Anti-Pattern de test

Anti-pattern : Assert manquant

- Pas d'assert :
  - un test qui ne teste rien : très souvent inutile
  - "test" affichant le résultat
    - illisible par les autres
    - perte d'automaticité
- Utilité possible : non pas de résultat attendu, mais vérification qu'il n'y a pas d'exception
  - non régression par rapport à un bug précédent

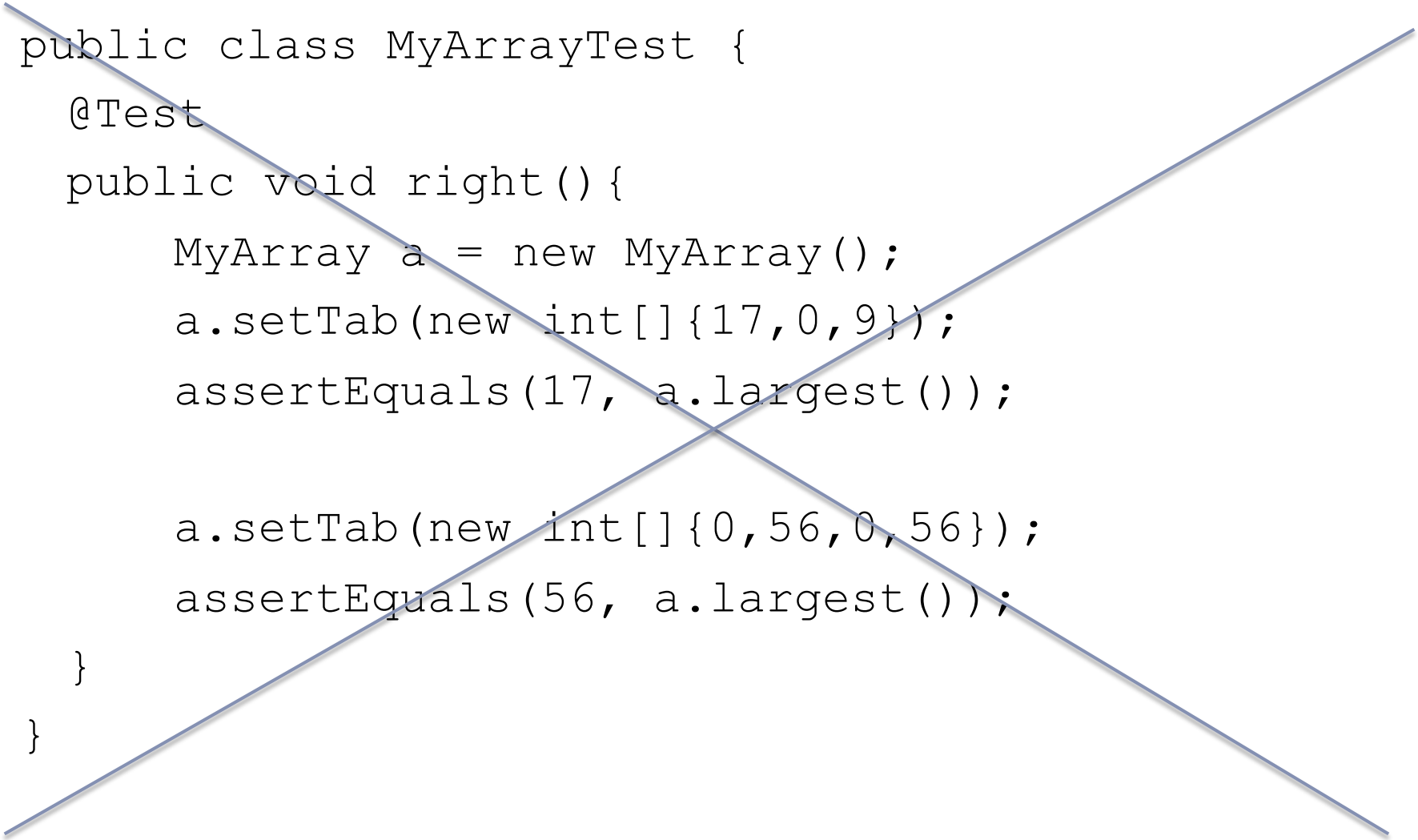
Anti-pattern :
Cas multiples dans une méthode

