Politecnico di Milano, A.A. 2015/2016

Software Engineering 2: **C**ode **I**nspection

Belotti Nicola 793419
Chioso Emanuele 791621
Colombo Andrea 853381

December 28, 2015

Contents

1 Assigned Methods

1.1 First Method: getConstructor

/** Returns a wrapped constructor element for the specified argument types * in the class with the specified name. If the specified class name is * a persistence-capable key class name which corresponds to a bean * with an unknown primary key class a dummy constructor will also be * returned. Types are specified as type names for primitive type * such as int, float or as fully qualified class names. * @param className the name of the class which contains the constructor * to be checked * @param argTypeNames the fully qualified names of the argument types * @return the constructor element * @see getClass */

```
{
   Object returnObject = null;

   if ((NameMapper.PRIMARY_KEY_FIELD ==
      getPersistenceKeyClassType(className)) &&
      Arrays.equals(argTypeNames, NO_ARGS))
   {
      returnObject = new MemberWrapper(className, null,
          Modifier.PUBLIC,
         (Class)getClass(className));
   }

   if (returnObject == null)
   {
      returnObject = super.getConstructor(className,
          argTypeNames);

      if (returnObject instanceof Constructor)  // wrap it
         returnObject = new
             MemberWrapper((Constructor)returnObject);
   }

   return returnObject;
}
```

1.2 Second Method: getMethod

/** Returns a wrapped method element for the specified method name and argument types in the class with the specified name. If the specified className represents a persistence-capable class and the requested methodName is readObject or writeObject, a dummy method will be returned. Similarly, if the specified class name is * a persistence-capable key class name which corresponds to a bean * with an unknown primary key class or a primary key field (in both * cases there is no user defined primary key class) and the requested * method is equals or hashCode, a dummy method will also be returned. * Types are specified as type names for primitive type such as int, * float or as fully qualified class names. Note, the method does not * return inherited methods. * @param className the name of the class which contains the method * to be checked * @param methodName the name of the method to be checked * @param argTypeNames the fully qualified names of the argument types * @return the method element * @see getClass */

```
public Object getMethod (final String className, final String
    methodName,
   String[] argTypeNames)
{
   int keyClassType = getPersistenceKeyClassType(className);
   Object returnObject = null;

   if (isPCClassName(className))
   {
      if ((methodName.equals("readObject") &&  // NOI18N
               Arrays.equals(argTypeNames, getReadObjectArgs()))
                   ||
         (methodName.equals("writeObject") &&  // NOI18N
                   Arrays.equals(argTypeNames,
                       getWriteObjectArgs())))
      {
         returnObject = new MemberWrapper(methodName,
            Void.TYPE, Modifier.PRIVATE,
                (Class)getClass(className));
      }
   }
   if ((NameMapper.UNKNOWN_KEY_CLASS == keyClassType) ||
      (NameMapper.PRIMARY_KEY_FIELD == keyClassType))
   {
      if (methodName.equals("equals") &&  // NOI18N
               Arrays.equals(argTypeNames, getEqualsArgs()))
      {
         returnObject = new MemberWrapper(methodName,
            Boolean.TYPE, Modifier.PUBLIC,
                (Class)getClass(className));
        }
        else if (methodName.equals("hashCode") &&  // NOI18N
                 Arrays.equals(argTypeNames, NO_ARGS))
        {
            returnObject = new MemberWrapper(methodName,
                Integer.TYPE, Modifier.PUBLIC,
                    (Class)getClass(className));
        }
    }

    if (returnObject == null)
    {
        returnObject = super.getMethod(className, methodName,
            argTypeNames);

        if (returnObject instanceof Method)  // wrap it
            returnObject = new MemberWrapper((Method)returnObject);
    }

    return returnObject;
}
```

1.3 Third Method: getFields

/** Returns a list of names of all the declared field elements in the * class with the specified name. If the specified className represents * a persistence-capable class, the list of field names from the * corresponding ejb is returned (even if there is a Class object * available for the persistence-capable). * @param className the fully qualified name of the class to be checked * @return the names of the field elements for the specified class */

```
public List getFields (final String className)
{
   final EjbCMPEntityDescriptor descriptor =
       getCMPDescriptor(className);
   String testClass = className;

   if (descriptor != null)  // need to get names of ejb fields
   {
      Iterator iterator =
          descriptor.getFieldDescriptors().iterator();
      List returnList = new ArrayList();

      while (iterator.hasNext())
         returnList.add(((FieldDescriptor)iterator.next()).getName());

      return returnList;
   }
   else
   {
      NameMapper nameMapper = getNameMapper();
      String ejbName =
         nameMapper.getEjbNameForPersistenceKeyClass(className);

      switch (getPersistenceKeyClassType(className))
      {
         // find the field names we need in the corresponding
         // ejb key class
         case NameMapper.USER_DEFINED_KEY_CLASS:
            testClass = nameMapper.getKeyClassForEjbName(ejbName);
            break;
         // find the field name we need in the abstract bean
         case NameMapper.PRIMARY_KEY_FIELD:
            return Arrays.asList(new String[]{
               getCMPDescriptor(ejbName).
               getPrimaryKeyFieldDesc().getName()});
         // find the field name we need in the persistence capable
         case NameMapper.UNKNOWN_KEY_CLASS:
            String pcClassName =
               nameMapper.getPersistenceClassForEjbName(ejbName);
```

```
        PersistenceFieldElement[] fields =
          getPersistenceClass(pcClassName).getFields();
        int i, count = ((fields != null) ? fields.length : 0);

        for (i = 0; i < count; i++)
        {
          PersistenceFieldElement pfe = fields[i];

          if (pfe.isKey())
            return Arrays.asList(new
               String[]{pfe.getName()});
        }
        break;
    }
  }

  return super.getFields(testClass);
}
```

