Politecnico di Milano

A.A. 2015-2016

Software Engineering 2 project

Code Inspection

Alessandro Pozzi (mat. 852358), Marco Romani (mat. 852361)

6 January 2016

Version 1.0

Summary

1. Classes and methods assigned

1.1 Main Class

InterceptorManager (appserver/ejb/ejb-container/src/main/java/com/sun/ejb/containers/interceptors/InterceptorManager.java)

1.2 Other Classes

These classes are defined in the same file of the InterceptorManager's class.

- AroundInvokeInterceptor
- BeanAroundInvokeInterceptor
- CallbackInterceptor

1.3 Methods

- load2xLifecycleMethods(ArrayList < CallbackInterceptor > [ ] metaArray)
- loadOnlyEjbCreateMethod(ArrayList < CallbackInterceptor > [ ] metaArray , int numPostConstructFrameworkCallbacks)
- AroundInvokeInterceptor(int index , Method method)
- intercept(final InterceptorManager . AroundInvokeContext invCtx) *in AroundInvokeInterceptor class*
- intercept(final InterceptorManager . AroundInvokeContext invCtx) *in BeanAroundInvokeInterceptor class*
- CallbackInterceptor(int index , Method method)

2. Functional role of the assigned classes

2.1 InterceptorManager

As the name may suggest, this class manages the interceptors in a java EE container. There is one and only one instance of this class for each container. Interceptors are auxiliary components "attached" to bean classes (but not only) by declaring some annotations like @AroundInvoke or @PostConstruct (the most relevant ones considered in our analysis).

The role of an interceptor is to intercept, of course, method calls on the target bean class or lifecycle events concerning the target bean class in order to execute some kind of pre-processing or security policy check that can lead to the discard or modification of the method call itself.

Interceptors can either be attached to a whole bean class or only to some of its methods. In both cases, for each instance of the target bean class, an instance of any interceptor declared is created. These interceptors' instances follow the lifecycle of the target class' instance.

As one might expect, the java code for this type of classes (interceptors and InterceptorManager) has to deal with abstractions of the concepts of classes and methods themselves, making it quite difficult to understand without a specific knowledge of the matter.

The two methods directly belonging to this class that we had to analyze are private methods called only once. They have auxiliary tasks that manage rare and uncommon situations:

- load2xLifecycleMethods(ArrayList < CallbackInterceptor > [ ] metaArray)

  This method is called to load interceptors that act accordingly to old 2.x versions of the EJB standard.

- loadOnlyEjbCreateMethod(ArrayList < CallbackInterceptor > [ ] metaArray , int numPostConstructFrameworkCallbacks)

  This method is called to load only interceptors for "ejbCreate" methods in the particular situation of a component that does not have a @PostConstruct annotation or does not even implement the EnterpriseBean interface.

2.2 AroundInvokeInterceptor

This type of interceptor has the role of intercepting method calls on the target class and doing some pre-processing “around” the context of the method call.

2.3 CallbackInterceptor

This type of interceptor has the role of intercepting lifecycle events concerning the target class and doing some operations on it, for example persistent data retrieval/management after the creation or before the destruction of the target class instance.

3. List of issues

3.1 Naming conventions

There are no relevant issues in this section. However, all the classes, methods and variables names are meaningful only to someone with a specific knowledge and comprehension of the scope and the tasks accomplished by the code. It could have been helpful to clarify the role of such elements with additional comments and/or documentations.

3.2 Indention

3.2.1 Intercept in AroundInvokeInterceptor

The following *return* statement is not aligned:

```
if( System.getSecurityManager() != null ) {
// Wrap actual value insertion in doPrivileged to
// allow for private/protected field access.
return java.security.AccessController
        .doPrivileged(new java.security.PrivilegedExceptionAction() {
            public java.lang.Object run() throws Exception {
                return method.invoke(interceptors[index], invCtx);
            }
        });
```

3.2.2 Intercept in CallbackInterceptor

The same as above.

3.2.3 CallbackInterceptor

The code inside the *try* statement is not aligned:

```
try {
final Method finalM = method;
if(System.getSecurityManager() == null) {
    if (!finalM.isAccessible()) {
        finalM.setAccessible(true);
    }
```

3.3 Braces

There are no relevant issues in this section.

