UNIVERSITÉ DE MONTRÉAL

Tâche 3 - Documentation

Par
Cédric Guévremont (20266532)
Rima Boujenane (20235550)

TRAVAIL PRÉSENTÉ À BENOIT BEAUDRY
DANS LE CADRE DU COURS IFT 3913
QUALITÉ DU LOGICIEL ET MÉTRIQUES

21 NOVEMBRE 2025

Documentation des modifications du Workflow GitHub Actions

L'objectif de cette tâche est d'intégrer un mécanisme automatisé de *mutation testing* avec PIT dans le pipeline GitHub Actions, d'extraire automatiquement le score de mutation, puis de faire échouer le workflow si le score courant est inférieur à celui obtenu lors de la dernière exécution réussie. Ce mécanisme permet de détecter immédiatement les régressions en qualité de test.

1. Choix de conception

1.1 Exécution ciblée de PIT

PIT est exécuté uniquement sur le module `core`, qui contient la majorité de la logique testée. Ce choix réduit le temps de CI et évite des erreurs sur les modules annexes. Les dépendances du module sont installées au préalable via :

```
mvn -B -DskipTests -DskipITs -pl core -am install
```

1.2 Extraction du score depuis le rapport HTML

Un script Bash (`scripts/pitest_score.sh`) parcourt le fichier `index.html` pour extraire la valeur de *Mutation Coverage* à l'aide de `awk`. Cette solution est simple, portable et indépendante du workflow.

1.3 Baseline et persistance du score

Le score de mutation de la dernière exécution réussie est stocké dans un fichier versionné :

`config/mutation_score.txt`

À chaque exécution du pipeline :

- le score courant est calculé,
- il est comparé à la baseline,
- si le score a diminué, le workflow échoue,
- sinon, la baseline est mise à jour.

Persistance entre exécutions. Le workflow met à jour le fichier localement *et le pousse dans le dépôt* depuis GitHub Actions. Ainsi, le prochain run utilisera toujours comme baseline le score du dernier workflow qui a réussi.

Impact de la matrice Java (24 et 25-ea). Les deux versions Java exécutent le pipeline, ce qui signifie que :

> Le dernier job à se terminer (24 ou 25-ea) met à jour la baseline, car il pousse la nouvelle valeur dans le dépôt.

Cette stratégie est suffisante pour le contexte du TP, car un seul fichier est modifié et il n'y a pas de contributeurs parallèles. Cependant, dans un environnement industriel, on réserverait la mise à jour de la baseline à une seule configuration stable (par exemple Java 24) afin d'éviter les interactions concurrentes entre jobs de la matrice.

2. Comparaison dans le pipeline

La comparaison se fait dans un step Bash :

```
previous=$(tr -d ' \n\r' < config/mutation_score.txt)
current=${{ steps.pit_coverage.outputs.current_pit_coverage }}

if [ "$current" -lt "$previous" ]; then
    exit 1
fi
```

En cas de succès, le fichier est mis à jour et commité par GitHub Actions, assurant ainsi la continuité du mécanisme.

3. Validation

3.1 Tests locaux

L'exécution de PIT, la génération du rapport HTML et l'extraction du score ont été validées localement avant l'intégration CI.

3.2 Validation sur GitHub Actions (succès attendu)

Un premier *push* a permis de confirmer que :

- PIT génère correctement ses rapports;
- le score est extrait et comparé;
- le pipeline reste vert lorsque le score est stable.

