Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

ATL – Ateliers logiciels

Analyse de la qualité d'un code

Quelques outils

Consignes

Ce TD va vous apprendre à utiliser Maven et ses plugins d'analyse de code. Le temps de travail estimé se compose de 2h de travail en classe et de 2h de travail à domicile. Les ressources nécessaires (fichiers java, fichiers xml) sont disponibles sur PoESI.

1 Maven le moteur de production

Dans une situation concrète, il n'est pas rare que plusieurs équipes développent ensemble une même application en utilisant des outils différents. Régulièrement, il faut mettre toutes les parties ensemble et les *intégrer* afin d'obtenir le produit final. Évidemment, tout ça doit se faire facilement et rapidement, d'où le besoin d'outils qui automatisent ces tâches.

Plusieurs outils — le terme consacré est « moteur de production » — sont utilisés actuellement dans le monde Java : Ant, Maven et Gradle.

Concentrons-nous sur Maven, un produit de la société Apache. C'est la solution la plus utilisée actuellement en entreprise. Le site consacré à Maven le présente ainsi :

> « Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information. »

Lorsque vous construisez votre projet avec Netbeans, cette construction est déléguée à un moteur de production. Par défaut, la version 8.1 de Netbeans, utilise Ant alors que la version 11 de Netbeans utilise Maven.

Installer l'outil

Afin de se familiariser avec Maven, commençons par l'utiliser via les lignes de commande. Vous pouvez installer Maven sur votre machine [1] mais il est possible d'utiliser la version fournie avec Netbeans en configurant correctement les variables d'environnement. Concrètement, il suffit d'ajouter au `PATH` le dossier (`bin`) contenant les binaires de Maven [2].

Ajoutez ce chemin à votre variable d'environnement et exécutez l'instruction suivante dans une invite de commande :

1. https://maven.apache.org/install.html
2. Sur les PC de l'école, ils sont dans le dossier `java/maven` de Netbeans

Version du document
3 février 2020

```
mvn --version
```

Le résultat attendu dépend de votre machine mais doit ressembler à ceci :

```
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: D:\apache-maven-3.6.3\apache-maven\bin\..
Java version: 1.8.0_232, vendor: AdoptOpenJDK, runtime: C:\Program
    ↪ Files\AdoptOpenJDK\jdk-8.0.232.09-hotspot\jre
Default locale: en_US, platform encoding: Cp1250
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

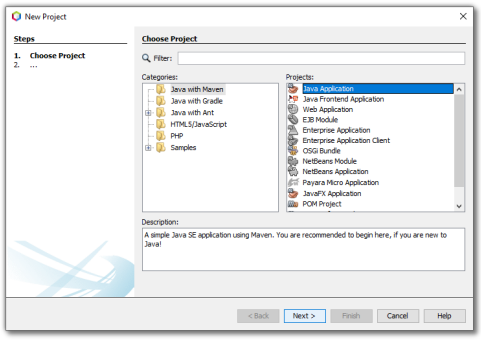
Création d'un projet

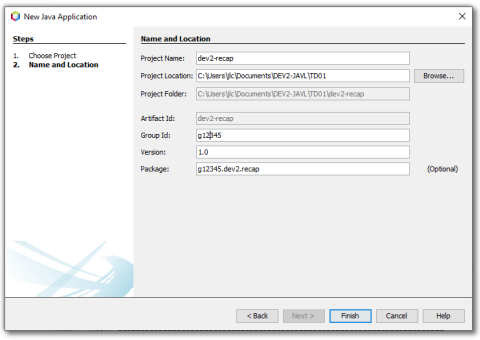
Clonez le dépôt GIT créé pour vous pour cette unité d'enseignement. L'adresse du dépôt doit avoir la forme suivante :

