Introduction Test

Ihab ABADI - UTOPIOS

Test et clean Test

- Le génie logiciel est une question de savoir-faire où nous devons écrire toutes les parties du logiciel avec le même soin, qu'il s'agisse de code de production ou de test.
- Rédiger des tests fait partie de notre savoir-faire.
- Nous ne pouvons avoir un code propre que si nous avons des tests propres.
- Un test propre se lit comme une histoire.
- Un test propre doit contenir toutes les informations nécessaires pour comprendre ce qui est testé.

Clean Test

- Imaginons que nous construisons une application de boutique en ligne. Les utilisateurs peuvent rechercher des produits, les sélectionner, collecter les produits dans un panier et enfin les acheter. Dans le cadre de l'application, nous avons un cas d'utilisation avec les spécifications suivantes :
- Étant donné un panier contenant des produits d'un total de 50 000 => Given
- Lorsque le total est calculé => When
- Ensuite, il renvoie 50 en tant que total => Then

Ecriture d’un test clean

- Le nom d'un test doit révéler le cas de test exact, y compris le système testé.
- Il doit spécifier l'exigence du cas de test aussi précisément que possible.
- L'objectif principal d'un bon nom de test est que si un test échoue, nous devrions être en mesure de récupérer la fonctionnalité cassée à partir du nom du test.
- Deux conventions de dénomination populaires :
  - **GivenWhenThen** (ex : GivenUserIsNotLoggedIn_whenUserLogsIn_thenUserIsLoggedInSuccessfully)
  - **ShouldWhen** (ex : ShouldHaveUserLoggedIn_whenUserLogsIn)

Ecriture d’un test clean

- L'utilisation de noms propres, significatifs et révélateurs d'intention dans les tests est aussi important que l'utilisation de noms propres dans le code de production. Par conséquent, nous devrions utiliser des dénominations propres dans des domaines tels que :
  - noms d'éléments logiciels (par exemple : noms de classe, de fonction, de variable)
  - scénarios de préparation
  - exécution du système testé
  - affirmations des comportements attendus
- Conseils pour nommer vos éléments logiciels :
  - suivre les conventions de nommage
  - ne polluez pas les noms avec des détails techniques
  - utiliser des dénominations fonctionnelles relatives au domaine métier
  - utiliser des constantes nommées pour les nombres/chaînes magiques
  - utiliser des noms prononçables
  - ne pas utiliser d'abréviations personnalisées
  - être explicite plutôt qu'implicite
  - révéler l'intention avec des fonctions bien nommées au lieu d'utiliser des commentaires

Ecriture d’un test clean - AAA

- Le modèle Arrange-Act-Assert est une manière descriptive et révélatrice d'intention de structurer des cas de test. Il prescrit un ordre des opérations:
  - La section Arrange doit contenir la logique de configuration des tests. Ici, les objets sont initialisés et préparés pour l'exécution des tests.
  - La section Act invoque le système que nous sommes sur le point de tester. Il peut s'agir par exemple d'appeler une fonction, d'appeler une API REST ou d'interagir avec certains composants.
  - La section Assert vérifie que l'action du test se comporte comme prévu. Par exemple, nous vérifions ici la valeur de retour d'une méthode, l'état final du test , les méthodes que le test a appelées, ou les éventuelles exceptions attendues et les résultats d'erreur.

Ecriture d’un test clean – F I R S T

- F.I.R.S.T est un acronyme contenant 5 caractéristiques importantes d'un test propre.
- **Fast**
- **Independent**
- **Repeatable**
- **Self-validating**
- **Thorough**