- Jamais une bonne idée, les tests doivent rester courts
  - + facile à comprendre
  - + facile d'identifier les causes des erreurs détectées par les tests
  - - de risque d'erreur dans les tests
- Si besoin de partage de code entre parties à tester
  - classes helper
    - ex : NetBean ne lance par défaut que les classe dont le nom finit par Test
  - méthodes non annotés @Test

Anti-pattern

```
public class MyArrayTest {
  @Test
  public void right(){
      MyArray a = new MyArray();
      a.setTab(new int[]{17,0,9});
      assertEquals(17, a.largest());

      a.setTab(new int[]{0,56,0,56});
      assertEquals(56, a.largest());
  }
}
```



Cas multiple

```
public class MyArrayTest {
  @Test
  public void simple(){
      MyArray a = new MyArray();
      a.setTab(new int[]{17,0,9});
      assertEquals(17, a.largest());

  @Test
  public void double(){
      MyArray a = new MyArray();
      a.setTab(new int[]{0,56,0,56});
      assertEquals(56, a.largest());
  }
}
```

Anti-pattern : Utilisation du mauvais assert

- Appel à assertTrue avec un true codé en dur : non
  - assertTrue("Object must be the same", expected == actual);
  - assertTrue("Objects must be equals", expected.equals(actual));
  - assertTrue("Object must be null", actual == null);
  - assertTrue("Object must be non null", actual != null);
- Utiliser l'assert correspondant
  - assertSame("Object must be the same", expected, actual);
  - assertEquals("Objects must be equals", expected, actual);
  - assertNull("Object must be null", actual);
  - assertNotNull("Object must be non null", actual);

Anti-pattern : Test trop compliqué

- Il ne faut pas avoir à tester le test pour se convaincre qu'il est bon
- Pas d'algorithmique compliqué
- Pas de ligne à rallonge
- Pas d'hésitation au copier-coller

Anti-pattern : Dépendances externes

- Il ne faut qu'il y ait des éléments trop compliqués à mettre en place
  - sinon les tests ne seront pas exécutés
  - toujours garder les tests automatiques
- Eviter les dépendances vers l'environnement
  - SGBD, fichiers, librairies, *socket*, …
- Faire en sorte d'inclure toutes les données nécessaires avec les tests
  - s'il faut vraiment un SGBD, envisager l'emploi de SGBD en mémoire(par ex. HSQLDB)

Anti-pattern : Récupération d'exceptions

- On ne récupère pas les exceptions !

```
@Test public void testCalculation() {
     try {
          deepThought.calculate();
          assertEquals(42, deepThought.getResult());
     } catch(CalculationException ex) {
          ex.printStackTrace();
     }
}
```

- Raté ! JUnit ne détectera pas que le test a échoué

Exceptions

- Si l'exception est attendue

```
@Test(expected=CalculationException.class)
public void testCalculation() {
     deepThought.calculate();
     fail("Pas d'exception");
}
```

- Si l'exception est inattendue

```
@Test
public void testCalculation() throws CalculationException {
     deepThought.calculate();
     assertEquals(42, deepThought.getResult());
}
```

Test unitaire et Injection de dépendance

Dépendance

- Dépendance
  - Une classe A dépends d'une classe B si B apparaît dans A.java (attribut, paramètre ou retour)
- **Dépendance avec un couplage fort**
  - Si A fait un new B()
  - Test unitaire en isolation impossible
    - Grave si B est non fiable
- Classes fiables
  - Classes SDK
  - Beans purs (too small to fail)

