Génie logiciel

Test unitaire

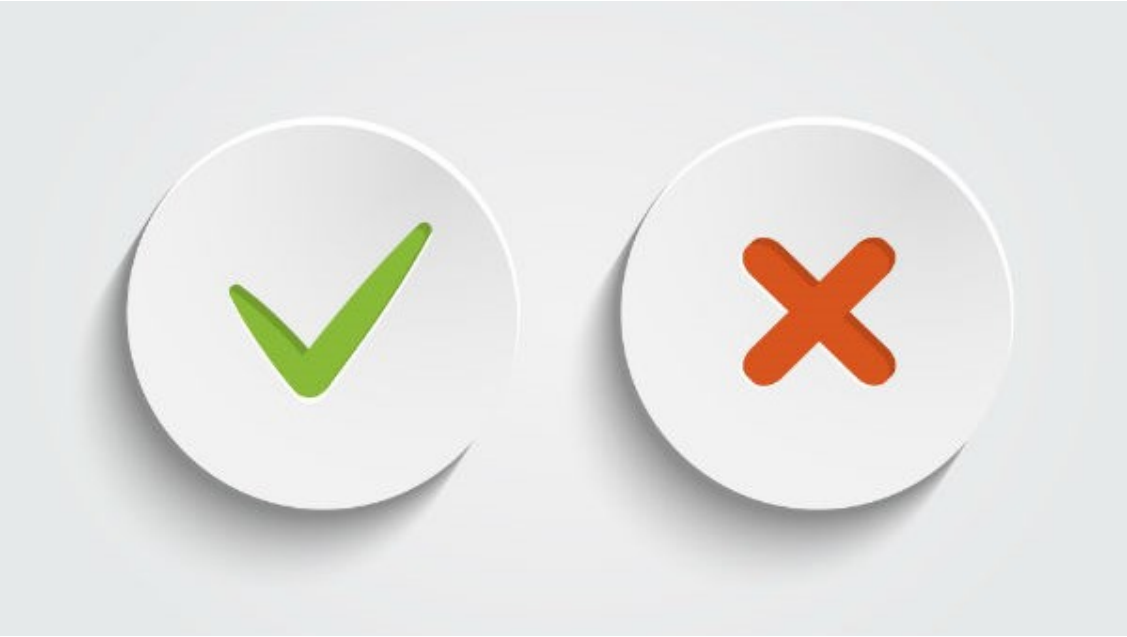
Louis-Edouard LAFONTANT

geodes

Université de Montréal

Test unitaire

Vérifier si une **unité individuelle (fonction, classe, module)** d'un programme est « apte à l'emploi »



Test unitaire

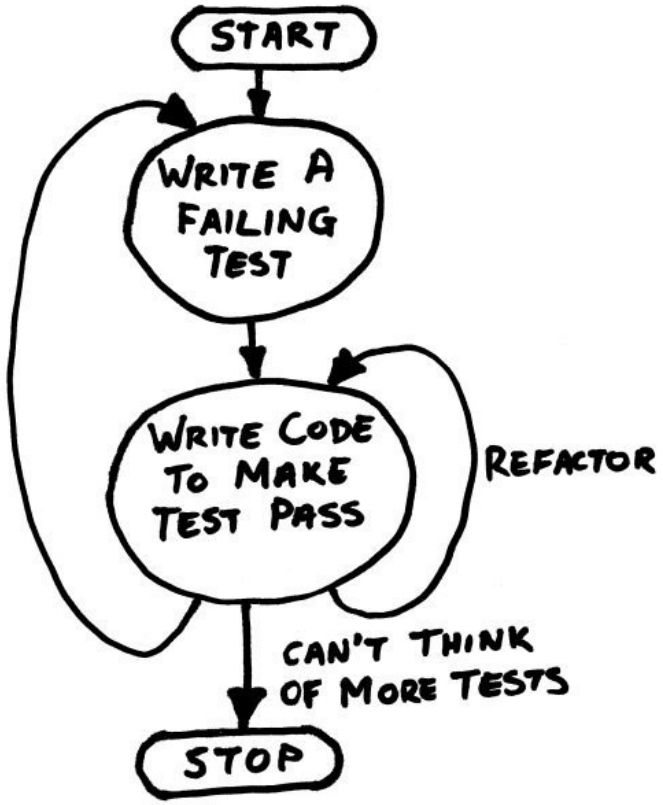
But : Tenter de démontrer que l'unité contredit sa spécification (résultat vs spécification)

- Les tests unitaires sont **structurels.**
  - Il faut connaitre la **structure interne** du programme
- Cependant, on ne teste pas le « contenu » de la méthode
  - La méthode est une **boîte noire**

Avantages

✓ Permet de tester plusieurs unités en **parallèle**

✓ Permet de tester une unité lorsque le système est encore incomplet : les tests unitaires sont **incrémentaux**



Processus

- Développement dirigé par les tests
- Tests fonctionnels initialement générés avant de commencer l'implémentation
  - Techniques de génération de tests automatique
- Les tests structurels sont ajoutés au fur et à mesure que l'implémentation progresse

Quand effectuer les tests unitaires?

