

# Mocks objets

## Simuler pour mieux tester

Clémentine Nebut

Faculté Des Sciences Université de Montpellier

2021

# Sommaire

Pourquoi simuler ?

Comment simuler ?

Mockito

Pallier les limites de Mockito : PowerMock

# Tester en présence de dépendance au temps

Test

```
...  
  
@Test  
public void testm(){  
    ...  
    boolean b=sut.m() ;  
    assertEquals( ???,b) ;  
    ...  
}
```



SUT

```
...  
  
public boolean m(){  
    ...  
    ...  
}
```

Si c'est l'heure,  
Retourne 0 ;  
Sinon :  
Faire plein de trucs  
Retourne truc compliqué

# Tester en présence d'aléatoire

Test

```
...  
@Test  
public void testm(){  
    ...  
    int i=sut.m() ;  
    assertEquals( ???,i) ;  
    ...  
}
```



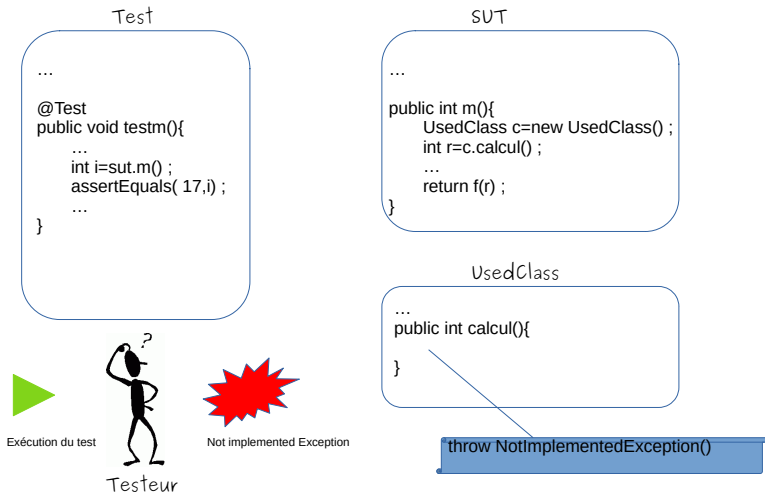
Testeur

SUT

```
...  
public int m(){  
    ...  
    ...  
}
```

alea=random()  
Si alea <12  
    k=calcul1  
Sinon  
    k=calcul2  
Retourner k

# Tester en présence de dépendance à du code non développé



# Tester en présence de dépendance à du code très lent

Test

```
...  
  
@Test  
public void testm(){  
    ...  
    int i=sut.m() ;  
    assertEquals( 17,i) ;  
    ...  
}
```

SUT

```
...  
  
public int m(){  
    UsedClass c=new UsedClass() ;  
    int r=c.calcul() ;  
    ...  
    return f(r) ;  
}
```

UsedClass

```
...  
public int calcul(){  
    ...  
}
```



Exécution du test



Testeur

Trèèèè loooong calcul

# Pourquoi simuler ?

- ▶ L'environnement du SUT est complexe ou coûteux à mettre en place (environnement matériel, base de données, ...).
- ▶ Mise en place de situations exceptionnelles difficiles à déclencher (out of memory, ...).
- ▶ L'environnement du SUT n'est pas encore disponible ou fiabilisé.
- ▶ Le SUT appelle du code lent.
- ▶ Le SUT fait appel à des méthodes non déterministes (fonction de l'heure, de nombres générés aléatoirement, ...)
- ▶ on simule alors tout ou partie de l'environnement qui pose problème.

# Sommaire

Pourquoi simuler ?