```
https://git.esi-bru.be/2019-2020-JLC/ATLG4/DXXX/g12345.git
```

Dans ce dépôt créez le dossier `ANLL` et le sous-dossier `TD01`. Dans ce dossier `TD01`, créez un projet via NETBEANS en utilisant MAVEN :

- ▷ Nom du projet : `atlg4-intro`.
- ▷ Project location : `ANLL/TD01`.
- ▷ GroupId : *<votreLogin>*.
- ▷ Version : `1.0`.
- ▷ Package : *<votreLogin>*`.atlg4.intro`.





Comme vous pouvez le voir dans NETBEANS, MAVEN a créé une structure de projet JAVA simple. Vous trouverez dans ce projet, sous le dossier PROJECT FILES dans NETBEANS le fichier de configuration (`pom.xml`) qui va permettre à MAVEN de connaître les opérations à effectuer pour construire le projet.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
       ↪ xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       ↪ xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       ↪ http://maven.apache.org/xsd/maven-4.0.0.xsd">
3      <modelVersion>4.0.0</modelVersion>
4      <groupId>g12345</groupId>
5      <artifactId>atlg4-intro</artifactId>
6      <version>1.0</version>
7      <packaging>jar</packaging>
8      <properties>
9          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
10         <maven.compiler.source>13</maven.compiler.source>
11         <maven.compiler.target>13</maven.compiler.target>
12     </properties>
13 </project>
```

Après cette première étape, le fichier de configuration ne contient que des informations sur les propriétés du projet (son nom, sa version, la version de JAVA utilisée,...). Nous reviendrons sur ce fichier par la suite.

Ajoutez dans votre projet une première classe JAVA qui contient une méthode permettant d'additionner deux entiers.

```java
public class App {

   public static void main(String[] args) {
      App app = new App();
      int result = app.sum(30, 12);
      System.out.println("Le résultat est " + result);
   }

   public int sum(int nb1, int nb2) {
      return nb1 + nb2;
   }

}
```

Dans une invite de commande, placez-vous dans le dossier qui contient votre fichier de configuration `pom.xml` et exécutez la commande suivante :

```
mvn package
```

Le projet passe alors par différentes phases : compilation, test... définies par défaut et produit, au final, le *jar* du projet.

Patience !

Ce processus peut prendre du temps la première fois car Maven va automatiquement télécharger ce dont il a besoin à partir d'un dépôt central. Ce ne sera plus le cas les fois suivantes. Tout ce qu'il télécharge est stocké dans le dépôt local, un dossier caché nommé `.m2`. Ce dépôt local est situé dans votre dossier utilisateur sur les machines de l'école.

Explorez maintenant la structure des dossiers généré par MAVEN. Vous y trouverez les dossiers :

- **src** contient toutes les sources au sens large (ce que le développeur fournit) ;
  - **src/main/java** contient les classes JAVA de l'application ;
  - **src/test/java** contient les classes JAVA des tests unitaires ;
- **target** contient tout ce qui est produit et notamment le fichier jar ;
  - **target/classes** contient les versions compilées des classes ;
  - **target/test-classes** contient les versions compilées des classes de test (qui ne seront pas dans le jar produit).

Ajoutez au projet un test unitaire qui valide le fonctionnement de la méthode `sum`.

```java
public void testSum() {
   System.out.println("testSum general case");
   //Arrange
   int nb1 = 10;
   int nb2 = 32;
   //Action
   App app = new App();
   int result = app.sum(nb1, nb2);
   //Assert
   int expected = 42;
   assertEquals(expected, result);
}
```

Exécutez à nouveau la commande suivante :

```
mvn package
```

Vous devriez voir dans les logs que MAVEN a automatiquement exécutez les tests unitaires afin de valider votre développement.

