

Politecnico di Milano

A.A. 2015-2016

Software Engineering 2: “myTaxiService”

Code Inspection

Luiza Bentivoglio, Michele Cantarutti

5 January 2016

Summary

**1. INTRODUCTION ............................................................................................................4**

**2. METHODS .....................................................................................................................4**

2.1 LIST OF METHODS ................................................................................................................11

2.2 FUNCTIONAL ROLE OF METHODS ......................................................................................12

**3. CHECKLIST .....................................................................................................................8**

*3.1 Naming conventions .................................................................................................................33*

*2.3.2 Indention ....................................................................................................................................33*

*2.3.3 Braces ..........................................................................................................................................34*

*2.3.4 File organization ....................................................................................................................35*

*2.3.5 Wrapping lines ........................................................................................................................36*

*2.3.6 Comments ..................................................................................................................................37*

*2.3.7 Java source files ......................................................................................................................33*

*2.3.8 Package and import statements .....................................................................................33*

*2.3.9 Class and interface declarations .....................................................................................34*

*2.3.10 Initializations and declarations ...................................................................................35*

*2.3.11 Method calls ..........................................................................................................................36*

*2.3.12 Arrays ......................................................................................................................................37*

*2.3.13 Object comparison .............................................................................................................33*

*2.3.14 Output format ......................................................................................................................33*

*2.3.15 Computation, comparisons and assignments .........................................................34*

*2.3.16 Exceptions ..............................................................................................................................35*

*2.3.17 Flow of control ....................................................................................................................36*

*2.3.18 Files ..........................................................................................................................................37*

**4. DEBUGGING ...............................................................................................................18**

**1. Introduction**

!!!NB va aggiunta altra roba qui?

This document is made up of two main parts. The first part will cover the description of the methods and their functional roles, whereas the second part is a checklist we used as a systematic way to analyze the code that was assigned to us. For the checklist we used this convention: the notes that don’t have a specific number refer to the whole class; the notes that are preceded by a number refer to a specific method, in this way:   
1- refers to method setEJBObjectTargetMethodInfo (line 3333)  
2- refers to method addLocalRemoteInvocationInfo (line 3425)  
3- refers to method addWSOrTimedObjectInvocationInfo (line 3553)

**2. Methods**

2.1 LIST OF METHODS

Here are the methods that were assigned to us.

1- First method, line 3332:

private void setEJBObjectTargetMethodInfo(InvocationInfo invInfo,

boolean isLocal,

Class originalIntf)

throws EJBException {

Class ejbIntfClazz = isLocal ?

javax.ejb.EJBLocalObject.class : javax.ejb.EJBObject.class;

Class[] paramTypes = invInfo.method.getParameterTypes();

String methodName = invInfo.method.getName();

// Check for 2.x Remote/Local bean attempts to override

// EJBObject/EJBLocalObject operations.

if( ejbIntfClazz.isAssignableFrom(originalIntf) ) {

try {

Method m = ejbIntfClazz.getMethod(methodName, paramTypes);

// Attempt to override EJBObject/EJBLocalObject method. Print

// warning but don't treat it as a fatal error. At runtime, the

// EJBObject/EJBLocalObject method will be called.

String[] params = { m.toString(),invInfo.method.toString() };

\_logger.log(Level.WARNING, ILLEGAL\_EJB\_INTERFACE\_OVERRIDE, params);

invInfo.ejbIntfOverride = true;

return;

} catch(NoSuchMethodException nsme) {

}

}

try {

invInfo.targetMethod1 = ejbClass.getMethod(methodName, paramTypes);

if( isSession && isStatefulSession ) {

MethodDescriptor methodDesc = new MethodDescriptor

(invInfo.targetMethod1, MethodDescriptor.EJB\_BEAN);

// Assign removal info to inv info. If this method is not

// an @Remove method, result will be null.

invInfo.removalInfo = ((EjbSessionDescriptor)ejbDescriptor).

getRemovalInfo(methodDesc);

}

} catch(NoSuchMethodException nsme) {

Object[] params = { logParams[0] + ":" + nsme.toString(),

(isLocal ? "Local" : "Remote"),

invInfo.method.toString() };

\_logger.log(Level.WARNING, BEAN\_CLASS\_METHOD\_NOT\_FOUND, params);

// Treat this as a warning instead of a fatal error.

// That matches the behavior of the generated code.

