DevOps & Continuous Testing

Robin ALONZO

Khadim GNING

Thomas JALABERT

Rémy KALOUSTIAN

Partie 1

Notre architecture

Comment ça marche ?

Nous disposons d'un pom.xml principal qui s'appelle "island-main" qui déclare les projets "Processors" et "AmazingExplorer" en tant que modules. Processors est le projet maven qui contient nos mutateurs, et AmazingExplorer est notre projet à muter. Processors a "spoon" en tant que dépendance, et AmazingExplorer a spoon en tant que plugin, qui permet de demander à spoon de générer les sources mutées quand on effectue "mvn generate-sources" sur le projet AmazingExplorer.

Chaque module contient un fichier pom.xml, le projet en lui-même contient un pom.xml qui ne fait que rediriger vers les deux modules. Le module des mutants dépend de spoon pour construire les processeurs alors que le module Island est dépendant de spoon mais aussi du module de mutateur. Pour résumer, si l'on souhaite muter le projet Island il faut ajouter un processeur dans les dépendances du projet. De plus, il est tout à fait possible d'utiliser plusieurs processeurs en même temps, les mutations s'additionnent dans les sources générées.

Quel script est exécuté ?

Le script "mutation.sh" crée un dossier "generated" qui a une liste de processeurs. Pour chaque processeur dans la liste :

- il créé un dossier à son nom dans le dossier generated,
- puis il modifie le pom.xml de AmazingExplorer pour inclure le processor dans la phase de mutation,
- puis il lance "mvn compile" (en fait "mvn generate-sources" aurait suffit je pense) pour générer les sources mutés,
- il copie dedans les soures générés (mutants), les tests (pas mutés) et un pom.xml qu'on a sauvegardé dans un dossier "ressources/pom.xml" qui permet de compiler et tester le projet normalement.
- puis il va dans ce dossier generated et lance "mvn test"

Dans quel répertoire on met le code généré depuis le mutant ?

Le code généré se situe dans le module Island. Un répertoire "generated-sources" contient un dossier "spoon" avec l'ensemble des sources généré pour un mutant.

Partie 2

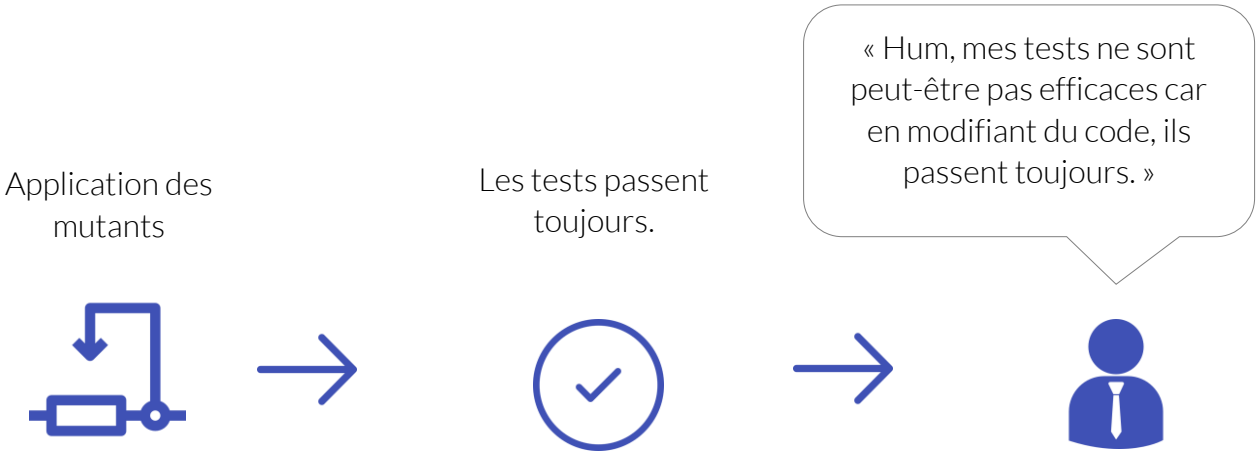
Prise de recul

Dans cette partie, nous allons effectuer une prise de recul sur l'approche par mutation de code. Nous verrons ensuite comment créer de bons mutators.

Commençons avec la question que tout le monde se pose : A quoi servent les mutators ?

