



# Test Driven Development en Java

**Dr. Ing. Chiheb Ameur ABID**

*Contact : chiheb.abid@gmail.com*

Janvier 2021

Version 1.1



## Plan

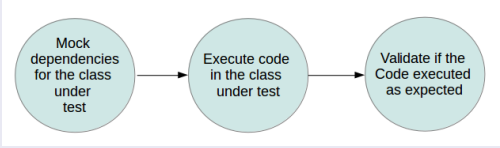




## Présentation de Mockito

### Pourquoi le mocking ?

- Certains objets "réels" dans les tests sont difficiles à instancier et à les configurer
- Parfois, seulement les interfaces existent ☞ Pas d'implémentation prête






## Présentation de Mockito

### Exemples d'utilisation

- Comportement non déterministe (heure courante, nombre généré aléatoirement, température ambiante, etc.)
- Initialisation longue (BD)
- Classe pas encore implémentée ou implémentation non stable
- États complexes difficiles à reproduire dans les tests (erreur réseau, exception sur fichiers)
- Pour tester, il faudrait patfois ajouter des attributs ou des méthodes aux classes applicatives (parasiter une classe applicative)

## Présentation de Mockito

### Définition

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

- Avec les mocks, on teste le comportement d'autres objets, réels, mais liés à un objet inaccessible ou non implémenté
- L'utilisation des Mocks dans les tests consiste à spécifier
  1. Quelles méthodes vont être appelées, avec quels paramètres et dans quel ordre
  2. Les valeurs retournées par le mock





## Présentation de Mockito

### Concepts

- **Dummy** (panin, factice) : objets vides qui n'ont pas de fonctionnalités implémentées. Ils sont transmis mais ne sont jamais réellement utilisés. Habituellement, ils sont juste utilisés pour remplir des listes de paramètres selon les paramètres
- **Stub** (bouchon) : classes qui renvoient en dur une valeur pour une méthode invoquée. Ils ne répondent généralement pas du tout à ce qui est en dehors de ce qui est programmé pour le test.
- **Mock** (factice) : des objets préprogrammés avec des attentes qui forment une spécification des appels qu'ils sont censés recevoir.
- **Fake** (subsBtut, simulateur) : implémentation partielle qui renvoie toujours les mêmes réponses
- **Spy** (espion) : les objets sont des répliques partielles d'objets réels : certaines méthodes sont moquées





## Présentation de Mockito

### Présentation de Mockito

- Mockito est un framework de simulation basé sur java, utilisé en conjonction avec d'autres frameworks de test tels que JUnit et TestNG, etc.
- Mockito est un générateur automatique de doublures
- Un Mock renvoie des données factices et évite les dépendances externes






## Génération des doublures avec Mockito

### Cycle de vie d'un mock dans un cas de test

- La procédure dutilisation de Mockito est très simple
  1. Création dun objet mock pour une classe ou une interface ;
  2. Description du comportement qu'il est censé imiter ;
  3. Utilisation du mock dans le code de test ;
  4. Si nécessaire, interrogation du mock pour savoir comment il a été utilisé durant le test.

# Intégration de Mockito

## Intégration de Mockito avec Maven

- Il est possible d'ajouter les fichiers `.jar` (Mockito et Junit) dans le projet
- Une manière plus simple consiste à utiliser `Maven`
- `Maven` est un outil créé par Apache, il permet de faciliter la gestion d'un projet Java
  - C'est un Outil de Gestion de Dépendance (Dependency Management Tool)
  - Il permet de mieux structurer un projet : séparer la partie liée au code du projet, le code des tests unitaires et autres fichiers statiques (images, PDF etc.)
  - L'ensemble du projet est géré à partir d'un seul fichier : `pom.xml`