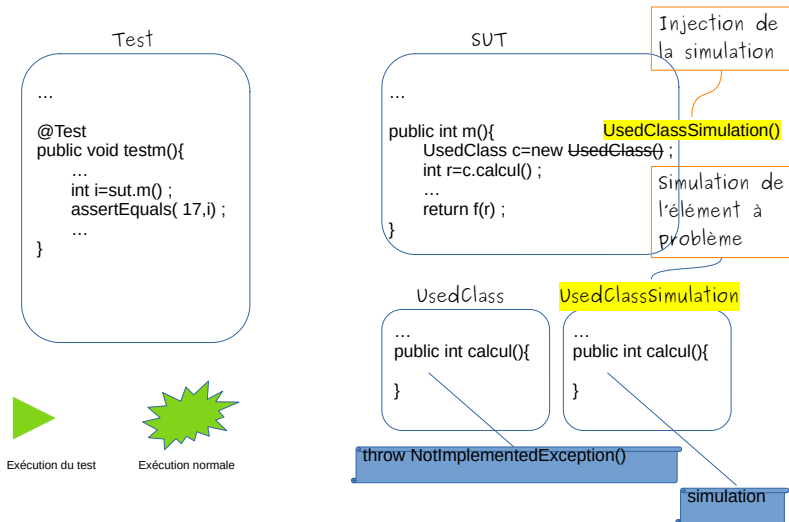
Comment simuler ?

Mockito

Pallier les limites de Mockito : PowerMock



# Simuler



# Simuler

- ▶ Écrire "à la main" les simulateurs est facile mais long  
→ Utiliser des outils
- ▶ Injecter les simulateurs
  - ▶ Remplacer le vrai élément par le simulateur dans le code sous test
  - ▶ MAIS sans modifier le code sous test
  - ▶ Utiliser de l'injection de dépendance
  - ▶ Prévoir l'injection de dépendance lors de l'écriture du code !  
(testabilité)

# Qu'est-ce qu'une doublure de test ?

- ▶ La définition d'objets factices se substituant lors du test aux objets réels
- ▶ On a donc des objets factices qui remplacent des objets réels pour faciliter le test.
- ▶ Ces objets peuvent être écrits à la main (leur classe) ou générés.
- ▶ Les objets factices doivent partager la même interface que les vrais objets, pour pouvoir s'y substituer.

# Types de doublures

- ▶ Dummy
- ▶ Stub
- ▶ Fake
- ▶ Spy
- ▶ Mock

Pas de définition consensuelle ... Dans la suite, définitions principalement inspirées de celles de Martin Fowler.

# Dummy

- ▶ Objets vides qui n'ont pas de fonctionnalités implémentées
- ▶ Les dummies sont “transmis” mais jamais réellement utilisés.
- ▶ En général ils sont utilisés pour remplir des listes de paramètres.

# Fake

- ▶ Le fake implémente de manière simpliste le comportement attendu d'une classe.
- ▶ Le fake est générique : il n'est pas spécifique à un test.
- ▶ Le fake met en place des raccourcis qui le rendent inutilisable en production

# Spy

- ▶ Un spy est une doublure capable de vérifier l'utilisation qui en est faite.
- ▶ Par exemple : appel au moins une fois de telle méthode avec tel paramètre.

# Bouchons de test (stubs)

- ▶ Un bouchon de test est une classe utilisée pour en simuler une autre.
- ▶ Il fournit des réponses pré-définies aux appels réalisés lors du test.
- ▶ Le bouchon de test est écrit grâce à la connaissance de la classe à simuler (boîte blanche).



# Mocks

- ▶ Les mocks sont des objets pré-programmés avec des pré-suppositions qui forment la spécification des appels qu'ils sont censés recevoir.
- ▶ Le testeur configure le mock de manière à lui donner le comportement souhaité.
- ▶ Le code du test met en place le mock, le configure, puis l'utilise pour paramétrer le SUT.
- ▶ Le mock permet de réaliser une vérification comportementale. Par exemple que telle méthode a bien été appelée.

