Test-Driven Development

James Hoffman

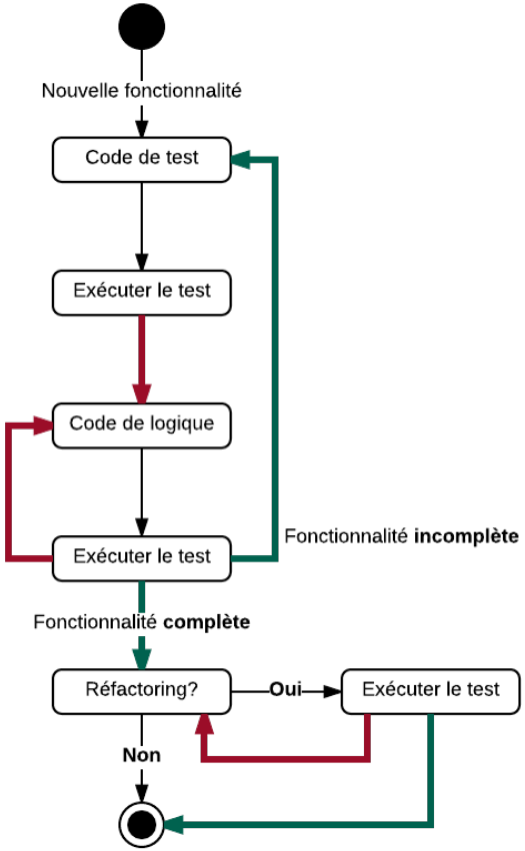
Introduction

- Les programmeurs s'efforcent d'écrire du code sans bugs.
- Dans le meilleur des cas le code ne compile tout simplement pas, à l'opposé, il compile mais produit un résultat inattendu. C'est alors qu'il faut déboguer!
- Ne serait-il pas merveilleux d'avoir un outil semblable au compilateur, mais qui valide le comportement du code...

La solution, le TDD!

- Valider l'intention du programmeur, pas seulement la validité syntaxique du code.
- Processus itératif qui force le développeur à écrire de façon incrémentale les tests d'un logiciel et le code permettant de répondre à ces tests.
- Le logiciel est divisé en petites unités logiques, plus simples à tester et à implémenter. On parle donc de tests unitaires.
- Tout le code implémenté doit l'être dans le but de réussir un test.

Cycle de développement



3 règles du TDD

- Écrire uniquement du code de logique qui permet à un test qui échoue de réussir
    - Avant d'écrire du code de logique, il faut écrire le code de test
- Écrire le minimum de code dans un test pour qu'il échoue, erreurs de compilation incluses.
- Écrire le code de logique minimum pour permettre au test qui échoue de réussir

Comment le mettre en pratique?

- Écrire le test avant le code
  - Toutes les fonctionnalités sont testées
  - Valide que le code répond au besoin
  - La qualité du logiciel est la priorité #1
  - Rédiger les tests avec soin
- Structure
  - Initialisation
  - Exécution
  - Validation
  - Nettoyage
- On teste tout! Ou presque...
  - Code qui exécute de la logique
  - Les méthodes privées peuvent habituellement être ignorées
  - L’interface utilisateur est difficile à tester et peu performante

Simuler les interactions externes

- Les interactions externes ne doivent pas être testées
  - Accès à une base de données;
  - Requêtes à vers un service web, etc.
- Utilisation de composants imitant le comportement original (mock)
  - Permets de réduire le temps d'exécution des tests
- Définir des contrats pour les interactions externes
  - L'inversion de dépendance permet de réduire le couplage
- Quand tester les interactions externes?
  - Lors des tests d'intégration

Exemple

```
@Test
public void multiplicationOfZeroIntegersShouldReturnZero() {

  // MyClass is tested
  MyClass tester = new MyClass();

  // assert statements
  assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
  assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
  assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
}
```

http://www.vogella.com/tutorials/JUnit/article.html

Avantages

- Encourage :
  - Une architecture simple, faiblement couplée
  - La séparation efficace des responsabilités
  - La conception des interfaces/contrats avant l'implémentation
  - La décomposition du logiciel en petites parties plus simple à coder
  - La détection précoce des bugs
- La confiance envers le produit rend le débogage et réfactoring moins intimidant
- Maintenabilité accrue, car le minimum de code est écrit pour répondre aux spécifications
- Les tests servent de documentation qui présente tous les usages possibles du code
- **La productivité est améliorée!**

Limitations

- Faux sentiment de sécurité. Le TDD ne signifie pas qu'on peut délaisser les autres types de tests (ex. Analyse, Intégration, Utilisateurs, etc.).
- Demande un effort constant pour maintenir la qualité et la validité des tests, surtout lors de l'ajout d'une fonctionnalité qui provoque une régression.
- Attention au code de tierces parties qui parfois n'est simplement pas testé ou qui introduit des bugs après une mise à jour.

Pièges à éviter

- Tests dépendants d’un état global
- Tests interdépendants
- Tester les détails d’implémentation
- Tests lourds ou lents à exécuter
  - Interface utilisateur
  - Données en temps réel
  - Accès à une base de données, au système de fichiers ou au réseau
- Délaisser les bonnes pratiques de génie logiciel pour faciliter les tests
- **Le code de test doit être produit et maintenu avec la même rigueur que le code de logique!**

Conclusion

- Bien entendu, les tests ont un impact positif, mais leur capacité à constamment augmenter de façon incrémentale la qualité globale du logiciel l'est encore plus.
- On peut ainsi affirmer que le TDD fait bien plus que seulement vérifier le fonctionnement du code, il en améliore également l'architecture.
- Devrait-on donc plutôt penser au TDD en terme de Test-Driven *Design*?

Aller plus loin

- Autres types de tests
  - Tests de spécifications
  - Tests d’analyses
  - Tests d’architecture
  - Tests système/d’intégration
  - Tests d’utilisateurs et d’approbation
- Méthodologies Agile; Scrum/XP

- Robert “Uncle Bob” C. Martin
- Martin Fowler
- Kent Beck

Références

- Introduction to Test Driven Development (TDD)
- James Shore: The Art of Agile Development: Test-Driven Development
- TestDrivenDevelopment
- Test Driven Development (TDD): Best Practices Using Java Examples | Technology Conversations
- ArticleS.UncleBob.TheThreeRulesOfTdd
- TDD Global Lifecycle - Test-driven development - Wikipedia, the free encyclopedia
- Test-Driven Design | Dr Dobb's
- The Full Life Cycle Object-Oriented Testing (FLOOT) Method