3.4 File Organization

3.4.1 LoadOnlyEjbCreateMethod

The variables declaration in this method might be separated from the rest of the code (the *for* block) in order to improve readability.

```
private void loadOnlyEjbCreateMethod(
        ArrayList<CallbackInterceptor>[] metaArray,
        int numPostConstructFrameworkCallbacks) {
    int sz = lcAnnotationClasses.length;
    for (int i = 0; i < sz; i++) {
        if (lcAnnotationClasses[i] != PostConstruct.class) {
            continue;
        }
```

3.4.2 Intercept in AroundInvokeInterceptor

In the *else* block there are two unnecessary blank lines.

```
} else {

    return method.invoke(interceptors[index], invCtx);

}
```

3.4.3 Intercept in CallbackInterceptor

There is an extra unnecessary blank line.

```
Object intercept(final InterceptorManager.AroundInvokeContext invCtx) throws Throwable {
    try {

        if( System.getSecurityManager() != null ) {
```

3.5 Wrapping Lines

3.5.1 Load2xLifecycleMethods

Line break after a parenthesis is not recommended. We recommend to break the line after *pre30LCMethodNames[i].*

```
try {
    Method method = beanClass.getMethod(
            pre30LCMethodNames[i], (Class[]) null);
```

Line break should occur before the operator.

```
if (method != null) {
    CallbackInterceptor meta =
            new BeanCallbackInterceptor(method);
```

3.5.2 LoadOnlyEjbCreateMethod

Line break after a parenthesis is not recommended.

```
if (method != null) {
    CallbackInterceptor meta = new BeanCallbackInterceptor(
            method);
```

3.6 Comments

There are only a few comments in the entire class and almost no documentation. Also, such comments do not describe at all what the code is doing but they are simply notes for the developer himself.

3.7 Java source file

There are two public interfaces (*AroundInvokeContext* and *InterceptorChain*) in the main public class (*InterceptorManager*). It will be more appropriate to put them into two separate source files.

As previously stated, the Javadoc is not complete: most of the public methods (not only the ones assigned) don't have any documentation, as well as the classes and interfaces. For example, the documentation of the *InterceptorManager* class does not provide any concrete additional information that cannot be inferred by the name of the class itself.

```
/**
 * UserInterceptorsManager manages UserInterceptors. There
 * is one instance of InterceptorManager per container.
 *
 * @author Mahesh Kannan
 */
public class InterceptorManager {
```

3.8 Package and Import Statements

No issues.

3.9 Class and Interface Declaration

Package level attributes should be declared before the private ones.

```
  private Class[] lcAnnotationClasses;

  private CallbackChainImpl[] callbackChain;

  // Optionally specified delegate to be set on SystemInterceptorProxy
  private Object runtimeInterceptor;

  List<InterceptorDescriptor> frameworkInterceptors = new LinkedList<InterceptorDescriptor>();
```

