DevOps & Continuous Testing

Robin ALONZO

Khadim GNING

Thomas JALABERT

Rémy KALOUSTIAN

Partie 1

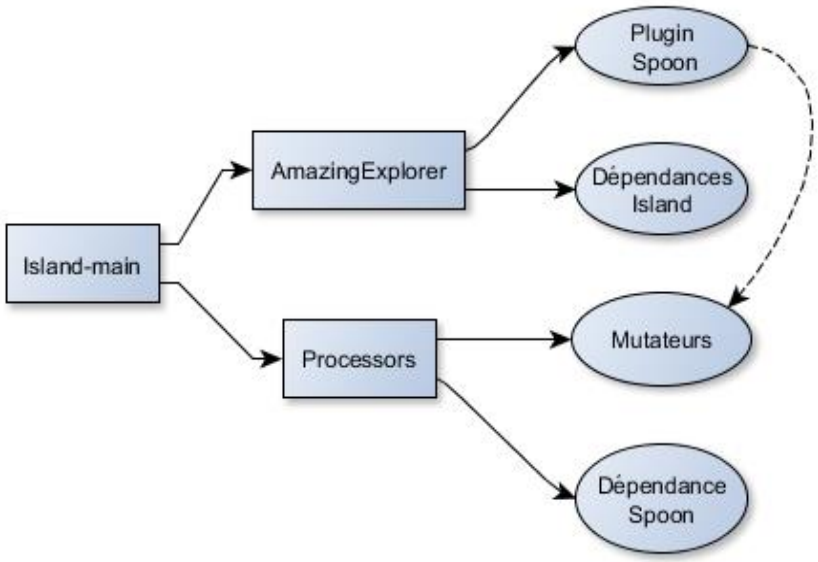
Notre architecture

Comment ça marche ?

Le projet est divisé en deux parties.

D'une part, le projet Island développé en SI3 qui contient un certain nombre de test dont on souhaite vérifier la validité. Auquel nous avons ajouté le plugin spoon pour pouvoir muter le code source. Dans les configurations de build, il est nécessaire de lier le projet à un mutateur pour générer les sources mutées

D'autre part, le projet processor contenant des mutations qui vont effectuer des mutations dans le code source de Island. Il a la dépendance maven "spoon" qui est nécessaire à sa compilation.



Ces deux parties sont intégré en tant que module dans projet principal, portant le nom de Island-main. Lorsque que l'on exécute "mvn compile" sur le projet Island-main.

Chaque module contient un fichier pom.xml, le projet principal contient aussi un pom.xml qui ne fait que rediriger vers les deux modules.

Quel script est exécuté ?

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

1. Il créé un dossier à son nom dans le dossier generated.
2. Il modifie le pom.xml de AmazingExplorer pour inclure le processeur dans la phase de mutation.
3. Puis il lance “mvn compile” pour générer les sources mutés.
4. Il copie dedans les sources 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.
5. Vérifie si le rapport à été fait
6. Déplace le fichier html
7. Et enfin, il va dans ce dossier generated et lance “mvn test”.

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

Le code est tout d’abord généré dans un dossier du module AmazingExplorer (target/generated-sources/spoon). Le script copie les sources mutées vers un autre dossier pour construire un projet maven. Le répertoire “generated” contient tous les projets générés.

