenceMap Map of all references
     * @param currentTrail The current trail of the recursive call
     * @param cycleLessReferences The trail of cycle less references, so we don't repeat searches
     * @param isConstructorInjection indicates if it's checking constructor injection.
     * @return true if there was a cycle, false if not.
     */
    private boolean checkCycle(String reference, HashMap< String, List<String> > referenceMap,
                               List<String> currentTrail, List<String> cycleLessReferences, boolean isConstructorInjection) {
        boolean cycleDetected = false;

        if ( cycleLessReferences.contains(reference) || referenceMap.get(reference).isEmpty() )  {
            //If the reference was already checked or doesn't have associated references, there is no cycle
            cycleDetected = false;
        } else if (currentTrail.contains(reference)) {
            //If the dependency was already in the trail, there is a cycle
            //But in setter injection, only a pure prototype cycle causes trouble
            System.err.println("CYCLE DETECTED: Checking if the cycle is fatal...");
            this.nonFatalCycle = true;
            if (isConstructorInjection) {
                cycleDetected = true;
            } else {
                cycleDetected =  checkIfInvalid(currentTrail, reference);
            }

        } else {
            //If the reference has associated references and is not in the trail
            //For every associated reference check if it causes a cycle

            currentTrail.add(reference); //Add the current dependency to the trail

            String associatedReference;
            List<String> associatedReferences = referenceMap.get(reference);

            for (int index = 0; index < associatedReferences.size() && !cycleDetected; index++) {

                associatedReference = associatedReferences.get(index);
                cycleDetected = checkCycle(associatedReference, referenceMap, currentTrail,
                        cycleLessReferences, isConstructorInjection);

            }

            currentTrail.remove(reference); //Remove the current dependency to the trail
        }

        if (!cycleDetected) {
            //If the reference doesn't cause a cycle, register it as cycle-less
            cycleLessReferences.add(reference);
        }

        return cycleDetected;
    }

    /**
     * Checks if the cycle has only prototypes
     * @param trail the trail of the recursive call
     * @return true if illegal cycle, false if not.
     */
    private boolean checkIfInvalid (List<String> trail, String reference) {
        int prototypeCount = 0;
        int referenceCount = 0;

        String dependency;

        //Start from the reference that causes the cycle
        for (int index = trail.indexOf(reference); index < trail.size(); index++) {
            dependency = trail.get(index);
            Bean currBean = this.findBean(dependency);

            if (currBean.getBeanScope() == Scope.Prototype) {
                prototypeCount++;
            }

            referenceCount++;
        }

        return prototypeCount == referenceCount;
    }

    // ---------------------------------------------------------------------------
    // Standard Setters and Getters section
    // ---------------------------------------------------------------------------

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

    public void setBeansMap(HashMap<String, Bean> beansMap) {
        this.beansMap = beansMap;
    }

}
```

XMLFactory

```java
package com.ci1330.ecci.ucr.ac.cr.factory;


import com.ci1330.ecci.ucr.ac.cr.bean.Bean;
import com.ci1330.ecci.ucr.ac.cr.readers.XmlBeanReader;

import java.util.HashMap;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * XMLFactory class which inherits from BeanFactory
 * and registers the XML file from which the configuration
 * must be read and tells the reader to parse it.
 */
public class XMLFactory extends BeanFactory{

    private XmlBeanReader xmlBeanReader;    // Instance of the
        XML configuration reader

    private String xmlFile;     //Path of the XML file which
        holds the configuration

    /**
     * Constructor of the class, it initializes the super-class
         attributes and
     * also the XML bean reader and the file.
     * @param xmlFile the name of the file
     */
    public XMLFactory(String xmlFile){
        super();
        this.xmlFile = xmlFile;
        this.xmlBeanReader = new XmlBeanReader(this);
        this.registerConfig();
        super.initContainer();
    }

    //Tells the reader to start parsing
    private void registerConfig(){
        this.xmlBeanReader.readBeans(this.getXmlFile());
    }


    /**
     * Return a bean instance from the super class.
     * @param id the beanId
     * @return the bean instance
     */
    @Override
    public Object getBean(String id) {
        return super.getBean(id);
    }

    /**
     * Adds a bean to the container
     * @param bean the {@link Bean} class
     */
    @Override
    public void addBean(Bean bean) {
        super.addBean(bean);
    }

    /**
     * Calls the super method for shutDownHook
     */
    @Override
    public void shutDownHook() {
        super.shutDownHook();
    }

    //
    -------------------------------------------------------------------------
    // Standard Setters and Getters section
    //
    -------------------------------------------------------------------------

    public String getXmlFile() {
        return xmlFile;
    }

    public XmlBeanReader getXmlBeanReader() {
        return xmlBeanReader;
    }

    public void setXmlBeanReader(XmlBeanReader xmlBeanReader) {
        this.xmlBeanReader = xmlBeanReader;
    }

    @Override
    public HashMap<String, Bean> getBeansMap() {
        return super.getBeansMap();
    }

    @Override
    public void setBeansMap(HashMap<String, Bean> beansMap) {
        super.setBeansMap(beansMap);
    }
}
```

AutowireEnum

```
package com.ci1330.ecci.ucr.ac.cr.bean;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Enumeration for NAIOCC Container.
 * Used for the different values of the AutowireEnum property.
 */
public enum AutowireEnum {

    byType,
    byName,
    constructor,
    none,
    annotation;

}
```

Bean

```
package com.ci1330.ecci.ucr.ac.cr.bean;

import com.ci1330.ecci.ucr.ac.cr.factory.BeanConstructorModule;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 11/09/2017
 *
```