There are many duplicates in the source file. In particular, the *AroundInvokeInterceptor* class is extremely similar to the *CallbackInterceptor*.

```
class AroundInvokeInterceptor {
    protected int index;
    protected Method method;

    AroundInvokeInterceptor(int index, Method method) {
        this.index = index;
        this.method = method;

        try {
            final Method finalM = method;
            if(System.getSecurityManager() == null) {
                if (!finalM.isAccessible()) {
                    finalM.setAccessible(true);
                }
            } else {
                java.security.AccessController
                        .doPrivileged(new java.security.PrivilegedExceptionAction() {
                    public java.lang.Object run() throws Exception {
                        if (!finalM.isAccessible()) {
                            finalM.setAccessible(true);
                        }
                        return null;
                    }});
            }
        } catch(Exception e) {
            throw new EJBException(e);
        }

    }
```

```
    Object intercept(final InterceptorManager.AroundInvokeContext invCtx) throws Throwable {
        try {
            final Object[] interceptors = invCtx.getInterceptorInstances();

            if( System.getSecurityManager() != null ) {
            // Wrap actual value insertion in doPrivileged to
            // allow for private/protected field access.
            return java.security.AccessController
                    .doPrivileged(new java.security.PrivilegedExceptionAction() {
                        public java.lang.Object run() throws Exception {
                            return method.invoke(interceptors[index], invCtx);
                        }
                    });
            } else {

                 return method.invoke(interceptors[index], invCtx);

            }
        } catch (java.lang.reflect.InvocationTargetException invEx) {
            throw invEx.getCause();
        } catch (java.security.PrivilegedActionException paEx) {
            Throwable th = paEx.getCause();
            if (th.getCause() != null) {
                throw th.getCause();
            }
            throw th;
        }
    }


class CallbackInterceptor {
    protected int index;
    protected Method method;

    CallbackInterceptor(int index, Method method) {
        this.index = index;
        this.method = method;

        try {
        final Method finalM = method;
        if(System.getSecurityManager() == null) {
            if (!finalM.isAccessible()) {
                finalM.setAccessible(true);
            }
        } else {
            java.security.AccessController
                    .doPrivileged(new java.security.PrivilegedExceptionAction() {
                public java.lang.Object run() throws Exception {
                    if (!finalM.isAccessible()) {
                        finalM.setAccessible(true);
                    }
                    return null;
                }});
        }
        } catch(Exception e) {
            throw new EJBException(e);
        }

    }
```

```
Object intercept(final CallbackInvocationContext invContext)
    throws Throwable {
    try {

        final Object[] interceptors = invContext
                .getInterceptorInstances();

        if( System.getSecurityManager() != null ) {
        // Wrap actual value insertion in doPrivileged to
        // allow for private/protected field access.
        return java.security.AccessController
                .doPrivileged(new java.security.PrivilegedExceptionAction() {
                    public java.lang.Object run() throws Exception {
                        return method.invoke(interceptors[index],
                                             invContext);
                    }
                });
        } else {
            return method.invoke(interceptors[index], invContext);

        }
    } catch (java.lang.reflect.InvocationTargetException invEx) {
        throw invEx.getCause();
    } catch (java.security.PrivilegedActionException paEx) {
        Throwable th = paEx.getCause();
        if (th.getCause() != null) {
            throw th.getCause();
        }
        throw th;
    }
}
```

Also, the methods *load2xLifecycleMethods* and *loadOnlyEjbCreateMethod* can be refactored using a common method (as suggested by the developer himself):

```
private void load2xLifecycleMethods(ArrayList<CallbackInterceptor>[] metaArray) {

    if (javax.ejb.EnterpriseBean.class.isAssignableFrom(beanClass)) {
        int sz = lcAnnotationClasses.length;
        for (int i = 0; i < sz; i++) {
            if (pre30LCMethodNames[i] == null) {
                continue;
            }
            try {
                Method method = beanClass.getMethod(
                        pre30LCMethodNames[i], (Class[]) null);
                if (method != null) {
                    CallbackInterceptor meta =
                            new BeanCallbackInterceptor(method);
                    metaArray[i].add(meta);
                    _logger.log(Level.FINE, "**## bean has 2.x LM: " + meta);
                }
            } catch (NoSuchMethodException nsmEx) {
                //TODO: Log exception
                //Error for a 2.x bean????
            }
        }
    }
}
```

```
//TODO: load2xLifecycleMethods and loadOnlyEjbCreateMethod can be
//  refactored to use a common method.
private void loadOnlyEjbCreateMethod(
        ArrayList<CallbackInterceptor>[] metaArray,
        int numPostConstructFrameworkCallbacks) {
    int sz = lcAnnotationClasses.length;
    for (int i = 0; i < sz; i++) {
        if (lcAnnotationClasses[i] != PostConstruct.class) {
            continue;
        }

        boolean needToScan = true;
        if (metaArray[i] != null) {
            ArrayList<CallbackInterceptor> al = metaArray[i];
            needToScan =  (al.size() == numPostConstructFrameworkCallbacks);
        }

        if (! needToScan) {
            // We already have found a @PostConstruct method
            // So just ignore any ejbCreate() method
            break;
        } else {
            try {
                Method method = beanClass.getMethod(pre30LCMethodNames[i],
                        (Class[]) null);
                if (method != null) {
                    CallbackInterceptor meta = new BeanCallbackInterceptor(
                            method);
                    metaArray[i].add(meta);
                    _logger.log(Level.FINE,
                            "**##[ejbCreate] bean has 2.x style ejbCreate: " + meta);
                }
            } catch (NoSuchMethodException nsmEx) {
                // TODO: Log exception
                //Error for a 2.x bean????
            }
        }
    }
```

As an additional note, the methods *intercept* in the classes *BeanAroundInvokeInterceptor* and *BeanCallBackInterceptor* (this class has not been assigned directly to our group) are completely identical, except for the type of parameter that they accept.