# Mocks, principe général d'utilisation

Test

```
...  
  
@Test  
public void testm(){  
    // 1.mise en place d'un mock de  
    UsedClass  
    // 2.paramétrage du mock pour la  
    méthode calcul (simulation de calcul)  
    // 3.injection du mock dans le SUT  
    // puis test « normal »  
    int i=sut.m() ;  
    assertEquals( 17,i) ;  
    ...  
}
```

SUT

```
private UsedClass c ;  
  
public SUT(UsedClass c){this.c=c;}  
  
public int m(){  
    int r=c.calcul() ;  
    ...  
    return f(r) ;  
}
```

UsedClass

```
...  
public int calcul(){  
  
}
```

throw NotImplementedException()

# Sommaire

Pourquoi simuler ?

Comment simuler ?

Mockito

Pallier les limites de Mockito : PowerMock

# Mockito

- ▶ Outil permettant de générer des mocks
- ▶ Le testeur peut facilement donner au mock le comportement recherché (paramétrage du mock).
- ▶ Le testeur peut facilement faire une vérification comportementale après exécution (le côté espion du mock)



# Principe général : création d'un mock

## Quand et pourquoi ?

- ▶ Quand on veut éviter (pour des raisons vues précédemment) de faire appel au vrai environnement du SUT
- ▶ Par exemple quand certains éléments de l'environnement dépendent du temps, ou d'un alea

# Principe général : création d'un mock

## Comment on fait ?

- ▶ Dans le test, au lieu d'utiliser une instance d'une classe "réelle" de l'environnement (et qu'on souhaite remplacer), on demande à Mockito d'utiliser une doublure.
- ▶ On paramètre la doublure pour qu'elle ait un comportement permettant le test.
- ▶ On écrit le test, qui utilisera donc la doublure.

# Principe général : création d'un mock

- ▶ Création d'un mock : utilisation de la méthode statique **mock** ou de l'annotation **@Mock**
- ▶ Nécessite la classe à mocker ou son interface
- ▶ Mockito, crée moi une instance mock pour cette classe ou cette interface

---

```
1  import org.mockito.Mock;
2  import static org.mockito.Mockito.*;
3  ...
4  C mock1 =mock(C.class); // C est une classe ou une interface
5  C mock2=mock(C.class , "nom");
6  @Mock C mock3;
7  @Mock(name="nom2") C mock4;
```

---

# Remarques basement techniques sur la création des mocks avec JUnit 5

- ▶ Pour utiliser les annotations mockito : ajouter `@ExtendWith(MockitoExtension.class)` sur la classe de test
- ▶ On ne peut pas utiliser l'annotation `@Mock` sur une variable locale (d'une méthode de test), seulement sur un attribut (normal, c'est une annotation)



# Mais que fait un mock fraîchement créé ?

Par défaut pas grand chose ...

Ses méthodes retournent quand on les appelle :

- ▶ pour les numériques : 0
- ▶ pour les booléens : false
- ▶ pour les collections : collections vides
- ▶ pour les objets quelconques : null

D'où la nécessité de paramétrer le mock

- ▶ pour qu'il réponde des choses utiles pour le test

# Principe général : paramétrage d'un mock

- ▶ On décrit le comportement du mock avec la méthode `when`
- ▶ On exprime quelque chose du genre : Mockito, quand (when !) le mock recevra tel appel, alors il faut répondre ceci
- ▶ On peut faire des vérifications comportementales avec la méthode `verify`
- ▶ On vérifie quelque chose du genre : Mockito, est-ce que telle méthode a bien été appelée au moins une fois avec tel paramètre ?
- ▶ On peut spécifier des comportements à vérifier un peu complexes grâce à des `matchers`

# Spécification du comportement du mock : méthodes avec retour

## Cas d'une méthode avec retour ; valeur unique

```
1  interface I { int m(); } // the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  when(myMock.m()).thenReturn(42);
```

► Mockito, quand myMock recevra un appel à la méthode m, retourne 42.

# Spécification du comportement du mock : méthodes avec retour

## Cas d'une méthode avec retour ; valeurs successives

```
1  interface I{ int m();} / the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  when(myMock.m()).thenReturn(42, 43, 44);
```

► Mockito, quand myMock recevra un appel à la méthode m, retourne 42 au premier appel, puis 43 au deuxième appel, puis 44 au 3ème appel.