```java
 * Bean class for NAIOCC Container.
 * Contains the Metadata of a Bean, manages the creation and
    destruction,
 * manages all the instances (if prototype), and the autowiring.
 */
public class Bean {

    private String id; //Uniquely identifies the bean
    private Class beanClass; //Used for different Java
        Reflection methods.
    private boolean lazyGen; //Flag used to indicate if the bean
         is lazy generated
    private AutowireEnum autowireEnum; //Indicates the type of
        autowiring the Bean uses.
    private Scope beanScope; //Indicates the scope of the Bean.

    private BeanFactory beanFactory;

    private Method initMethod; //Initialization method, called
        after the injection of dependencies.
    private Method destroyMethod; //Destroy method, called when
        the container is going to be destroyed.

    private BeanConstructor beanConstructor; //Class used for
        constructor-injections
    private List<BeanAttribute> beanAttributeList; //List of
        classes that are used for setter-injection

    /*The stack is used for keeping track of the different
        instances of a bean.
    The top bean instance is considered as the current one.*/
    private Stack<Object> beanInstanceStack;

    /**
     * Constructor of the class, initializes the Instances Stack
         and the BeanAttribute List.
     */
    public Bean (BeanFactory beanFactory) {
        this.beanInstanceStack = new Stack<>();
        this.beanAttributeList = new ArrayList<>();
        this.beanFactory = beanFactory;
    }

    /**
     * Initializes an instance of a bean, and appends the new
         instance to end of
     * the beanInstanceStack.
     */
    public void createNewInstance() {
        if (this.beanScope == Scope.Singleton && this.beanInstanceStack.size() > 0) {
            System.err.println("Invalid initialization: The Singleton Bean has already been initialized.");
            System.exit(1);
        }

        Object currInstance = this.newInstance();
        this.beanInstanceStack.push(currInstance);
    }

    /**
     * Autowires all the properties of the bean
     */
    public void autowire () {
        //Autowire by constructor or, Atomic-autowire all parameters and register the constructor
        if (this.beanConstructor != null) {
            List<BeanParameter> beanParameterList = this.beanConstructor.getBeanParameterList();
            if (beanParameterList.size() > 0) {
                //If the parameter list has parameters, they are autowired (if necessary) and the constructor is registered
                for (BeanParameter beanParameter : beanParameterList) {
                    beanParameter.autowireProperty();
                }
                BeanConstructorModule.registerConstructor(this);

            } else {
                //If there are no paramters, but the constructor isn't null, it's because the user indicated
                //autowire by constructor to a single constructor
                BeanAutowireModule.autowireSingleConstructor(this.beanConstructor, this.beanFactory, this.id);
            }
        }
        //Atomic-autowire all attributes
        for (BeanAttribute beanAttribute : this.beanAttributeList) {
            beanAttribute.autowireProperty();
        }

        //Class autowiring
        BeanAutowireModule.autowireBean(this);
    }

    /**
     * Checks if all the properties of the bean are correct
     */
    public void checkBeanProperties() {
        for (BeanAttribute beanAttribute : this.
            beanAttributeList) {
            beanAttribute.checkProperty();
        }

        if (this.beanConstructor != null) {
            for (BeanParameter beanParameter : this.
                beanConstructor.getBeanParameterList()) {
                beanParameter.checkProperty();
            }
        }
    }

    /**
     * Creates an instance, by injecting the constructor, if any
        .
     * If there is no specified constructor, it uses the default
         one.
     * @return The new bean instance
     */
    private Object newInstance() {
        Object currInstance = null;
        if (this.beanConstructor == null) {
            try {
                currInstance = this.beanClass.newInstance();

            } catch (InstantiationException e) {
                System.err.println("Instantiation␣Error:␣There␣
                    was␣an␣exception␣trying␣to␣instantiate␣the␣
                    bean␣" + this.beanClass.toString() + ".");
                e.printStackTrace();
                System.exit(1);
            } catch (IllegalAccessException e) {
                System.err.println("Instantiation␣Error:␣There␣
                    was␣an␣exception␣trying␣to␣access␣the␣
                    instance␣bean␣" + this.beanClass.toString() +
                     ".");
                e.printStackTrace();
                System.exit(1);
            }
        }
        else {
            currInstance = this.beanConstructor.newInstance();
        }
```

```java

        return currInstance;
    }

    /**
     * Make all the setter-injections by iterating the attribute list.
     * It pops the top of the stack, makes all the injections, and then
     * it pushes back to the stack.
     */
    public void injectDependencies () {
        Object currInstance = this.getInstance();
        for (BeanAttribute currBeanAttribute : this.beanAttributeList) {
            currBeanAttribute.injectDependency(currInstance);
        }
    }

    /**
     * Calls the initialization method for the current bean instance, if any.
     */
    public void initialize () {
        if (this.initMethod != null) {
            Object currInstance = this.getInstance();
            try {

                this.initMethod.invoke(currInstance);
            } catch (IllegalAccessException e) {
                System.err.println("Initialize␣Error:␣There␣was␣an␣exception␣trying␣to␣access␣the␣init␣method.");
                e.printStackTrace();
                System.exit(1);
            } catch (InvocationTargetException e) {
                System.err.println("Initialize␣Error:␣There␣was␣an␣exception␣trying␣to␣invoke␣the␣init␣method.");
                e.printStackTrace();
                System.exit(1);
            }
        }
    }