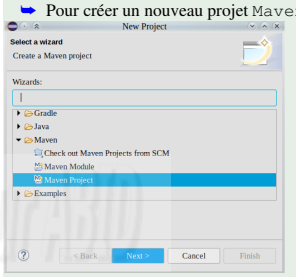


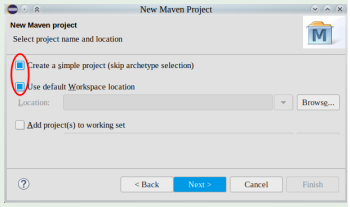



# Utilisation de Mockito

## Exemple illustratif : Intégration de Mockito dans un projet Java

- Pour créer un nouveau projet `Maven`, aller au menu `File->New->Project`



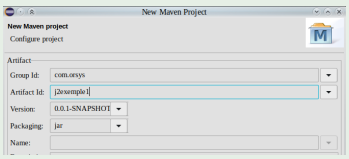




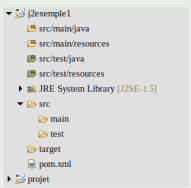

# Utilisation de Mockito

## Exemple illustratif : Intégration de Mockito dans un projet Java

- En général, le `Group Id` correspond au nom de l'organisation, et l'`Artifact Id` correspond au nom du projet




- `Maven` prend en charge la structuration du projet



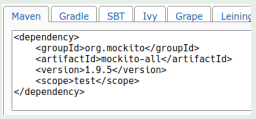





# Utilisation de Mockito

## Exemple 1 : Intégration de Mockito dans un projet Java

- Aller au site `www.mvnrepository.com` et récupérer les méta-données relatives au framework `Mockito`




- On peut ajouter les dépendances `Mockito` de deux manières
  1. En Insérant directement les méta-données dans le fichier `pom.xml`

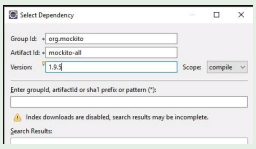
## Utilisation de Mockito

### Exemple illustratif : Intégration de Mockito dans un projet Java

- On peut ajouter les dépendances `Mockito` de deux manières
  1. En Insérant directement les méta-données dans le fichier `pom.xml`

```
<dependencies>
  <dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-all</artifactId>
    <version>1.9.5</version>
  </dependency>
</dependencies>
```

  2. Après avoir ouvert le fichier `pom.xml`, choisir l'onglet `Dependencies`. Puis, cliquer sur le bouton `Add` : une boîte de dialogue s'affiche pour saisir les informations de dépendances








## Génération des doublures avec Mockito

### Mocking : création de doublure

- Lutilisation de Mockito nécessite au préalable limportation statique du paquet `Mockito` `import static org.mockito.Mockito.*;`
- Deux manières de créer des doublures, soit avec la méthode `mock()` ou bien avec l'annotation `@Mock`.

### Avertissement

Mockito ne peut pas mocker ou espionner : les classes déclarées final, les méthodes déclarées final, les énumérations enums, les méthodes statiques, les méthodes privées (déclarées private), les méthode hashCode() et equals(), les classes anonymes, et les types primitifs.





## Génération des doublures avec Mockito

### Création d'une doublure avec la méthode `mock()`

- Création d'un Mock sans nom
  `MaClasse monMock = mock(MaClasse.class);`
- Création d'un Mock avec attribution de nom
  `MaClasse mockAvecNom = mock(MaClasse.class, "Mon mock");`

### Exemple

```
public interface Calcul {
        int Somme(int x,int y);
}
/**********************************/
import org.junit.Test;
import static org.mockito.Mockito.*;

public class ExempleTest {
        @Test
        public void testMock() {
                Calcul monMock=mock(Calcul.class);
                assertTrue(true);
        }
}
```





## Génération des doublures avec Mockito

### Création d'une doublure avec l'annotation `@Mock`

- Lannotation se met au-dessus de la déclaration de lattribut du type du mock. Le nom du mock est automatiquement celui de lattribut.
  `@Mock`
  `private MaClasse monMock;`
- Il ne faut pas oublier d'initialiser linterpréteur des annotations de Mockito
  1. Initialisation par lappel de méthode `initMocks()` annotés par `@Mock`, qui se trouvent dans la classe passée en paramètre.
  2. Initialisation par le moteur dexécution `MockitoJUnitRunner`
  3. Initialisation par la règle `MockitoRule`