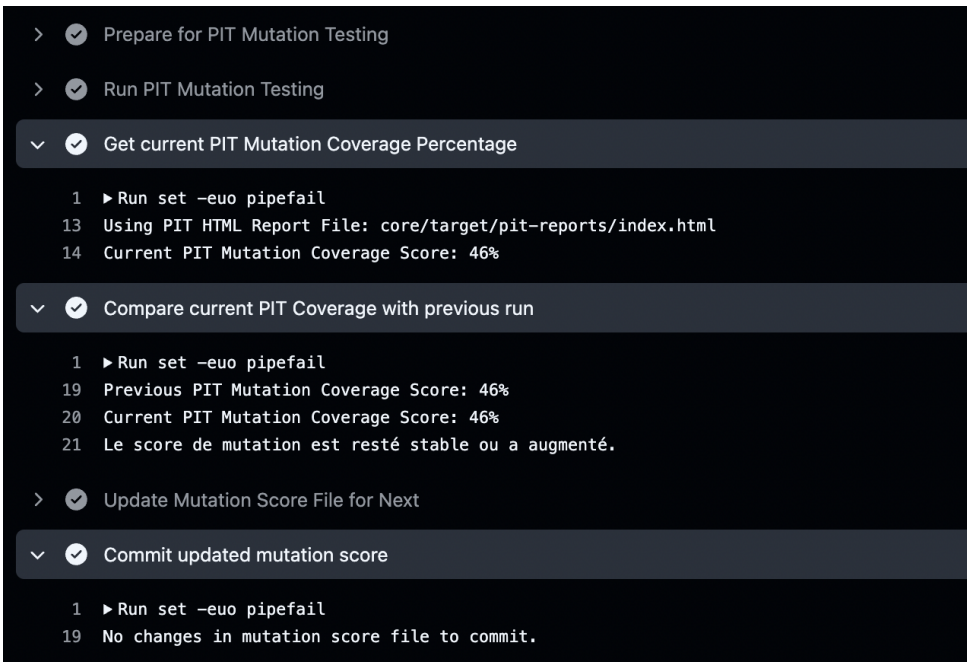

Figure 1: Exécution du pipeline montrant le comportement du pipeline en cas de score stable

3.3 Validation d'une régression PIT

En augmentant manuellement la valeur du fichier `mutation_score.txt`, une régression simulée a fait échouer le pipeline comme prévu.

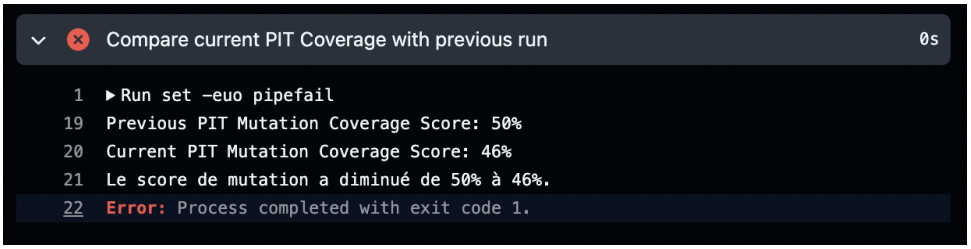

Figure 2: Exécution du pipeline montrant la détection d'une régression PIT.

4. Action humoristique : « Baudry-Roll »

4.1 Objectif et choix de conception

Afin de rendre le pipeline plus informatif (et plus amusant) en cas d'échec, nous avons créé une *action composite GitHub* personnalisée appelée « Rickroll on failure ». Cette action affiche dans les logs un encadré ASCII contenant un message humoristique ainsi qu'un lien vers la célèbre vidéo de Rick Astley.

Nous avons choisi de nommer l'étape principale affichée dans les logs *« Baudry-Roll Activated »*, un clin d'œil à Monsieur Baudry, notre professeur, qui apprécie particulièrement l'humour dans le contexte du cours.

L'action est invoquée automatiquement lorsqu'un job échoue, ce qui permet de terminer l'exécution du pipeline sur une note humoristique tout en signalant clairement la présence d'une erreur.

Plutôt que d'utiliser une action publique générique, nous avons choisi :

- de créer une action locale (`.github/actions/rickroll`),
- d'y définir un format visuel personnalisé lié au contexte du cours,
- d'assurer une intégration simple à l'aide de `if: failure()` dans le workflow.

Cette conception permet de garder un contrôle complet sur le style, le texte affiché et l'intégration dans le pipeline.

