POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR

Mocks

Philippe Collet **Polytech Nice Sophia – PS5**

(Dice code from S. Mosser and S. Urli)

Dice Game

Dice game : spécifications

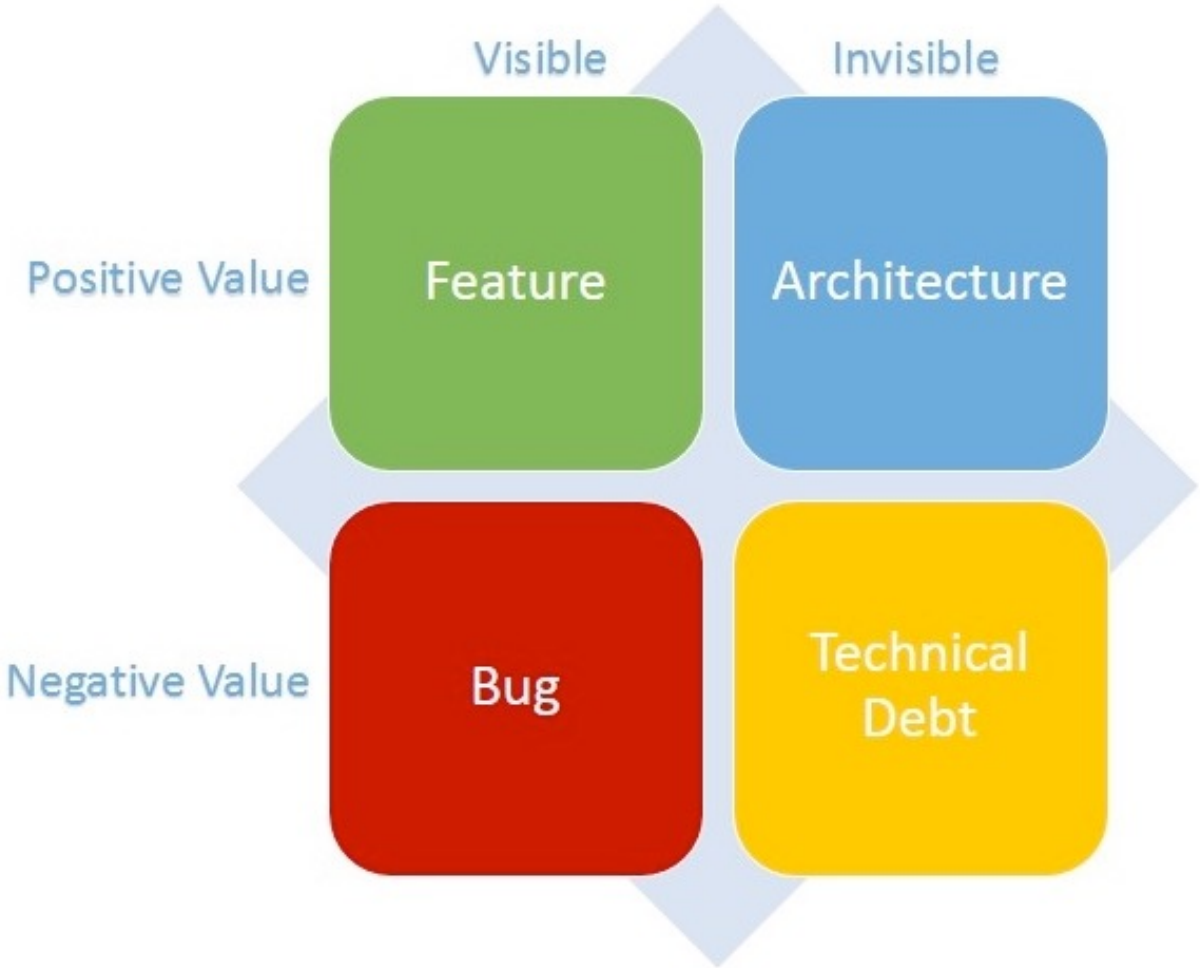
- **Etre capable de jeter un dé**
  - Critère d'acception : le dé a 6 faces, et retourne un nombre aléatoire en 1 et 6.
- **Associer un jet de dé à un joueur précis**
  - Critère d'acception : un joueur a un nom, et expose la valeur obtenue de son propre dé
- **Le joueur lance deux dés et garde le maximum**
  - Critère d'acception : le dé est lancé juste 2 fois, et seulement la valeur maximale est conservée
- **Le jeu de dés se joue à 2, et le joueur qui obtient la valeur maximale après un lancer gagne** (ex-aequo entraîne une relance, pas de vainqueur après 5 matches ex-aequo)
  - Critère d'acception : le jeu expose un vainqueur en suivant les règles

Tâche 1 : Etre capable de jeter un dé

- Critère d'acception : le dé a 6 faces, et retourne un nombre aléatoire en 1 et 6.