    /**
     * Calls the destruction method for all the beans instances, if any, and leaves
     * the stack empty.
     */
    public void destroyAllInstances() {
        Object currInstance;
        while (!this.beanInstanceStack.empty()) {
            currInstance = this.beanInstanceStack.pop();
            if (this.destroyMethod != null) {
                try {

                    this.destroyMethod.invoke(currInstance);

                } catch (IllegalAccessException e) {
                    System.err.println("Destruction Error: There
                        was an exception trying to access the
                        destroyAllInstances method.");
                    e.printStackTrace();
                    System.exit(1);
                } catch (InvocationTargetException e) {
                    System.err.println("Destruction Error: There
                        was an exception trying to invoke the
                        destroyAllInstances method.");
                    e.printStackTrace();
                    System.exit(1);
                }
            }
        }
    }

    /**
     * Peeks the top of the stack.
     * @return Returns the current bean.
     */
    public Object getInstance () {
        if (this.beanInstanceStack.empty()) {
            return null;
        } else {
            return this.beanInstanceStack.peek();
        }
    }

    /**
     * Appends an attribute to the end of the attribute list.
     * @param beanAttributeToAppend bean attribte to apend
     */
    public void appendAttribute (BeanAttribute
        beanAttributeToAppend) {
        this.beanAttributeList.add(beanAttributeToAppend);
    }


    //
    ________________________________________________________________________

    // Standard Setters and Getters section
    //
    ________________________________________________________________________


    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public void setBeanClass(Class beanClass) {
        this.beanClass = beanClass;
    }

    public Class getBeanClass () {
        return beanClass;
    }

    public boolean isLazyGen() {
        return lazyGen;
    }

    public void setLazyGen(boolean lazyGen) {
        this.lazyGen = lazyGen;
    }

    public AutowireEnum getAutowireEnum() {
        return autowireEnum;
    }

    public void setAutowireEnum(AutowireEnum autowireEnum) {
        this.autowireEnum = autowireEnum;
    }

    public void setBeanScope(Scope beanScope) {
        this.beanScope = beanScope;
    }

    public Scope getBeanScope() {
        return beanScope;
    }

    public BeanFactory getBeanFactory() {
```

```java
        return beanFactory;
    }

    public void setBeanFactory(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    public void setInitMethod(Method initMethod) {
        this.initMethod = initMethod;
    }

    public void setDestroyMethod(Method destroyMethod) {
        this.destroyMethod = destroyMethod;
    }

    public void setBeanConstructor(BeanConstructor
        beanConstructor) {
        this.beanConstructor = beanConstructor;
    }

    public List<BeanAttribute> getBeanAttributeList () {
        return beanAttributeList;
    }

    public BeanConstructor getBeanConstructor() {
        return beanConstructor;
    }
}
```

BeanAttribute

```java
package com.ci1330.ecci.ucr.ac.cr.bean;

import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 14/09/2017
 *
 * BeanAttribute class for NAIOCC Container.
 * Contains the Metadata of an attribute and manages the setter
    injection of a
 * dependency for a Bean.
 */
public class BeanAttribute extends BeanProperty {

    private Method setterMethod; //Used for invoking the
        respective class setter

    /**
     * Constructor of the class, initializes the class and super
         -class attributes.
     * @param beanRef init value for the super's beanRef
         attribute
     * @param beanFactory init value for the super's beanFactory
          attribute
     * @param value init value for the super's value attribute
     * @param setterMethod init value for the bean's setter
         method
     */
    public BeanAttribute(String beanRef, Class beanRefClass,
        BeanFactory beanFactory, Object value, AutowireEnum
        atomic_autowire, Method setterMethod) {
        super(beanRef, beanRefClass, beanFactory, value,
            atomic_autowire);
        this.setterMethod = setterMethod;
    }

    /**
     * Receives an object, an injects a dependency to the object
         .
     * The dependency is fetched by using the super.getInstance
         method
     * @param objectToInject The bean instance without
         injections
     */
    void injectDependency(Object objectToInject) {
        Object dependency = super.getInstance();
        try {
            this.setterMethod.invoke(objectToInject, dependency)
                ;

        } catch (IllegalAccessException e) {
            System.err.println("Setter Error: There was an
                exception trying to access the setter method for
                :\n"
                    + "\t" + this.setterMethod.toString() + ".")
                        ;
            e.printStackTrace();
            System.exit(1);
        } catch (InvocationTargetException e) {
            System.err.println("Setter Error: There was an
                exception trying to invoke the setter method for
                :\n"
                    + "\t" + this.setterMethod.toString() + ".")
                ;
            e.printStackTrace();
            System.exit(1);
        }

    }

    //
    -----------------------------------------------------------------------

    // Standard Setters and Getters section
    //
    -----------------------------------------------------------------------


    public void setSetterMethod(Method setterMethod) {
        this.setterMethod = setterMethod;
    }

}
```

BeanAutowireModule

```
package com.ci1330.ecci.ucr.ac.cr.bean;

import com.ci1330.ecci.ucr.ac.cr.exception.BeanAutowireException
    ;
import com.ci1330.ecci.ucr.ac.cr.exception.
    BeanTypeConflictException;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;
import com.thoughtworks.paranamer.AdaptiveParanamer;
import com.thoughtworks.paranamer.Paranamer;

import java.lang.reflect.*;
import java.util.ArrayList;
import java.util.List;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * The class is in charge of autowiring (at class level) a bean.
 * It also has the capability to autowire a single constructor.
 */
public class BeanAutowireModule {

    /**
     * Determines which type of autowiring needs to be done
     * @param bean the bean to autowire
     */
```

