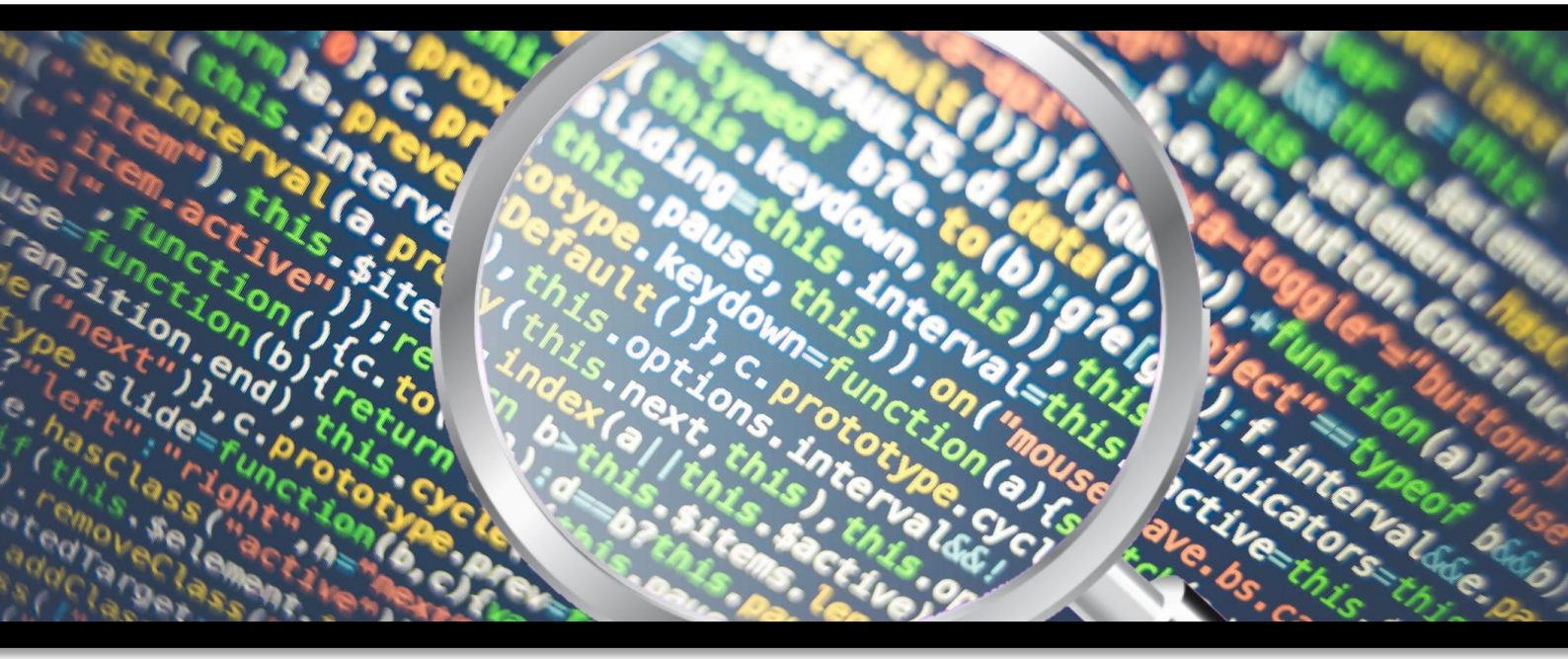
# Projet de validation et vérification : Code Mutation

Par **Charles LARZILLIÈRE** et **Quentin ANGELI**

Sous la direction de Mme Caroline LANDRY et M. Oscar Luis VERA PEREZ

Février 2019



Master 2 IL / ILa

istic Informatique Électronique

UNIVERSITÉ DE RENNES 1

# Présentation du projet : Code mutation

Ce projet de validation et vérification (V&V) nous proposait une liste de différentes méthodes de tests automatiques à mettre en place (disponible ici ). Nous avons choisi de mettre en place l'analyse de mutation (*Mutation Analysis*). Le principe de la mutation est simple : changer certains morceaux du code compilé, puis exécuter la suite de test existante pour voir s'ils détectent les nouvelles erreurs. Une liste de modification à mettre en place nous était proposée :

- Remplacer tous les booléens par *false* dans un premier temps, puis par *true*
- Enlever tout le corps de la méthode d'une fonction *void*
- Remplacer le corps d'une méthode booléennes par *return true* ou *return false*
- Substituer les *+* des méthodes par des - , les * par des */* et vice versa
- Substituer les < par<= et vice versa
- Supprimer les choix alternatifs des booléens (a et b) par des choix simple (a)
- Remplacer les retours d'appel de méthode par une valeur arbitraire
- Remplacer les constantes par une variation de celle ci

Nous avons choisi de mettre en place trois de ces choix : les changement sur les additions, soustractions, multiplications et divisions; les changement sur les supérieur ou inférieur; et la suppression du corps de fonction des méthodes*void*.

# Implémentation

Notre projet est organisé en 2 sous modules par défaut :

*CodeMutation*, qui contient toutes les logiques de modification de code et d'exécution de test.