- Package god (Game Of Dices)
- Classe Dice
  - Méthode roll
- Reproductibilité du jeu ?
  - Objet random fourni à la création
- Si random ne marche pas bien ?
  - RuntimeException…
  - C'est de la **dette technique** !

Dette technique ?



```java
package god;

import java.util.Random;

public class Dice {

    private final static int FACES = 6;
    private Random rand;

    public Dice(Random rand) { this.rand = rand; }

    public int roll() {
        int result = rand.nextInt(FACES) + 1;
        if (result < 1 || result > FACES)
            throw new RuntimeException("Dice returns an incompatible value");
        return result;
    }

}
```

Tester

- Modéliser les critères d'acceptation
  - Bonne propriété : quand les tests passent, la tâche est terminée !
- On a besoin de tester :
  - Lancer un dé rend une valeur ?
  - Une valeur hors de [1-6] lance une exception

Créer sa propre classe Random ?

- Pas terrible de redéfinir une classe juste pour les tests
  - Si elle était plus compliquée, ce serait quasiment infaisable

Maîtriser l’aléatoire…

```java
package god;

import org.junit.jupiter.api.Test;
import java.util.Random;

import static org.junit.jupiter.api.Assertions.assertThrows;
import static org.junit.jupiter.api.Assertions.assertTrue;
import static org.mockito.Mockito.*;

public class DiceTest {

    Dice theDice;
```

Et si je pouvais simuler un Random qui rend toujours la même valeur…

https://site.mockito.org/

Définition

- Mock = Objet factice
- les mocks (ou Mock object) sont des objets simulés qui reproduisent le comportement d'objets réels de manière contrôlée

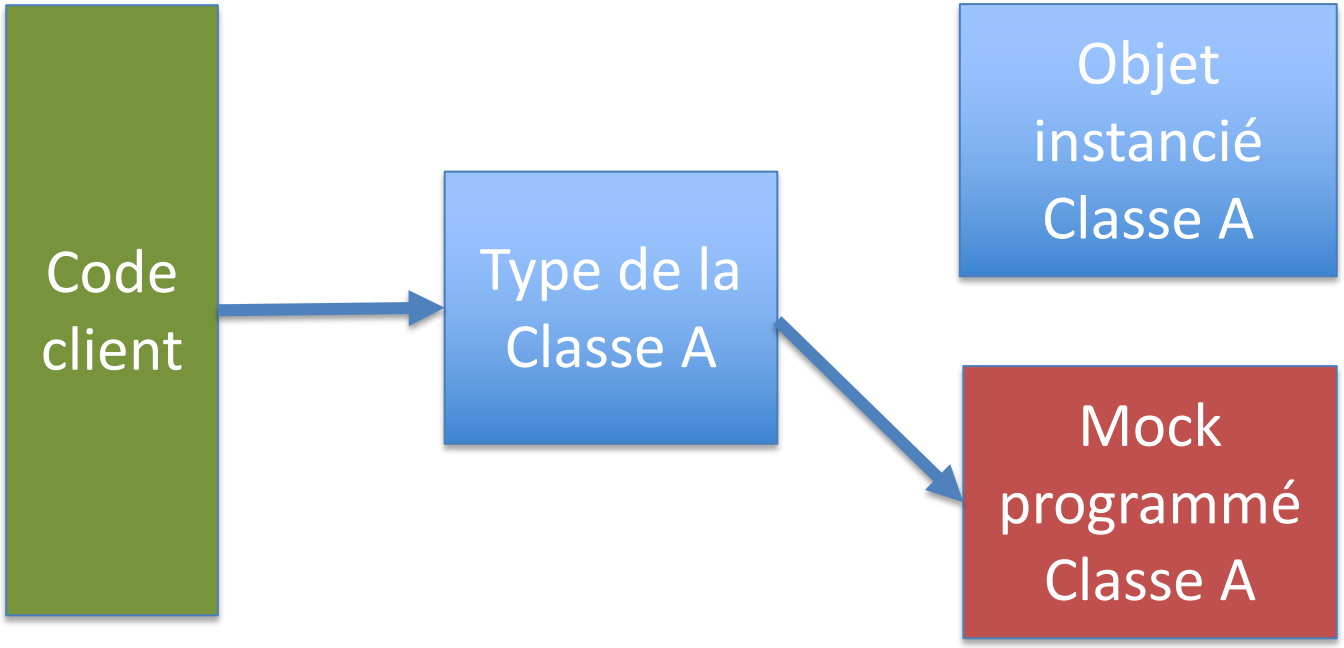
- On teste ainsi le comportement d'autres objets, réels, mais liés à un objet simulé, le mock