Partie 2

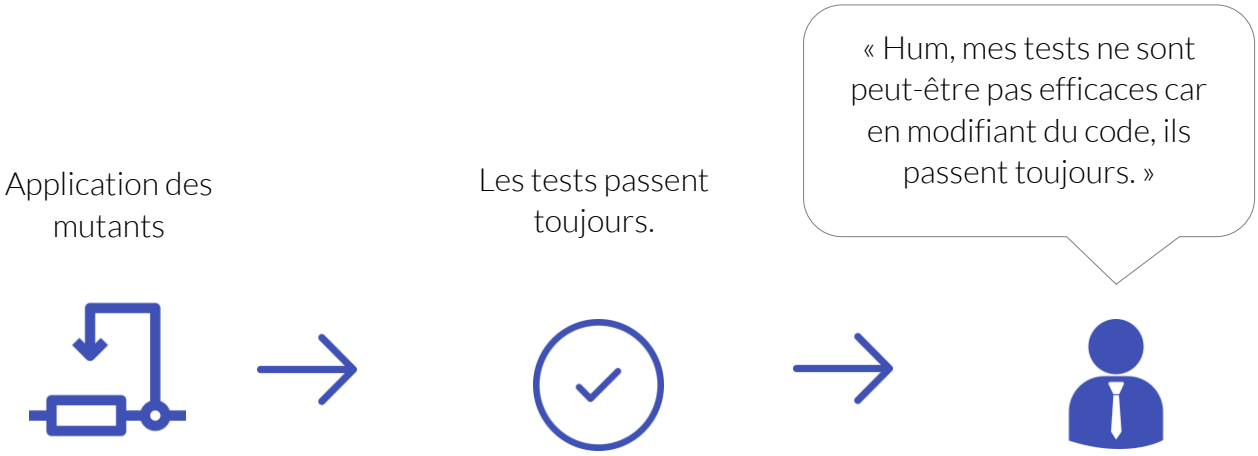
Prise de recul

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 ?

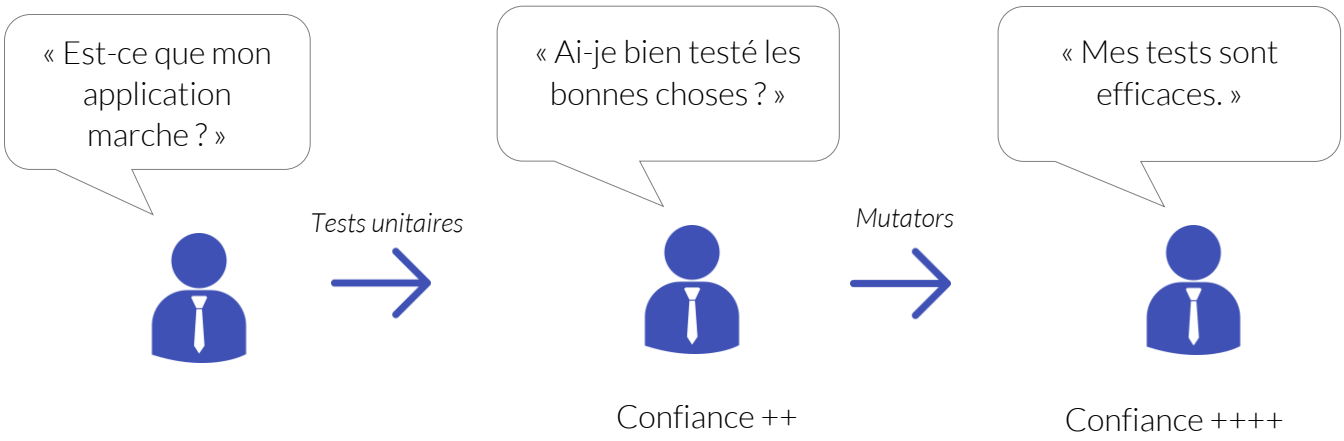
Rappelons tout d'abord l'utilité des tests dans une application. Les tests unitaires servent à s'assurer que l'application produit le comportement attendu. Comment être sûr que les tests s'effectuent sur des parties cruciales de l'application ? C'est là tout l'enjeu des mutators : valider la pertinence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

En bonus, un schéma d'explication :

