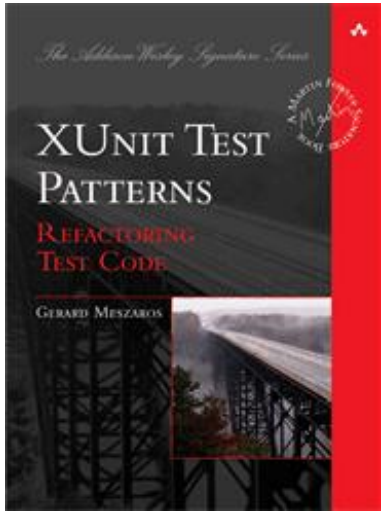
Doublures de test

(*dummy, stub, mock, spy, fake*)

Isabelle BLASQUEZ
@iblasquez

Doublure de test (Test Double) : Définition

Doublure de Test : Tout objet qui remplace un objet réel lors d'un test
(Meszaros, 2007)



Une doublure permet de reproduire l'état et/ou le comportement du composant dont dépend le code à tester.

Ces objets peuvent être **écrits à la main** (leur classe)
ou **générés** (à l'aide d'un outil comme Mockito en Java par exemple)

Version de xUnit Test Patterns en ligne (section Test Double) accessible via : http://xunitpatterns.com/Test%20Double.html

Mais, par abus de langage, une doublure de test est parfois appelée …

Stub (bouchon) : Terme générique utilisé dans la littérature pour identifier une implémentation squelettique ou spéciale d'un composant logiciel déployée pour développer ou tester un composant qui l'utilise ou qui en est dépendant **(IEEE, 1990**

… car une doublure de tests permet de bouchonner une dépendance …

Ou…

Mock : parfois utilisé comme terme générique en raison de l'article historique sur le sujet : **Mock Roles, not Objects** (http://www.jmock.org/oopsla2004.pdf)

Mock Roles, not Objects

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes
ThoughtWorks UK
Berkshire House, 168-173 High Holborn
London WC1V 7AA
{sfreeman, npryce, tmackinnon, jwalnes} @thoughtworks.com

ABSTRACT
Mock Objects is an extension to Test-Driven Development that supports good Object-Oriented design by guiding the discovery of a coherent system of types within a code base. It turns out to be less interesting as a technique for isolating tests from third-party libraries than is widely thought. This paper describes the process of using Mock Objects with an extended example and reports best and worst practices gained from experience of applying the process. It also introduces jMock, a Java framework that embodies our collective experience.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques, Object-Oriented design methods

General Terms
Design, Verification.

Keywords
Test-Driven Development, Mock Objects, Java..

1. INTRODUCTION
Mock Objects is misnamed. It is really a technique for identifying types in a system based on the roles that objects play.

In [10] we introduced the concept of *Mock Objects* as a technique to support Test-Driven Development. We stated that it encouraged expressed using Mock Objects, and shows a worked example. Then we discuss our experiences of developing with Mock Objects and describe how we applied these to jMock, our Mock Object framework.

1.1 Test-Driven Development
In Test-Driven Development (TDD), programmers write tests, called *Programmer Tests*, for a unit of code before they write the code itself [1]. Writing tests is a *design* activity, it specifies each requirement in the form of an executable example that can be shown to work. As the code base grows, the programmers refactor it [4], improving its design by removing duplication and clarifying its intent. These refactorings can be made with confidence because the test-first approach, by definition, guarantees a very high degree of test coverage to catch mistakes.

This changes design from a process of *invention*, where the developer thinks hard about what a unit of code should do and then implements it, to a process of *discovery*, where the developer adds small increments of functionality and then extracts structure from the working code.

Using TDD has many benefits but the most relevant is that it directs the programmer to think about the design of code from its intended use, rather than from its implementation. TDD also tends to produce simpler code because it focuses on immediate requirements rather than future-proofing and because the emphasis on refactoring allows developers to fix design weaknesses as their understanding of the domain improves.

Et pourtant Mock Aren’t Stub …

Mocks Aren't Stubs

The term 'Mock Objects' has become a popular one to describe special case objects that mimic real objects for testing. Most language environments now have frameworks that make it easy to create mock objects. What's often not realized, however, is that mock objects are but one form of special case test object, one that enables a different style of testing. In this article I'll explain how mock objects work, how they encourage testing based on behavior verification, and how the community around them uses them to develop a different style of testing.