```java
    static void autowireBean (Bean bean) {
        switch (bean.getAutowireEnum()) {
            case byName:
                autowireByName(bean);
                break;
            case byType:
                autowireByType(bean);
                break;
            case constructor:
                autowireConstructor(bean);
                break;
            case none:
                break;
            default:
                try {
                    throw new BeanAutowireException("Autowire␣Module␣Error:␣Unexpected␣value␣recieved␣while␣trying␣to␣autowire␣the␣bean␣" + bean.getId());
                } catch (BeanAutowireException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
        }
    }

    /**
     * Iterates all the fields of a class. For every field it searches that the field name matches a beanId
     * in the container, if found, creates a {@link BeanAttribute} for the field.
     * @param bean the bean to use
     */
    private static void autowireByName (Bean bean) {
        Class currInstanceClass = bean.getBeanClass();
        BeanFactory beanFactory = bean.getBeanFactory();
        List<BeanAttribute> registeredAttributes = bean.getBeanAttributeList();

        Method currAttributeSetter;
        String currAttributeName;
        Class currAttributeType;

        //For every field of the class
        for (Field currAttribute: currInstanceClass.getDeclaredFields()) {
            //If the field is private, make it accessible
            if(Modifier.isPrivate(currAttribute.getModifiers())) {
                currAttribute.setAccessible(true);
            }

            currAttributeName = currAttribute.getName();
            currAttributeType = currAttribute.getType();

            if(beanFactory.findBean(currAttributeName) != null){
                currAttributeSetter = findSetter(
                    currAttributeName, currAttributeType, bean);

                //If the attribute isn't already registered in
                    the Bean (the user didn't overwrite the
                    autowiring for the
                //attribute), put it in the bean.
                if (!attributeIsAlreadyRegistered(
                    registeredAttributes, currAttributeName)) {
                    BeanAttribute beanAttribute = new
                        BeanAttribute(currAttributeName,
                        currAttributeType, beanFactory, null,
                        AutowireEnum.none, currAttributeSetter);
                    bean.appendAttribute(beanAttribute);
                }
            }
        }
    }

    /**
     * Finds the setter method for an attribute
     * @param attributeName the name of the attribute used
     * @param attributeClass the type of the attribute
     * @param bean used to recover the class of the bean
     * @return the setter method
     */
    private static Method findSetter (String attributeName,
        Class attributeClass, Bean bean) {
        Method[] beanMethods;
        Class[] methodParameterTypes;

        //Search every method in the bean
        beanMethods = bean.getBeanClass().getDeclaredMethods();
        for (Method method : beanMethods) {
            //If private, make it accessible
            if(Modifier.isPrivate(method.getModifiers())){
                method.setAccessible(true);
            }

            //Check if it has set at the start and contains the
                name of the attribute
            if (method.getName().startsWith("set") && method.
                getName().toLowerCase().contains(attributeName.
                toLowerCase())) {

                    methodParameterTypes = method.getParameterTypes
                        ();
                    //Check the parameters are valid
                    if (method.getParameterCount() == 1 &&
                        methodParameterTypes[0].equals(attributeClass
                        )) {
                        return method;
                    }

                }

            }

            try {
                throw new BeanAutowireException("Autowire␣Module␣
                    Error:␣The␣field␣" + attributeName +"␣of␣" + bean
                    .getBeanClass().toString() +
                        "␣matches␣with␣autowiring,␣but␣no␣setter␣
                            method␣was␣found␣for␣it.");
            } catch (BeanAutowireException e) {
                e.printStackTrace();
                System.exit(1);
            }
            return null; //Keep the compiler happy
        }

        /**
         * Iterates the list of {@link BeanAttribute} and returns
            true if the bean reference is already found,
         * and false if not.
         * @param registeredAttributes The list of BeanAttributes
         * @param beanRef the bean reference to search
         * @return the result of the search
         */
        private static boolean attributeIsAlreadyRegistered (List<
            BeanAttribute> registeredAttributes, String beanRef) {
            for (BeanAttribute registeredAttribute :
                registeredAttributes) {
                if (registeredAttribute.getBeanRef().equals(beanRef)
                    ) {
                    return true;
                }
            }
            return false;
        }

```

```java
    /**
     * Iterates every field in the bean class and tries to
         search for a bean that match in type with
     * the field. If there are multiple definitions of beans
         with that type, an exception is thrown.
     * @param bean the bean to autowire
     */
    private static void autowireByType (Bean bean) {

        Class currInstanceClass = bean.getBeanClass();
        BeanFactory beanFactory = bean.getBeanFactory();
        List<BeanAttribute> registeredAttributes = bean.
            getBeanAttributeList();

        Method currAttributeSetter;
        String currAttributeName;
        Class currAttributeClass;

        Bean typeLikeBean = null;

        //Iterates all the fields
        for (Field currAttribute: currInstanceClass.
            getDeclaredFields()) {
            //If the field is private, make it accesible
            if(Modifier.isPrivate(currAttribute.getModifiers()))
                {
                currAttribute.setAccessible(true);
            }

            currAttributeClass = currAttribute.getType();

            //If there are multiple beans with that type, exit
                abnormally.
            try {
                typeLikeBean = beanFactory.findBean(
                    currAttributeClass);
            } catch (BeanTypeConflictException e) {
                e.printStackTrace();
                System.exit(1);
            }

            //If the bean was found
            if(typeLikeBean != null){

                currAttributeName = currAttribute.getName();
                currAttributeSetter = findSetter(
                    currAttributeName, currAttributeClass, bean);

