Tests unitaires et Mocks

Licence Professionnelle IDSE
2015-2016

IUT de Nice

Simon Urli
urli@i3s.unice.fr

Inspiré du cours de Philippe Collet.

Validation & Vérification

- Conformité avec la définition : **Validation**
  - Faisons nous le bon produit ?
  - Défauts par rapport aux besoins que le produit doit satisfaire
- Correction d'une phase ou de l'ensemble : **Vérification**
  - Faisons nous le produit correctement ?
  - Tests
  - Erreurs par rapport aux définitions précises établies lors des phases antérieures

Validation & Vérification

- Spécifications fonctionnelles définies lors de l'analyse des besoins : ce sont les intentions
- Vérifier le produit consiste à vérifier la conformité vis-à-vis de ces spécifications fonctionnelles
- Valider le produit consiste à vérifier que le produit correspond aux besoins clients

Test : définition

- Une expérience **d'exécution** pour mettre en évidence un défaut ou une erreur
  - **Diagnostic** : quel est le problème
  - Besoin d'un **oracle** qui indique si le résultat est conforme aux intentions
  - **Localisation** de l'erreur
- Les tests mettent en évidence des erreurs
- **On ne doit pas vouloir démontrer qu'un programme marche à l'aide de tests !**
- **Ne pas trouver d'erreur ne signifie pas qu'il n'y en a pas !**

Constituant d'un test

- Nom, objectif, commentaires, auteur
- Données : jeu de test
- Du code : cas de test
- Des oracles : vérifications de propriétés
- Des traces parfois
- Un compte rendu

- Coût : autant que le programme !

Test vs Essai vs Débogage

- Le coût du test est amorti car il est **reproductible**
- Un essai est «one-shot»
- Le débogage est une enquête : il cherche à résoudre un problème (peut être décelé grâce à un test)

Stratégies de test OO

- Les classes sont la plus petite unité testable
- L'héritage introduit de nouveaux contextes pour les méthodes
  - Les méthodes doivent être testées pour chaque classe !
- Les objets ont des états : les procédures de test doivent en tenir compte

Que faut-il tester ?

- Principe **Right-BICEP** :
  - **Right** : est-ce que les résultats sont corrects ?
  - **B (Boundary)** : est-ce que les conditions aux limites sont correctes ?
  - **I (Inverse)** : est-ce que l'on peut vérifier la relation inverse ?
  - **C (Cross-check)** : est-ce que l'on peut vérifier le résultat autrement ?
  - **E (Error condition)** : est-ce que l'on peut forcer l'occurence d'erreurs ?
  - **P (Performance)** : est-ce que les performances sont prévisibles ?

Right-BICEP

- B : Boundary conditions :
  - Identifier les conditions aux limites de la spécification
  - Que se passe-t-il lorsque les données sont :
    - anachroniques ex. !*W@V»
    - non correctement formatées ex : fred@foobar.
    - vides ou nulles ex : 0, 0.0, «», null
    - extraordinaires ex : 1000 pour un âge
    - dupliquées ex : doublon dans un set
    - non conformes ex : listes ordonnées qui ne le sont pas
    - désordonnées ex : commande imprimer avant d'être connecté

Right-BICEP

- Principe «**CORRECT**» :
  - **Conformance** : test avec données en dehors du format attendu
  - **Ordering** : test avec données sans l'ordre attendu
  - **Range** : test avec données hors de l'intervalle
  - **Reference** : test des dépendances avec le reste de l'application (précondition)
  - **Existence** : test sans valeur attendu (pointeur nul)
  - **Cardinality** : test avec des valeurs remarquables (bornes, nombre max)
  - **Time** : test avec des cas où l'ordre à une importance

Right-BICEP

- Inverse - Cross check
  - Identifier : les relations inverses et les algorithmes équivalents
    - Exemple : test de la racine carrée en utilisant la fonction de mise au carré
- Error condition - Performance
  - Identifier ce qui se passe quand :
    - Le disque, la mémoire etc sont pleins

Test unitaire en Java

- Référence : librairie JUnit
- Inclus par défaut dans eclipse

Utilisation de JUnit

- Importer la librairie dans une classe
- Créer une classe
- Utiliser l'annotation @Test sur les méthodes de test
- Exploiter les oracles avec les méthodes assert* :
  - assertEquals
  - assertTrue
  - assertFalse
  - fail
  - ...