# Spécification du comportement du mock, méthode avec paramètres

## Spécifier le comportement selon la valeur reçue en paramètre

---

```
1  interface I{ int m(int i);} // the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  when(myMock.m(1)).thenReturn(42);
6  when(myMock.m(42)).thenReturn(1);
```

---

► Mockito, quand myMock recevra l'appel à la méthode m avec pour paramètre 1, retourne 42, et avec comme paramètre 42, retourne 1.

# Spéc. du comportement du mock pour lever des exceptions

## Cas d'une méthode avec retour avec levée d'exception

---

```
1  interface I{ int m() throws E;} // the interface to mock
2  ...
3  @Mock I myMock;
4  ...
5  when(myMock.m()) .thenThrow(new E());
```

---

► Mockito, quand myMock reçoit un appel à m, jette une exception de type E

## Cas d'une méthode sans retour avec levée d'exception

---

```
1  interface I{ void m(int i) throws E;} // the interface to mock
2  ...
3  @Mock I myMock; // the mock
4  ...
5  doThrow(New E()) .when(myMock).m(1);
```

---

► Mockito, quand myMock reçoit un appel à m avec pour paramètre 1, jette une exception de type E

# Spécification du comportement du mock, exemple de combinaison

## Cas d'une méthode avec paramètres; combinaison de then

---

```
1  interface I { int m(int i) throws E; }
2  ...
3  @Mock I myMock;
4  ...
5  when(myMock.m(1)).thenReturn(42).thenThrow(new E());
6  when(myMock.m(42)).thenReturn(1).thenReturn(99);
```

---

► Mockito, quand myMock recevra un appel à m avec comme param 1, jette une exception de type E, et quand tu reçois un appel à m avec pour param 42, retourne 99.

# Spécification du comportement du mock, mock de méthodes statiques

## Cas d'une méthode sans paramètres

---

```
1  class A{ static int m() {return "blabla";}}
2  ...
3
4  ...
5  try (MockedStatic<A> utilities = Mockito.mockStatic(A.class)) {
6      utilities.when(A::m).thenReturn("Hop");
7      ...
8  }
```

---

► Mockito, quand myMock recevra un appel à la méthode statique m , retourne "Hop".

- try avec ressource
- nécessite l'utilisation de mockito-inline



# Spécification du comportement du mock, mock de méthodes statiques

## Cas d'une méthode avec paramètres

```
1    class A{ static int m(int i) {return "blabla"+i;}}
2    ...
3
4    ...
5    try (MockedStatic<A> utilities = Mockito.mockStatic(A.class)) {
6        utilities.when(() -> A.m(1)).thenReturn("Hop hop");
7        // ici A.m(1) vaut "Hop hop"
8    }
9    // là A.m(1) vaut "blabla1"
```

► Mockito, quand myMock recevra un appel à la méthode statique m avec pour paramètre 1, retourne "Hop hop".

- try avec ressource
- nécessite l'utilisation de mockito-inline

# Vérification du comportement : verify

## Vérifier qu'une méthode est appelée 3 fois

```
1 verify(mock1, times(3)).m();
```

## Vérifier qu'une méthode est appelée au moins/au plus 3 fois

```
1 verify(mock1, atLeastOnce()).m();  
2 verify(mock1, atMost(3)).m();
```

## Vérifier qu'une méthode n'est jamais appelée

```
1 verify(mock1, never()).m();
```

# Vérification du comportement : verify

## Vérifier l'ordre d'appel

---

```
1 InOrder ordre =inOrder(mock1, mock2);  
2 ordre.verify(mock1).m();  
3 ordre.verify(mock2).m();
```

---

# Mock et spy

- ▶ On veut espionner un objet instance d'une classe réelle (pas un mock) ...
- ▶ ... et éventuellement en altérer le comportement
- ▶ Par exemple pour juste mocker une méthode (remplacer le comportement réel par un autre, et garder tous les autres comportements réels)
- ▶ On utilise alors un spy, dont le comportement par défaut est celui de la classe de l'objet espionné