                //And it wasn't already in the container,
                    register it
                if (!attributeIsAlreadyRegistered(registeredAttributes, currAttributeName)) {
                    BeanAttribute beanAttribute = new BeanAttribute(typeLikeBean.getId(), currAttributeClass, beanFactory, null, AutowireEnum.none, currAttributeSetter);
                    bean.appendAttribute(beanAttribute);
                }

            }
        }
    }

    /**
     * Iterates all constructors. For every constructor, searches that its parameters' names, match with a bean's id.
     * If they all match, the constructor is selected. If there is more than one matched constructor, an exception is thrown.
     * @param bean the bean to autowire
     */
    private static void autowireConstructor (Bean bean) {
        if (bean.getBeanConstructor() == null) { //If the user already defined the constructor explicitly this process is omitted
            Constructor[] classConstructors = bean.getBeanClass().getDeclaredConstructors();
            BeanFactory beanFactory = bean.getBeanFactory();

            Parameter[] constructorParameters;
            String[] parameterNames;
            Constructor matchedConstructor = null;
            boolean allParamsMatched, allParamsClassesMatched;
            Paranamer paranamer = new AdaptiveParanamer(); // Utility to recover parameter names

            List<BeanParameter> beanParameterList = new ArrayList<>();
            for (Constructor classConstructor : classConstructors) {
                //If it has parameters
                if (classConstructor.getParameterCount() > 0) {
                    allParamsMatched = true;

                    parameterNames = paranamer.lookupParameterNames(classConstructor);
                    constructorParameters = classConstructor.
                getParameters();
            //Look if the names match
            for (String  parameter : parameterNames) {
                if (beanFactory.findBean(parameter) ==
                    null) {
                    allParamsMatched = false;
                    break;
                }
            }

            //If they all matched
            if (allParamsMatched) {
                //And the constructor didn't match
                    already
                if (matchedConstructor == null) {
                    //Check that the types also match
                    allParamsClassesMatched =
                        checkParametersTypes(beanFactory,
                         constructorParameters,
                        parameterNames, beanParameterList
                        );

                    if (allParamsClassesMatched) {
                        matchedConstructor =
                            classConstructor;
                    } else {
                        try {
                            throw new
                                BeanAutowireException("
                                Autowire␣Module␣Error:␣
                                parameter␣types␣mismatch␣
                                for␣autowiring␣by␣
                                constructor.␣Bean:␣" +
                                bean.getId());
                        } catch (BeanAutowireException e
                            ) {
                            e.printStackTrace();
                            System.exit(1);
                        }
                    }

                } else {
                    //If it did, exit abnormally
                    try {
                        throw new BeanAutowireException(
                            "Autowire␣Module␣Error:␣there
                            ␣are␣multiple␣constructors␣
                            that␣match␣in␣their␣
                            parameters␣names,␣in␣
```

```
                                    autowiring by constructor. 
                                    Bean: " + bean.getId());
                        } catch (BeanAutowireException e) {
                            e.printStackTrace();
                            System.exit(1);
                        }
                    }

                }
            }
        }

        if (matchedConstructor != null) {
            BeanConstructor beanConstructor = new
                BeanConstructor(matchedConstructor);
            beanConstructor.setBeanParameterList(
                beanParameterList);
            bean.setBeanConstructor(beanConstructor);
        }
    }

}

/**
 * For a specific constructor searches that its parameters'
    names, match with a bean's id.
 * If they all match, the BeanParameters are created. If the
     parameters didn't match, exits abnormally.
 * @param beanConstructor the constructor to match
 * @param beanFactory the factory to use
 * @param beanId the id of the bean (used for throwing the
    error)
 */
public static void autowireSingleConstructor(
    BeanConstructor beanConstructor, BeanFactory beanFactory,
     String beanId) {
    Constructor classConstructor = beanConstructor.
        getConstructorMethod();

    //If it has parameters
    if (classConstructor.getParameterCount() > 0) {
        Boolean allParamsMatched = true;
        Paranamer paranamer = new AdaptiveParanamer();

        String[] parameterNames = paranamer.
            lookupParameterNames(classConstructor);
        Parameter[] constructorParameters = classConstructor
            .getParameters();

        List<BeanParameter> beanParameterList =
            beanConstructor.getBeanParameterList();

        //Look if the names match
        for (String parameter : parameterNames) {
            if (beanFactory.findBean(parameter) == null) {
                allParamsMatched = false;
                break;
            }
        }

        //If they don't match, exit abnormally.
        if (!allParamsMatched) {
            try {
                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);
            } catch (BeanAutowireException e) {
                e.printStackTrace();
                System.exit(1);
            }
        } else {
            Boolean allParamsClassesMatched =
                checkParametersTypes(beanFactory,
                constructorParameters, parameterNames,
                beanParameterList);

            //If the types didn't match, exit abnormally.
            if (!allParamsClassesMatched) {
                try {
                    throw new BeanAutowireException("
                        Autowire Module Error: parameter
                        types mismatch for autowiring by
                        constructor. For bean: " + beanId);
                } catch (BeanAutowireException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
            }
        }

    }
}

/**
 * Checks that the types of the parameters, match the types
    of the beans.
 * @param beanFactory the factory to use
 * @param constructorParameters the array of parameters
 * @param constructorParameterNames the array of parameter
    names
 * @param beanParameterList the list of bean parameters, in
    which we will start to append if a parameter matches
 * @return true if they all match, false if they don't.
 */
private static boolean checkParametersTypes(BeanFactory
    beanFactory, Parameter[] constructorParameters, String[]
    constructorParameterNames, List<BeanParameter>
    beanParameterList) {
    boolean allParamsClassesMatched = true;
    int parameterIndex = 0;