## Création d'une doublure avec l'annotation `@Mock`

### Initialisation par lappel de méthode `initMocks()`

- C'est la méthode la plus classique
- La méthode MockitoAnnotations.initMocks(this) initialise tous les attributs annotés par @Mocks, qui se trouvent dans la classe passée en paramètre.

```
public void FooTest {
        private Foo foo;
        @Mock
        private MaClasse monMock;
        @Before
        public void setUp() {
                MockitoAnnotations.initMocks(this);
                foo = new Foo(monMock);
                ... // Testing
        }

}
```





## Création d'une doublure avec l'annotation `@Mock`

### Initialisation par le moteur dexécution MockitoJUnitRunner

- Cette option n'est utilisable que si Mockito est le seul moteur utilisé. Ceci exclut par exemple l'utilisation des paramètres de test JUnit

```
@RunWith(MockitoJUnitRunner.class)
public void FooTest {
        private Foo _foo;
        @Mock
        private MaClasse _monMock;
        @Before
        public void setUp() {
                _foo = new Foo(_monMock);
                ... // Testing
        }
        ...
}
```





## Création d'une doublure avec l'annotation `@Mock`

### Initialisation par la règle MockitoRule

- La règle JUnit invoque implicitement la méthode `initMocks(this)`

```
public void FooTest {
        private Foo _foo;
        @Mock
        private MaClasse _monMock;
        @Rule
        public MockitoRule mockitoRule = MockitoJUnit.rule();
        @Before
        public void setUp() {
                _foo = new Foo(_monMock);
        }
        ...
}
```





## Génération des doublures avec Mockito

### Stubbing

- Le stubbing consiste à définir le comportement des méthodes dun mock.
- Appel d'une méthode avec une valeur de retour unique
  `when(monMock.retourneUnEntier()).thenReturn(3);`
- Appel d'une méthode avec des valeurs de retour consécutives
  `when(monMock.retourneUnEntier()).thenReturn(3, 4);`
  `when(monMock.retourneUnEntier()).thenReturn(3).thenReturn(4);`

### Exemple

```
public class ExempleTest {
        @Test
        public void testMock() {
                Calcul monMock=mock(Calcul.class);
                when(monMock.Somme(4,3)).thenReturn(7);
                when(monMock.Incrementer()).thenReturn(0).thenReturn(1);

                System.out.println("Somme(4,3) retourne : "+monMock.Somme(4,3));
                System.out.println("Incrementer : "+monMock.Incrementer());
                System.out.println("Incrementer : "+monMock.Incrementer());
                assertTrue(true);
        }
}
```

## Génération des doublures avec Mockito

### Stubbing : Appel d'une méthode avec n'importe quel paramètre

- Il est possible de spécifier un appel sans que les valeurs des paramètres aient vraiment dimportance. On utilise pour cela des **Matchers**
  - `any(), anyObject()`
  - `anyBoolean(), anyDouble(), anyFloat() , anyInt(), anyString()`, etc.
  - `anyList(), anyMap(), anyCollection(), anyCollectionOf(), anySet(), anySetOf()`

### Avertissement

Si on utilise des Matchers, tous les arguments doivent être des Matchers





## Génération des doublures avec Mockito

### Stubbing : Levée d'exception

- Il est possible de lever une exception lorsqu'une méthode est appelée
  - `when(monMock.methode()).thenThrow(new BidonException);`
  - `doThrow(new BidonException()).when(monMock).methode();`
- Il est possible de cumuler le retour dune valeur donnée puis la levée dune exception `when(monMock.methode()).thenReturn(3).thenThrow(new BidonException());`
- Un test JUnit permettant de vérifier si l'appelle d'une méthode a levé une exception

```
@Test(expected = BidonException.class)
public void should_throw_exception() {
        monMock.methode();
}
```

### Exemple

```
@Test(expected=IllegalArgumentException.class)
public void testMockException() {
        Calcul monMock=mock(Calcul.class);
        when(monMock.Diviser(5,0)).thenThrow(new IllegalArgumentException("Argument nul"));
        when(monMock.Diviser(5,2)).thenReturn(2);
        System.out.println("Diviser (5,2) -"+monMock.Diviser(5,2));
        System.out.println("Diviser (5,0) -"+monMock.Diviser(5,0));
}
```

