IUT Lyon 1

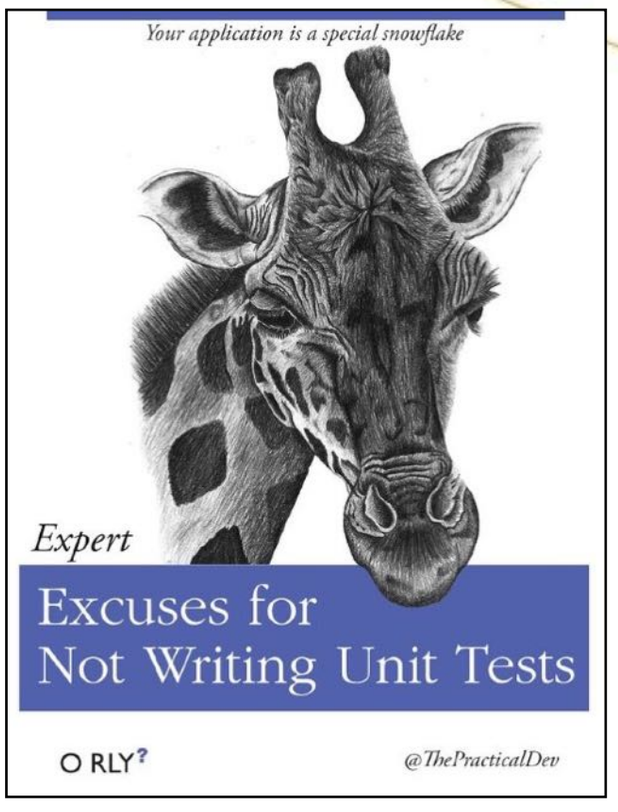
l'excellence technologique

Tests unitaires

LP IEM

❝ *No matter how slow you are writing clean code,*
you will always be slower if you make a mess. ❞

Uncle Bob Martin



Tests unitaires

- Qu’est-ce qu’un test unitaire ?
  - Les tests fonctionnels, d’intégration ou de validation testent les fonctionnalités d’un logiciel d’un point de vue externe ➪ tests “clients”
  - Tests unitaires s’appliquent au niveau d’une classe
  - Tests unitaires ➪ “tests programmeur”
  - Test des fonctionnalités élémentaires
    - Logique d’un algorithme
    - Rejet des entrées hors domaine
    - …

Tests unitaires

- Pourquoi des frameworks de tests unitaires ?
  - Généralement la première version d'un code est testée en détail (pas à pas, sortie de chaîne de caractère sur la console, ...)
  - Les problèmes arrivent lors des modifications (ajout de fonctionnalités, corrections de bugs,...)
  - Nécessité d'écrire des tests automatiques
    - ce n'est pas le programmeur qui vérifie les sorties du programme
    - les tests sont construits en utilisant les entrées, un contexte et le résultat attendu
    - le test vérifie que le résultat est conforme au résultat attendu

Tests unitaires

- Pourquoi des frameworks de tests unitaires ?
  - Au cours du développement on construit des suites de test qui sont exécutées plusieurs dizaines de fois par jour
    ➪ vérifier que chaque modification significative ne casse pas le programme
  - Automatisation des tests
    - un clic pour lancer une suite de tests
    - Les tests peuvent s'exécuter avant un commit sur le gestionnaire de configuration ou/et après
    - Les tests peuvent être exécutés par le serveur d'intégration continue (CI)

Tests unitaires

- JUnit
  - Framework de tests unitaire spécifique à Java
  - Intégré dans tous les IDE (IntelliJ, Netbeans, Eclipse, ...)
  - Trois versions coexistent
    - Version 3 : utilise l'héritage pour définir les tests
    - Version 4 : s'appuie sur les annotations (introduites avec Java 5)
    - Version 5 : s'appuie sur les expressions lambda introduites avec Java 8

JUnit 4 - exemple

```
import static org.junit.Assert.*;
import org.junit.*;

public class SlidingPuzzleTest {

    Connection c;

    @Before
    public void setUp() {
        c = Connection.newInstance();
    }

    @Test
    public void testConnectionOpening() {
        assertTrue(c.isClosed());
        c.open();
        assertFalse(c.isClosed());
    }
```

Tests unitaires

- JUnit 4
  - Chaque méthode de test qui doit être invoquée automatiquement est marquée avec l'annotation `@Test`
  - Si on veut tester qu'une méthode lance bien une exception dans certaines conditions on annote la méthode avec `@Test(expected = Exception.class)`
    - On peut préciser la classe de l'exception attendue

Tests unitaires

- JUnit 4
  - Setup
    - L'annotation `@BeforeClass` pour une initialisation commune à tous les tests unitaires de la classe de tests
    - L'annotation `@Before` permet de dire à JUnit d'exécuter une méthode avant chaque test
  - Teardown
    - Pour nettoyer après les tests (fermeture de connexions, fichiers, …) il existe les annotations `@AfterClass` et `@After`