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

> Lorsque vous compilez votre projet avec MAVEN, les tests unitaires sont automatiquement exécutés. Autrement dit un projet dont les tests unitaires échouent, génère une erreur lors de la compilation. Par défaut MAVEN exécute les tests unitaires contenu dans les classes dont le nom se termine par TEST et dont le nom de méthode commence par test. Ces conventions de nommages peuvent être paramétrées : http://maven.apache.org/surefire/maven-surefire-plugin/examples/inclusion-exclusion.html

Si vous consultez le fichier de configuration `pom.xml`, vous devriez voir de nouvelles balises qui gèrent les dépendances dont MAVEN a eu besoin pour construire votre projet.

```xml
<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>5.3.1</version>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-params</artifactId>
      <version>5.3.1</version>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.3.1</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

Les étapes du développement

MAVEN standardise également les étapes par lesquelles passe un projet lors de sa construction. Voici les principales :

clean	Nettoie le projet (détruit le dossier `target`)
compile	Compile les sources
test	Lance les tests unitaires
package	Crée l'artifact (le jar)
install	Copie l'artifact dans le dépôt local
site	Génère la documentation du projet

Toutes ces phases (à part *clean*) se suivent et s'enchaînent. Ainsi, si vous lancez l'étape *package*, il lancera les tests unitaires avant.

Les plugins

L'exécution n'est pas une phase standard du processus [3]. Cette phase est souvent introduite via un **plug-in**. Un plugin peut ajouter de nouvelles phases ou servir à modifier le comportement d'une phase standard.

Par exemple, notre code Java pourrait s'exécuter via la commande

```
mvn exec:java -Dexec.mainClass="g12345.atlg4.intro.App"
```

La différence étant qu'on passe ici par Maven qui se chargera de le construire avant de l'exécuter, si nécessaire.

Via ces plugins nous allons pouvoir par exemple exécuter un projet JavaFX, générer un rapport sur la couverture de test de notre code ou encore générer un rapport contenant toutes les mauvaises pratiques de programmation contenues dans le projet.

Les portées

Comme vous l'avez vu les dépendances permettent d'indiquer ce dont le projet a besoin. mais on peut également préciser quand le projet en a besoin (via l'attribut `scope`). Voici quelques *scopes* fréquemment rencontrés :

compile
: Nécessaire à la compilation et par la suite (scope par défaut). La dépendance est incorporée à l'artifact produit.

runtime
: Nécessaire uniquement pour l'exécution. La dépendance est incorporée à l'artifact produit.

provided
: Nécessaire à la compilation et par la suite (comme `compile`) mais la dépendance est supposée exister dans l'environnement d'exécution et n'est donc pas incorporée à l'artifact produit.

test
: Nécessaire à la compilation et l'exécution des tests. La dépendance n'est *pas* incorporée à l'artifact produit.

Plugin JavaFX

Ajoutons une interface graphique JavaFX à notre projet. Pour ce faire, il faut commencer par ajouter en dépendance les libraires JavaFX. Modifiez votre fichier de configuration `pom.xml` avec la dépendance suivante :

3. Maven a été pensé avant tout pour *construire* un projet.

```xml
<dependency>
   <groupId>org.openjfx</groupId>
   <artifactId>javafx-controls</artifactId>
   <version>13</version>
</dependency>
```

Créez au sein du package `g12345.atlg4.intro` la classe `HelloFX` [4]

```java
package g12345.atlg4.intro;

import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloFX extends Application {

   @Override
   public void start(Stage stage) {
      String javaVersion = System.getProperty("java.version");
      String javafxVersion = System.getProperty("javafx.version");
      Label l = new Label("Hello, JavaFX " + javafxVersion + ", running on Java " +
       ↪ javaVersion + ".");
      Scene scene = new Scene(new StackPane(l), 640, 480);
      stage.setScene(scene);
      stage.show();
   }

   public static void main(String[] args) {
      launch();
   }

}
```

Afin de permettre à Maven d'exécuter votre projet, ajoutez au fichier de configuration `pom.xml` une nouvelle section build qui va définir comment créer l'interface JavaFX via un plugin nommé `javafx-maven-plugin`. Pour créer cette interface, le plugin a besoin de connaître l'emplacement de la méthode `main` de votre application. N'oubliez pas d'adapter le nom du package dans le fichier xml pour votre projet.

4. Cette classe provient de https://openjfx.io/openjfx-docs