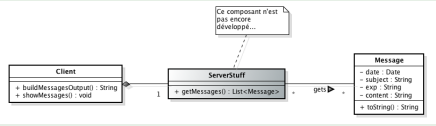




## Exemple 1 : utilisation de Mockito

### Exemple 1 : utilisation de Mockito

On considère la structure de classes ci-après décrivant un système de messagerie simplifié




- La classe `Client` décrit un utilisateur de la messagerie
- La classe `ServerStuff` représente le serveur de messagerie associé à un client. Cependant, cette classe n'est pas encore implémentée. On sait seulement qu'elle dispose d'une méthode qui permet de récupérer les nouveaux messages du client associé.
- Un message est décrit par une instance de la classe `Message`
- Notre objectif est de tester la classe `Client`





## Exemple 1 : utilisation de Mockito

### Code de la classe `Message`

```
public class Message {
  Date date;
  String subject;
  String exp;
  String content;

  public Message(Date date, String subject, String exp, String content) {
    this.date = date;
    this.subject = subject;
    this.exp = exp;
    this.content = content;
  }

  public String toString() {
    return subject + ", reçu le " + date + ", de : " + exp + "\n" + content;
  }
}
```

### Code de l'**interface** `ServerStuff` !!

```
import java.util.List;

public interface ServerStuff {
        List<Message> getMessages();
}
```

## Exemple 1 : utilisation de Mockito

### Code de la classe `Client`

```
public class Client {

  ServerStuff serverStuff;

  public Client(ServerStuff serverStuff) {
      this.serverStuff = serverStuff;
  }

  public String buildMessagesOutput() {
      List messages = this.serverStuff.getMessages();

      String output = "";

      output += "====================\n";
      output += "Vos nouveaux messages\n";
      output += "====================\n";
      for (Message message  messages) {
          output += message + "\n";
      }
      output += "====================";

      return output;
  }
}
```





## Exemple 1 : utilisation de Mockito

### Code de la classe de test

```
package j2exemple1;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import java.util.*;
class ClientTest {
  private Client client;
  @BeforeEach
  void setUp() throws Exception {
    // Mocking
        ServerStuff serverStuff = mock(ServerStuff.class);
        // Stubbing
        when(serverStuff.getMessages()).thenReturn((List<Message>) Arrays.asList(
                        new Message(new Date(2010, 11, 7), "Bonjour !", "bob@u-picardie.fr", "Comment ca va ?"),
                        new Message(new Date(2010, 11, 8), "Un test", "testeur@u-picardie.fr", "
                            Bonjour le test !")));
        // Instancier le client
        this.client = new Client(serverStuff);
  }

  @Test
  void test() {
        // Checking
        String output = this.client.buildMessagesOutput();
        assertTrue(output.equals("====================\nVos nouveaux messages\n====================\
              nBonjour !, recu le Wed Dec 07 00:00:00 CET 3910, de : bob@u-picardie.fr\nComment ca va
              ?\nUn test, recu le Thu Dec 08 00:00:00 CET 3910, de : testeur@u-picardie.fr\nBonjour
              le test !\n===================="));
  }
}
```