02 January 2007

Martin Fowler

Translations: French · Italian · Spanish · Portuguese · Korean

Find **similar articles** to this by looking at these tags: popular · testing

Contents

Regular Tests
Tests with Mock Objects
  Using EasyMock
The Difference Between Mocks and Stubs
Classical and Mockist Testing
Choosing Between the Differences
  Driving TDD
  Fixture Setup
  Test Isolation
  Coupling Tests to Implementations
  Design Style
So should I be a classicist or a mockist?
Final Thoughts

I first came across the term "mock object" a few years ago in the Extreme Programming (XP) community. Since then I've run into mock objects more and more. Partly this is because many of the leading developers of mock objects have been colleagues of mine at ThoughtWorks at various times. Partly it's because I see them more and more in the XP-influenced testing literature.

Un des articles de références …

https://martinfowler.com/articles/mocksArentStubs.html

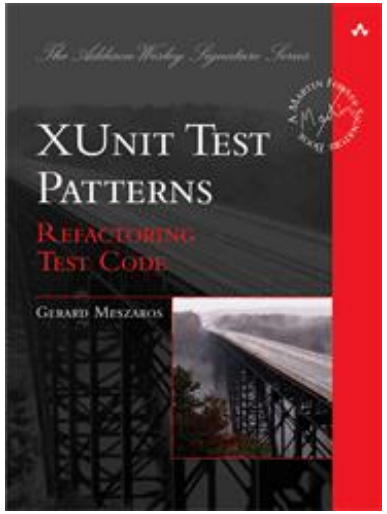
Différents styles de doublures pout mettre en place
Différents styles de tests

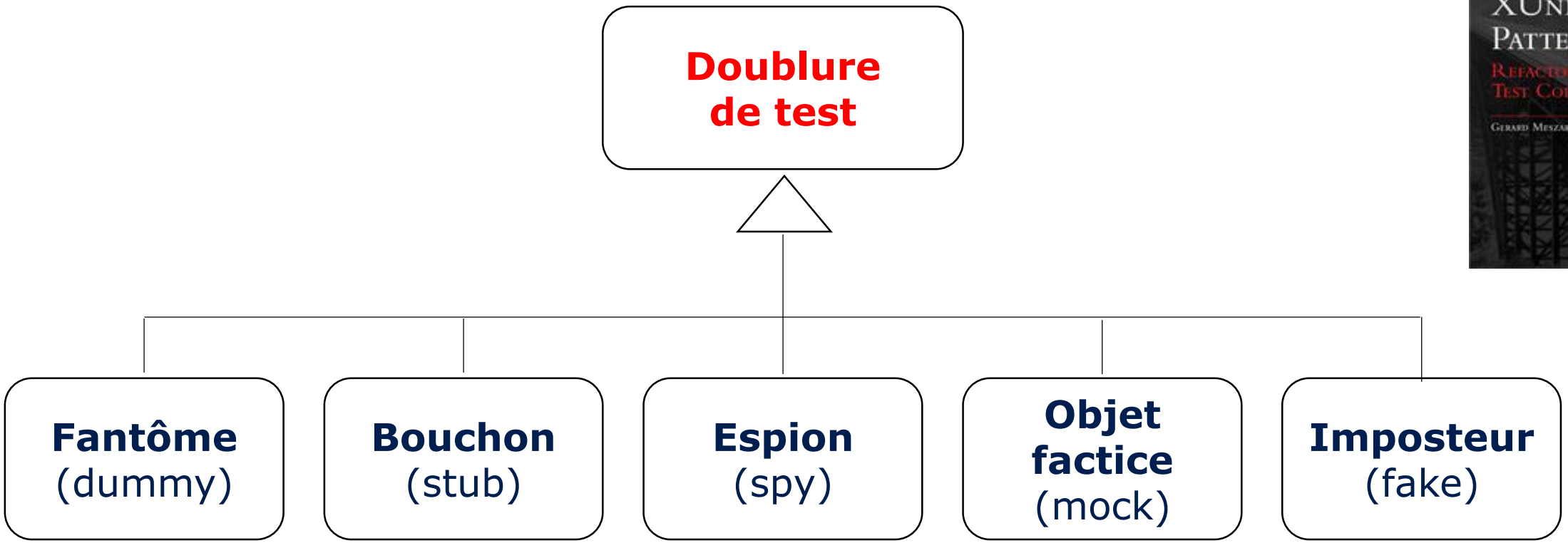
A lire !