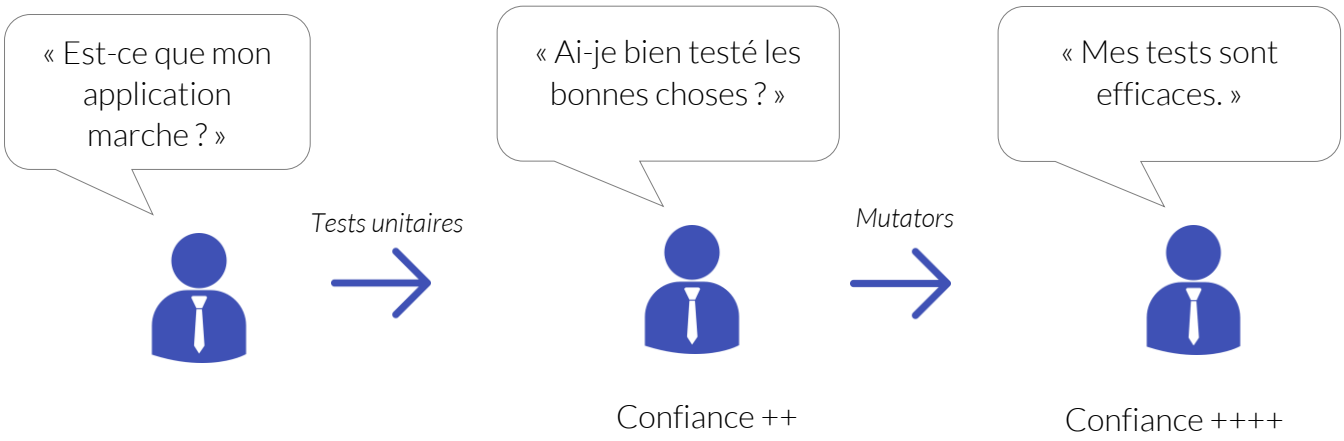
Rappelons tout d'abord l'utilité des tests dans une application. Les tests unitaires servent à s'assurer que l'application produit le comportement attendu. Comment être sûr que les tests s'effectuent sur des parties cruciales de l'application ? C'est là tout l'enjeu des mutators : valider la pertinence des tests.

En bonus, un schéma d'explication :