Exemples d'utilisation

- Comportement non déterministe (l'heure, un senseur)
- Initialisation longue (BD)
- Classe pas encore implémentée ou implémentation changeante, en cours
- Etats complexes difficiles à reproduire dans les tests (erreur réseau, exception sur fichiers)
- Si pour tester, il faudrait ajouter des attributs ou des méthodes

Principe

- 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



Principe

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**

Comment ça marche puisque l'objet est **FAUX** ?

Principes : un mode **espion**

import static org.mockito.Mockito.*

1. Création des mocks
   - méthode **mock** ou annotation @mock
2. Description de leur comportement
   - Méthode **when**
3. Mémorisation à l'exécution des interactions
   - Utilisation du mock dans un code qui teste un comportement spécifique
4. Interrogation, à la fin du test, des mocks pour savoir comme ils ont été utilisés
   - Méthode **verify**

Création

- Par une interface ou une classe (utilisation de .class)
  - UneInterface mockSansNom = **mock**(UneInterface.class);
  - UneInterface mockAvecNom =
    mock(UneInterface.class, "ceMock");
  - **@Mock** UneInterface ceMock**;**

Stubbing

- Pour remplacer le comportement par défaut des méthodes

- Pour une méthode qui a un type de retour :
  - **when + thenReturn** ;
  - when + thenThrow ;
- Pour une méthode de type void :
  - doThrow + when ;

```java
public class DiceTest {

    Dice theDice;

    @Test
    public void identifyBadValuesGreaterThanNumberOfFaces() {
        Random tooMuch = mock(Random.class);
        when(tooMuch.nextInt(anyInt())).thenReturn(7);
        theDice = new Dice(tooMuch);
        assertThrows(RuntimeException.class, () -> {
            theDice.roll();
        } );
    }

    @Test
    public void identifyBadValuesLesserThanOne() {
        Random notEnough = mock(Random.class);
        when(notEnough.nextInt(anyInt())).thenReturn(-1);
        theDice = new Dice(notEnough);
        assertThrows(RuntimeException.class, () -> {
            theDice.roll();
        } );
    }
```

Matchers

- Mockito vérifie les valeurs en arguments en utilisant equals()
- Assez souvent, on veut spécifier un appel dans un « when » ou un « verify » sans que les valeurs des paramètres aient vraiment d'importance

- Mockito fourni des fonctions *Matchers*
  - when(mockedList.get(**anyInt**())).thenReturn("element");
  - verify(mockedList).get(**anyInt**());

Compléments sur les *Matchers*

- Les plus utilisés :
  - any() , static <T> T anyObject()
  - anyBoolean(), anyDouble(), anyFloat() , anyInt(), anyString()…
  - anyList(), anyMap(), anyCollection(), anyCollectionOf(java.lang.Class<T> clazz) ,anySet(), <T> java.util.Set<T> anySetOf(java.lang.Class<T> clazz)

- **Si on utilise des matchers, tous les arguments doivent être des matchers :**

  verify(mock).someMethod(anyInt(), anyString(), "third argument");

  n'est pas correct !

Tâche 2 : Associer un jet de dé à un joueur précis

- Critère d'acception : un joueur a un nom, et expose la valeur obtenue de son propre dé

- Classe Player
  - Constructeur avec le nom du joueur et le dé
  - Attribut lastValue (et getter) pour la dernière valeur du dé (-1 sinon)
  - Méthode play lance le dé et stocke la valeur

- Test ?