Version française :
http://bruno-orsier.developpez.com/mocks-arent-stubs/

Classification des doublures de test

En effet, selon Meszaros un mock et un stub sont des types particuliers de ***doublure de tests*** *ayant des usages différents …*





Une doublure,

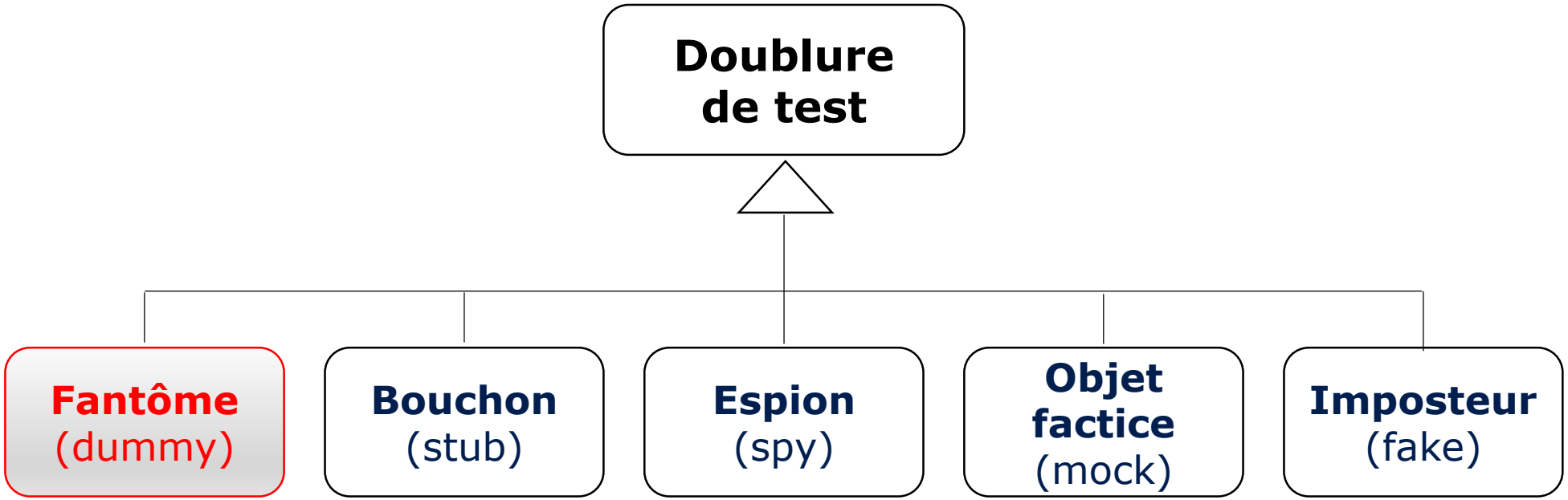
Oui, mais pour quel usage ?

*Dans cette partie, les doublures seront **écrites à la main**…*

… et la présentation des différentes doublures s'inspire de :
https://8thlight.com/blog/uncle-bob/2014/05/14/TheLittleMocker.html

Qu'est-ce qu'un Dummy ?

(Fantôme)



Dummy : Définition …

Une interface métier…

```
interface Authorizer {
  public Boolean authorize(String username, String password);
}
```

Un Dummy (fantôme)

Un objet qui simule l'implémentation d'un objet de type `Authorizer` *et ne fait rien …*

```
public class DummyAuthorizer implements Authorizer {
  public Boolean authorize(String username, String password) {
    return null;
  }
}
```

→ Doublure de test la plus simple

→ Objet « vide », ne contient ***aucune implémentation***

Dummy : Utilisation (*Que faire avec un Fantôme ?*)

Les dummies sont des objets qui ne font rien (d'où le `null`)
puisqu'ils sont simplement utilisés comme paramètres d'appel à une méthode
(***transmis*** mais ***jamais réellement utilisés***).
En général ils sont utilisés pour remplir des listes de paramètres.