```
...
      <dependency>
         <groupId>org.openjfx</groupId>
         <artifactId>javafx-controls</artifactId>
         <version>13</version>
      </dependency>
</dependencies>

<build>
   <plugins>
      <plugin>
         <groupId>org.openjfx</groupId>
         <artifactId>javafx-maven-plugin</artifactId>
         <version>0.0.4</version>
         <configuration>
            <mainClass>g12345.atlg4.intro.HelloFX</mainClass>
         </configuration>
      </plugin>
   </plugins>
</build>

<properties>
   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
...
```

Dans une invite de commande, exécuter la commande

```
mvn clean javafx:run
```

Si vous utilisez le bouton d'exécution du projet au sein de NETBEANS, une erreur survient.

```
Error: JavaFX runtime components are missing, and are required to run this
    ↪ application
Command execution failed.
org.apache.commons.exec.ExecuteException: Process exited with an error: 1
    ↪ (Exit value: 1)
```

En effet NETBEANS est configuré pour utiliser par défaut le plugin `exec-maven-plugin`. Vous pouvez changer ce comportement en modifiant les `Actions` du projet via ses `propriétés`.

N'hésitez pas à consulter la documentation du plugin si vous avez besoin d'une configuration particulière : https://github.com/openjfx/javafx-maven-plugin.

La documentation du projet : mvn site

Afin de demander à MAVEN de générer un compte rendu de l'état du projet nous allons ajouter au fichier de configuration `pom.xml` une section REPORTING.

Commencez par ajoutez à votre `pom.xml` le plugin `maven-site-plugin` pour que MAVEN soit en mesure de générer un rapport après la construction du projet.

```xml
<build>
   <plugins>
      <plugin>
         <groupId>org.openjfx</groupId>
         <artifactId>javafx-maven-plugin</artifactId>
         <version>0.0.4</version>
         <configuration>
            <mainClass>g12345.atlg4.intro.HelloFX</mainClass>
         </configuration>
      </plugin>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-site-plugin</artifactId>
         <version>3.7.1</version>
      </plugin>
   </plugins>
</build>
```

Ensuite ajoutez une nouvelle section REPORTING après la section BUILD.

```xml
<reporting>
   <plugins>
   <!-- Reporting plugin -->
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-project-info-reports-plugin</artifactId>
         <configuration>
            <argLine>-Xmx1024m</argLine>
         </configuration>
         <version>3.0.0</version>
         <reportSets>
            <reportSet>
               <reports><!-- select reports -->
                  <report>index</report>
               </reports>
            </reportSet>
         </reportSets>
      </plugin>

      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-jxr-plugin</artifactId>
         <version>2.3</version>
      </plugin>

      <!-- To publish JUnit test results -->
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-report-plugin</artifactId>
         <version>3.0.0-M3</version>
      </plugin>
   </plugins>
</reporting>
```

Dans une invite de commande exécutez la commande

```
mvn site
```

Vous pouvez à présent consulter la documentation de votre projet qui contient pour l'instant le rapport de tests, c'est à dire le nombre de tests unitaires exécutés, réalisés et erronés. Ce rapport ne semble pas très utile pour l'instant. L'objectif du TD suivant est d'alimenter ce site avec des données pertinentes.

2 Couvertures de tests

> « La couverture de code est une mesure utilisée pour décrire le taux de code source exécuté d'un programme quand une suite de test est lancée. Un programme avec une haute couverture de code, mesurée en pourcentage, a davantage de code exécuté durant les tests ce qui laisse à penser qu'il a moins de chance de contenir de bugs logiciels non détectés, comparativement à un programme avec une faible couverture de code » (wikipedia)

Pour un logiciel développé en JAVA l'outil qui calcule une telle couverture est JAVA CODE COVERAGE[5]. Vous pouvez le téléchargez et l'exécutez en ligne de commande mais la façon la plus pratique de l'utiliser est d'intégrer son plugin dans le fichier de configuration `pom.xml` et de demander à MAVEN de générer un rapport avec le détail de la couverture de tests.

Commencez par ajouter le plugin suivants à votre fichier de configuration `pom.xml` dans la section build du fichier