Optional en Java 8

- Renvoyer -1, c'est moche et cela peut être mal interprété -> Dette technique encore…
- Optional : encapsuler un objet qui peut être *null*

- Modifier la classe Player
  - lastValue est un Optional

- Impact sur les tests
  - Vérifier juste que la valeur est présente ou pas…

```java
package god;
import java.util.Optional;

public class Player {

    private String name;
    private Dice dice;
    private Optional<PlayResult> lastValue = Optional.empty();

    public Player(String name, Dice dice) {
        this.name = name;
        this.dice = dice;
    }

    public Optional<PlayResult> getLastValue() { return lastValue; }
```

PlayResult ?

Le même genre d'abstraction que dans le résultat de la comparaison dans la main de Poker !

```java
package god;

public class PlayResult implements Comparable<PlayResult> {

    private int value;

    public PlayResult(int val) { this.value = val; }

    public int get() { return value; }

    public int compareTo(PlayResult o) {
        int otherVal = o.get();
        return value < otherVal ? -1 : value > otherVal ? +1 : 0;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof PlayResult)
            return (value == ((PlayResult)obj).get());
        return false;
    }
```

```java
public class PlayResultTest {

    PlayResult pr1;
    PlayResult pr2;

    @BeforeEach
    public void setUp() {
        pr1 = new PlayResult( val: 5);
        pr2 = new PlayResult( val: 2);
    }

    @Test
    public void testCompareTo() {
        assertTrue( condition: pr1.compareTo(pr2) > 0);
        assertTrue( condition: pr2.compareTo(pr1) < 0);
        assertTrue( condition: pr1.compareTo(pr1) == 0);
        assertTrue( condition: pr1.compareTo(new PlayResult( val: 5)) == 0);
    }

    @Test
    public void testEquals() {
        assertEquals(pr1,pr1);
        assertEquals(pr1,new PlayResult( val: 5));
        assertNotEquals(pr1,pr2);
    }

}
```

```java
public class PlayerTest {

    Player p;

    @Test
    public void lastValueNotInitialized() {
        p = new Player( name: "John Doe", new Dice(new Random()));
        assertFalse(p.getLastValue().isPresent());
    }

    @Test
    public void lastValueInitialized() {
        p = new Player( name: "John Doe", new Dice(new Random()));
        p.play();
        assertTrue(p.getLastValue().isPresent());
    }
```

La méthode play n'a qu'à stocker le jet de dé dans lastValue

Tâche 3 : Le joueur lance deux dés et garde le maximum

- Critère d'acception : le dé est lancé juste 2 fois, et seulement la valeur maximale est conservée

- Modifier la méthode play pour lancer deux dés et garder le max

- Tests ?
  - Le joueur suit bien les règles
  - Le joueur garde la valeur max

```java
package god;
import java.util.Optional;

public class Player {

    private String name;
    private Dice dice;
    private Optional<PlayResult> lastValue = Optional.empty();

    public Player(String name, Dice dice) {
        this.name = name;
        this.dice = dice;
    }

    public void play() {
        int a = dice.roll();
        int b = dice.roll();
        this.lastValue = Optional.of(new PlayResult(Math.max(a, b)));
    }

    public Optional<PlayResult> getLastValue() { return lastValue; }

}
```

```java
@Test
public void lastValueNotInitialized() {
    p = new Player( name: "John Doe", new Dice(new Random()));
    assertFalse(p.getLastValue().isPresent());
}

@Test
public void lastValueInitialized() {
    p = new Player( name: "John Doe", new Dice(new Random()));
    p.play();
    assertTrue(p.getLastValue().isPresent());
}

@Test
public void throwDiceOnlyTwice() {
    Dice d = mock(Dice.class);
    p = new Player( name: "John Doe", d);
    p.play();
    verify(d, times( wantedNumberOfInvocations: 2)).roll();
}
```

Comportements par défaut

```
assertEquals("ceMock", monMock.toString());

assertEquals("type numerique : 0 ", 0,
                                monMock.retourneUnEntier());

assertEquals("type booleéen : false",
                             false, monMock.retourneUnBooleen());

assertEquals("type collection : vide",
                                0, monMock.retourneUneList().size());
```

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());
```

Un comportement de mock non exécuté ne provoque pas d'erreur

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);
```

```java
@Test
public void keepTheMaximum() {
    Dice d = mock(Dice.class);
    p = new Player( name: "John Doe", d);

    when(d.roll()).thenReturn( t: 2, ...ts: 5);
    p.play();
    assertEquals(p.getLastValue().get(), new PlayResult( val: 5));

    when(d.roll()).thenReturn(6).thenReturn(1);
    p.play();
    assertEquals(p.getLastValue().get(), new PlayResult( val: 6));

}
```