## Spy

---

```
1 @Spy LinkedList<String> spy;  
2 ...  
3 spy.add('ajout1');  
4 spy.add('ajout2');  
5 verify(spy).add("ajout1");  
6 when(spy.isEmpty()).thenReturn(false);  
7 assertFalse(spy.isEmpty());
```

---

# Limites et remarques

## Limites

- ▶ On ne peut pas mocker les méthodes privées, ni les méthodes final
- ▶ On ne peut pas mocker les méthodes equals et hashCode (utilisées en interne par Mockito)
- ▶ On ne peut pas mocker les classes final (avant la version 3.4) ou anonymes

## Remarques

- ▶ Un comportement non utilisé du mock ne provoque pas d'erreur
- ▶ Verify : si la vérification échoue, le test échoue.

# Argument matchers

- ▶ Permettent une spécification de paramètres flexible dans les when ou les verify
- ▶ Ne plus utiliser Matchers (deprecated pour cause de conflit de nommage avec hamcrest)
- ▶ <https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/ArgumentMatchers.html>

---

```
1 public interface I { public int m(int i); }
2 @Mock I mock;
3 ...
4 when(mock.m(anyInt())).thenReturn(42);
5 assertEquals(mock.m(12), 42);
```

---

- ▶ autres argument matchers : eq, contains, ...
- ▶ limitation : si on utilise des argument matchers, tous les arguments doivent être des argument matchers.

# Sommaire

Pourquoi simuler ?

Comment simuler ?

Mockito

Pallier les limites de Mockito : PowerMock

# PowerMock

- ▶ Framework de test permettant de contourner les limitations classiques des frameworks de mock comme Mockito.
- ▶ Avec PowerMock, on peut :
  - ▶ mocker des méthodes privées, statiques, et/ou final
  - ▶ mocker des classes final
  - ▶ ...
- ▶ “Please note that PowerMock is mainly intended for people with expert knowledge in unit testing. Putting it in the hands of junior developers may cause more harm than good.”



# Préparation de la classe de test et de la classe mockée

---

```
1 package tpmocks;
2 import org.junit.Assert;
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.mockito.Mockito;
6 import org.powermock.api.mockito.PowerMockito;
7 import org.powermock.core.classloader.annotations.PrepareForTest;
8 import org.powermock.modules.junit4.PowerMockRunner;
9
10 @RunWith(PowerMockRunner.class)
11 @PrepareForTest({A.class})
12 public class MesTestsPowerMockSurA {...}
```

---

# Mock d'une méthode finale

---

```
1  public class A{ final String jeSuisFinal(){return "AF";} ...}
2  ...
3  @Test
4  public void mockFinalTest() {
5      A mock = PowerMockito.mock(A.class);
6      final String resultMock = "valeur_mockee";
7      Mockito.when(mock.jeSuisFinal()).thenReturn(resultMock);
8      // on verifie que la methode final a bien ete mockee
9      Assert.assertEquals(mock.jeSuisFinal(), resultMock);
```

---

# Mock d'une méthode statique

---

```
1    public class A{ static String jeSuisStatique(){return "AS";} ...}  
2    ...  
3    @Test  
4    public void mockStaticTest() {  
5        PowerMockito.mockStatic(A.class);  
6        String resultMock = "valeur_mockee";  
7        Mockito.when(A.jeSuisStatique()).thenReturn(resultMock);  
8        // on verifie que la methode statique a bien ete mockee  
9        Assert.assertEquals(A.jeSuisStatique(), resultMock);  
10    }
```

---

# Injecter les mocks

**Injecter le mock** : faire en sorte que l'instance sous test utilise le mock plutôt que le vrai objet

Test

```
...  
  
@Test  
public void testm(){  
    // 1.mise en place d'un mock de UsedClass  
    UsedClass mock=mock(UsedClass.class)  
    // 2.paramétrage du mock  
    when(mock.calcul()).thenReturn(42);  
    // 3.injection du mock dans le SUT  
    sut=new SUT(mock);  
    // puis test « normal »  
    int i=sut.m();  
    assertEquals( 17,i);  
    ...  
}
```