## Génération des doublures avec Mockito

### Exercice

- On veut modéliser un jeu de casino : le jeu de la boule
- Le jeu de la boule est un jeu de casino simplifié du jeu de la roulette. Il utilise les chiffres de 1 à 9. Le joueur qui joue fait un pari en misant une somme.
- Il est possible de miser sur rouge, noir, manque, passe, pair ou impair.
- Les chiffres 1, 3, 7, 9 sont impairs. Les chiffres 2, 4, 6, 8 sont pairs.
- Les chiffres 1, 3, 6, 8 sont noirs. Les chiffres 2, 4, 7, 9 sont rouges.
- Les chiffres 1, 2, 3, 4 sont "manque" ("on a manqué de dépasser 5"). Les chiffres 6, 7, 8, 9 sont "passe" ("on a dépassé 5").
- Ces chances sont des chances simples : si la chance simple misée sort, le joueur gagne une fois la mise (qui lui est restituée), sinon la mise est perdue.
- Le chiffre 5 n'est ni pair, ni impair, ni manque, ni passe, ni rouge, ni noir. Si le 5 sort, la mise jouée sur une chance simple est perdue.
- Le joueur peut miser sur un numéro. Si celui-ci sort, le joueur gagne 7 fois la mise qui lui est restituée sinon la mise est perdue.
- Un joueur peut évidemment miser sur plusieurs cases (et même sur rouge et noir !). Il ne peut miser que des quantités entières (des jetons de valeur entière en euro).





## Génération des doublures avec Mockito

### Exercice

- Pour modéliser ce jeu on utilise aux moins deux classes : la classe `Joueur` qui modélise un joueur et `CroupierBoule` qui modélise le gestionnaire du jeu de la boule : le croupier.
- La classe `CroupierBoule` possède la méthode `public int getNumSorti()` qui retourne le numéro sorti

1. Un joueur est lié au casino et c'est le casino qui lui indique combien il a gagné ou perdu. Il peut savoir quel numéro est sorti en le demandant au casino (ou au représentant du casino). Donner le code de la classe `Joueur`. C'est la classe à tester et on veut l'isoler de la classe qui modélise le casino (son mock).
2. Écrire les tests pour les cas suivants :
   - Le joueur gagne : Le joueur n'a joué que sur le 8 avec 3 jetons et le 8 est sorti
   - Le joueur perd : Le joueur n'a joué que sur le 8 avec 3 jetons et le 9 est sorti

## Génération des doublures avec Mockito

### Espionnage

- Mockito garde trace de tous les appels de méthode. Vous pouvez utiliser la méthode `verify()` sur un objet mock pour vérifier qu'une méthode a été appelée et avec certaines valeurs de paramètre.
- Ce type de test correspond à un test de comportement
- Il est possible d'espionner un objet classique à l'aide de la méthode `spy()` ou l'annotation `@Spy`



## Fonctionnement de la méthode `verify()`

### Fonctionnement de la méthode `verify()`

- Elle permet de vérifier :
  - Quelles méthodes ont été appelées sur un mock,
  - Combien de fois,
  - Avec quels paramètres,
  - Dans quel ordre.
- Si la vérification échoue, il y a une levée d'exception et le test unitaire échoue



## Fonctionnement de la méthode `verify()`

### Vérification du nombre d'appels

- Vérifier qu'une méthode est appelée exactement une fois
  `verify(monMock).retourneUnBooleen();`
- Vérifier qu'une méthode est appelée exactement `n` fois
  `verify(monMock,times(n)).retourneUnBooleen();`
- Vérifier qu'une méthode est appelée au moins une fois
  `verify(monMock,atLeastOnce()).retourneUnBooleen();`
- Vérifier qu'une méthode est appelée au plus `n` fois
  `verify(monMock,atMost(n)).retourneUnBooleen();`
- Vérifier qu'une méthode n'est jamais appelée
  `verify(monMock,never()).retourneUnBooleen();`



## Fonctionnement de la méthode `verify()`

### Vérification de l'ordre des appels

- Cette vérification nécessite dimporter la classe InOrder
  `import org.mockito.InOrder;`
- Vérifier qu'un appel est effectué avant un autre
```
InOrder ordre = inOrder(monMock);
ordre.verify(monMock).retourneUnEntierBis(4, 2);
ordre.verify(monMock).retourneUnEntierBis(5, 3);
```
- On peut utiliser Vérifier qu'une méthode est appelée au moins une fois
  `verify(monMock,atLeastOnce()).retourneUnBooleen();`

### Vérification des appels avec nimporte quel paramètre

- On utilise pour cela des Matchers
  `verify(mockedList).someMethod(anyInt());`
- Si on utilise des Matchers, tous les arguments doivent être des Matchers
  ✖ `verify(mock).someMethod(anyInt(), anyString(), "un argument effectif");`