```
<plugin>
   <groupId>org.jacoco</groupId>
   <artifactId>jacoco-maven-plugin</artifactId>
   <version>0.8.5</version>
   <executions>
      <execution>
         <goals>
            <goal>prepare-agent</goal>
         </goals>
      </execution>
      <execution>
         <id>report</id>
         <phase>prepare-package</phase>
         <goals>
            <goal>report</goal>
         </goals>
      </execution>
   </executions>
</plugin>
```

Ensuite au sein de la section REPORTING ajouter le plugin suivant :

5. https://www.eclemma.org/jacoco/trunk/index.html

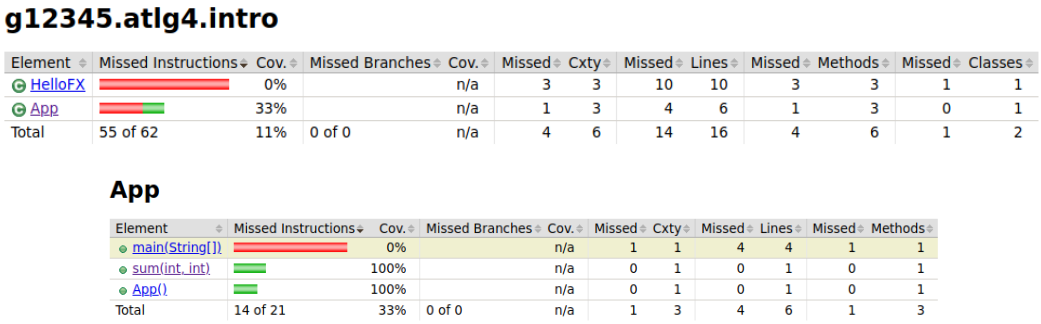
```xml
<!-- To publish Java code coverage -->
<plugin>
   <groupId>org.jacoco</groupId>
   <artifactId>jacoco-maven-plugin</artifactId>
   <version>0.8.5</version>
   <configuration>
      <goals>
         <goal>prepare-agent</goal>
         <goal>report</goal>
      </goals>
   </configuration>
   <reportSets>
      <reportSet>
         <reports>
            <!-- select non-aggregate reports -->
            <report>report</report>
         </reports>
      </reportSet>
   </reportSets>
</plugin>
```

La configuration de ce plugin peut sembler obscure. heureusement nous n'aurons pas besoin de la modifier. Pour plus d'information vous pouvez jet un œil à la documentation.

Générez le rapport via `mvn site`

Lecture du rapport

Si vous consultez le rapport produit, la page d'index du site donne accès au rapport de Java Code Coverage. Ce rapport présente le pourcentage de couverture de tests de chaque package et chaque classe.



g12345.atlg4.intro

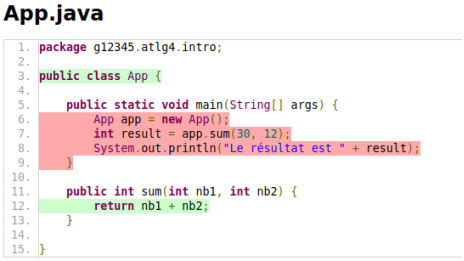
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
HelloFX		0%		n/a	3	3	10	10	3	3	1	1
App		33%		n/a	1	3	4	6	1	3	0	1
Total	55 of 62	11%	0 of 0	n/a	4	6	14	16	4	6	1	2

App

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
main(String[])		0%		n/a	1	1	4	4	1	1
sum(int, int)		100%		n/a	0	1	0	1	0	1
App()		100%		n/a	0	1	0	1	0	1
Total	14 of 21	33%	0 of 0	n/a	1	3	4	6	1	3

On peut y lire également différentes statistiques sur le nombre de ligne testées, le nombre d'embranchement testés ou sur la complexité cyclomatique [6].

Sélectionnez la classe `App.java` pour avoir accès au détail de l'analyse.



App.java