Utilisation de JUnit

- Possibilité d'initialiser un contexte (fixture) commun à toutes les méthodes de tests ou au contraire de les nettoyer :
  - Annotation **@Before** et **@After**
  - Chaque test s'exécute dans son propre contexte après avoir appelé la méthode annotée par le **Before** et exécute ensuite la méthode annotée par le **After**

Test unitaire : quelques conseils

- Structure du code pour accéder aux méthodes «protected» dans les tests
- Les noms de tests doivent être sans ambiguité
- Renseigner les messages de fail dans les assert
- Utilisez le assertEquals quand c'est possible : donne beaucoup d'informations en cas de fail
- Coder/tester, coder/tester, ...
- En cas de découverte d'un bug, faire un cas de test pour mettre en avant le bug !
- Privilégiez les tests des parties critiques du code

Exemple

- (Exemple live : cf code joint)

Tests et Mocks

- Mock = objet factice
- les mocks (ou Mock object) sont des objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée
- On teste ainsi le comportement d'autres objets, réels, mais liés à un objet inaccessible ou non implémenté
- Cet objet est remplacé par un mock

Principe des mocks

- Un mock a la même interface que l'objet qu'il simule
- L'objet client ignore s'il interagit avec un objet réel ou un objet simulé
- La plupart des frameworks de mock permettent :
  - De spécifier quelles méthodes vont être appelées, avec quels paramètres et dans quel ordre
  - De spécifier les valeurs retournées par le mock

Librairie de Mock en Java : Mockito

- Léger
  - Focalisation sur le comportement recherché et la vérification après exécution
- Simple
  - Un seul type de mock
  - Une seule manière de les créer

Principe de fonctionnement

- Création des mocks
  - méthode mock ou annotation @mock
- Description du comportement
  - méthode when
- Mémorisation à l'exécution des interactions
  - Utilisation du mock dans un code qui teste un comportement spécifique
- Interrogation à la fin du test des mocks pour savoir comment ils ont été utilisés
  - Méthode verify

Création

- A partir d'une interface ou d'une classe (utilisation de .class)
  - UneInterface mockSansNom = mock(UneInterface.class)
  - UneInterface mockAvecNom = mock(UneInterface.class, «ceMock»);
    - @Mock UneInterface ceMock;
- Comportements par défaut :
  - assertEquals(«ceMock», monMock.toString());
  - assertEquals("type numerique : 0 ", 0, monMock.retourneUnEntier());
  - assertEquals("type booléen : false", false, monMock.retourneUnBooleen());

Stubbing

- Remplacement du comportement par défaut des méthodes
- Deux possibilités :
  - Méthode qui a un type de retour :
    - when + thenReturn
    - when + thenThrow
  - Méthode de type void :
    - doThrow + when

Stubbing : retour d'une valeur unique

```
// stubbing
when(monMock.retourneUnEntier()).thenReturn(3);

// description avec JUnit
assertEquals("une premiere fois 3", 3, monMock.retourneUnEntier());
assertEquals("une deuxieme fois 3", 3, monMock.retourneUnEntier());
```

Stubbing : valeurs de retour consécutives

```
// stubbing
when(monMock.retourneUnEntier()).thenReturn(3, 4, 5);

// description avec JUnit
assertEquals("une premiere fois : 3", 3, monMock.retourneUnEntier());
assertEquals("une deuxieme fois : 4", 4, monMock.retourneUnEntier());
assertEquals("une troisieme fois : 5", 5, monMock.retourneUnEntier());

when(monMock.retourneUnEntier()).thenReturn(3, 4);
// raccourci pour     .thenReturn(3).thenReturn(4);
```

Stubbing : levée d'exceptions

```
public int retourneUnEntierOuLeveUneExc() throws BidonException;
// stubbing
when(monMock.retourneUnEntierOuLeveUneExc()).thenReturn(3)
                    .thenThrow(new BidonException());

// description avec JUnit
assertEquals("1er appel : retour 3",
                3, monMock.retourneUnEntierOuLeveUneExc());

try {
    monMock.retourneUnEntierOuLeveUneExc(); fail();
} catch (BidonException e) {
    assertTrue("2nd appel : exception", true);
}
```

La méthode verify

- Permet de vérifier
  - Quelles méthodes ont été appelées sur un mock
    - Combien de fois, avec quels paramètres, dans quel ordre
  - Une exception est levée si la vérification échoue : le test échouera aussi

Verify

- Méthode appelée une seule fois :
  - verify(monMock).retourneUnBooleen();
  - verify(monMock, times(1)).retourneUnBooleen();
- Méthode appelée au moins/au plus une fois :
  - verify(monMock, atLeastOnce()).retourneUnBooleen();
  - verify(monMock, atMost(1)).retourneUnBooleen();
- Méthode jamais appelée :
  - verify(monMock, never()).retourneUnBooleen();
- Avec des paramètres spécifiques :
  - verify(monMock).retourneUnEntierBis(4,2);