4.2 Intégration dans le workflow

L'action est appelée uniquement si l'étape Maven échoue :

```
- name: Build ${{ matrix.java-version }}
  run: mvn -B clean test

- name: Rickroll on Failure
  if: failure()
  uses: ./.github/actions/rickroll
```

4.3 Validation

Pour valider l'action, un test volontairement erroné a été ajouté dans le module `core`. Cela a permis de confirmer que :

- le pipeline échoue bien lorsque les tests ne passent pas ;
- l'action « Baudry-Roll » s'exécute et apparaît correctement dans les logs.

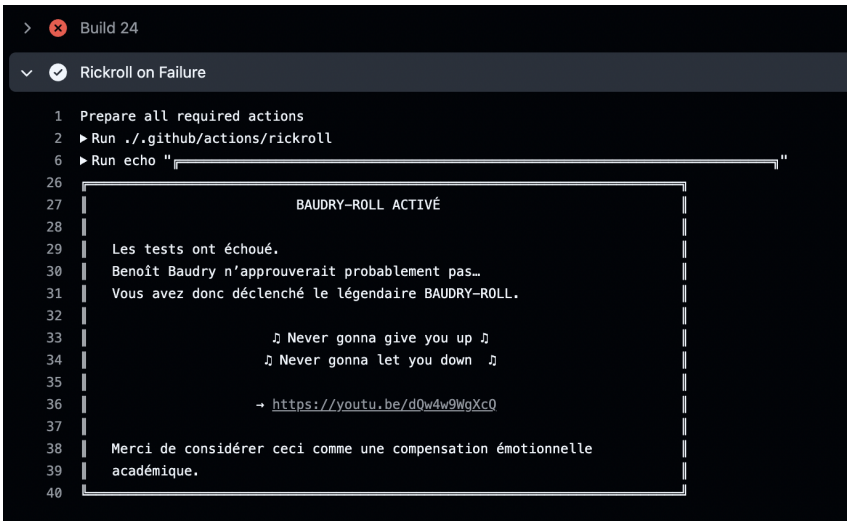

Figure 3: Affichage obtenu du « Baudry-Roll » dans GitHub Actions

Documentation des cas de test – `GHUtility`

Contexte général. La classe `GHUtility` regroupe diverses méthodes utilitaires utilisées dans plusieurs composantes de GraphHopper. Elle est volumineuse, hétérogène et contient plusieurs méthodes statiques comportant des branches conditionnelles ou des comportements dépendants d'objets complexes comme `Graph`, `NodeAccess`, `EdgeIteratorState` ou encore `BaseGraph`. Dans le cadre du devoir, nous avons sélectionné trois méthodes présentant des comportements différents (valide/invalide/exception), adaptées à des tests unitaires avec Mockito et permettant d'augmenter la couverture.

Les tests portent sur :

- `getAdjNode` : méthode simple mais avec une ramification cruciale.
- `getProblems` : méthode complexe itérant sur un graphe, sensible à divers cas limites.
- `getCommonNode` : méthode détectant des erreurs structurelles et lançant des exceptions.

Les classes suivantes ont été simulées (*mockées*) car elles nécessitent un graphe complet ou des structures internes difficiles à instancier à la main : `Graph`, `NodeAccess`, `EdgeExplorer`, `EdgeIterator`, `EdgeIteratorState`, `BaseGraph`. Leur comportement a été réduit à ce qui est strictement nécessaire pour atteindre les branches de code ciblées.

Test 1 – `testGetAdjNode_withValidEdge`

Intention : Vérifier que lorsque l'identifiant d'arête est valide, la méthode `getAdjNode` interroge correctement le graphe et retourne le noeud adjacent fourni par `EdgeIteratorState`.

Choix des classes simulées :

- `Graph` est mocké car un vrai graphe nécessite une structure interne complexe.
- `EdgeIteratorState` est mocké pour contrôler précisément la valeur retournée par `getAdjNode()`.

Définition des mocks :

- `graph.getEdgeIteratorState(edge, adjNode)` retourne un `EdgeIteratorState` simulé.
- `edgeState.getAdjNode()` retourne une valeur arbitraire (42), permettant de vérifier l'aiguillage du code.