- **Avant l'implémentation**
  - Force de détailler les exigences de manière implémentable
- **Pendant l'implémentation**
  - Prévient de coder en trop: quand les tests passent, la fonctionnalité est complétée
- **Pendant la réingénierie *(refactoring)* du code**
  - Assure que la nouvelle version se comporte comme l'ancienne
    - Test de régression
- **Quand on programme en équipe**
  - Augmente la confiance que le code soumis ne brisera pas celui des autres

Types de cas tests unitaires

Pour chaque méthode, on test:

Son succès
test pour un succès

Son échec
test pour un échec

Son invariance
test sanitaire

Tester pour un succès

- La sortie est correcte pour une entrée correcte
- Exemple de conversion :

  `convertFromRoman("VII") = 7`
  - Retourne `true`, une bonne valeur, sans erreur
- Exemple du triangle :

  `isEquilateral(new Triangle(4,4,4))`
  - $A = B = C$ doit annoncer un triangle équilatéral

Tester pour un échec

- Échouer, tel qu'attendu, pour une mauvaise entrée
- Exemple de conversion :
  `convertFromRoman("IIIII")` `= NaN`
  - Retourne `false`, lance une exception, retourne un message d'erreur
- Exemple du triangle :
  `TriangleFactory.createTriangle(2,3,7)`
  - $A + B > C$ doit échouer

