VV - Code coverage

Antoine Leval, Damien Vansteen

February 10, 2019

Abstract

Le projet de VV consistait à développer un outil de couverture de test, à réaliser avec Spoon, Javassist ou ASM. Le code source du projet est téléchargeable à l'adresse suivante : https://github.com/Viinyard/Coberturajour.

1 Choix de l'outil

Pour le projet nous avons dans un premier temps tenté de développer l'outil avec Spoon. Mais cet outil était trop contraignant car nous ne pouvions manipuler que du code source. Cela impliquait de penser à tous les cas à parser, ce qui devenait très compliquer pour les lambdas expressions.

De plus cela nous obligeait à compiler deux fois le projet, une fois le projet sans code ajouté, et une fois le projet avec code transformé puis de lancer les tests sur le projet avec code transformé pour evaluer la couverture de test.

Nous avons donc choisi l'outil Javassist. Cet outil offrait l'avantage de pouvoir analyser du byte code java tout en ayant la possibilité d'ajouter du code source dans les fichiers compilé grâce au compilateur intégré à l'outil. Pas besoin donc d'avoir des notions très avancé de byte code Java.

2 Développement

Pour développer notre solution nous avons décidé d'utiliser un POJO maven, de sorte que l'analyse de couverture de test se lance automatiquement à chaque fois que l'on construirait le projet à partir de maven.

Pour la structure du projet nous nous sommes inspiré du projet maven-javassist https://github.com/papegaaij/maven-javassist, qui offre la possibilité d'implémenter un ClassTransformer, appliqué à toute les classes du projet maven.

Pour analyser la couverture de test nous avons écrit une classe statique `Intrumenting.java` qui manipule une collection de type `Map<String, Map<Integer, Boolean>>`. La première map fait références aux classes Java. La deuxième map référence le numéro de ligne de chaque début de block de code.

```
public static Map<String, Map<Integer, Boolean>> lines = new HashMap<String, Map<Integer,
    Boolean>>();
public static void addInstrumentedClass(String qualifiedName) {
   registerClass(qualifiedName);
}
public static void addInstrumentedStatement(String qualifiedName, int position) {
   registerClass(qualifiedName);
   lines.get(qualifiedName).put(position, false);
}
public static void isPassedThrough(String outputDirectory, String qualifiedName, final
    int position) {
   registerClass(qualifiedName);
   lines.get(qualifiedName).put(position, true);
}
private static void registerClass(String qualifiedName) {
   if (lines.get(qualifiedName) == null) {
      lines.put(qualifiedName, new HashMap<Integer, Boolean>());
   }
```

```
}
```

Chaque fois que l'on rencontre un nouveau block, on enregistre cette entrée grâce au nom de la classe, le numéro de ligne, et un boolean à false, pour indiquer qu'il n'est pas couvert.

```
public void applyTransformations(ClassPool classPool, CtClass classToTransform) throws
    TransformationException {
   Instrumenting.addInstrumentedClass(classToTransform.getName());
   for(CtMethod ctMethod : classToTransform.getDeclaredMethods()) {
      try {
        ControlFlow cf = new ControlFlow(ctMethod);
        ControlFlow.Block[] blocks = cf.basicBlocks();
        List<Integer> listLine = new ArrayList<>();
        for(ControlFlow.Block b : blocks) {
           getLogger().info(b.toString());
           int line =
               ctMethod.insertAt(ctMethod.getMethodInfo().getLineNumber(b.position()),
               false, "{ fr.istic.vv.maven.javassist.Instrumenting.isPassedThrough(\""+
               this.getBaseDirectory()+"\", \""+classToTransform.getName()+"\", "+
               b.position()+"); }");
           listLine.add(line);
           getLogger().info("Line : " + line + ", " +
               (ctMethod.getMethodInfo().getLineNumber(b.position() )+ b.index()));
           Instrumenting.addInstrumentedStatement(classToTransform.getName(), line);
        }
        for(int i : listLine) {
           ctMethod.insertAt(i , true, "{
               fr.istic.vv.maven.javassist.Instrumenting.isPassedThrough(\""+
               this.getBaseDirectory()+"\", \""+classToTransform.getName()+"\", "+ i +");
               }");
        }
      } catch (BadBytecode badBytecode) {
        badBytecode.printStackTrace();
      } catch (CannotCompileException e) {
        e.printStackTrace();
      }
   }
}
```

Ici grâce à l'outil ControlFlow offert par Javassist on récupère tous les blocks. Un block est un morceau de code au début ou à la fin duquel un jump est réalisé. On peut voir quels sont les blocks possible de sortie et les blocks possible d'entrée.