Tâche 4 : Le jeu de dés se joue à 2…

et le joueur qui obtient la valeur maximale après un lancer gagne (ex-aequo entraîne une relance, pas de vainqueur après 5 matches ex-aequo)

- Critère d'acception : le jeu expose un vainqueur en suivant les règles

- Création d'une classe Game
  - Deux joueurs (left, right)

- Tests ?
  - Cas « pas de vainqueur »

  - Cas « un vainqueur »

```java
public class Game {

    private Player left;
    private Player right;

    public Game(Player left, Player right) {
        this.left = left;
        this.right = right;
    }

    public Optional<Player> play() {
        int counter = 0;
        while(counter < 5) {
            left.play();  PlayResult l = left.getLastValue().get();
            right.play(); PlayResult r = right.getLastValue().get();

            int cmp = l.compareTo(r);

            if(cmp > 0 )      { return Optional.of(left);  }
            else if (cmp < 0) { return Optional.of(right); }

            counter++;
        }
        return Optional.empty();
    }
```

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);

Espionner un objet classique

- Pour espionner autre chose qu'un objet mock (un objet « réel »):
  - Obtenir un objet espionné par la méthode spy :

    List list = new LinkedList();

    List spy = **spy**(list);
  - Appeler des méthodes « normales » sur le spy :

    spy.add("one");

    spy.add("two");
  - Vérifier les appels à la fin :

    verify(spy).add("one");

    verify(spy).add("two");

```java
public class GameTest {

    Game g;

    @Test
    public void noWinnerAfter5Attempts() {
        Dice single = mock(Dice.class);
        when(single.roll()).thenReturn(1);

        Player p1 = spy(new Player( name: "John", single));
        Player p2 = spy(new Player( name: "Jane", single));

        g = new Game(p1,p2);
        assertFalse(g.play().isPresent());
        verify(p1, times( wantedNumberOfInvocations: 5)).play();
        verify(p2, times( wantedNumberOfInvocations: 5)).play();
    }
```

```java
@Test
public void andTheWinnerIsP1() {

    Player p1 = mock(Player.class);
    when(p1.getLastValue()).thenReturn(Optional.of(new PlayResult( val: 5)));

    Player p2 = mock(Player.class);
    when(p2.getLastValue()).thenReturn(Optional.of(new PlayResult( val: 2)));

    g = new Game(p1,p2);
    assertEquals(p1, g.play().get());
}

@Test
public void andTheWinnerIsP2() {

    Player p1 = mock(Player.class);
    when(p1.getLastValue()).thenReturn(Optional.of(new PlayResult( val: 1)));

    Player p2 = mock(Player.class);
    when(p2.getLastValue()).thenReturn(Optional.of(new PlayResult( val: 6)));

    g = new Game(p1,p2);
    assertEquals(p2, g.play().get());
}
```

GRASP!
Pas besoin de Dice

Maven avec Java 17 / JUnit 5.8 ?

- Démo et setup :

  https://github.com/collet/dice-mockito-demo

- Dans votre projet, ajouter au pom.xml :

```xml
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>17</java.version>
    <maven.compiler.source>${java.version}</maven.compiler.source>
    <maven.compiler.target>${java.version}</maven.compiler.target>
    <junit.jupiter.version>5.8.0</junit.jupiter.version>
    <mockito.version>3.12.4</mockito.version>
</properties>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-inline</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

Mockito-inline
version spécifique qui est capable de mocker les enums, les types et méthodes « final » (comme celle de Random)

Avec Java 11 ?

- pom.xml :

```xml
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>11</java.version>
    <maven.compiler.source>${java.version}</maven.compiler.source>
    <maven.compiler.target>${java.version}</maven.compiler.target>
    <junit.jupiter.version>5.6.2</junit.jupiter.version>
    <mockito.version>3.6.0</mockito.version>
</properties>

<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>${mockito.version}</version>
    <scope>test</scope>
</dependency>
```

Mockito-core
(version standard)

Fonctionnalités avancées

https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html

- Mocker des classes abstraites, des méthodes statiques (> 3.4), des constructions d'objets (> 3.5)
- Answer: pour implémenter des réponses non précablées
- …