Léna Francisco Do Carmo 71702495

Léia Lebail 71702346

Tests : unitaires (JUnit) vs à base de propriétés (QuickCheck) vs par comparaison (Monolith)

Rapport d'exposé

Programmation Comparée

Nous allons aborder le sujet des tests, en particulier les tests unitaires avec JUnit, les tests à base de propriétés avec QuickCheck et les tests par comparaison avec Monolith. Nous allons donc faire un rappel sur ce qu'est un test et à quoi servent-t-il, puis présenter ces trois librairies et démontrer leurs avantages et inconvénients.

Qu'est-ce qu'un test ?

Un test désigne une procédure de vérification partielle d'un système. Son objectif principal est d'identifier un nombre maximal de comportements problématiques du logiciel. Il permet ainsi, dès lors que les problèmes identifiés seront corrigés, d'en augmenter la qualité.
D'une manière plus générale, le test désigne toutes les activités qui consistent à rechercher des informations quant à la qualité du système afin de permettre la prise de décisions.

Pourquoi tester ?

Selon le projet et le temps les raisons du test sont nombreuses.

On peut tester pour :

- trouver des bugs
- mesurer la qualité et la fiabilité du code / du projet (par exemple pour respecter les normes ISO9126 et ISO25000)
- mesurer les performances du projet, la fluidité ou la réactivité du projet
- s'assurer de l'acceptance du client, vérifier la conformité par rapport au cahier des charges
- vérifier la sécurité, par exemple les fuites de données
- l'interopérabilité, pour permettre une meilleure intégration des mises à jour futures ou les programmes qui utilise le projet

JUnit et tests unitaires

JUnit est un framework de test unitaire pour le langage de programmation Java, créé par Kent Beck et Erich Gamma et dont la première version est sortie en 2002, repose sur des assertions qui vérifient le résultat attendu : exécute les tests et compare avec le résultat attendu.

JUnit définit deux types de fichiers de tests. Les TestCase (cas de test) sont des classes contenant un certain nombre de méthodes de tests. Un TestCase sert généralement à tester le bon fonctionnement d'une classe. Une TestSuite permet d'exécuter un certain nombre de TestCase déjà définis.

Dans un TestCase il n'y a pas de *main* méthod, chaque test étant indépendant.

Il y a plusieurs fonctionnalités de JUnit que nous avons eu l'occasion d'utiliser et d'étudier dans différentes matières notamment Génie Logiciel Avancé en M1 :

- Les annotations

Les annotations sont un mécanisme de JUnit qui a été "copié" de java pour permettre d'indiquer clairement comment traiter une méthode.

Le @Test qui permet d'indiquer les méthodes de test.

Le @Before ou @BeforeEach qui permet d'exécuter une méthode avant d'exécuter les tests ce qui permet de mettre en place un environnement particulier pour réaliser les tests.

Le @After ou @AfterEach qui permet d'exécuter une méthode après avoir exécuté les tests ce qui permet de supprimer l'environnement mis en place.

- Les assertions

À l'aide des méthodes assertEqual() (ou assertTrue()…) de la librairie org.junit.Assert, il est possible de déterminer dans quels cas le test doit échouer ou réussir.

- Suites de tests

Le système des suites de tests permet d'englober plusieurs tests unitaires ensemble et de les lancer ensemble.

- L'intégration continue

Les verdicts des tests peuvent être utilisés dans le cadre de l'intégration continue afin de s'assurer que le code ne régresse pas.

QuickCheck et tests à base de propriété

La première version de QuickCheck est sortie en 1999. Cette librairie, créé par Koen Claessen et John Hughes, a pour but de tester des programmes Haskell automatiquement. Elle a été remaniée a de nombreuses reprises par la communauté pour l'adapter dans plus de 30 langages différents tel que C, C++, Java, JavaScript, OCaml, PHP...

QuickCheck utilise le principe des tests à base de propriété : cela signifie qu'au lieu de s'appuyer sur des valeurs prédéfinies pour vérifier l'exactitude d'un résultat, un test de propriété générera de façon aléatoire les valeurs données en entrée. Ainsi, on pourra tester les différents comportements du code grâce à ces valeurs variées.

QuickCheck utilise également le principe du "shrinking" : lorsque, lors d'un test, une valeur cause un fail, QuickCheck va reprendre la valeur qui cause un problème et va essayer de repasser un test avec une valeur similaire ou réduite (par exemple dans le cas d'un tableau à plusieurs éléments, un second test va donc être retenté avec une valeur retirée du tableau), le tout afin d'essayer de réduire au minimum et trouver le ou les éléments perturbateur qui seront la raison de ce fail, et ce à plusieurs reprises. Ainsi, lors de l'exécution d'un fail, on pourra voir comme message "Failed! Falsified (after x tests and y shrinks)", ce qui signifie que notre test aura rencontré un élément causant le fail au bout de x test, et aura réalisé y shrinks afin de trouver l'élément.

Monolith et tests par comparaisons

Monolith est une bibliothèque Ocaml qui a été créé pour tester d'autres bibliothèques Ocaml grâce à une génération aléatoire de données et de fuzzing.

Le fuzzing (ou test à données aléatoires) est une technique pour tester des logiciels. L'idée est d'injecter des données aléatoires dans les entrées d'un programme. Si le programme échoue (par exemple en plantant ou en générant une erreur), alors il y a des défauts à corriger.

Elle a été créée par François Pottier (créateur entre autres de Menhir) et est sortie en juin 2020, dont la dernière mise à jour date d'août 2021.

Monolith utilise le principe du test par comparaison. Cela nécessite de prendre deux implémentations, une implémentation candidate (celle qu'on veut tester) et une implémentation référente (qui correspond au comportement attendu), de les exécuter de manière simultanée et de comparer les résultats pour obtenir un verdict.

Pour générer des scénarios de tests automatiquement, Monolith, en plus d'une implémentation candidate et une implémentation référente, a besoin d'un fichier qui indique la signature des opérations des deux implémentations et les types nécessaires aux opérations. C'est également dans ce fichier qu'on lance les tests.

Monolith exécute chaque scénario généré sur les deux implémentations en simultané et vérifie que les résultats obtenus sont équivalents, si ce n'est pas le cas, il présente ce scénario sous forme de code ocaml.

Comparaisons

	JUnit	**QuickCheck**	**Monolith**

Langage	Java	Haskell	Ocaml
Type de test	Unitaire	À base de propriété	Par comparaison
Avantages	• Test Unitaires plus intuitifs • Grosse communauté d'utilisateurs	• Génération automatique de données aléatoires • Shrinking	• Génération automatique de données aléatoires • Approche par comparaison plus efficace dans certains cas
Inconvénients	• Tests écrits à la main, fastidieux • Pas de données aléatoires	• Peu intuitif • Difficulté à déterminer les propriétés d'un programme	• Communauté récente donc peu de documentation et d'exemples

Bibliographie

Introduction à JUnit

JUnit

Shrinking

QuickCheck

QC exemple

Test à base de propriété

Monolith

Documentation Monolith

Test par comparaison