3.10 Initialization and Declaration

3.10.1 LoadOnlyEjbCreateMethod

The boolean *needToScan* should be declared at the beginning of the *for* block.

```
for (int i = 0; i < sz; i++) {
    if (lcAnnotationClasses[i] != PostConstruct.class) {
        continue;
    }

    boolean needToScan = true;
    if (metaArray[i] != null) {
        ArrayList<CallbackInterceptor> al = metaArray[i];
        needToScan =  (al.size() == numPostConstructFrameworkCallbacks);
    }
```

3.11 Method Calls

As far as we know, there are no misplaced method calls and the returned values seem to be used correctly. However, this can only be verified with a higher-level inspection to be applied to a wider area of the application.

3.12 Arrays

In principles, sometimes there is no check on the "index of-out-bound" errors, like in the *load2xLifecyclesMethods*. Here, the range of the index *i* is defined accordingly to the length of the *lcAnnotationClasses* arrays, but is used to access to two different arrays (without any checks): *pre30CLMethodNames* and *metaArray*. Of course, it probably relies on a set of preconditions and invariants that, unfortunately, are not explicitly declared.

```
int sz = lcAnnotationClasses.length;
for (int i = 0; i < sz; i++) {
    if (pre30LCMethodNames[i] == null) {
        continue;
    }
    try {
        Method method = beanClass.getMethod(
                pre30LCMethodNames[i], (Class[]) null);
        if (method != null) {
            CallbackInterceptor meta =
                    new BeanCallbackInterceptor(method);
            metaArray[i].add(meta);
```

Below it is exhibited another example. Here, the *index* used in the *interceptors[index]* is provided when the object is created. Therefore, it is not possible for us to check accurately the correctness of the array indexing.

```
class AroundInvokeInterceptor {
    protected int index;
    protected Method method;

    AroundInvokeInterceptor(int index, Method method) {
        this.index = index;
        this.method = method;
```

```
Object intercept(final InterceptorManager.AroundInvokeContext invCtx) throws Throwable {
    try {
        final Object[] interceptors = invCtx.getInterceptorInstances();

        if( System.getSecurityManager() != null ) {
        // Wrap actual value insertion in doPrivileged to
        // allow for private/protected field access.
        return java.security.AccessController
                .doPrivileged(new java.security.PrivilegedExceptionAction() {
                    public java.lang.Object run() throws Exception {
                        return method.invoke(interceptors[index], invCtx);
                    }
                });
        } else {

             return method.invoke(interceptors[index], invCtx);

        }
```

3.13 Object comparison

No issues.

3.14 Output format

No issues.

3.15 Computation, Comparison and Assignement

No issues.

3.16 Exceptions

3.16.1 Load2xLifecycleMethods

First of all, the *NoSuchMethodException* is catched but not handled. Also, the *beanClass.getMethod* method can raise a *SecurityException* that is not even catched.

```
try {
    Method method = beanClass.getMethod(
            pre30LCMethodNames[i], (Class[]) null);
    if (method != null) {
        CallbackInterceptor meta =
                new BeanCallbackInterceptor(method);
        metaArray[i].add(meta);
        _logger.log(Level.FINE, "**## bean has 2.x LM: " + meta);
    }
} catch (NoSuchMethodException nsmEx) {
    //TODO: Log exception
    //Error for a 2.x bean????
}
```

3.16.2 LoadOnlyEjbCreateMethod

This method has the same issues of the *Load2xLifecycleMethods.*

3.16.3 AroundInvokeInterceptor

In this method, it might have been better to catch more specific exceptions (like the *SecurityException* thrown by *finalM.setAccessible().*

```
try {
    final Method finalM = method;
    if(System.getSecurityManager() == null) {
        if (!finalM.isAccessible()) {
            finalM.setAccessible(true);
        }
    } else {
        java.security.AccessController
                .doPrivileged(new java.security.PrivilegedExceptionAction() {
            public java.lang.Object run() throws Exception {
                if (!finalM.isAccessible()) {
                    finalM.setAccessible(true);
                }
                return null;
            }});
    }
} catch(Exception e) {
    throw new EJBException(e);
}
```