Exemple :

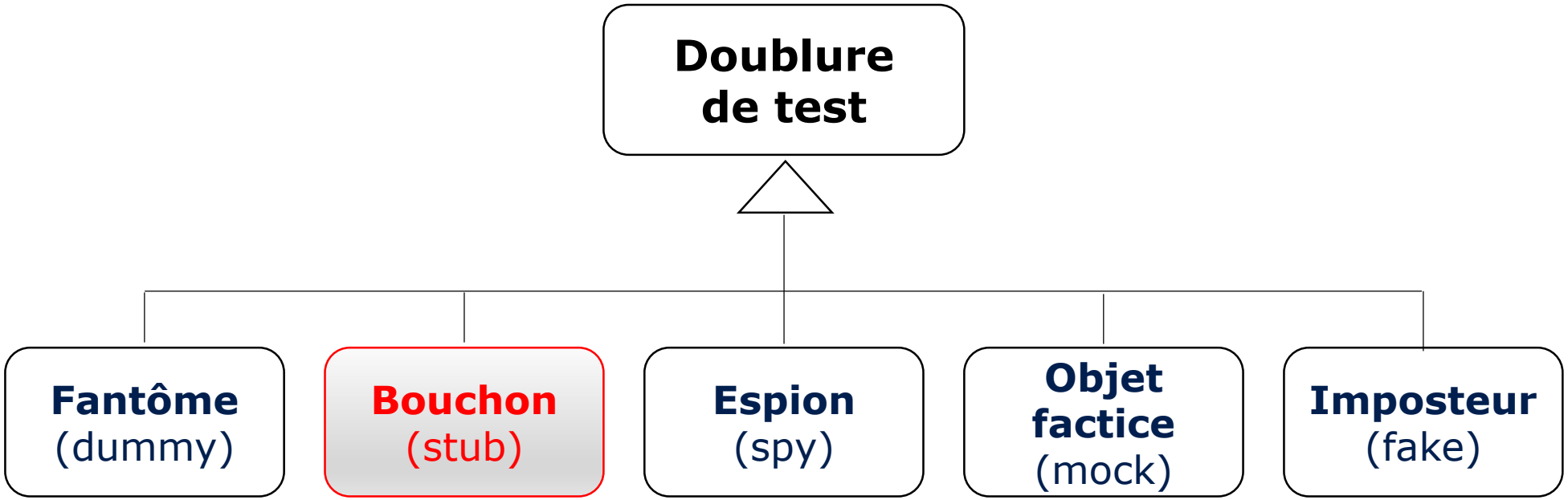
```
public class System {
    public System(Authorizer authorizer) {
        this.authorizer = authorizer;
    }

    public int loginCount() {
        //returns number of logged in user
    }
}

@Test
public void newlyCreatedSystem_hasNoLoggedInUsers() {
  System system = new System(new DummyAuthorizer());
  assertThat(system.loginCount(), is(0));
}
```

*La doublure est utilisée dans **un test** pour permettre de tester le **SUT** (**S**ystem **U**nder **T**est) ici l'objet `system` de la classe `System`*

*Transmission de **la doublure** (sans utilisation de l'objet par la suite)*
⇒ Le Dummy suffit !!!

Qu'est-ce qu'un Stub ?

(Bouchon)



Stub : Définition …

Une interface métier…

```
interface Authorizer {
  public Boolean authorize(String username, String password);
}
```

Un Stub (bouchon)

Cette fois-ci retourne `true`
(c-a-d quelque chose ≠ null
pour simuler un login correct
lors d'un futur appel du stub …)

```
public class AcceptingAuthorizerStub implements Authorizer {
  public Boolean authorize(String username, String password) {
    return true;
  }
}
```

→ fait l'objet d'une ***implémentation minimale*** afin
de fournir des réponses prédéfinies lorsque l'objet sous test a besoin de ses services.
L'implémentation tient compte du contexte exact pour lequel la doublure sera utilisée

→ Le bouchon de test est écrit grâce à la connaissance de la classe à simuler
(**boîte blanche** ⇒ **comportement attendu** en sortie)
(avec le **`true`** on considère ici que le login est correct).

Stub : Utilisation …

→ Pour **écrire un test sur une partie du système qui requiert un login correct**,
il suffit d'injecter dans le test le bouchon `AcceptingAuthorizerStub`.