## Injection des Mocks et des Spy

### Injection des Mocks et des Spy

- L'annotation `InjectMock` permet l'injection des Mocks et des Spy
- L'injection des Mocks est utile lorsqu'on souhaite tester une classe qui dépend d'une autre où la dernière doit être mockée
- Mockito tentera d'injecter des Mocks uniquement par injection de constructeur, injection de setter ou injection de propriété
  - ☞ Si l'une des stratégies suivantes échoue, Mockito ne signalera pas d'échec ; c'est-à-dire que vous devrez fournir les dépendances vous-même



## Injection des Mocks et des Spy

### Exemple : Injection des Mocks et des Spy

```
public class Two {
        public void doSomething() {
                System.out.println("Un_autre_service_fonctionne...");
        }
}
public class One {
        private Two _two;
        public One( Two two ) {
                _two = two;
        }
        public void work() {
                System.out.println("Un_premier_service_fonctionne");
                _two.doSomething();
        }
}
//////////////////////////
public class OneTest {
    @Mock
    private Two _two;
    @InjectMocks
    private One _one;
    @Before
    public void setup() {
        MockitoAnnotations.initMocks(this);
    }
    @Test
    public void oneCanWork() throws Exception {
        one.work();
        Mockito.verify(_another).doSomething();
    }
}
```



## Injection des Mocks et des Spy

### Exemple : Injection des Mocks et des Spy

- On crée une doublure pour la classe `Two`, puis on l'injecte dans `One`

```
public final class OneTest {
  @Mock
  private Two _two;
  @InjectMocks
  private One _one;
  @Before
  public void setup() {
        MockitoAnnotations.initMocks(this);
  }
  @Test
  public void oneCanWork() throws Exception {
        _one.work();
        Mockito.verify(_one).doSomething();
  }
}
```



## Utilisation de Mockito : Connexion à une BD avec Mockito

### Exemple 2 : Objectif

- On se propose de simuler la connexion à une BD avec `Mockito`
  - ☞ On simule la création d'une connexion à une BD et l'exécution d'une requête

### Exemple 2 : Création du projet

- On crée un projet `Maven` en incluant les dépendances relatives à `Mockito` et les pilotes pour la connexion à une BD Mysql
  - ☞ Récupérer les méta-données relatives aux dépendances de `Mysql Connector`

Maven | Gradle | SBT | Ivy | Grape | Leiningen | Bu

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.21</version>
</dependency>
```

## Utilisation de Mockito :Connexion à une BD avec Mockito

### Exemple 2 : Code applicatif

- On crée la classe à tester `DatabaseService`
- Cette classe contient deux méthodes
  - La première méthode sera chargée de créer la session de base de données.
  - La deuxième méthode sera responsable de l'exécution de la requête.

```
package j2exemple2;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DatabaseService {
        private Connection dbConnection;
        public void getDbConnection() throws ClassNotFoundException, SQLException {
                dbConnection = DriverManager.getConnection("http://127.0.0.1:3306/mockito_db","
                        root","passwd");
        }
        public int executeQuery(String query) throws SQLException {
                return dbConnection.createStatement().executeUpdate(query);
        }

}
```





## Utilisation de Mockito :Connexion à une BD avec Mockito

### Exemple 2 : Classe de test

- On crée la classe de test `DatabaseServiceTest`
  - Aller au menu `File -> New -> JUnit Test Case`