Motivation des valeurs : Un identifiant d'arête positif (`edge = 5`) force la prise de la branche "valide". Le noeud adjacent retourné (42) est volontairement distinct de `adjNode = 7` pour s'assurer qu'il provient bien du mock.

Oracle : La méthode doit retourner la valeur donnée par `edgeState.getAdjNode()`, soit 42.

Test 2 – `testGetAdjNode_invalidEdge`

Intention : Valider que lorsqu'un identifiant d'arête invalide est fourni, la méthode ne tente pas d'interroger le graphe et retourne directement le noeud adjoint passé en paramètre.

Choix des classes simulées : Seul `Graph` est mocké, mais il ne doit jamais être appelé. Ce test vérifie une propriété importante : éviter des appels invalides dans le cas d'une arête non valide.

Motivation des valeurs : Nous utilisons `edge = -1`, valeur explicitement non valide selon `Edge.isValid`. `adjNode = 7` permet ensuite de vérifier que la méthode renvoie bien ce paramètre tel quel.

Oracle :

- `getAdjNode` retourne exactement 7.
- `graph.getEdgeIteratorState` ne doit jamais être invoqué (vérification Mockito).

Test 3 – `testGetProblems_InvalidLatitude`

Intention : Vérifier que la méthode `getProblems` détecte correctement une latitude hors bornes et ajoute un message d'erreur à la liste retournée.

Choix des classes simulées :

- `Graph` pour retourner un nombre minimal de nœuds.
- `NodeAccess` pour fournir des coordonnées invalides.
- `EdgeExplorer` et `EdgeIterator` pour simuler l'existence ou non d'arêtes.

Définition des mocks :

- `graph.getNodes()` retourne 1 pour limiter l'itération.
- `na.getLat(0)` retourne `200.0`, valeur invalide ($> 90°$).
- `iter.next()` retourne `false` pour indiquer l'absence d'arêtes et isoler le test sur la borne de latitude.

Motivation des valeurs : Une latitude à 200.0 permet de franchir explicitement la condition :

$$lat > 90 \text{ ou } lat < -90$$

Ainsi, nous testons un cas d'erreur simple, reproductible, et sans dépendance à une structure réelle de graphe.

Oracle : La liste retournée doit contenir un unique message mentionnant « latitude ».

Test 4 – `testGetCommonNode_LoopEdge_ThrowsException`

Intention : Vérifier que la méthode `getCommonNode` détecte un cas d'arête boucle (même nœud base et adjoint) et lance l'exception `IllegalArgumentException` attendue.

Choix des classes simulées :

- `BaseGraph` est mocké pour retourner deux états d'arêtes personnalisés.
- `EdgeIteratorState` est mocké pour définir l'arête considérée comme une boucle.

Définition des mocks :

- `e1.getBaseNode()` et `e1.getAdjNode()` retournent tous deux 4, créant ainsi une boucle.
- `e2` est défini comme une arête normale pour isoler le test uniquement sur le cas d'erreur de `e1`.

Motivation des valeurs : Le cas d'arête boucle est explicitement traité comme erreur dans le code source, ce qui en fait un excellent candidat pour tester la robustesse de la méthode.

Oracle : `getCommonNode(...)` doit lancer immédiatement une `IllegalArgumentException`.

Conclusion

L'extension du pipeline GitHub Actions avec un module de mutation testing automatisé démontre comment des outils DevOps modernes peuvent être exploités pour garantir une qualité constante du code. La mise en place d'une baseline évolutive, l'extraction automatisée du score à partir des rapports HTML et l'intégration conditionnelle d'actions additionnelles montrent la puissance des workflows configurables.

Les validations effectuées confirment que l'approche détecte efficacement les régressions, réagit correctement en cas d'erreur et fournit un retour immédiat aux développeurs. En somme, cette tâche a permis d'implémenter un pipeline CI/CD plus intelligent, plus autonome et marqué par une touche d'humour, tout en respectant les bonnes pratiques professionnelles.