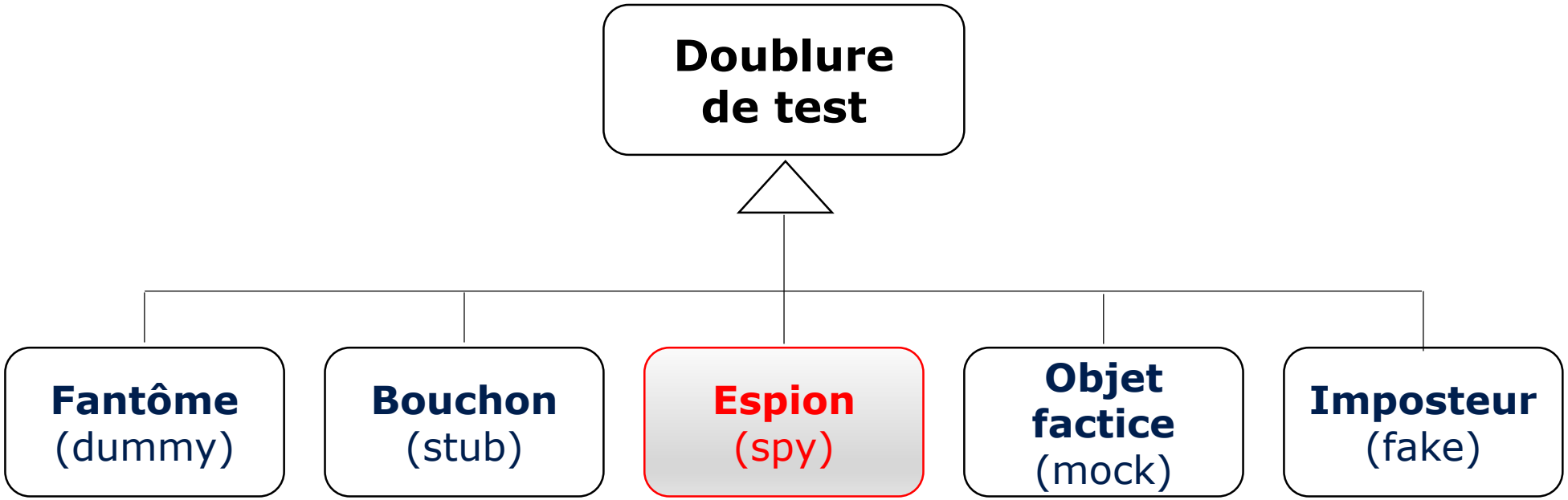
→ Pour écrire un test sur une partie du système qui gère des **utilisateurs non autorisés**, il faudrait écrire un autre bouchon `AcceptingUnauthorizerStub` qui renverrait `false` (signifiant ainsi un login incorrect)

Le stub permet de reproduire le comportement du composant dont dépend le code à tester

Le stub est une classe écrite pour en simuler un autre
qui sera **utilisée dans le test** à moment donné pour simuler ce comportement …

Qu'est-ce qu'un Spy ?

(Espion)



Spy : Définition et utilisation

Une interface métier…

```
interface Authorizer {
  public Boolean authorize(String username, String password);
}
```

Un Spy (espion)

```
public class AcceptingAuthorizerSpy implements Authorizer {
  public boolean authorizeWasCalled = false;

  public Boolean authorize(String username, String password) {
    authorizeWasCalled = true;
    return true;
  }
}
```