```
1.  package g12345.atlg4.intro;
2.
3.  public class App {
4.
5.      public static void main(String[] args) {
6.          App app = new App();
7.          int result = app.sum(30, 12);
8.          System.out.println("Le résultat est " + result);
9.      }
10.
11.     public int sum(int nb1, int nb2) {
12.         return nb1 + nb2;
13.     }
14.
15. }
```

6. https://fr.wikipedia.org/wiki/Nombre_cyclomatique

Les lignes vertes représentent les lignes couvertes par des tests tandis que les lignes rouges représentent les lignes non couvertes. Dans notre cas seul le contenu de la méthode `main` n'est pas validé par des tests unitaires.

Ajoutez maintenant une nouvelle méthode `int div(int nb1, int nb2)` à votre classe `App`. Cette méthode permet de diviser deux entiers non nuls.

```
public int div(int nb1, int nb2) {
   if (nb1 == 0 || nb2 == 0) {
      throw new IllegalArgumentException("Un des nombres vaut 0 " + nb1 + " " +
    ↪ nb2);
   }
   return nb1 - nb2;
}
```

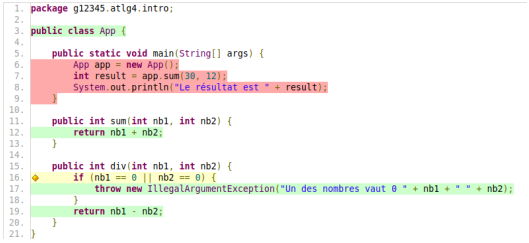
La valeur retournée est incorrecte. Le but est de voir si les tests unitaires vont permettre de découvrir cette erreur. Ajoutez le test unitaire suivant pour valider la méthode `div` :

```
@Test
public void testDiv__when__divisorIsNull() {
   System.out.println("testDiv nb2 is null");
   //Arrange
   int nb1 = 10;
   int nb2 = 0;
   //Assert
   Assertions.assertThrows(IllegalArgumentException.class, () -> {
      //Action
      App app = new App();
      app.div(nb1, nb2);
   });
}

@Test
public void testDiv1() {
   System.out.println("testDiv general case");
   //Arrange
   int nb1 = 4;
   int nb2 = 2;
   //Action
   App app = new App();
   int result = app.div(nb1, nb2);
   //Assert
   int expected = 2;
   assertEquals(expected, result);
}
```

Générez le rapport via `mvn site` et consultez le détail de la couverture de tests de la méthode `div`. Vous constatez l'apparition d 'une ligne jaune.

App.java



```
1.  package g12345.atlg4.intro;
2.
3.  public class App {
4.
5.      public static void main(String[] args) {
6.          App app = new App();
7.          int result = app.sum(30, 12);
8.          System.out.println("Le résultat est " + result);
9.      }
10.
11.     public int sum(int nb1, int nb2) {
12.         return nb1 + nb2;
13.     }
14.
15.     public int div(int nb1, int nb2) {
16.         if (nb1 == 0 || nb2 == 0) {
17.             throw new IllegalArgumentException("Un des nombres vaut 0 " + nb1 + " " + nb2);
18.         }
19.         return nb1 - nb2;
20.     }
21. }
```

Cette ligne jaune signale que seul un cas de la condition a été testé.

Limite de ce rapport

Java Code Coverage est assez pratique pour avoir une diée de la couverture de tests d'un logiciel, et donc avoir une première estimation de sa qualité potentielle. Cependant vous constatez que rien n'a permit d'identifier l'erreur de la méthode `div`[7].

Pour aller plus loin

D'autres outils permettent d'analyser plus en profondeur la qualité des tests d'une application. Une autre méthode est appelée le `Mutation testing`[8]. Le principe général est de soumettre les méthodes de votre programme à de petites modifications et de vérifier que les tests unitaires ne sont plus valides.

Imaginons que votre projet contient une classe dont les tests unitaires sont tous en succès, Cette classe contient une méthode qui présente le test suivant