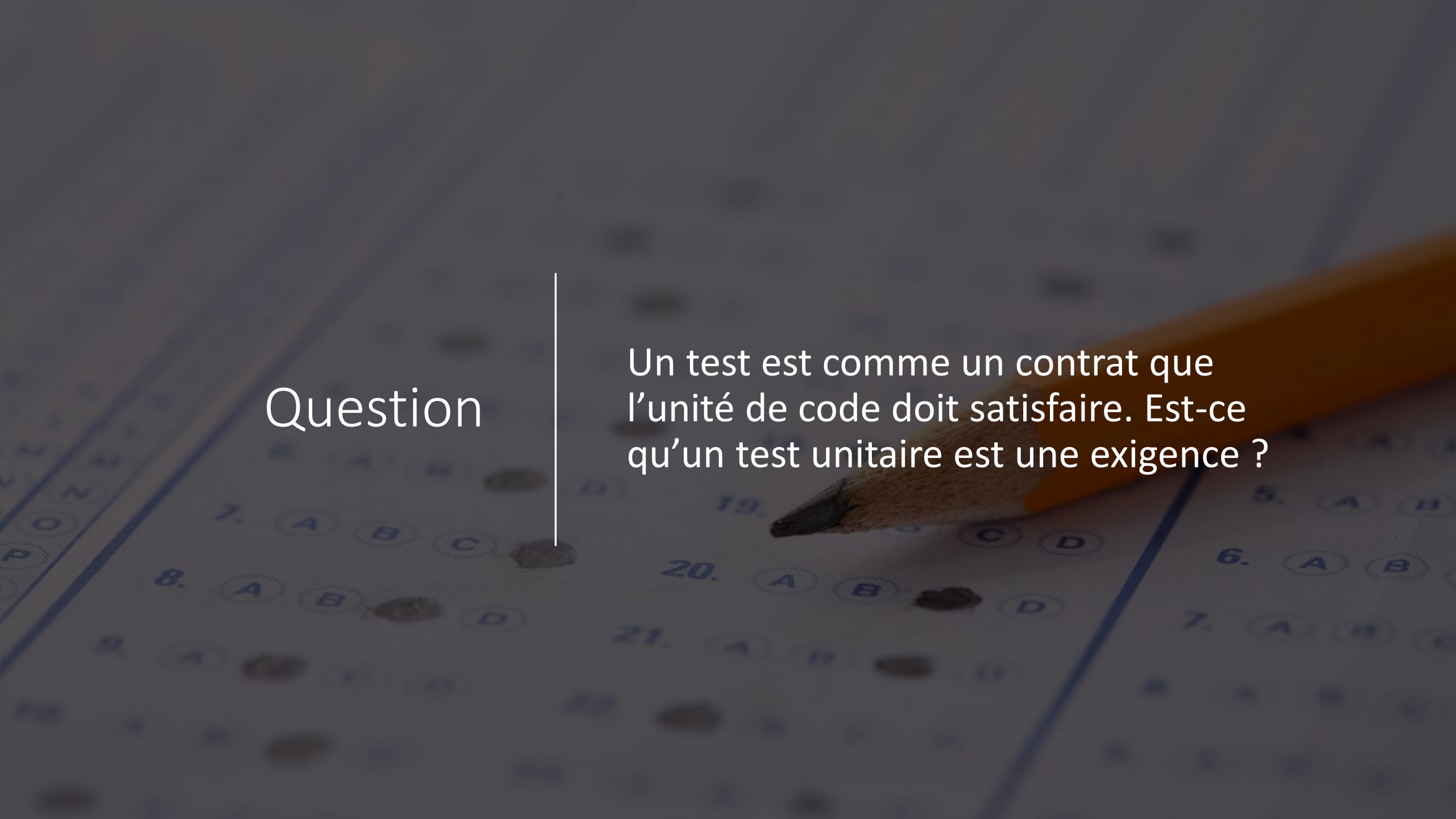
Test sanitaire

- Vérifier l'identité et l'invariance par composition
  - Exécuter une méthode suivit de son inverse
  - Exécuter une méthode à répétition
- Exemple de conversion :

  `convertToRoman(convertFromRoman("MMXVII")) = "MMXVII"`
- Exemple du triangle : non applicable

Question

Un test est comme un contrat que l’unité de code doit satisfaire. Est-ce qu’un test unitaire est une exigence ?



Réponse

TL;DR: Non

- *Exigence : expression d'un besoin documenté sur ce que le système doit faire*
- Le test est une technique de vérification du code, faite à posteriori, contrairement aux exigences
- Les tests unitaires s'assurent que les fonctionnalités exigées fonctionnent

Terminologie

- **Fixture de test :** Collection de cas de tests qui **testent une seule classe** du système
  Peut créer des objets qui sont recréés pour chaque test
- **Cas de test:** Plus petite unité de test qui s'assure d'une **réponse spécifique à un ensemble d'entrées donné**
- **Oracle**: couple de l'entrée contrôlée et de la sortie attendue
  Ex: pour $A = 1$ et $B = 2$, le résultat attendu est 3
- **Suite de test:** Collection de cas de test
- **Exécuteur de test:** Orchestre l'exécution et fournit le résultat de l'exécution de tous les cas de test