// Mark the target methods as null. If this method is

// invoked at runtime it will be result in an exception from

// the invocation handlers.

invInfo.targetMethod1 = null;

}

}

2.2 FUNCTIONAL ROLE OF METHODS

Now we are going to describe the functional role of our methods.

1- private void setEJBObjectTargetMethodInfo ( InvocationInfo invInfo , boolean isLocal , Class originalIntf ) throws EJBException

At a first glance, we can see the method is private and therefore it can only be invoked from our class. Moreover the method returns void, which means it is only invoked to manipulate some data but it does not actually return anything.

Lastly it can possibly throw an exception.

The first step of our method is creating an object called ejbIntfClazz and checking whether the variable isLocal is true or not, and depending on this ejbIntfClazz is set either as a local object or a remote one. After this, it retrieves the method associated with the information it received as a parameter and it gathers information about this method: firstly the types of its parameters and then its name. Here the code splits in two different branches and if the first one succeeds, the method will return at once. The condition upon which the first branch is entered is a check that checks whether the type (Class) of ejbIntfClazz is either the same as or is a superclass of originalIntf (which has been received as a parameter). In this branch, the method (whose name we got earlier) of ejbIntfClazz is retrieved and stored in a temporal variable called m. The descriptions of method m and the one of the method of parameter invInfo are passed on to the logger, which prompts a warning if an override cannot be done (because the descriptions of the two methods are different). If these operations succeed, the method returns here, otherwise it goes to the second and last branch.

This try-block uses the name and parameters from earlier to retrieve the specified method from ejbClass and this method is set as the target method of parameter invInfo. If there’s an ongoing session and the session is stateful, a new method descriptor is created for the aforementioned target method. The descriptor is assigned to the removal info of invInfo. Should this block fail (because no method was found from instruction invInfo.targetMethod1), an exception will be thrown and the catch-block will start. The role of the catch-block is to notify (through the logger ) that the target method was not found and to set the target method as null.

**3. Checklist**

**3.1 Naming conventions**

The constants declared in our class from line 262 to line 277 aren’t declared using all uppercase.

1- We think all class names, interface names, method names, class variables, method variables and constants have meaningful names, with the exception of an object called ejbIntfClazz, which we found in setEJBObjectTargetMethodInfo, because we feel as though the name is not very clear and doesn’t suggest anything as to what its function might be (the declaration is found at line 3338). Also, at line 3348, a one-character variable is declared and initialized, and this should be avoided.

2- At line 3546, constant EJBLocalObject\_remove is used but its name is not made up of all uppercase.

3- At line 3571, method entrySet() is used but its name is not a verb.

**3.2 File organization**

1- Line 3353 has 84 characters, even though most of the length is due to the long name of a constant.

2- The lengths of lines 3520, 3521, 3522 exceed 80 characters.

3- Line 3571 is 99 characters long and therefore is too long.

**3.3 Comments**

Although there is a comment that explains what the class does, we believe it should be more detailed and thorough because it doesn’t give a clear idea of what the class does.

1- There are some comments inside the method but there isn’t a descriptive comment of the whole method.

2- There’s no comment explaining what the method does at the beginning of the method. Only a few sparse comments are used throughout the method.

3- There’s no description of what the method does and there are no comments inside the method either.

**3.4 Java source files**

Our java source file does not contain one single public class. In fact, two more public classes are declared in the file. The former (PreInvokeException) is declared at line 4998, the latter (ContainerInfo) is declared at line 5010.

The javadoc is not complete, in fact, there’s none at the beginning of our methods.

**3.5 Class and interface declarations**

The class variables declared in our class are declared in a mixed order, so they do not follow the conventional order ( that would be public, protected, package-level, and lastly private).

**3.6 Method Calls**

2- Method addInvocationInfo is often invoked inside our method, but the returned value, which is of type InvocationInfo, is never used. This method is invoked at line 3434, 3442, 3457, 3467, 3488. Lastly, it’s invoked at line 3479, where the value is actually stored in an object called info, but then again this object is passed on to another method (named postProcessInvocationInfo) which doesn’t use the object for anything, as we can see in the code below. Therefore the returned value is useless.

protected InvocationInfo postProcessInvocationInfo ( InvocationInfo invInfo) {

return invInfo;

}

**CHECKLIST METODO 2**

**BUG**

**Si hanno errori nel caso in cui gli oggetti utilizzati all’interno dei blocchi if risultato esser vuoti (il metodo non garantisce l esistenza o meno degli oggetti, poiché le variabili locali del metodo fanno riferimento alle variabili globali della classe).**