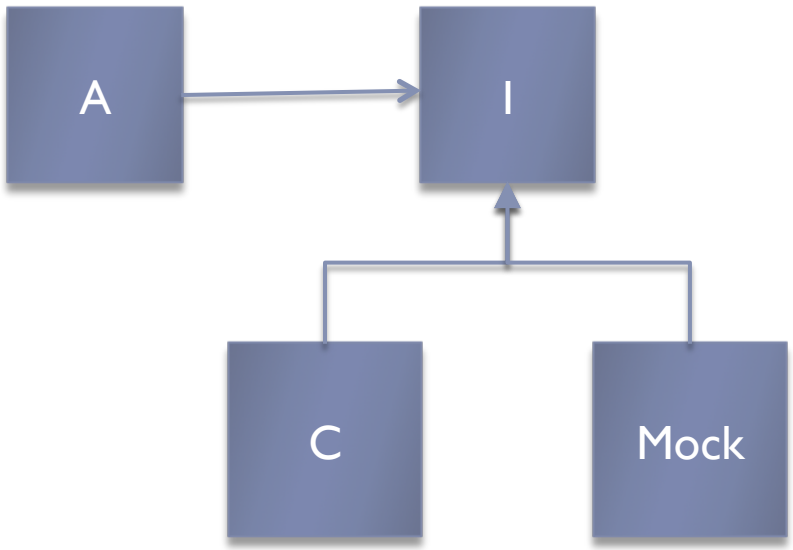
Solution

1. Rendre les dépendances explicites et paramétrable
   - Par construction ou ou par setter
2. EVITER LES NEW (utiliser une factory)
3. Définir des interfaces
   - une interface est fiable
   - on ne peut avoir de couplage fort avec une interface
4. Remplacer les dépendances par des imitations

Tout ca !

Principe du Simulacre (**MOCK**)

- Appelé aussi **imitation**, ou bouchon
- A dépends d'une interface I
- Remplacement d'un classe C implémentant une interface I par une autre classe



Utilisation d'un mock

- Tests unitaires de A
  - A la base on substitue tout classe non-fiable dont on dépend
- Tests d'intégration : on substitue tout objet qui
  - a un comportement non déterministe (ex. cours d'action)
  - est difficile à configurer
  - a un comportement difficile à reproduire (ex. erreur réseau)
  - est lent/couteux
  - a une interface utilisateur
  - dépend de l'heure/date courante
  - produit un effet de bord non vérifiable automatiquement
  - produit un effet de bord non désirable lors d'un test
  - n'existe pas encore lorsque le test est exécuté

Exemple : classe à tester

```
public class Checker {
  private Environmental env;
      public Checker(Environmental anEnv) {
      env = anEnv;
  }
  /** Après 17h, lancer une alarme en jouant un
  fichier de sons */
  public void reminder() {
      Calendar cal = Calendar.getInstance();
      cal.setTimeInMillis(env.getTime());
      int hour = cal.get(Calendar.HOUR_OF_DAY);
      if (hour >= 17)
            env.playWavFile("quit_whistle.wav");
  }
}
```

Exemple : mock "à la main"

```
public interface Environmental {
  public long getTime();
  public void playWaveFile(String filename);
}

public class MockEnvironment implements Environmental {
  private long time;
  public long getTime() { return time; }
  public void setTime(long time) { this.time = time; }

  private boolean played = false;
  public void playWaveFile(String filename
  { played=true; }
  public boolean wasPlayed() { return played; }
}
```

Exemple : test

```
public class TestChecker {
  @Test
  public void testQuittingTime() {
      long t1 = ...;            // 16h57
      MockEnvironment env = new MockEnvironment();
      env.setTime(t1);
      Checker checker = new Checker(env);
      checker.reminder();
      assertFalse(env.wasPlayed());
      t1 += (5 * 60 * 1000);  // Advance the time
      env.setTime(t1);
      checker.reminder();
      assertTrue(env.wasPlayed());
  }
}
```