Ecriture d’un test clean

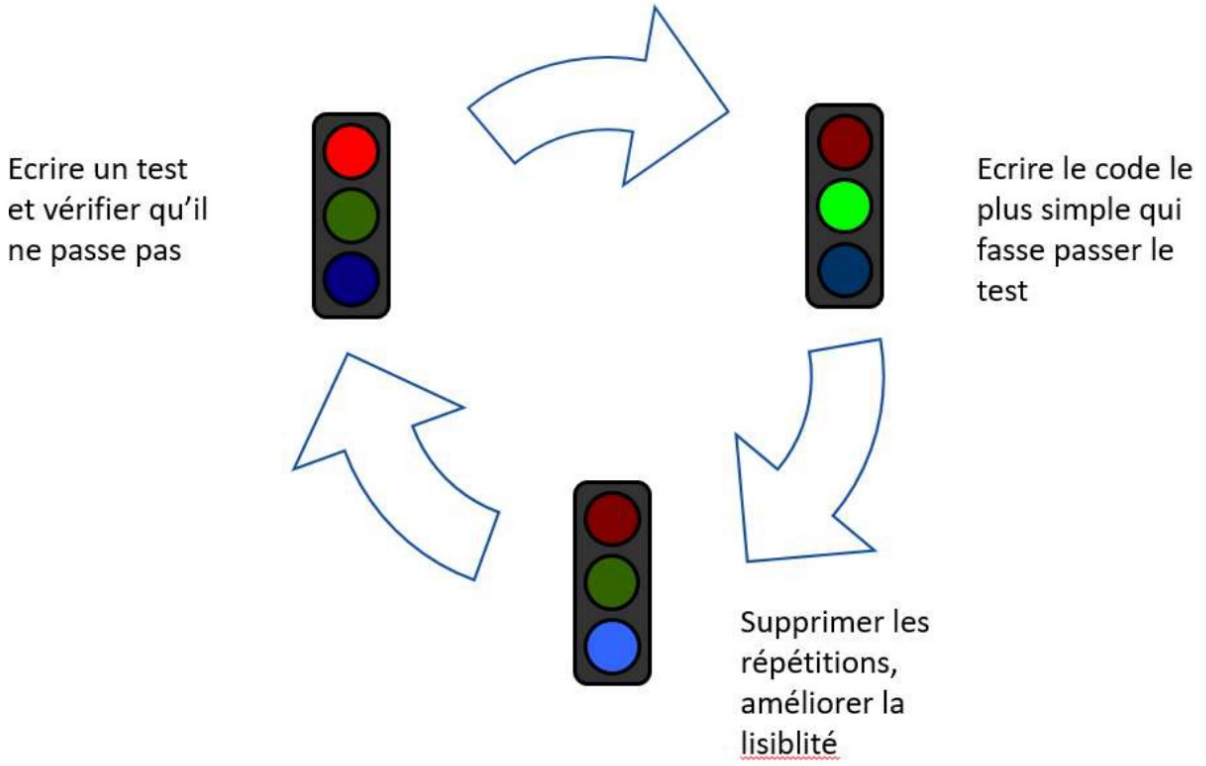
- Un test doit vérifier un seul comportement. Un même comportement peut contenir une ou plusieurs lignes d'assertions dans le code. Un test doit être couplé à un comportement fonctionnel et non à une action technique ou à une modification du code.
- Utiliser des données de test significatives
  - Les tests sont des exemples d'utilisations de code. Ils doivent utiliser des données de test significatives relatives au domaine de l'entreprise, résultant en des exemples lisibles, utilisables et réels. Par conséquent, révéler la connaissance du domaine en utilisant des données de test significatives est essentiel pour produire des tests propres.
- Masquer les données non pertinentes pour le test
  - Ne polluez pas vos tests avec des données de test non pertinentes. De telles informations ne font qu'augmenter la charge mentale cognitive, ce qui entraîne des tests gonflés. Au lieu de cela, masquez les données non pertinentes en utilisant des générateurs de données de test.

TDD

TDD

- C’est une technique de développement
- Les tests aident à spécifier du code, et non pas à le valider
- 3 étapes pour développer une fonctionnalité ou corriger un bug
  - Je pose un test unitaire et je le fais passer au rouge
  - J’écris le code nécessaire pour faire passer mon test au vert
  - Je nettoie mon code et le refactorise
- La couverture de code devient alors un bénéfice, et non plus un objectif
- Le vrai indicateur à regarder est : la NON-couverture de code

TDD


Ecrire un test
et vérifier qu'il
ne passe pas
Ecrire le code le
plus simple qui
fasse passer le
test
Supprimer les
répétitions,
améliorer la
lisiblité

TDD

- TDD renverse le modèle classique
  - Besoins -> Spécifications -> Codage/Tests Unitaires -> Tests d'intégration -> Maintenance
  - Scénarios Utilisateurs -> Test/Code/Refactor -> Tests d'Intégration
- TDD remplace une approche « industrielle »...
  - Concevoir, puis produire, puis valider, puis analyser, puis corriger
  - Rationnaliser la production de code : le mieux est l'ennemi du bien
- ...par une approche « artisanale »
  - Le code est un matériau de production, on cherche l'excellence dans le geste
  - Tester et maintenir le code au plus près de son écriture
  - Principes KISS et SOLID (Clean code)

TDD – quelques principes de Clean code

- KISS : Keep It Simple Stupid
- SOLID
  - Single Responsibility Principe : une classe doit avoir une seule responsabilité
  - Open/Closed : une classe doit être ouvert à l'extension mais fermée à la modification
  - Liskov Substitution : utilisez le type le plus « haut » possible (classes abstraites, interfaces...)
  - Interface segregation : préférez utiliser plusieurs interfaces spécifiques pour chaque client plutôt
    qu'une interface générale
  - Dependency Inversion : injection de dépendances. Il faut dépendre des abstractions et non pas des
    implémentations
- DRY : Don't Repeat Yourself

TDD - Récap

- Commencez toujours par un test automatisé qui échoue
- N’ajoutez jamais de tests sur la barre rouge
- Eliminez toute duplication