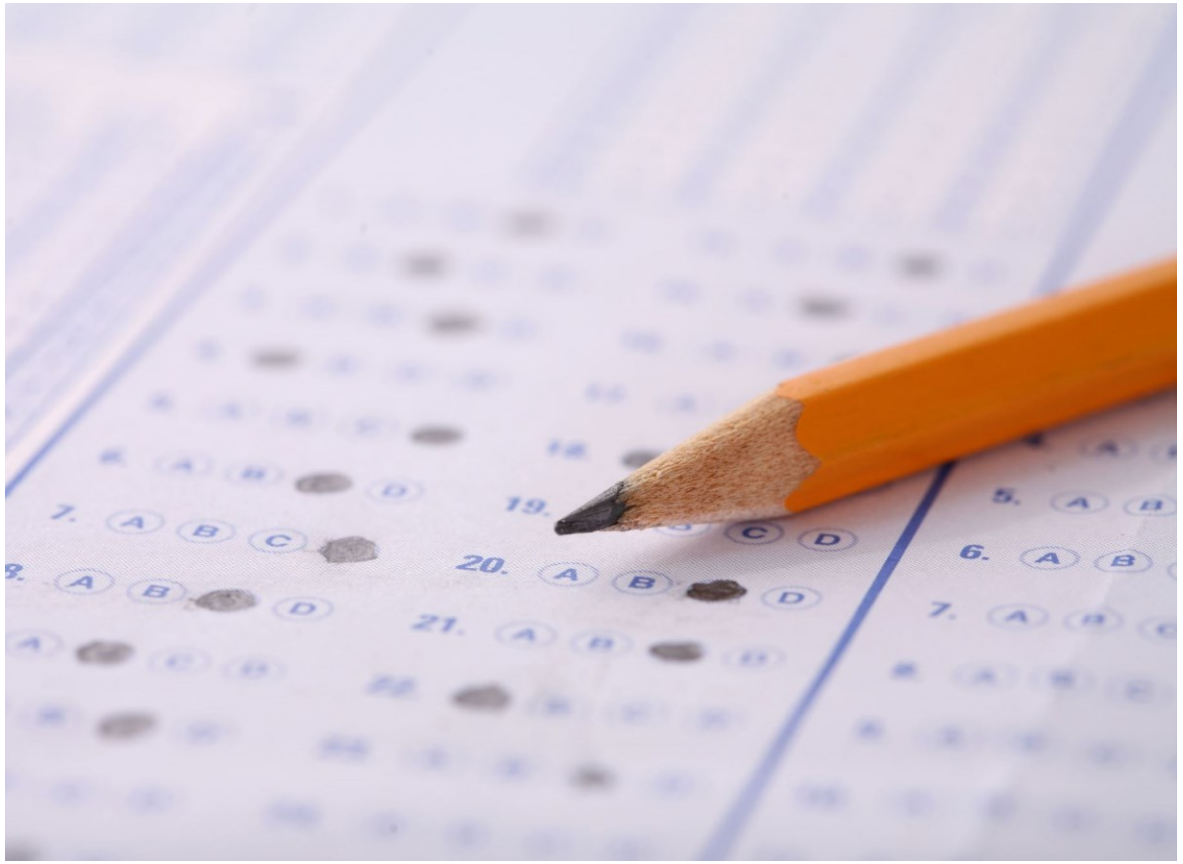
Couverture des tests pour une unité

- **Valeurs “normales”**
  Valeurs aléatoires raisonnables couvrant toutes les partitions d’entrée
- **Les cas limites**
  `0`, `Integer.MAX_VALUE`, tableau vide, string vide
- **Valeurs inattendues**
  `null`, caractères invalides dans un string, index négatif
- **Différentes catégories** d’entrées
  Entier positif, négatif, zéro
- **Différents comportements** possibles
  Chaque message d’erreur, toutes les options d’un menu

Question

Effectuer les tests unitaires de la classe suivante

```java
public class Calculator {
  public int add(int a, int b) {
    return a + b;
  }
  public int subtract(int a, int b) {
    return a - b;
  }
  public int multiply(int a, int b) {
    return a * b;
  }
  public int divide(int a, int b) {
    return a / b;
  }
```