Mock écrit à la main

- Conclusion sur l'exemple précédent
  - mock class écrite à la main
  - utilisation simple du mock object
- Besoins
  - interfaces avec + de méthodes (ex. HttpServletRequest >50 méthodes)
  - vérifier l'enchaînement des méthodes
- Générateur automatique de mock
  - plusieurs frameworks existants
  - EasyMock, jMock, Mocquer

Framework d'objet simulé (en Java)

- **EasyMock**
- **JMock**
- JMockit
- **Mockito**
- Mockachino
- **Mocquer**
- Moxie
- rMock
- MockLib
- SevenMock
- PowerMock

EasyMock

- http://www.easymock.org
  - package org.easymock
- Principe du scénario attendu / scénario de test
- Dans l'exemple précédent qu'était-il attendu de Checker.reminder() ?
  - appel de env.getTime()
  - appel de env.playWaveFile() si après 17h
- http://www.easymock.org/EasyMock3_0_Documentation_fr.html

Scénarios

```
public class Simple {
  @Test
  public void ring() {
      Environmental mock =
  EasyMock.createStrictMock(Environmental.class);
      Checker classTested = new Checker(mock);
      long t1 = ...;       // Avant 17h
      long t2 = ...;         // Apres 17h
      EasyMock.expect(mock.getTime()).andReturn(t1);
      EasyMock.expect(mock.getTime()).andReturn(t2);
      mock.playWaveFile("quit_whistle.wav");
      EasyMock.replay(mock);
      classTested.reminder();
      classTested.reminder();
      EasyMock.verify(mock);
  }
}
```

Scénario attendu

Scénario de test

Ecriture d'un test mocké

- Principe général
  - on crée le mock : `createMock(Class<T> c)`
  - on décrit le scénario attendu : méthodes du mock
  - on ré-initialise le mock : `replay`
  - on applique le scéario de test
  - on vérifie le scénario : `verify`
- Le scénario obtenu doit être strictement identique à celui attendu
  - pas d'appel en plus ou en moins
  - sinon exception AssertionError (standard JUnit)

Syntaxe d'un scénario

- Simple appel de méthode
  - `mock.method(args);`
- Appel de méthode + comportement
  - `EasyMock.expect(mock.method(args)).andXXX(x);`
  - comportement possibles
    - `andReturn(value)` : retour d'une valeur
    - `andThrow(except)` : levée d'une exception
- Ordre du scénario
  - sans ordre `createMock()`
  - avec ordre `createStrictMock()`

Autres trucs

- Réutilisation d'un mock (à éviter)
  - `mock.reset()`, `mock.resetToStrict()`, `mock.resetToDefault()`
- Plusieurs mocks
  - `EasyMock.verify()` : appel statique => 1 seul mock

```
IMocksControl ctrl = createStrictControl();
mock1 = ctrl.createMock(class1);
mock2 = ctrl.createMock(class2);
…
ctrl.verify()
```

Appels de méthodes

- Plusieurs appels :
  - `times(int nb)`
- Entre min and max appels :
  - `times(int min, int max)`
- Au moins un appel :
  - `atLeastOnce()`
- Une quantité non définie d'appels :
  - `anyTimes()`
- Enchainement :

```
EasyMock.expect(mock.voteForRemoval("Document"))
        .andReturn((byte) 42).times(3)
        .andThrow(new RuntimeException(), 4)
        .andReturn((byte) -42);
```