*SourceCode*, qui correspond au code modifié et testé. Dans une implémentation complète, c'est le dossier qui contiendrait le programme source.

Lors de l'exécution de notre programme, un troisième sous module apparait sous le nom de *SourceModifiedCode*. C'est dans ce dossier qu'est recopier les fichiers .class de *SourceCode* et qui sont par la suite modifiés.

Pour la bonne exécution du programme, il est important de modifier les valeurs contenus dans le fichierDefaultConst.java , présent dans*CodeMutation/main/java/constantes/* avec les chemins correspondants (et plus particulièrement le chemin du dossier où le projet a été cloné pour la valeur *ABSOLUTE_PATH_TO_PROJECT*).

## Additions, soustractions, multiplications et divisions

Pour l'implémentation de la mutation sur les opérations mathématiques simples, nous utilisons *javassist* pour modifier les fichiers .class de notre Source, avant d'exécuter les tests. Le fichier *_MainRunner* s'occupe des boucles d'exécutions.

Dans un premier temps, nous dupliquons le dossier de tests du projet source afin de pouvoir les exécuter tout au long de notre programme. Cela est fait simplement via la fonction *initTestInOutputFolder():void* :

```
private static void initTestInOutputFolder() {
    DefaultConst constantes = new DefaultConst();
    List<String> testsClassesName = new ArrayList<String>();
    testsClassesName.add("MathOperationTest");
    testsClassesName.add("MathSupInfTest");
    try {
        new InstructionModifier().rewriter( inputFolder: constantes.ABSOLUTE_PATH_TO_PROJECT + constantes.defaultPathToTestClasses,
                outputFolder: constantes.ABSOLUTE_PATH_TO_PROJECT + constantes.defaultPathToModifiedTestClasses,
                testsClassesName
                );
    } catch (CannotCompileException e) {
        System.out.println("Error while compiling tests classes: " + e.getReason());
        e.printStackTrace();
    } catch (IOException e) {
        System.out.println("An Error occured while duplicating the tests: " + e.getMessage());
        e.printStackTrace();
    }
    catch (NotFoundException e){
        System.out.println("Error while duplicating tests classes.\nAre you sure the path in Constantes.class is correct?");
        System.out.println("Error message: " + e.getMessage());

    }
}
```

Cette fonction se base sur les chemins par défaut, récupère les classes présente dans le fichier *SourceModifiedCode/target/test-classes/* et les recopie dans le dossier *SourceModifiedClass*. En cas d'erreur, un message console indique à l'utilisateur une des possibilités de l'erreur.

Une fois les tests recopiés, nous exécutons les fonctions de modification des classes. Chaque fonction correspond à la modification d'une variable vers sont opposé. Nous allons vous présenter dans ce rapport le fonctionnement de la modification de l'additions vers la soustraction, les autres modifications faisant appels à des système similaires.

### Exécution des tests

Dans un premier temps, nous exécutons les tests sans aucune modification. Cette exécution se fait grâceà notre classe *TestExecutor* et sa fonction *executeTestFromADistance*:

```
public void executeTestFromADistance(String pathToTestClasses, String pathToImplementation, List<String> classesName)
    JUnitCore core = new JUnitCore();

    core.addListener(new RunListener() {...});

    URL classUrl = new URL( spec: "file://" + pathToTestClasses);
    URL classUrl2 = new URL( spec: "file://" + pathToImplementation);
    URL[] classUrls = {classUrl, classUrl2};
    URLClassLoader ucl = new URLClassLoader(classUrls, getClass().getClassLoader());
    Class c;

    for(String classToLoad : classesName){
       c = ucl.loadClass(classToLoad);
       core.run(c);
    }
```

Cette fonction prend en paramètre le chemin vers les classes de tests (*pathToTestClasses*), le chemin vers les classes testées (*pathToImplementation*) et une liste de noms de classe (*ClassesName*). Nous créons un *JUnitCore*, et allons charger les chemins des classes via un *URLClassLoader*. Ensuite, nous prenons chacune des classes de *ClassesName*, et nous les exécutons.

Nous avons gardé le format de sortie console présent dans les exemples de code pour afficher nos résultats :

```
RUN FINISHED
| IGNORED: 0
| FAILURES: 0
| RUN: 5
```

En cas d'échec du test, le message d'erreur apparait au-dessus des résultats. Après chaque modification, nous exécuterons les tests de la même façon.

## Modification des classes

La modification d'une classe se passe en quatre étapes :

- On compte le nombre d'apparition de l'opérateur à modifier
- On modifie un opérateur
- On exécute les tests
- On annule le changement fait à l'opérateur

On répète ces opérations pour chaque opérateur a modifier.
Pour compter le nombre d'apparition d'un opérateur dans une fonction, on fait appels à la fonction *countOperationInClass* qui prend un paramètre le chemin et le nom de la classe, le nom de la methode et l'opération à compter (*add, sub, div, mul*). Cette fonction renvoie un *int,* différent de 0 si la fonction contient l'opérateur recherché.
Nous bouclons ensuite sur la valeur renvoyée par cette fonction.