1.4 Fourth Method: getField

/** Returns a wrapped field element for the specified fieldName in the * class with the specified className. If the specified className * represents a persistence-capable class, a field representing the * field in the abstract bean class for the corresponding ejb is always * returned (even if there is a Field object available for the * persistence-capable). If there is an ejb name and an abstract bean * class with the same name, the abstract bean class which is associated * with the ejb will be used, not the abstract bean class which * corresponds to the supplied name (directly). * @param className the fully qualified name of the class which contains * the field to be checked * @param fieldName the name of the field to be checked * @return the wrapped field element for the specified fieldName */

lstlisting public Object getField (final String className, String fieldName) String testClass = className; Object returnObject = null;

if (className != null) NameMapper nameMapper = getNameMapper(); boolean isPCClass = isPCClassName(className); boolean isPKClassName = false; String searchClassName = className; String searchFieldName = fieldName;

// translate the class name field names to corresponding // ejb name is abstract bean equivalents if necessary if (isPCClass) searchFieldName = nameMapper. getEjbFieldForPersistenceField(className, fieldName); searchClassName = getEjbName(className); else // check if it is a pk class without a user defined key class String ejbName = nameMapper.getEjbNameForPersistenceKeyClass(class

switch (getPersistenceKeyClassType(className)) // find the field we need in the corresponding // abstract bean (translated below from ejbName) case NameMapper.PRIMARY$_KEY_FIELD : testClass = ejbName; searchClassName = ejbName; isPKClassName = true; break; //findthefieldweneedbycalledupdateFieldWrapper//be$
$needtousethe//persistence-capableclassnameandflagtocallthat//code, soweconfigureitherecaseN$
$testClass = nameMapper.getPersistenceClassForEjbName(ejbName); isPCClass = true; isPKClassName = true; break;$

if (nameMapper.isEjbName(searchClassName)) searchClassName = nameMapper. getAbstractBeanClassForEjbName(searchClassName);

returnObject = super.getField(searchClassName, searchFieldName);

if (returnObject == null) // try getting it from the descriptor returnObject = getFieldWrapper(testClass, searchFieldName); else if (returnObject instanceof Field) // wrap it returnObject = new MemberWrapper((Field)returnObject);

if (isPCClass) returnObject = updateFieldWrapper((MemberWrapper)returnObject, testClass, fieldName); // when asking for these fields as part of the // persistence-capable is key class, we need to represent the // public modifier which will be generated in the inner class if (isPKClassName (returnObject instanceof MemberWrapper)) ((MemberWrapper)returnObject).$_modifiers = Modifier.PUBLIC;$

return returnObject;

1.5 Fifth Method: getFieldType

/** Returns the field type for the specified fieldName in the class * with the specified className. This method is overrides the one in * Model in order to do special handling for non-collection relationship * fields. If it's a generated relationship that case, the returned * MemberWrapper from getField contains a type of the abstract bean and * it's impossible to convert that into the persistence capable class name, so here * that case is detected, and if found, the ejb name is extracted and * used to find the corresponding persistence capable class. For a * relationship which is of type of the local interface, we do the * conversion from local interface to persistence-capable class. In the * case of a collection relationship (generated or not), the superclass' * implementation which provides the java type is sufficient. * @param className the fully qualified name of the class which contains * the field to be checked * @param fieldName the name of the field to be checked * @return the field type for the specified fieldName */ public String getFieldType (String className, String fieldName) String returnType = super.getFieldType(className, fieldName);

if (!isCollection(returnType) isPCClassName(className)) NameMapper nameMapper = getNameMapper(); String ejbName = nameMapper.getEjbNameForPersistenceCl
String ejbField = nameMapper.getEjbFieldForPersistenceField(className, fieldName);

if (nameMapper.isGeneratedEjbRelationship(ejbName, ejbField)) String[] inverse = nameMapper.getEjbFieldForGeneratedField(ejbName, ejbField);

returnType = nameMapper. getPersistenceClassForEjbName(inverse[0]);

if (nameMapper.isLocalInterface(returnType)) returnType = nameMapper.getPersistenceClassForLocalInterface(className, fieldName, returnType);

return returnType;

1.6 Sixth Method: getFieldWrapper

non c' commento.. asd

```
private MemberWrapper getFieldWrapper (String className, String
    fieldName)
   {
      EjbCMPEntityDescriptor descriptor =
          getCMPDescriptor(className);
      MemberWrapper returnObject = null;

      if (descriptor != null)
      {
         PersistenceDescriptor persistenceDescriptor =
            descriptor.getPersistenceDescriptor();

         if (persistenceDescriptor != null)
         {
            Class fieldType = null;

            try
            {
               fieldType =
                   persistenceDescriptor.getTypeFor(fieldName);
            }
            catch (RuntimeException e)
            {
               // fieldType will be null - there is no such field
            }

            returnObject = ((fieldType == null) ? null :
               new MemberWrapper(fieldName, fieldType,
               Modifier.PRIVATE, (Class)getClass(className)));
         }
      }

      return returnObject;
   }
```