Spiegazione metodo 2

* protected void addLocalRemoteInvocationInfo() throws Exception
* nessun parametro, non torna nulla.
* **Se l’oggetto che ha chiamato la funzione è remoto**

**if ( isRemote ) {**

* + *Se l’oggetto è l interfaccia di home remota,*
  + Allora prendo tutti i metodi di remoteIntf e li passo ad addInvocationInfo ed indico al metodo appena chiamato, che si tratta dell’interfaccia remota e gli passo anche remoteIntf.

if( hasRemoteHomeView ) {

// Process Remote intf

Method[] methods = remoteIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method, MethodDescriptor.EJB\_REMOTE,

remoteIntf);

}

* + Inoltre prendo tutti i metodi di homeIntf e li passo ad addInvocationInfo ed indico al metodo appena chiamato, che si tratta dell’interfaccia remota e gli passo anche homeIntf.

// Process EJBHome intf

methods = homeIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method, MethodDescriptor.EJB\_HOME,

homeIntf);

}

*Se l’oggetto è l’interfaccia di business remota,*

per ciascun oggetto di tipo RemoteBusinessIntfInfo contenuto in remoteBusinessIntfInfo:

* + prendo tutti i metodi degli oggetti e li passo ad addInvocationInfo ed indico al metodo appena chiamato, che si tratta dell’interfaccia di business remota, e gli passo anche ciascun oggetto.

if( hasRemoteBusinessView ) {

for(RemoteBusinessIntfInfo next :

remoteBusinessIntfInfo.values()) {

// Get methods from generated remote intf but pass

// actual business interface as original interface.

Method[] methods =

next.generatedRemoteIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method,

MethodDescriptor.EJB\_REMOTE,

next.remoteBusinessIntf);

}

}

Inoltre, prendo tutti i metodi di remoteBusinessHomeIntf e li passo ad addInvocationInfo e gli passo anche remoteBusinessHomeIntf, ed indico che si tratta di un metodo remoto di home.

// Process internal EJB RemoteBusinessHome intf

Method[] methods = remoteBusinessHomeIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method, MethodDescriptor.EJB\_HOME,

remoteBusinessHomeIntf);

}

}

}

**Se l’oggetto è locale**, faccio una cosa analoga

if ( isLocal ) {

if( hasLocalHomeView ) {

// Process Local interface

Method[] methods = localIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

InvocationInfo info = addInvocationInfo(method, MethodDescriptor.EJB\_LOCAL,

localIntf);

postProcessInvocationInfo(info);

}

// Process LocalHome interface

methods = localHomeIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method,

MethodDescriptor.EJB\_LOCALHOME,

localHomeIntf);

}

}

if( hasLocalBusinessView ) {

// Process Local Business interfaces

for(Class localBusinessIntf : localBusinessIntfs) {

Method[] methods = localBusinessIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method,

MethodDescriptor.EJB\_LOCAL,

localBusinessIntf);

}

}

// Process (internal) Local Business Home interface

Method[] methods = localBusinessHomeIntf.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method,

MethodDescriptor.EJB\_LOCALHOME,

localBusinessHomeIntf);

}

}

if (hasOptionalLocalBusinessView) {

// Process generated Optional Local Business interface

String optClassName = EJBUtils.getGeneratedOptionalInterfaceName(ejbClass.getName());

ejbGeneratedOptionalLocalBusinessIntfClass = optIntfClassLoader.loadClass(optClassName);

Method[] methods = ejbGeneratedOptionalLocalBusinessIntfClass.getMethods();

for ( int i=0; i<methods.length; i++ ) {

Method method = methods[i];

addInvocationInfo(method,

MethodDescriptor.EJB\_LOCAL,

ejbGeneratedOptionalLocalBusinessIntfClass,

false, true);

}

// Process generated Optional Local Business interface

Method[] optHomeMethods = ejbOptionalLocalBusinessHomeIntf.getMethods();

for ( int i=0; i<optHomeMethods.length; i++ ) {

Method method = optHomeMethods[i];

addInvocationInfo(method,

MethodDescriptor.EJB\_LOCALHOME,

ejbOptionalLocalBusinessHomeIntf);

}

}

if( !hasLocalHomeView ) {

**// Add dummy local business interface remove method so that internal**

**// container remove operations will work. (needed for internal 299 contract)**

addInvocationInfo(this.ejbIntfMethods[EJBLocalObject\_remove],

MethodDescriptor.EJB\_LOCAL,

javax.ejb.EJBLocalObject.class);

}

}

}