    Class currBeanClass;

    //For every parameter
    for (Parameter constructorParameter :
        constructorParameters) {
        currBeanClass = beanFactory.findBean(
            constructorParameterNames[parameterIndex]).
            getBeanClass();
        //If the type of the bean is the same as the type of
             the parameter
        if ( currBeanClass == constructorParameter.getType()
            ) {
            //Append it
            beanParameterList.add(new BeanParameter(
                constructorParameterNames[parameterIndex],
                currBeanClass, beanFactory, null,
                AutowireEnum.none, parameterIndex,
                constructorParameter.getType().toString()));
        } else {
            allParamsClassesMatched = false;
            break;
        }

        parameterIndex++;
    }

    return allParamsClassesMatched;
}
}
```

BeanConstructor

```
package com.ci1330.ecci.ucr.ac.cr.bean;

import java.lang.reflect.Constructor;
```

```java
import java.lang.reflect.InvocationTargetException;
import java.util.ArrayList;
import java.util.List;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 15/09/2017
 *
 * Bean Constructor for NAIOCC Container.
 * Contains the Metadata of a Bean's constructor, manages the constructor injection.
 */
public class BeanConstructor {
    private Constructor constructorMethod;
    private List<BeanParameter> beanParameterList;

    /**
     * Constructor of the class, initializes the Parameter list and sets the constructor method value.
     * @param constructorMethod init value for the construction method.
     */
    public BeanConstructor (Constructor constructorMethod) {
        this.constructorMethod = constructorMethod;
        this.beanParameterList = new ArrayList<>();
    }

    /**
     * Creates a new instance of a bean, with constructor injection.
     * @return The injected bean instance.
     */
    public Object newInstance() {
        Object[] parameterInstances = new Object[this.beanParameterList.size()];
        Object beanInstance = null;
        for (BeanParameter currBeanParameter : this.beanParameterList) {
            parameterInstances[currBeanParameter.getIndex()] = currBeanParameter.getInstance();
        }
        try {
            beanInstance = this.constructorMethod.newInstance(parameterInstances);

        } catch (InstantiationException e) {
            System.err.println("Construction Error: There was an exception trying to instantiate a bean with the constructor method for:\n"
                    +  "\t" + this.constructorMethod.toString() + ".");
            e.printStackTrace();
            System.exit(1);
        } catch (IllegalAccessException e) {
            System.err.println("Construction Error: There was an exception trying to access the constructor method for:\n"
                    +  "\t" + this.constructorMethod.toString() + ".");
            e.printStackTrace();
            System.exit(1);
        } catch (InvocationTargetException e) {
            System.err.println("Construction Error: There was an exception trying to invoke the constructor method for:\n"
                    + "\t" + this.constructorMethod.toString() + ".");
            e.printStackTrace();
            System.exit(1);
        } catch (IllegalArgumentException e) {
            System.err.println("Construction Error: There was an exception trying to invoke the constructor method for:\n"
                    + "\t" + this.constructorMethod.toString() + "\n"
                    + " with " + parameterInstances[0].getClass() +".");
            e.printStackTrace();
            System.exit(1);
        }

        return beanInstance;
    }

    /**
     * Appends a bean parameter to the list.
     * @param beanParameter
     */
    public void append(BeanParameter beanParameter){
        this.beanParameterList.add(beanParameter);
    }
    // -----------------------------------------------------------------------
    // Standard Setters and Getters section
    // -----------------------------------------------------------------------
```

```java

    public void setConstructorMethod(Constructor
        constructorMethod) {
         this.constructorMethod = constructorMethod;
    }

    public Constructor getConstructorMethod() {
        return constructorMethod;
    }

    public void setBeanParameterList(List<BeanParameter>
        beanParameterList) {
         this.beanParameterList = beanParameterList;
    }

    public List<BeanParameter> getBeanParameterList() {
        return beanParameterList;
    }

}
```

BeanParameter

```java
package com.ci1330.ecci.ucr.ac.cr.bean;

import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 15/09/2017
 *
 * BeanParameter class for NAIOCC Container.
 * Contains the Metadata of a Bean's constructor's parameter.
 */
public class BeanParameter extends BeanProperty {

    private int index; //The position of the parameter in the
        constructor.
    private String explicitTypeName;

    /**
     * Constructor of the class, initializes the class and super
         -class attributes.
     * @param beanRef init value for the super's beanRef
         attribute
     * @param beanFactory init value for the super's beanFactory
          attribute
     * @param value init value for the super's value attribute
     * @param index init value for the parameter's index.
     */
    public BeanParameter(String beanRef, Class beanRefClass,
        BeanFactory beanFactory, Object value, AutowireEnum
        atomic_autowire, int index, String explicitTypeName) {
         super(beanRef, beanRefClass, beanFactory, value,
             atomic_autowire);
         this.index = index;
         this.explicitTypeName = explicitTypeName;
    }

    //
        ____________________________________________________________________________
    // Standard Setters and Getters section
    //
        ____________________________________________________________________________

    public void setIndex(int index) {
         this.index = index;
    }

    public int getIndex() {
         return index;
    }

    public String getExplicitTypeName() {
         return explicitTypeName;
    }

    public void setExplicitTypeName(String explicitTypeName) {
         this.explicitTypeName = explicitTypeName;
    }