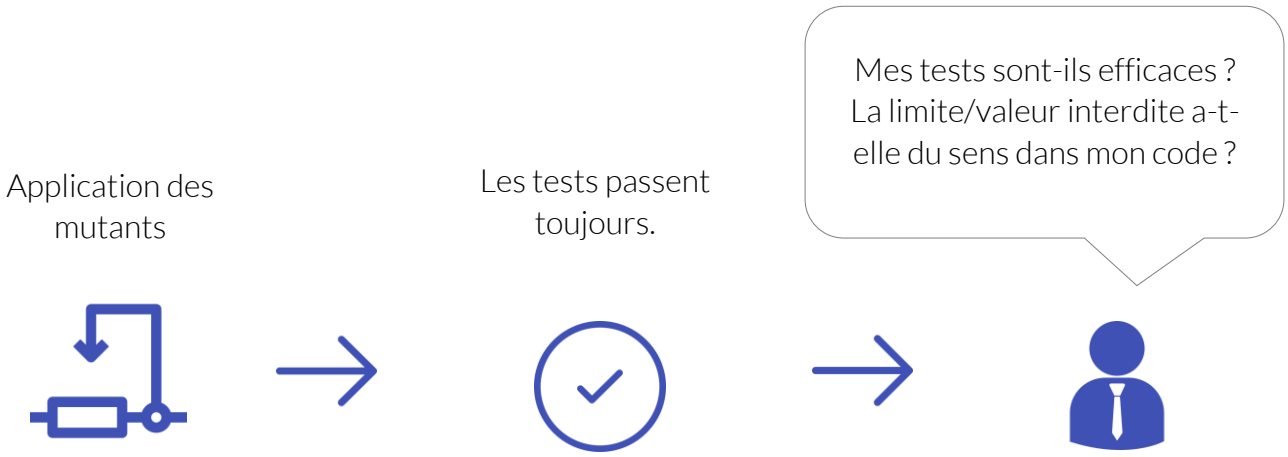


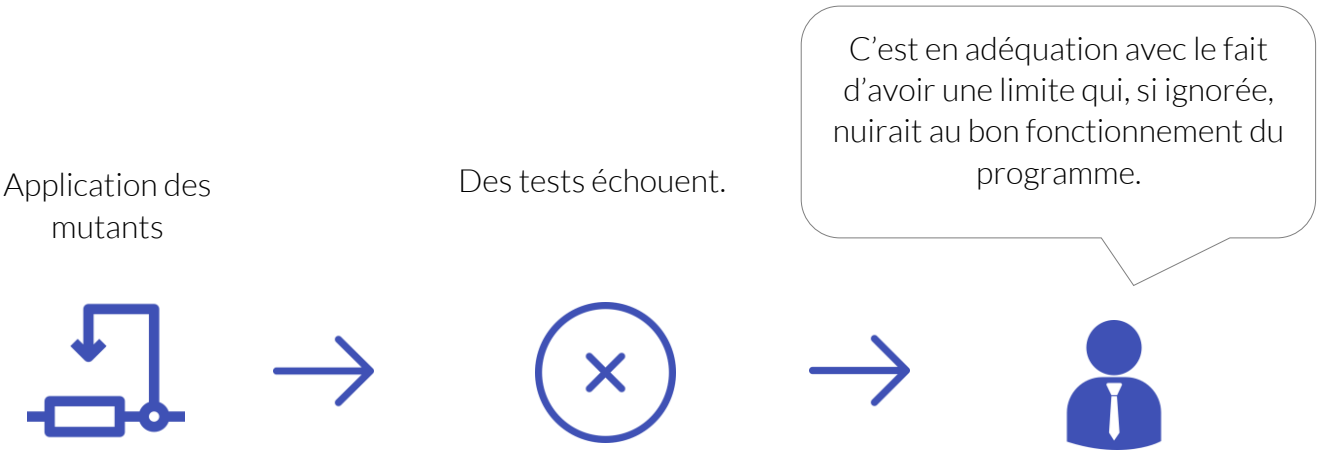
Les mutators sont ainsi un ajout au niveau de la qualité du logiciel.

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.

Mutators développés

Nous avons développé deux mutators qui agissent sur les opérateurs. Le premier remplace un opérateur par son opérateur opposé (division par multiplication …). Le deuxième intervertit l'opérande gauche et l'opérande droite de toutes les opérations binaires.

Nous pensons que ces deux mutators ont de la valeur, car le projet Island dispose de plusieurs éléments cruciaux qui reposent sur des opérations numériques. En effet, elles sont utilisées pour la gestion des points d'actions, la gestion des ressources, et la gestion des coordonnées, trois éléments qui sont au cœur de ce projet. Les transformer via des mutators nous permet alors de voir si nos tests encadrent bien ces éléments.

Le troisième mutator ajoute un à tous les nombres du projet. Il permet ainsi de vérifier les tests sur les aspects mathématiques du projet et les effets de bords sur les conditions. Comme présenté plus tôt, ce mutator est intéressant car il nous permet de créer des cas où l'on sortirait de la carte, ou bien dépasserait une limite d'une unité, ce qui serait suffisant pour nuire au bon déroulement du programme. Nous pouvons ainsi vérifier comment nos tests s'en sortent à partir d'un comportement inattendu.

NB : Afin de visualiser les rapports de test il faut ouvrir le fichier index.html se trouvant dans le dossier generated/reports généré à l'issue de l'exécution du script mutation.sh. Ce fichier contient un lien vers chacun des rapports produits.