SUT

```
private UsedClass c ;  
  
public SUT(UsedClass c){this.c=c;}  
  
public int m(){  
    int r=c.calcul();  
    ...  
    return f(r);  
}
```

UsedClass

```
...  
public int calcul(){  
    ...  
}
```

throw NotImplementedException()

# Injecter les mocks

- ▶ Se ramène au classique problème d'injection de dépendance (Dependency Injection)
- ▶ On veut que le SUT utilise un mock plutôt qu'un vrai objet de type A ...
- ▶ ... sachant que le mock "est un" A du point de vue du typage ...
- ▶ Options possibles de manière générale :
  - ▶ Injection par constructeur : l'objet que l'on veut simuler est reçu en paramètre du constructeur du SUT
  - ▶ Injection par accesseur/mutateur : l'objet que l'on veut simuler est injecté dans le SUT via l'appel à un accesseur en écriture (mutateur)
- ▶ Option possible en Mockito (dans certains cas) : l'annotation `@InjectMocks`

# Injecter les mocks avec @InjectMocks, exemple – SUT

---

```
1 import java.util.Random;
2
3 public class ClassUtilisantRandom {
4     private Random r;
5
6     public ClassUtilisantRandom() {
7         r=new Random(System.currentTimeMillis());
8     }
9
10    public int calculAvecAlea() {
11        int randomInt=r.nextInt(20);
12        int result;
13        // un calcul qui dépend de randomInt
14        if (randomInt==10) {
15            result=0;
16        } else {
17            result=randomInt%2;
18        }
19        return result;
20    }
21 }
```

---

# Injecter les mocks avec @InjectMocks, exemple – Test

```
1 import static org.junit.jupiter.api.Assertions.assertEquals;
2 import static org.mockito.Mockito.when;
3 import java.util.Random;
4 import org.junit.jupiter.api.Test;
5 import org.junit.jupiter.api.extension.ExtendWith;
6 import org.mockito.InjectMocks;
7 import org.mockito.Mock;
8 import org.mockito.junit.jupiter.MockitoExtension;
9
10 @ExtendWith(MockitoExtension.class)
11 public class TestInjectionMock {
12
13     @Mock
14     Random mockrandom;
15
16     @InjectMocks
17     ClasseUtilisantRandom sut=new ClasseUtilisantRandom();
18
19     @Test
20     void testCalculRetourne0QuandAleaVaut10() {
21         when(mockrandom.nextInt(20)).thenReturn(10);
22         assertEquals(0, sut.calculAvecAlea());
23     }
24
25     @Test
26     void testCalculRetourne0QuandAleaEstPairEtDifferentDe10() {
27         when(mockrandom.nextInt(20)).thenReturn(12);
28         assertEquals(0, sut.calculAvecAlea());
29     }
30
31     @Test
32     void testCalculRetourne1QuandAleaEstPair() {
33         when(mockrandom.nextInt(20)).thenReturn(13);
34         assertEquals(1, sut.calculAvecAlea());
35     }
36 }
```

# Sommaire

Pourquoi simuler ?

Comment simuler ?

Mockito

Pallier les limites de Mockito : PowerMock



# Conclusion

- ▶ Les mocks permettent de simuler certains comportements, afin d'en tester d'autres
- ▶ Situations où les mocks sont utiles :
  - ▶ présence d'aléatoire ou dépendance au temps
  - ▶ accès à du code lent
  - ▶ accès à des bases de données
  - ▶ accès à du code non encore développé
  - ▶ test en isolation

# Conclusion

- ▶ Des outils comme Mockito permettent de créer facilement des simulateurs spécifiques à un test : les mocks
- ▶ Ces mocks doivent être (en général) injectés dans le système sous test
- ▶ Si l'injection n'est pas possible, le système est alors difficilement testable.