}
```

BeanProperty

```
package com.ci1330.ecci.ucr.ac.cr.bean;

import com.ci1330.ecci.ucr.ac.cr.exception.
    BeanAtomicAutowireException;
import com.ci1330.ecci.ucr.ac.cr.exception.BeanPropertyException
    ;
import com.ci1330.ecci.ucr.ac.cr.exception.
    BeanTypeConflictException;
import com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
```

```java
 *
 * BeanParameter class for NAIOCC Container.
 * Contains the Metadata of a Bean's property. Manages the fetching from the factory, if
 * the property references a Bean.
 */
public abstract class BeanProperty {

    private String beanRef; //The beanId that references a bean
    private Class beanRefType;
    private BeanFactory beanFactory;
    private Object value; //The explicit value, specified by the end-user.
    private AutowireEnum atomic_autowire; //Specifies the atomic autowiring for the property

    /**
     * Constructor of the class, initializes the class attributes.
     * @param beanRef init value for the property's beanRef attribute
     * @param beanFactory init value for the property's beanFactory attribute
     * @param value init value for the property's value attribute
     */
    BeanProperty(String beanRef, Class beanRefType, BeanFactory beanFactory, Object value, AutowireEnum atomic_autowire) {
        this.beanRef = beanRef;
        this.beanRefType = beanRefType;
        this.beanFactory = beanFactory;
        this.value = value;
        this.atomic_autowire = atomic_autowire;
    }

    /**
     * The bean instance can either be an explicit value, or be fetched from the BeanFactory
     * @return instance of the value or instance sent by the container for the reference
     */
    Object getInstance () {
        if (this.value == null) {
            return this.beanFactory.getBean(this.beanRef);
        } else {
            return this.value;
        }
    }

    /**
     * According to the value of /atomic_autowire, autowires the property
     */
    void autowireProperty () {
        switch (this.atomic_autowire) {
            case byName:
                //This case is mostly for parameter autowiring, in which the type is known until the container
                //is fully created
                this.autowireByName();
                break;
            case byType:
                //This case is for both parameters and attributes.
                //It searches the container for a bean that matches with its type, if found stores its ID.
                this.autowireByType();
                break;
            case annotation:
                //This case is exclusive for attributes that were autowired using an annotation.
                //It first tries to autowire byType, if it fails, tries to autowire byName
                this.autowireByAnnotation();
                break;
        }
    }

    /**
     * Searches for a bean with the name of beanRef, if not found, throws an exception
     * If found, puts beanRefType (if null) to the type of the recovered bean
     */
    private void autowireByName () {
        if (this.beanFactory.findBean(this.beanRef) == null) {
            try {
                throw new BeanAtomicAutowireException("Bean␣Atomic-Autowire␣Error:␣At␣atomic-autowiring␣byName␣no␣bean␣was␣found␣for␣the␣beanId␣" + this.beanRef);
            } catch (BeanAtomicAutowireException e) {
                e.printStackTrace();
                System.exit(1);
            }
```

```java
        } else if (this.beanRefType == null) {
            //If the beanRefType was null, put it as the type of the recovered bean
            this.beanRefType = this.beanFactory.findBean(this.beanRef).getBeanClass();
        }
    }

    /**
     * Searches for a bean with the type of beanRefType, if not found, throws an exception
     * If found, stores the recovered bean's ID
     */
    private void autowireByType () {
        try {
            if (this.beanFactory.findBean(this.beanRefType) == null) {

                try {
                    throw new BeanAtomicAutowireException("Bean Atomic-Autowire Error: At atomic-autowiring byType no bean was found for the type " + this.beanRefType);
                } catch (BeanAtomicAutowireException e) {
                    e.printStackTrace();
                    System.exit(1);
                }

            } else {
                //If a bean exists, store the property's ID
                this.beanRef = this.beanFactory.findBean(this.beanRefType).getId();
            }
        } catch (BeanTypeConflictException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    /**
     * Special autowire for annotation, first tries to do autowire byType, if it fails, it does autowire byName
     */
    private void autowireByAnnotation () {
        try {
            //Sees if there exists a bean with that type (autowire byType)
            if (this.beanFactory.findBean(this.beanRefType) == null) {

                //Sees if there exists a bean with that reference (autowire byName)
                if (this.beanFactory.findBean(this.beanRef) == null) {
                    try {
                        throw new BeanAtomicAutowireException("Bean␣Atomic−Autowire␣Error:␣At␣atomic−autowiring␣byName␣no␣bean␣was␣found␣for␣the␣beanId␣" + this.beanRef);
                    } catch (BeanAtomicAutowireException e) {
                        e.printStackTrace();
                        System.exit(1);
                    }
                }
                //We don't have to do anything if the beanRef is valid, because the type is already assign
                //And the checkProperty method will check that everything matches.

            } else {
                this.beanRef = this.beanFactory.findBean(this.beanRefType).getId();
            }
        } catch (BeanTypeConflictException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    /**
     * Checks if the metadata of a bean is correct, if not, throws an exception.
     */
    public void checkProperty() {
        if (value == null) {
            boolean thereIsProblem = true;
            //If the reference is not null and exists a reference for it in the container
            if (this.beanRef != null && this.beanFactory.findBean(this.beanRef) != null) {

                //If the type is not null and the bean returned by the factory matches with the declared type
                if (this.beanRefType != null && this.beanRefType == this.beanFactory.findBean(this.beanRef).getBeanClass()) {

                    thereIsProblem = false;
                }

            }

            if (thereIsProblem) {
                try {
                    throw new BeanPropertyException("Bean Property Error: There was an unexpected exception with the reference and type of the property " + this.beanRef);
                } catch (BeanPropertyException e) {
                    e.printStackTrace();
                    System.exit(1);
                }
            }
        }
    }