```
package j2exemple2;
import static org.junit.jupiter.api.Assertions.*;
import java.sql.*;
import org.junit.jupiter.api.*;
import org.mockito.*;
class DatabaseServiceTest {
        @InjectMocks
        private DatabaseService dbConnection;
        @Mock
        private Connection mockConnection;
        @Mock
        private Statement mockStatement;
        @BeforeEach
        public void setUp() {
                MockitoAnnotations.initMocks(this);
        }
        @Test
        void test() throws Exception {
                Mockito.when(mockConnection.createStatement()).thenReturn(mockStatement);
                Mockito.when(mockConnection.createStatement().executeUpdate(Mockito.anyString())
                        ).thenReturn(1);
                int value=dbConnection.executeQuery("");
                assertEquals(1, value);
        }
}
```





## Injection des mocks

### Exercice

- On sintéresse à un jeu de hasard et dargent
- Le joueur possède une somme dargent qui lui permet de jouer à un jeu (la somme étant physiquement étalée devant lui, on suppose que le jeu na aucun contrôle à effectuer sur le montant de la mise).
- Le jeu qui nous intéresse se joue avec 2 dés et une banque qui gère les pertes et les gains. Les dés sont du modèle classique à tirage aléatoire entre 1 et 6.
- La banque est censée être toujours solvable. Néanmoins, il arrive que le casino soit débordé par un joueur chanceux et narrive plus à suivre : la banque saute. Le gain est quand même donné au joueur, mais le jeu ferme immédiatement.
- La règle du jeu est la suivante. On ne peut jouer quà un jeu ouvert. Le joueur qui joue fait un pari en misant une somme. Il 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 deux fois la mise, somme créditée au joueur. Si le pari a fait sauter la banque, le jeu ferme immédiatement. Si la somme des lancers ne vaut pas 7, le joueur a perdu sa mise





## Injection des mocks

### Exercice

- Le sujet consiste à tester en isolation la méthode `public void jouer() throws ...` de la classe qui représente le jeu. On ne demande pas décrire ni de tester les autres classes : limitez-vous à des interfaces et utilisez des doublures
- Exemples de scénarios à tester
  - Le plus simple : la banque est fermée
  - Le joueur perd : vérifier notamment que le joueur a été débité de sa mise, laquelle a été créditée à la banque, et que le jeu reste ouvert.

## Plan





## Organisation des tests

### Organisation des tests

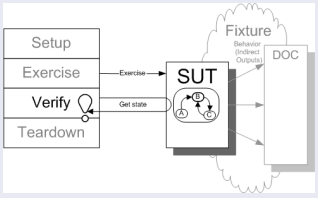
- Les tests peuvent être organisés de différentes façons
  1. Classe de test par classe : on met toutes les méthodes de test pour une classe de système sous test (SUT) sur une seule Classe de test
  2. Classe de test par fonctionnalité : on regroupe les méthodes de test sur les classes de test en fonction de la fonction testable de le SUT qu'ils exercent.
  3. Classe de test par fixature : on organise les méthodes de test dans des classes de test basées sur la similitude du test fixation.



## Principales stratégies pour vérifier les résultats dun test

### Vérification de l'état

- On détermine si la méthode exercée a fonctionné correctement en examinant l'état du SUT et de ses collaborateurs après la a été exercée
  - L'état d'un objet est défini par les valeurs de ses attributs
- Approche de test classique






## Principales stratégies pour vérifier les résultats dun test

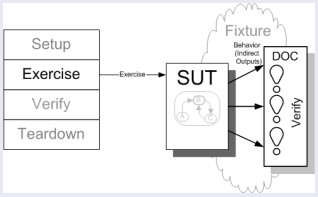
### Vérification de l'état : principe de fonctionnement

1. Initialisation : Instancier le SUT et ses collaborateurs
2. L'exécution : l'étape où la méthode à vérifier est exécutée
3. La vérification : l'exécution **des assertions** qui permettent de vérifier l'**état** du SUT et de ses collaborateurs.
4. Le nettoyage (teardown en anglais) : l'étape où les données de test sont réinitialisées, supprimées ou autre.

# Principales stratégies pour vérifier les résultats d'un test

## Vérification comportementale

- Elle consiste à vérifier le comportement du SUT et de sa collaboration avec les autres objets.
  - Elle garantit que le SUT se comporte vraiment comme spécifié plutôt que juste se retrouver dans le bon état post-exercice
- Elle est basée sur l'utilisation des Mocks






# Principales stratégies pour vérifier les résultats dun test

## Vérification comportementale : principe de fonctionnement

1. Initialisation
   - Instancier le SUT et les **mocks** à partir des classes interfaces qui représentent les collaborateurs
   - Intialiser les attentes (les "Expectations") : on écrit le comportement attendu par l'objet mocké
2. L'exécution et vérification : la méthode à vérifier est exécutée ce qui induit que les appels des méthodes des Mocks sont surveillés par Mockito
4. Le nettoyage (teardown en anglais) : l'étape où les données de test sont réinitialisées, supprimées ou autre.



# Exemple : vérification de l'état vs comportementale

## Exemple : énoncé

Il y a deux objets : un entrepôt (warehouse) et une commande (order). L'entrepôt contient différents types de produits dans des quantités limitées. Une commande concerne un produit pour une quantité donnée. S'il y'a suffisamment de stock dans l'entrepôt, la commande est **passée** ; sinon, **rien ne se passe**.

## Principe

- L'objet à tester (SUT) : une commande
  - Vérifier que la commande change d'état lorsqu'il y a suffisamment de stock dans l'entrepôt
- L'objet collaborateur : un entrepôt



# Principales stratégies pour vérifier les résultats dun test

## Exemple : Méthodes de la classe de tests