Tests unitaires

- JUnit 4
  - Les tests s'écrivent en utilisant les méthodes statiques de la classe Assert :
    - assertEquals(int arg1 , int arg2)
    - assertEquals(Object o1, Object o2)
      - définie pour tous les types primitifs et les objets
    - assertTrue(boolean condition)
    - assertFalse(boolean condition)

Tests unitaires

- JUnit 4
  - Méthodes statiques de la classe Assert (suite)
    - assertNull(Object object)
    - assertNotNull(Object object)
    - assertSame(Object expected, Object actual)
      - utilise == alors que assertEquals utilise equals
    - assertNotSame(Object unexpected, Object actual)
  - Toutes les méthodes statiques de la classe Assert existent aussi avec un argument supplémentaire de type String affiché en cas d'échec du test
    - assertTrue("Valeur négative", x < 0)
    - assertEquals("Nb de mesures different", m.length, l.size())

- JUnit 4
  - Attention aux problèmes d'arrondis dans les tests d'égalité
    - Utiliser la méthode appropriée qui permet de spécifier une tolérance (delta) sur le test d'égalité

```
public static void assertEquals(String message,
                                double expected,
                                double actual,
                                double delta)
```

  - Exemple :

```
assertEquals("Diagonale du carre", Math.sqrt(2), Carre.diagonale(2), 1e-6);
```

Tests unitaires

- JUnit 4
  - Suite de tests
    - Fichier JUnit qui permet de lancer plusieurs classes de test

```
package fr.lp.iem.test;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({CircleTest.class, RectangleTest.class, PointTest.class})
public class AllTests {
}
```

    - Il est également possible de créer une configuration qui lance tous les tests d'un dossier sous IntelliJ

- JUnit 4 + hamcrest
  - La bibliothèque hamcrest est en partie intégrée dans JUnit
    - Elle permet d'écrire des tests plus proches d'une phrase en anglais

```
assertThat(actual, is(equalTo(expected)));
assertThat(actual, is(not(equalTo(expected))));
assertThat(actual, is(expected));
assertThat(actual, is(not(expected)));

assertThat(Conversion.rpmToRadPerSec(60), closeTo(6.283185, DELTA) );
```

  - Il faut souvent la compléter en ajoutant la librairie complète
    - Project Settings -> Global Libraries -> + -> import from maven : org.hamcrest:hamcrest-library
    - Dans les options Dependencies du module, passer la librairie hamcrest avant JUnit

IUT Lyon 1

l'excellence technologique

Objets immuables

LP IEM

Objets immuables

- Objets immuables
  - Avantages
    - Simples à construire, tester et utiliser
    - automatiquement thread-safe (pas de pb de synchronisation)
    - n’ont pas besoin d’être copiés défensivement
    - n’ont pas besoin d’implémenter un constructeur de copie ou la méthode clone
    - permettent à la méthode hashcode de mettre son résultat en cache
    - font de bonnes clés pour les Map et de bons éléments pour les Set (ces objets ne doivent pas changer quand ils sont dans ces collections)

Objets immuables

- Objets immuables
  - Avantages
    - invariants de classe testés à la construction et n’ont plus besoin d’être testés par la suite (exemple : objet de type Range avec une borne min toujours inférieure à sa borne max)
      - Les objets ne peuvent pas exister dans un état invalide ou incohérent

Objets immuables

- Objets immuables
  - Règles pour créer une classe immuable
    - interdire l’héritage (marquer la classe final ou utiliser des fabriques statiques avec un constructeur private)
    - rendre les champs private et final
    - forcer les utilisateurs à créer la classe en une seule étape
    - ne pas fournir de méthodes permettant de modifier un objet
    - si la classe possède une référence vers un objet qui n’est pas immuable il doit faire l’objet d’une copie défensive à chaque fois qu’il est passé entre différentes classes

Objets immuables

Objets immuables - exercice

- Créer une classe immuable Point avec un constructeur prenant les coordonnées x et y en paramètres (de type int)
  - Ajouter une méthode double distance(Point p)
  - Ajouter une méthode move qui prendra en paramètre la valeur du déplacement dx et dy
  - Implémenter les méthodes hashcode() et equals(Object o)
  - Ecrire des tests unitaires pour :
    - valider les méthode distance et move
    - valider le caractère immuable et le bon comportement de l'identité et de l'égalité

Objets immuables

Objets immuables - exercice

- Créer une interface Shape (ne pas utiliser java.awt.Shape) avec :
  - une méthode boolean contains(Point p) qui renvoie vrai si les coordonnées du point (p.x,p.y) sont à l'intérieur de la forme
  - une méthode double area() retournant l'aire
- Créer des classes Circle, Rectangle implémentant l'interface Shape
- Constructeurs :
  - Circle (Point center, int radius)
  - Rectangle(Point topLeftCorner, int width, int height)
- Ecrire des tests unitaires pour valider les méthodes contains et area