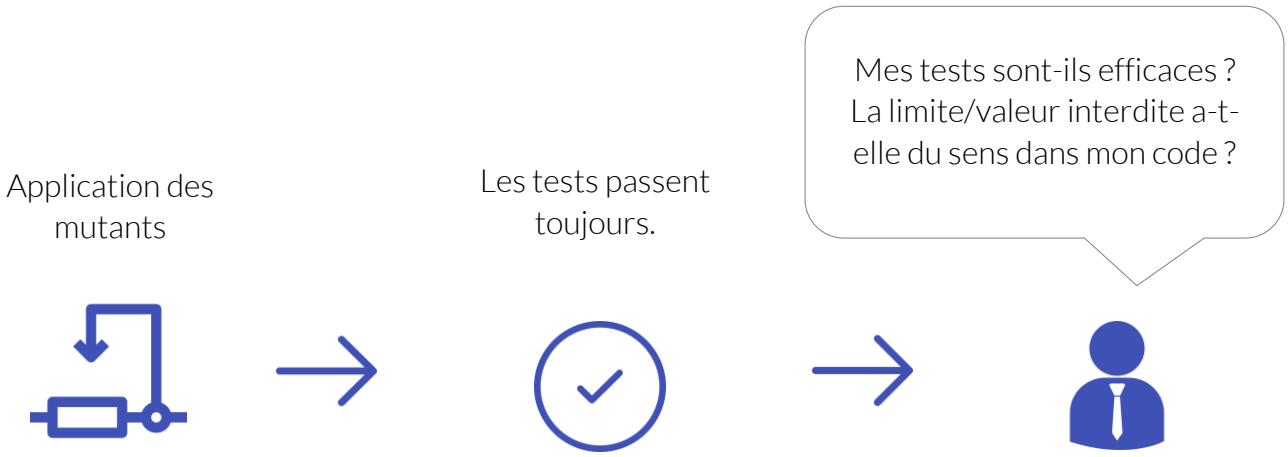


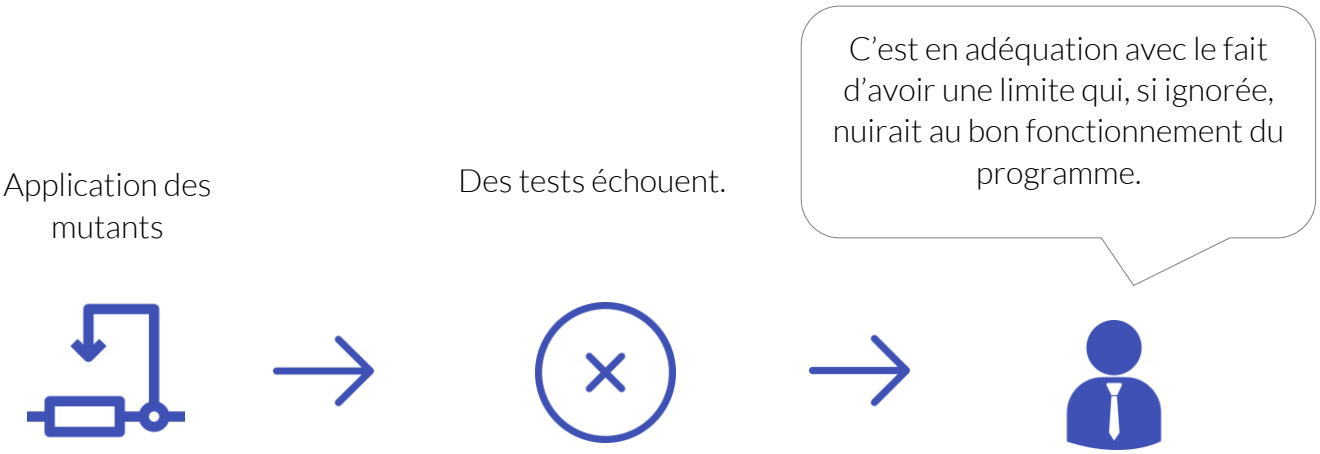
Les mutators sont ainsi un ajout au niveau de la qualité du logiciel.

Premièrement, ils permettent d'agrandir la confiance que l'on a à propos du code. Si les tests unitaires permettent de savoir que l'application a le comportement attendu, les mutators permettent de donner plus de recul sur la pertinence des tests et leur efficacité. Ainsi, les mutators deviennent des testeurs de tests, et accentuent la confiance que l'on a dans notre code.



Ensuite les mutators permettent de détecter des problèmes de cohérence de données. Par exemple, imaginons que l'on mette une limite sur un certaine variable (ou une valeur interdite). Après avoir créé un mutator qui met cette variable au-dessus de la limite (ou lui attribue la valeur interdite) on peut vérifier le comportement de nos tests et distinguer deux cas :





Les problèmes de cohérence de données sont alors mis en avant dans le premier cas.

Enfin, les mutators permettent de détecter du code qui n'est pas testé, ou tout simplement pas utilisé. Par exemple, on réalise un mutator qui s'effectue sur une certaine classe. En lançant les tests, on voit que tous passent. Soit on est dans le cas énoncé précédemment (une modification qui ne change pas le comportement), soit le code muté n'est pas utilisé/testé. Il faut alors remédier à ce problème.

Voyons à présent comment écrire des mutators pertinents.

Trouver un mutator pertinent est plus difficile qu'il n'y paraît. En effet, il faut modifier le code tout en faisant en sorte qu'il puisse toujours être compilé. Cela implique de ne pas faire des mutators trop généraux, qui toucheraient à tout notre code.

Le principal critère de pertinence pour un mutator serait l'importance de ce qu'il mute. Il faut muter les parties du code qui sont critiques pour s'assurer qu'elles sont bien testées.

Par exemple, dans notre projet de QGL, nous devons déplacer un drone sur un grille, de case en case. Si on déplace le drone en dehors de la grille, il explose et la partie est terminée. Si le drone se trompe de coordonnées à une case près, nous perdons.

Pour vérifier cette partie cruciale, nous avons réalisé un mutator qui ajoute 1 à tous les entiers. Nous attendons alors un pourcentage de tests réussis plus faible.