```
if(nb<=0)
```

Les outils de `Mutation testing` vont modifier le code pour qu'il devienne

```
if(nb<0)
```

Ce changement infime doit provoquer l'erreur doit moins un test unitaire vu que vous avez modifier la logique d'une méthode. Si aucun tests unitaires n'est en erreur après la mutation, vos tests sont considérés comme insuffisant.

Nous n'aborderons pas cette technique au sein de ce cours, mais n'hésitez pas à consulter la documentation de la librairie pitest si le sujet vous intéresse : https://pitest.org/

3 Analyse statique de code

Lors du développement d'un logiciel, le développeur ne bénéficie pas du recul nécessaire sur son propre travail pour en déceler tous les défauts. Lorsqu'on travaille en équipe, il est d'usage d'avoir une personne responsable de la revue de code. Cette personne relit le code écrit et le critique pour en déceler les erreurs formelles ou pour signaler les difficultés de maintenance du code.

Beaucoup de ces défauts peuvent être repérés automatiquement pour peu qu'on possède une liste de ces défauts. Les outils de detection de défauts les plus connus pour Java sont FindBugs, CheckStyle et PMD.

Concentrons nous sur Programming Mistake Detector[9] et ajoutons le plugin nécessaire au sein de la section reporting du fichier de configuration `pom.xml`.

7. L'exemple est volontairement évident.
8. https://en.wikipedia.org/wiki/Mutation_testing
9. https://pmd.github.io/

```xml
<plugin>
<artifactId>maven-pmd-plugin</artifactId>
<version>3.12.0</version>
<configuration>
   <targetJdk>1.8</targetJdk>
   <includes>
      <include>**/*.java</include>
   </includes>
   <outputDirectory>${project.reporting.outputDirectory}</outputDirectory>
   <targetDirectory>${project.build.directory}</targetDirectory>
   <rulesets>
      <ruleset>${basedir}/pmd_ruleset.xml</ruleset>
   </rulesets>
   <minimumTokens>40</minimumTokens>
   <reportSets>
      <reportSet>
         <reports>
            <report>pmd</report>
            <report>cpd</report>
         </reports>
      </reportSet>
   </reportSets>
</configuration>
</plugin>
```

Encore une fois, la configuration est assez complexe, mais nous ne nous attarderons pas sur le détail de chaque balise [10].

La liste des défauts a analyser par Programming Mistake Detector est fournie via un fichier xml intitulé `pmd_ruleset.xml`. Copiez ce fichier à la racine du projet (c'est à dire à coté du `pom.xml`).

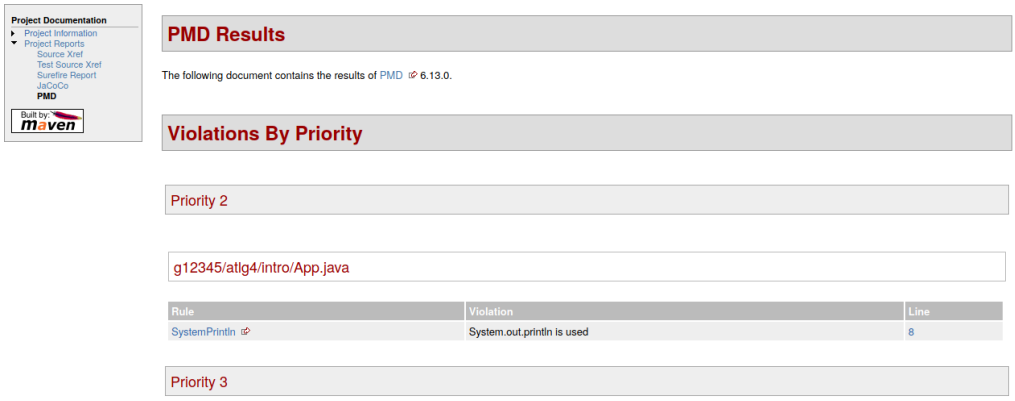
```xml
<?xml version="1.0"?>
<ruleset name="Custom ruleset"
       xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0
      http://pmd.sourceforge.net/ruleset_2_0_0.xsd">