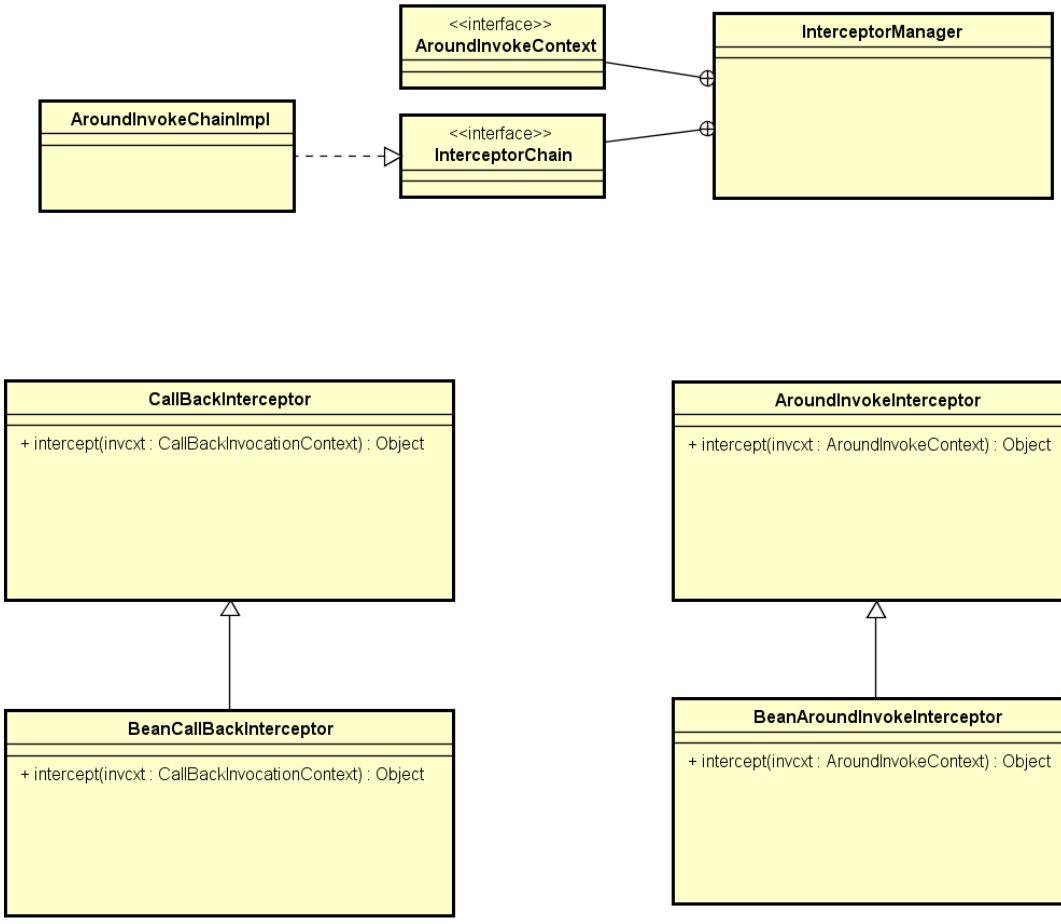
3.16.4 CallbackInterceptor

The same as *AroundInvokeInterceptor.*

4. Other problems

4.1.1 Class Hierarchy

As stated before, the classes *CallBackInterceptor* and *AroundInvokeInterceptor* are identical: they have the very same methods and attributes. The only difference is in the parameter that their methods *intercept* require as argument. The following class diagram clarify the relationship between all the classes in the Java Source File *InterceptorManager*:



The subclasses *BeanCallBackInterceptor* and *BeanAroundInvokeInterceptor* are almost identical as well: their *intercept* methods have really few differences.

This, in our opinion, suggests that the class hierarchy should be reorganized in such a way to redefine *CallBackInterceptor* and *AroundInvokeInterceptor* under a common abstract class that defines the *intercept* method once for all. Even merging the two classes may be a solution, but it probably may compromise their functional role.

5. Appendix

5.1 Software and tools used

- Microsoft Word (https://products.office.com/it-it/word) to redact and to format this document.
- GitHub (https://github.com) to share the working material of this project.
- Eclipse Luna (https://eclipse.org) to visualize the Glassfish's code.

5.2 Hours of work

Alessandro Pozzi: ~10 hours

Marco Romani: ~10 hours