Arguments d'appel

- `mock.method(args)` **ou** `expect(mock.method(args))`
  - Comparaison des arguments : `.equals()`
- `anyBoolean(), anyByte(), anyChar(), anyDouble(), anyFloat(), anyInt(), anyLong(), anyObject(), anyObject(Class clazz), anyShort()`
  - Laisse passer n'importe quelle valeur de ce type
- `eq(X value, X delta)`
  - Même valeur, plus ou moins un delta (float et double)
- `aryEq(X value)`
  - Utilise Arrays.equals()
- `isNull(), notNull()`
  - Valeur nulle, ou non nulle
- `isNull(Class clazz), notNull(Class clazz)`
  - Valeur nulle, ou non nulle d'un certain type

Arguments d'appel (2)

- `same(X value)`
  - Comparaison ==
- `isA(Class clazz)`
  - instance de clazz ou d'une classe hérite ou implémente clazz
- `lt(X value), leq(X value), geq(X value), gt(X value)`
  - <, <=, >=, > à la valeur attendue (types numériques et implémentations de Comparable)
- `startsWith(String prefix), contains(String substring), endsWith(String suffix)`
  - Chaîne commencant par/contient/se terminant par la valeur attendue
- `matches(String regex), find(String regex)`
  - Vérifie que la valeur reçue/une sous-chaîne de la valeur reçue correspond à l'expression régulière

Arguments d'appel (3)

- `and(X first, X second)`
  - Est valide si les résultats des deux comparateurs sont vérifiés
- `or(X first, X second)`
  - Est valide si l'un des résultats des deux comparateurs est vérifié
- `not(X value)`
  - Est valide si le résultat du comparateur utilisé dans value est négatif
- **exemple** : `mock.method(not(and(geq(40), lt(50))))`
  - appel de method mais avec un argument non compris entre 40 et 49
- `capture(Capture<T> capture)`
  - Laisse passer n'importe quelle valeur mais la capture dans le paramètre capture pour un usage ultérieurs (utile si T est complexe).

Mock de classe

- Permet de faire un mock partiel
- On précise les méthodes à mocker

```
ToMock mock=EasyMock.createMockBuilder(ToMock.class)
    .addMockedMethod("mockedMethod").createMock();
```

- Les autres auront le comportement de la classe
- Appel d'un constructeur particulier

```
ToMock mock=createMockBuilder(ToMock.class)
    .withConstructor(args);
```

- A réserver à des cas vraiment particulier
  - Code de la classe non refactorable

Exercice

- Un jeu de dés :
  - Les dés sont du modèle classique à tirage aléatoire entre 1 et 6.
  - La banque possède un fond, et un fond minimum à ne pas dépasser. Si elle passe en dessous de ce niveau suite à un pari réussi, le gain est quand même donné au joueur, mais la banque saute et le jeu ferme immédiatement.
- La règle du jeu est la suivante :
  - On ne peut parier que sur un jeu ouvert.
  - Le joueur qui a parié est débité du montant de son pari qui est encaissé par la banque.
  - Ensuite les 2 dés sont lancés. Si la somme des lancers vaut 7, alors le joueur gagne : la banque paye cinq fois la mise, somme créditée au joueur. Si la somme des lancers ne vaut pas 7, le joueur a perdu sa mise.

Interfaces

```
public interface De {
    // Retour aléatoire entre 1 et 6
    byte lancer();
}
public interface Joueur {
    // crédite de c grâce à un
  numéro de transaction visa
    void credit(long visaTrNb, int
  c);
    // débite de c
    void debit(int c);
}public interface Banque {
  boolean fondMinAtteint();
   // Débite de c, produit un
  numéro de transaction Visa
    long debit(int c) throws
  PlusdeFond;
    // Crédite de c
    void credit(int c);
}
```

```
public interface Jeu {
    // positionne les dés
    void setDe1(De d);
    void setDe2(De d);

    // ouvre le jeu
    void ouvre();

    // le jeu est-il ouvert ?
    boolean estOuvert();

    // positionne la banque
    void setBanque(Banque b);

    // Le joueur j parie un montant
  de m
    void jouer(Joueur j, int m)
  throws JeuNonOuvert;
}
```