On parcours donc chaque block un par un, on ajoute la première ligne de code de chaque block dans une liste, en simulant l'ajout d'une ligne de code (avec la méthode `ctMethod.insertAt()` avec le paramètre modify à false. On référence également cette ligne dans notre classe Instrumenting (initialisé à false pour indiqué qu'elle n'as pas encore été couverte).

Après avoir parcourus tous les blocks d'une méthode on ajoute notre ligne qui appelle `Instrumenting.isPassedThroug` à l'entrée de chaque block.

En ajoutant cette méthode lorsque maven lancera l'éxécution de tous les tests, à chaque fois qu'un test passera dans un block il appelera cette méthode et passera donc ce block à true, pour indiquer qu'il a été couvert.

Ici nous avons dû procéder en deux temps, premièrement stocker les numéros de lignes où nous devions ajouter la ligne de code, puis l'ajout des lignes de code. Car si nous ajoutions les lignes de code au fur et à mesure que l'on parcourait les blocks, nous avions des comportement étrange à cause de la méthode `ctMethod.getMethodInfo().getLineNumber(...)` qui prenait en compte le code ajouté à la volée.

3 Execution

A l'exécution le premier problème que nous avions rencontré était le fait que si le projet n'était pas clean par maven à chaque fois, alors les lignes de code d'instrumentation était ajouté de nouveau

à chaque packaging de l'application.

Pour contrer ce problème, et aussi pour conserver le fait que le packaging finale de l'application ne soit pas polué par notre instrumentation de code qui pourrait baisser les performance et alourdir le code inutilement, nous avous écris nos fichiers .class modifié dans un dossier à part dans le dossier target/classes/jassist.

Nous avons donc dû manipuler le plugin de test maven pour qu'il lance les tests à partir de ce dossier.

Pour faire cela nous avons ajouté ceci dans notre pom.xml

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>3.0.0-M3</version>
    <configuration>
        <classesDirectory>${project.build.outputDirectory}/jassist</classesDirectory>
    </configuration>
</plugin>
```

Mais également dans notre plugin javassist afin que ce dossier puisse rester configurable.

```
<plugin>
    <groupId>fr.istic.vv.maven.javassist</groupId>
    <artifactId>javassist-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <configuration>
        <packageName>pro.vinyard</packageName>
        <outputDirectory>${maven-javassist-directory}</outputDirectory>
    </configuration>
</plugin>
```

Donc lorsqu'on lance le packaging de l'application par maven, au moment de la compilation les classes monitorée sont écrites dans ce dossier, puis au moment des tests, maven appelle nos classes monitorée. Nous avons donc un fichier ou chaque block couvert par les tests est à true et chaque block non couvert est à false.

Afin de générer le rapport de couverture de test nous devions écrire une sauvegarde de celui-ci à la fin de la compilation, (à la fin de notre POJO), puis à chaque appel à la méthode d'instrumentation `isPassedThrought(..)`, nous éditions ce fichier.

En effet l'éxécution du POJO étant terminé les données de la classe d'instrumentation n'était pas persistée.

A l'éxecution des tests il était donc difficile de récupérer tous les blocks non couvert, et de sauvegarder le rapport à la fin de tests.

Nous aurions eu besoin d'un POJO maven s'éxécutant avant de goal de test pour récupérer ces données, puis après le goal de test pour générer le rapport.

L'éxécution d'un POJO ne pouvant dépasser le temps d'éxécution du goal auquel il est assigné on peut difficilement persister des données entre plusieurs goals.

Pour rémédier à cela nous aurions dû utiliser les java-agent, ou créer un serveur local avec qui communiquer tout au long de l'exécution des tests. Nous n'avons pas mis en place cette solution plus optimale par manque de temps.

Pour palier à cela nous avons persisté les données dans un fichier que nous mettons à jour à chaque exécution de la méthode `isPassedThrought(..)`.

Voici un example de rapport générer par notre outil :

```
pro.vinyard.ClassToTest;18;true;
pro.vinyard.ClassToTest;22;true;
pro.vinyard.ClassToTest;24;false;
pro.vinyard.ClassToTest;27;true;
pro.vinyard.ClassToTest;30;true;
pro.vinyard.App;11;false;
```