

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. CLASSES ...........................................................................................................................4**

2.1 LIST OF CLASSES ........................................................................................................................11

2.2 FUNCTIONAL ROLE OF CLASSES ..................................................................................................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**

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. Classes**

2.1 LIST OF CLASSES

2.2 FUNCTIONAL ROLE OF CLASSES

setEJBObjectTargetMethodInfo( InvocationInfo invInfo , boolean isLocal , Class originalIntf )

* metodo private, quindi lo chiama solo la classe dove sta, che restituisce void, quindi manipola qualcosa.
* Eccezione EJBException.
* Se la variabile isLocal è true, allora ejbIntfClazz è uguale a un oggetto locale, altrimenti è un oggetto remoto.
* Ottiene le informazioni del metodo (prima i tipi dei parametri e poi il nome del metodo)
* Contolla se ejbIntfClazz è dello stesso tipo o estende originalIntf (parametro passato al metodo)

Se si,

parte un try-catch

estraggo il metodo da ejbIntfClazz, identificato tramite il nome e i tipi dei parametri presi prima->

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

Poi invio la dichiarazione del metodo m e quella del parametro passato al metodo al logger, che farà partire un warning se non sarà possibile effettuare un override. (ovvero se risulteranno due dichiarazioni diverse),

Se il try va a buon fine, finisce il metodo, altrimenti va nel prossimo try.

(( invInfo.ejbIntfOverride = true;

return; ))

Altrimenti

Va direttamente al prossimo try.

**Prendo il metodo interessato, direttamente da ejbClass , che è stato ottenuto dai parametri passati al costruttore della classe**

**// get Class objects for creating new EJBs**

**ejbClass = loader.loadClass(ejbDescriptor.getEjbImplClassName());**

try {

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

public class **MethodDescriptor**

extends [FeatureDescriptor](http://docs.oracle.com/javase/7/docs/api/java/beans/FeatureDescriptor.html)

**A MethodDescriptor describes a particular method that a Java Bean supports for external access from other components. ----🡪 (viene generate una descrizione del metodo)**

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. -🡪 non ho capito a che serve un remove method, potremmo dire semplicemente che rimuove le informazioni di invInfo...**

invInfo.removalInfo = ((EjbSessionDescriptor)ejbDescriptor).

getRemovalInfo(methodDesc);

}

**Non trova nessun metodo dall’istruzione**

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

} catch(NoSuchMethodException nsme) {

**(((((( logParams[0] = ejbDesc.getName(); ->>inizializzto nel costruttore, quindi dovrebbe esserci qualcosa ))))))))))**

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;

}

}

**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 Indention**

**3.3 Braces**

**3.4 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.

**3.5 Wrapping lines**

**3.6 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.

**3.7 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.

**3.8 Package and import statements**

**3.9 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).

**3.10 Initializations and declarations**

**3.11 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;

}

// 20 RIGA 4998 E 5010 DUE CLASSI PUBBLICHE , 21 OK

// 23 MANCA,

// 26 E 27 DA RIGUARDARE (IMPOS)

//28 E 29 CHIEDERE E DA 30 A 33 OK, 34,35,36 DA FARE

//53, riga 3356 il catch non fa nulla… è il comportamento voluto?

**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);

}

}

}