Nous faisons ensuite appel à une *PreciseOperation*, qui diffère selon le type de modification que nous voulons apporter. Dans notre exemple, nous modifions les + (*add*) en - (*sub).*

```java
public int addToSubPreciseOperation(String pathToSourceClasses, String pathToOutputFolder, String className, String functionName, int operationNumberWanted)

    ClassPool pool = ClassPool.getDefault();
    //Choose the folder where the sources are
    pool.appendClassPath(pathToSourceClasses);

    // select the class to change
    CtClass functions = pool.get(className);

    //select the function to change
    CtMethod twice = functions.getDeclaredMethod(functionName);
    int currentPosition = 0;
    CodeAttribute codeAttribute = twice.getMethodInfo().getCodeAttribute();
    byte[] code = codeAttribute.getCode();

    int positionModified = -1;

    for (int i = 0; i < code.length; i++) {
        switch (code[i]) {...}
    }
    functions.writeFile(pathToOutputFolder);
    pool.clearImportedPackages();
    functions.defrost();
    return positionModified;
}
```

Cette fonction prend en paramètre un nom de classe et de fonction, ainsi que le numéro de l'opération désiré. Elle changera l'opérateur *add* correspondant en *sub*. Pour détecterles opérateur, nous prenons la méthode en byte code, et comparons ce byte code aux valeurs de la bibliothèque java *Opcode.*
*Opcode* contient les valeurs en byte code *de add, mul, div* et*sub* selon le type. Ansi, une addition entre deux *int*sera un *IADD*, alors qu'une addition entre deux *float* sera un *FADD*.

```
switch (code[i]) {
    case Opcode.IADD:       //int
        if (currentPosition == operationNumberWanted) {
            code[i] = Opcode.ISUB;
            currentPosition++;
            positionModified = i;
        } else {
            currentPosition++;
        }
        break;
```

Ici, un exemple pour les *int*.

Cette fonction retourne la position de l'opérateur modifié, qui sera utilisé dans la fonction *inverseOperatorAtPosition*.

Après chaque modification, les tests sont exécutés comme vu précédemment.

La fonction *inverseOperatorAtPosition*utilise le même switch/case que l'exemple ci-dessus pour inverser directement l'opérateur a la position renvoyée par *addToSubPreciseOperation.*

Une fois l'opération changer avec sa valeur d'origine, la boucle recommence et on passe à l'opération suivante.

### Affichage du résultat

Le résultat s'affiche sur la console pour chaque modification. Le format est le suivant :

```
TESTS AVANT LES MODIFICATIONS - SUB TO ADD
RUN FINISHED
| IGNORED: 0
| FAILURES: 0
| RUN: 5

Debut des modification de la classe MathOperation
Modification de la methode add
Fin de la modification de add
Modification de la methode addDouble
Fin de la modification de addDouble
Modification de la methode substract
Modification de variable position 0
FAILURE: function substract is incorrect: 3 was received when 1 was expected
RUN FINISHED
| IGNORED: 0
| FAILURES: 1
| RUN: 5
Fin de la modification de substract
Modification de la methode add3TimesAndSub1
Modification de variable position 0
FAILURE: function Add 3x -1 is incorrect : 6 was received when 4 was expected
RUN FINISHED
| IGNORED: 0
| FAILURES: 1
| RUN: 5
Fin de la modification de add3TimesAndSub1
Modification de la methode multiply
Fin de la modification de multiply
Modification de la methode divide
Fin de la modification de divide

//////////////////   Fin de l'execution du programme   //////////////////
```

### Supérieur et inférieur

Les modifications des signes <, > ,<= et >= se font de la même façon que pour les *add, sub, mul, div.*

Le code est très similaire, avec pour seul changement les valeurs des *Opcode* en valeurs fixes

### Méthodes void

Comme pour les deux autres fonctionnalités, nous récupérons les méthodes d'une classe, nous vérifions pour chacune si elles sont du type *void* grâce à la méthode *getReturnType*().equals(CtClass.voidType), puis nous remplaçons le code contenu dans la classe grâce à la *méthodesetBody*.

## Améliorations possibles de notre programme

Notre programme est améliorable en plusieurs points :

- Ajouter une interface utilisateur. Celles-cidevrais permettre de simplement gérer les notions de chemin de classe et afficher les résultats d'exécutions
- Ecrire les résultats de l'exécution dans un ou plusieurs fichiers. Nous pourrions garder la structure actuelle de résultats, en consoles, mais dans des fichiers externe, et par type de modifications possible.
- Ajouter de nouvelles mutations, comme celles présentées dans l'introduction de ce rapport
- Changer complètement la structure du code, notamment avec l'utilisation de fonction lambda, pour diminuer le nombre de lignes.
- Ajouter des statistiques sur les résultats des tests.

## Conclusion

Notre projet présente les résultats attendus. Nous arrivons à modifier le code compilé d'un projet source afin de modifier son comportement, et vérifier ainsi la bonne couverture des tests déjà présents dans le projet source.

Finalement, la difficulté fut de mettre en place une structure de projet solide, avec la création de fonction pour lire nos fichiers externes, les modifier et exécuter les tests d'un projet cible depuis un autre projet. Ce dernier point fut pour nous la première difficulté majeure.