    // ---------------------------------------------------------------------------
    // Standard Setters and Getters section
    // ---------------------------------------------------------------------------

    public void setBeanRef(String beanRef) {
        this.beanRef = beanRef;
    }

    public void setBeanFactory(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    public BeanFactory getBeanFactory() {
        return beanFactory;
    }

    public void setValue(Object value) {
        this.value = value;
    }

    public String getBeanRef() {
        return this.beanRef;
    }

    public Class getBeanRefType() {
        return this.beanRefType;
    }
```

```java

    public Object getValue() {
        return value;
    }
}
```

Scope

```java
package com.ci1330.ecci.ucr.ac.cr.bean;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Enumeration for NAIOCC Container.
 * Used for the different values of the Scope property.
 */
public enum Scope {
    Singleton,
    Prototype;
}
```

Stereotype

```java
package com.ci1330.ecci.ucr.ac.cr.bean;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Enumeration for NAIOCC Container.
 * Used for the different values of the Stereotype Annotations.
 */
public enum Stereotype {
    Bean,
    Controller,
    Service,
    Repository;
}
```

AnnotationsBeanReaderException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception in {@link com.ci1330.ecci.ucr.ac.cr.
    readers.AnnotationsBeanReader}
 */
public class AnnotationsBeanReaderException extends Exception {
    public AnnotationsBeanReaderException() {
    }

    public AnnotationsBeanReaderException(String message) {
        super(message);
    }
}
```

BeanAtomicAutowireException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception in {@link com.ci1330.ecci.ucr.ac.cr.
     bean.BeanProperty} while trying
 * to atomicly autowire.
 */
public class BeanAtomicAutowireException extends Exception {

    public BeanAtomicAutowireException() {
        super();
    }

    public BeanAtomicAutowireException(String message) {
        super(message);
    }
}
```

BeanAutowireException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception in the {@link com.ci1330.ecci.ucr.ac.
     cr.bean.BeanAutowireModule}
 */
public class BeanAutowireException extends Exception{

    public BeanAutowireException() {
        super();
    }

    public BeanAutowireException(String message) {
        super(message);
    }
}
```

BeanConstructorConflictException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when there is a conflict in {@link com
    .ci1330.ecci.ucr.ac.cr.factory.BeanConstructorModule}
 */
public class BeanConstructorConflictException extends Exception{

    public BeanConstructorConflictException() {
        super();
    }

    public BeanConstructorConflictException(String message) {
        super(message);
    }

}
```

BeanConstructorNotFoundException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when the {@link com.ci1330.ecci.ucr.ac
    .cr.factory.BeanConstructorModule} can not
 * find a Constructor.
 */
public class BeanConstructorNotFoundException extends Exception
    {

    public BeanConstructorNotFoundException() {
        super();
    }

    public BeanConstructorNotFoundException(String message) {
        super(message);
    }

}
```

BeanPropertyException

```
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when something goes wrong in {@link
     com.ci1330.ecci.ucr.ac.cr.bean.BeanProperty}
 */
public class BeanPropertyException extends Exception {

    public BeanPropertyException() {
        super();
    }

    public BeanPropertyException(String message) {
        super(message);
    }

}
```

BeanTypeConflictException

```
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when trying to invoke findBean method
     of {@link com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory}
 * and there are two beans of the same type.
 */
public class BeanTypeConflictException extends Exception {

    public BeanTypeConflictException() {
        super();
    }

    public BeanTypeConflictException(String message) {
        super(message);
    }
}
```

IdNotFoundException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Throws an exception if someone tries to recover a bean from {
    @link com.ci1330.ecci.ucr.ac.cr.factory.BeanFactory}
 * and the id does not exist.
 */
public class IdNotFoundException extends Exception {

    public IdNotFoundException() {
    }
    public IdNotFoundException(String message) {
        super(message);
    }
}
```

InvalidPropertyException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when {@link com.ci1330.ecci.ucr.ac.cr.
    factory.BeanCreator}
 * receives invalid property information.
 */
public class InvalidPropertyException extends Exception {

    public InvalidPropertyException() {
        super();
    }

    public InvalidPropertyException(String message) {
        super(message);
    }

}
```

RepeatedIdException

```java
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when {@link com.ci1330.ecci.ucr.ac.cr.
   factory.BeanCreator} receives a repeated bean id.
 */
public class RepeatedIdException extends Exception{

    public RepeatedIdException(){
    }

    public RepeatedIdException(String message){
        super(message);
    }

}
```

SetterMethodNotFoundException

```
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when the {@link com.ci1330.ecci.ucr.ac
   .cr.factory.BeanCreator} does not find the setter method
 * of an attribute.
 */
public class SetterMethodNotFoundException extends Exception{

    public SetterMethodNotFoundException() {
        super();
    }

    public SetterMethodNotFoundException(String message) {
        super(message);
    }
}
```

XmlBeanReaderException

```
package com.ci1330.ecci.ucr.ac.cr.exception;

/**
 * @author Elias Calderon, Josue Leon, Kevin Leon
 * Date: 13/09/2017
 *
 * Indicates an exception when {@link com.ci1330.ecci.ucr.ac.cr.
   readers.XmlBeanReader} finds an error
 */
```

```java
public class XmlBeanReaderException extends Exception {
    public XmlBeanReaderException() {
    }

    public XmlBeanReaderException(String message) {
        super(message);
    }
}
```

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. [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].