Implémentation minimale du bouchon…

… et enregistrement de l'appel de la méthode …

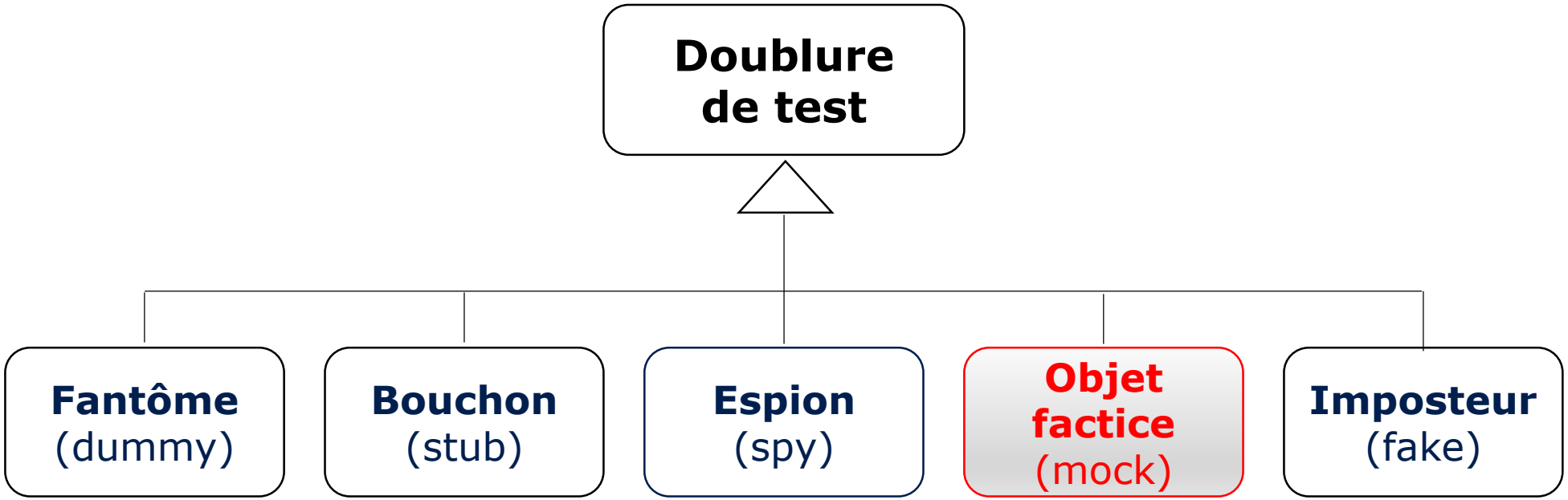
→ Bouchon qui, **en plus** de reproduire un comportement, ***enregistre les paramètres de ses appels*** pour un traitement ultérieur.

→ Un spy est donc une doublure capable **vérifier comment cette doublure est utilisée dans le test** : elle *espionne* le test…

(Par exemple : appel au moins une fois de telle méthode avec tel paramètre)

Qu'est-ce qu'un Mock ?

(Objet Factice)



Mock : Présentation

Un mock(objet factice)

```
public class AcceptingAuthorizerVerificationMock implements Authorizer {
  public boolean authorizeWasCalled = false;

  public Boolean authorize(String username, String password) {
    authorizeWasCalled = true;
    return true;
  }

  public boolean verify() {
    return authorizedWasCalled;
  }
}
```

Implémentation minimale pour donner comportement souhaité (stub)

… enregistrement de l'appel de la méthode (spy)

… ET ajout d'une méthode de contrôle pour vérifier le test …

*Un peu comme si **l'assertion attendue dans le test était déplacée dans l'objet lui-même…***
C'est l'objet qui sait ce qui est tester et qui va contrôler (`verify`), à la place du test (`assert`)

→ Objet proche d'un ***stub espion***…
.. qui permet **en plus de faire une vérification comportementale** c-a-d de vérifier quelle fonction a été appelée, avec quel(s) argument(s), combien de fois ….