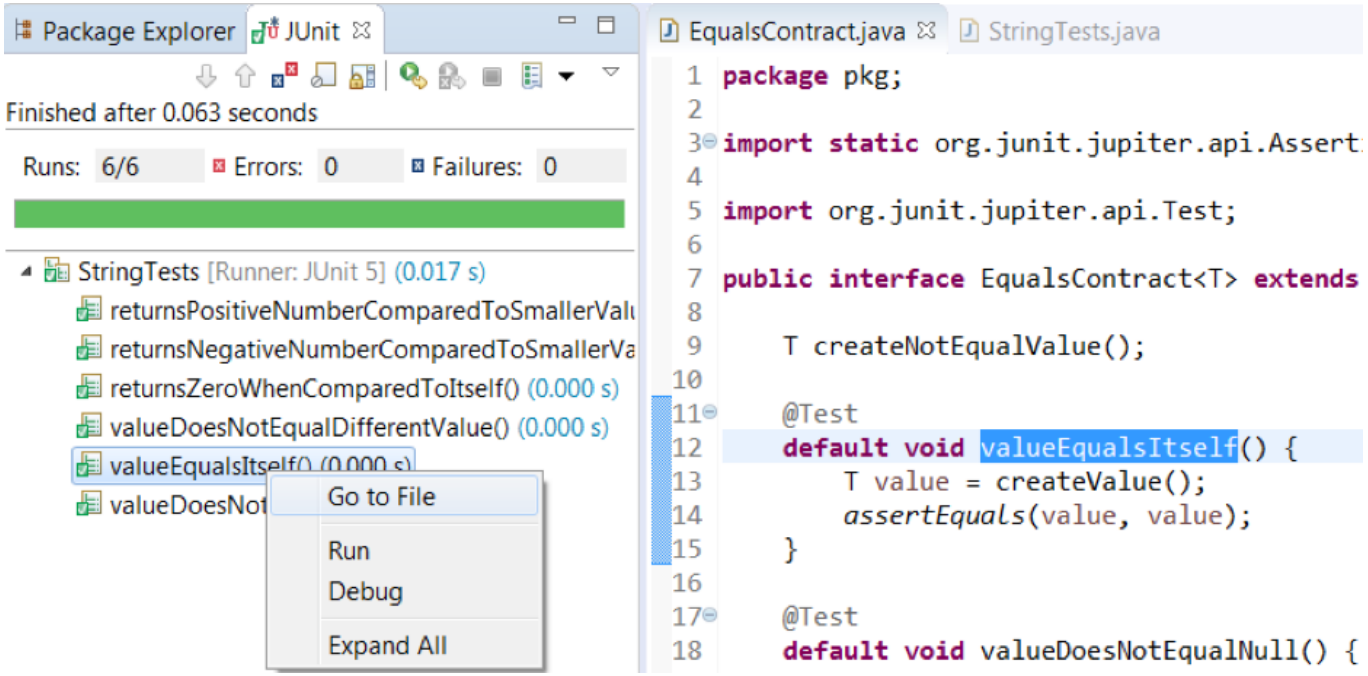


Réponse

- **`add`** [Succès]
  $a \neq b$; $a = b$; $a < 0$; $a, b < 0$; $a = 0$; $a = b = 0$
- **`subtract`** [Succès]
  même chose ; $a \times b < 0$; $a > b$; $a < b$
- **`multiply`** [Succès]
  comme add ; $a \times b < 0$
- **`divide`**
  comme multiply [Succès] $b = 0$ [Échec]
- **Combinaisons** [Succès/Échec]
  $a + b - b$; $a - b + b$; $a \times b / b$; $a / b \times b$ [Sanitaire]

Outils de tests unitaires

- SUnit (pour SmallTalk) par Kent Beck en 1989
- Beck l'a évolué pour créer **JUnit (pour Java)**
- Prolifération de *xUnit*



Tester en isolation

Comment tester une unité qui dépend d'autres unités?

Méthode qui a besoin d'objets qui n'ont pas encore été créés
Ex: Fonction qui envoie un email requiert une authentification

Il faut créer ces objets pour le cas d'utilisation qui **simule la présence de l'objet réel**

- **Object passif** (dummy): pour remplir les paramètres
- **Faux objet**: prendre des raccourcis (BD en mémoire)
- **Objet proxy** (stub) **mock**: pré-programmé pour les besoins du cas de test uniquement. Surtout utile quand les tests précèdent le développement

Préparation des objets dépendants

- **`setUp`** est appelé avant tous les tests
  - Prépare un environnement commun à tous les tests
- **`teardown`** est appelé après tous les tests.
  - Nettoie/Détruit tous les éléments créés pour les tests et par les tests.

```
public class TestGame extends TestCase {
  private Game game;
  private Ship fighter;

  public void setUp(  ) throws BadGameException {
    this.game = new Game(  );
    this.fighter = this.game.createFighter("001");
  }

  public void tearDown(  ) {
    this.game.shutdown(  );
  }

  public void testCreateFighter(  ) {
    assertEquals("Fighter did not have the correct identifier",
        "001", this.fighter.getId(  ));
  }

  public void testSameFighters(  ) {
    Ship fighter2 = this.game.createFighter("001");
    assertSame("createFighter with same id should return same object",
        this.fighter, fighter2);
  }

  public void testGameInitialState(  ) {
    assertTrue("A new game should not be started yet",
        !this.game.isPlaying(  ));
  }
}
```

Organisation du code

- Garder les classes de tests dans le même projet que le code
  - Les test sont compilés avec le reste du code
  - Aide à actualiser les tests
- Grouper les tests dans le même paquet, mais un dossier différent des fichiers source
  - Ex: **`src/`** , **`tests/`** , `docs/` , `readme` , `license`
  - Permet aux tests d'accéder aux entités visibles seulement dans leur paquetage
- Utiliser une nomenclature descriptive et standardisée :
  - Ex: `ParserTest` teste la classe `Parser`

Bonne pratique

Questions typiques

- **Comment tester les méthode privées ?**
  En général, elles ne devraient pas être testées directement, mais c'est tout de même possible de les tester à l'aide de mécanismes de réflexion
- **Doit-on tester tous les « getters » et « setters » ?**
  En général c'est inutile, mais on doit tester les getters et setters dont le comportement n'est pas trivial et/ou dans les cas limites
- **Comment tester des classes abstraites?**
  À l'aide de classes abstraites contenant des tests qui sont implémentés par les fixtures concrètes

Tout cas de test doit…

- ✓S’exécuter sans intervention humaine : doit être **automatisé**
- ✓Déterminer tout seul si l’unité qu’il test est un **succès ou un échec**, sans qu’un humain n’ait à interpréter les résultats
- ✓Tester exactement **une seule fonctionnalité** pertinente
- ✓S’exécuter en **isolation**, indépendamment des autres cas de tests, même s’ils testent la même unité

But: déterminer la cause de l’erreur de façon unique !