   <description>
      This ruleset checks my code for classic code smells.
   </description>

   <rule ref="category/java/bestpractices.xml"></rule>
   <rule ref="category/java/codestyle.xml"></rule>
   <rule ref="category/java/design.xml">
      <exclude name="LoosePackageCoupling"/>
   </rule>
   <rule ref="category/java/documentation.xml"></rule>
   <rule ref="category/java/errorprone.xml"></rule>
   <rule ref="category/java/multithreading.xml"></rule>
   <rule ref="category/java/performance.xml"></rule>
   <rule ref="category/java/security.xml"></rule>
</ruleset>
```

Générez le rapport via la commande `mvn site` et consultez la partie relative à Programming Mistake Detector, La liste des défauts du programme sont triés par priorité. Les défauts de priorités numéro 1 étant les plus importants.

10. Remarquez que le plugin n'a pas été mis à jour pour les versions récentes de Java.



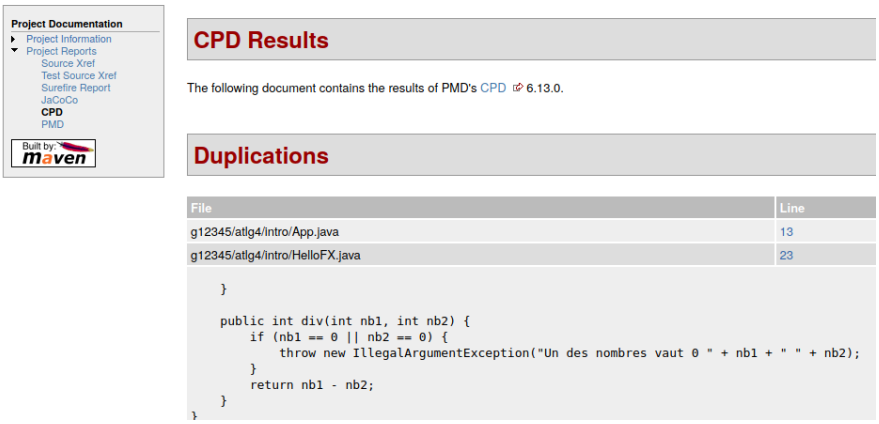
Lorsque vous cliquez sur un défaut, vous êtes dirigé vers la documentation de Programming Mistake Detector Cette documentation explique le détail du défaut et propose une solution pour le corriger.

Consultez la documentation des différents défauts de la classe `App.java`.

Une fois ces corrections effectuées, modifiez votre classe `HelloFX.java` et ajoutez la nouvelle méthode suivante

```java
public int div(int nb1, int nb2) {
   if (nb1 == 0 || nb2 == 0) {
      throw new IllegalArgumentException("Un des nombres vaut 0 " + nb1 + " " +
    ↪ nb2);
   }
   return nb1 - nb2;
}
```

Générez le rapport via la commande `mvn site` et vous constaterez que les doublons ont été détectés et répertoriés dans le menu Copy/Paste Detector.



La détection de code redondant a une certaine sensibilité, configurable via une balise du plugin :

```
<minimumTokens>40</minimumTokens>
```

4 Exercice

Créez un nouveau projet avec Maven et placez-y le code de votre dernier projet Java. Modifiez le fichier de configuration `pom.xml` en y ajoutant les balises vues durant le TD.

Générez la documentation de ce projet et essayez d'améliorer la qualité de ce projet en corrigeant les défauts de celui-ci.