Mock : Définition

[1] **Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

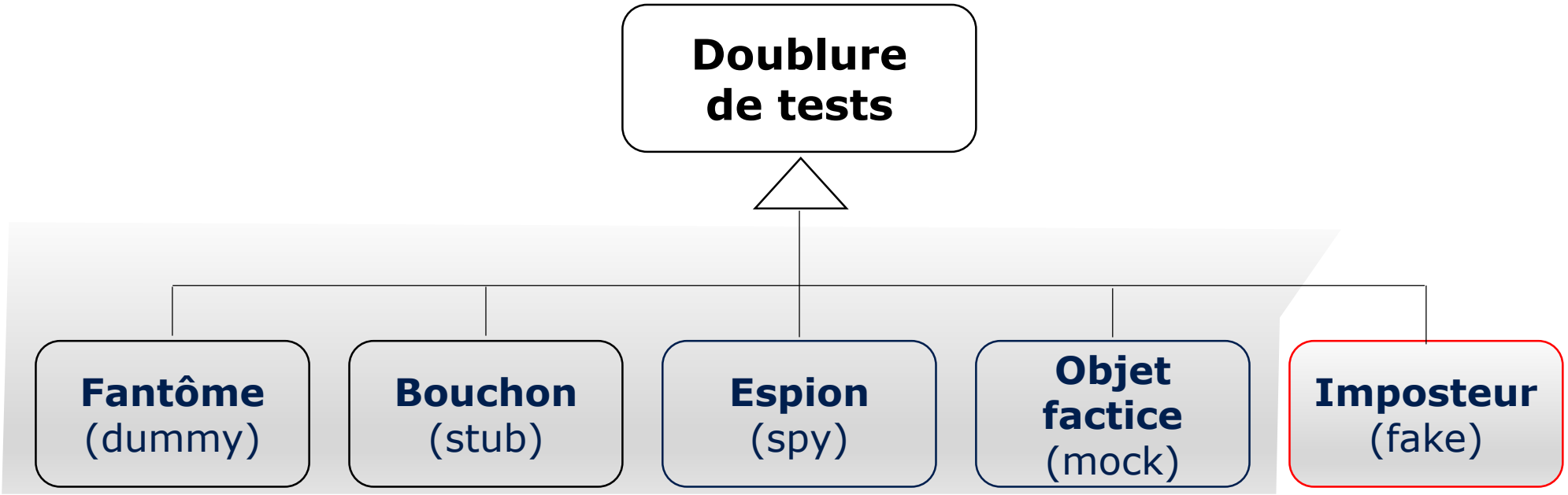
[2] **Objet factice** (mock) : objet simulé dont le ***comportement est décrit spécifiquement pour un test unitaire dans un test unitaire***.
De plus, il a la capacité à vérifier la validité et l'enchaînement d'appels de méthode sur l'objet simulé par un langage d'expectations.

[1] Définition proposé par Matin Fowler sur https://martinfowler.com/bliki/TestDouble.html

[2] « Les tests dans le développement logiciel, du cycle en V aux méthodes agiles » I. Blasquez, H.Leblanc, C.Percebois

Qu'est-ce qu'un Fake ?

(Imposteur)



Fake : Présentation

Un fake
(imposteur)

Implémente de manière simpliste le comportement de la classe.

```
public class AcceptingAuthorizerFake implements Authorizer {
      public Boolean authorize(String username, String password) {
        return username.equals("Bob");
      }
}
```

→ Un fake n'est pas un stub. Il est plus générique :
un *imposteur* a un **vrai comportement métier.**
(aucune des doublures précédentes n'avait de réel comportement métier)

→ Un fake contient une implémentation alternative opérationnelle.
Il est utilisé pour simplifier une dépendance, par exemple une base de données en mémoire au lieu d'une base de données réelle.

→ Le fake met en place des raccourcis qui le rendent inutilisable en production…

Mais pourquoi utiliser une doublure dans un test ?

Pourquoi utiliser une doublure dans un test

→L'environnement du SUT (**S**ystem **U**nder **T**est) 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 stabilisé.
Le SUT appelle du code lent.

→Le SUT fait appel à des méthodes non déterministes
(fonction de l'heure, de nombres générés aléatoirement, ...)

Extrait : http://www.lirmm.fr/~dony/enseig/IL/testEtMock.pdf

Comment manipuler une doublure de test générée à l'aide de Mockito ?

Page officielle du framework Mockito : http://mockito.org/

javadoc et documentation pour faciliter l'utilisation du framework :
https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html

En Java, utilisation possible de Mockito pour générer des doublures de test …

Pour un projet maven, ajouter au **pom.xml** la dépendance vers Mockito :

```
<dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>2.10.0</version>
        <scope>test</scope>
</dependency>
```

Extrait : http://mockito.org/

Pour un projet graddle, voir la rubrique **How** de http://mockito.org/ pour la declaration de la dépendance …

Comment créer une doublure de test (dans Mockito)

```java
public class MaClasse {
}
```

Classe (à doubler)
(code source : src/main/java)

→ via une annotation (@Mock) :

```java
import org.mockito.Mock;

public class TestMockito {

   @Mock
   MaClasse doublureDeMaClasse;

    //… utilisation de la doublure dans les tests à  suivre…
}
```

OU

→ via une méthode statique (*mock*)

```java
import org.junit.Test;
import static org.mockito.Mockito.*;

public class TestMockito {

@Test
public void unTestAvecUneDoublure()  {
   MaClasse doublureDeMaClasse = mock(MaClasse.class);
   //…
}
```

Code Test
(src/test/java)