```
private static String GUINESS = "Guiness";
private static String CHIMAY = "Chimay";
private Warehouse warehouse;

// Initialization, processed before each test thanks to the Before annotation
@BeforeEach
protected void setUp() throws Exception {
  // The fixtures or, collaborators
  warehouse = new Warehouse();
  warehouse.add(GUINESS, 50);
  warehouse.add(CHIMAY, 25);
}

// Teardown, precessed after each test thanks to the After annotation
@AfterEach
protected void tearDown() throws Exception {
  warehouse = null;
}
```

# Vérification de l'état

## Vérifier l'état d'une commande

- Vérifier qu'une commande a été effectuée lorsque le stock contient suffisamment de produits

```
public void testOrderIsFilledIfEnoughInWarehouse() {
  // The System Under Testing
  Order order = new Order(GUINESS, 50);
  // Exercise
  order.fill(warehouse);

  // Check
  assertTrue(order.isFilled());
  assertEquals(0, warehouse.getInventory(GUINESS));
}
```

## Vérifier l'état d'une commande

- Vérifier qu'une commande n'était pas effectuée lorsque le stock ne contient pas suffisamment de produits

```
public void testOrderDoesNotRemoveIfNotEnough() {
  Order order = new Order(GUINESS, 51);
  order.fill(warehouse);
  assertFalse(order.isFilled());
  assertEquals(50, warehouse.getInventory(GUINESS));
}
```





# Vérification comportementale

## Principe

- L'objet à tester (SUT) : une commande
- L'objet collaborateur : un entrepôt
  - ☞ Cet objet sera mocké et surveillé





# Vérification comportementale

## Vérifier une commande

- Vérifier qu'une commande a été effectuée lorsque le stock contient suffisamment de produits

```
public void testFillingRemovesInventoryIfInStock() {
  // Setup
  Order order = new Order(GUINESS,50);
  Warehouse warehouseMock=mock(Warehouse.class);

  // stubs
  when(warehouseMock.hasInventory(GUINESS, 50)).thenReturn(true);

  // exercise
  order.fill(warehouseMock);

  // verify
  InOrder inOrder = inOrder(warehouseMock);
  inOrder.verify(warehouseMock).hasInventory(GUINESS,50);
  inOrder.verify(warehouseMock).remove(GUINESS,50);
  assertTrue(order.isFilled());
}
```





# Vérification comportementale

## Vérifier une commande

- Vérifier qu'une commande n'était pas effectuée lorsque le stock ne contient pas suffisamment de produits

```
public void testFillingDoesNotRemoveIfNotEnoughInStock() {
  Order order = new Order(GUINESS, 51);
  Warehouse warehouseMock = mock(Warehouse.class);

  // stubs
  when(warehouseMock.hasInventory(GUINESS, 51)).thenReturn(false);

  // exercise
  order.fill(warehouseMock);

  // verify
  verify(warehouseMock).hasInventory(GUINESS, 51);
  verify(warehouseMock,never()).remove(anyString(),anyInt());
}
```

## MERCI POUR VOTRE ATTENTION

Questions ?