Que faire sur des doublures ? …

→**Spécifier le comportement d'une doublure (mot clé `when`)**
c-a-d prédire le(s) résultat(s) souhaité(s) après l'appel d'une méthode.
*Cette opération est appelée **stubbing** (ou bouchonnage)*

→**Utiliser une doublure dans un code qui teste un comportement spécifique.**
***Mockito** encapsule et contrôle tous les appels effectués sur la doublure.*

→**Vérifier le« bon » comportement** d'une doublure (**mot clé `verifiy`*)***
*Cette opération est en quelque sorte une **assertion** pour vérifier le(s) appel(s) à une méthode de la doublure.*

Comment manipuler une doublure de test générée à l'aide de Mockito ?

Spécifier le comportement d'une doublure (stubber)

Isabelle BLASQUEZ

Spécifier le comportement d'une doublure : (1/2)
Un exemple simple pour comprendre le *bouchonnage*

Le contexte :

Une classe ***User*** *qui propose un service* ***getLogin*** *encore non implémenté …*

```
public class User {

public Object getLogin() {
    // TODO Auto-generated method stub
    // Rien d'implémenter pour l'instant ;-)
 return null;
}

}
```

Le problème :

Ecrire un test qui fait appel à la méthode ***getLogin*** *(qui n'existe pas encore réellement)*

La solution :

Utiliser une doublure *de* ***User*** *(un faux, un simulacre) pour* ***forcer le comportement attendu*** *et* ***faire comme si*** *ce service* ***getLogin*** *était implémenté et répondait correctement aux attentes …*

…Autrement dit …

QUAND ***getLogin*** *est appelé sur la doublure alors la doublure* ***RENVOIE*** ***la*** *« bonne »* ***valeur***

Exemples extraits de : https://www.docdoku.com/blog/2015/03/23/tests-unitaires-avec-mockito/

Spécifier le comportement d'une doublure : (2/2)
Un exemple simple pour comprendre le *bouchonnage*

```
public class TestMockitoDecrireComportement {

@Test
public void testStubberUneMethodeQuiAUnTypeDeRetour()  {

  User user = mock(User.class);
  when(user.getLogin()).thenReturn("alice");

  assertEquals(user.getLogin(), "alice"); //Test AU VERT !!!
  assertEquals(user.getLogin(), "bob"  ); //Test AU ROUGE !!!
}

}
```

1. Création de la doublure

2. *Bouchonnage* de la doublure :
Paramétrisation du comportement à simuler
lors de l'appel de ce service dans ce test (bouchonnage)

3. Utilisation de la doublure comme tout autre objet du test
(Si on faisait une assertion sur le stub, ce qu'on devrait obtenir …)

QUAND **`getLogin`** *est appelée sur la doublure alors la doublure* ***RENVOIE*** ***la*** *« bonne »* ***valeur***

méthode statique when …

… enchaînée avec thenReturn

Va permettre de *stubber* le composant `User` via une *doublure* pour les besoins de ce test …

Exemples extraits de : https://www.docdoku.com/blog/2015/03/23/tests-unitaires-avec-mockito/

Stubber avec la méthode statique when (extrait javadoc)

```
public static <T> OngoingStubbing<T> when(T methodCall)
```

Enables stubbing methods. Use it when you want the mock to return particular value when particular method is called.
Simply put: "**When** the x method is called **then** return y".

```
when(mock.someMethod()).thenReturn(10);
```

Utilisation de when pour des méthodes qui ont un *type de retour*
→ when *suivi de* thenReturn
→ when *suivi de* thenThrown

```
//setting exception to be thrown:
when(mock.someMethod("some arg")).thenThrow(new RuntimeException());
```

Levée d'exception (thenThrown)

```
//you can set different behavior for consecutive method calls.
//Last stubbing (e.g: thenReturn("foo")) determines the behavior of further consecutive calls.
when(mock.someMethod("some arg"))
 .thenThrow(new RuntimeException())
 .thenReturn("foo");
```

Enchainement de comportements pour des appels successifs :
→ premier appel : levée d'exception
→ autres appels : retour de la valeur

```
//Alternative, shorter version for consecutive stubbing:
when(mock.someMethod("some arg"))
 .thenReturn("one", "two");
//is the same as:
when(mock.someMethod("some arg"))
 .thenReturn("one")
 .thenReturn("two");
```

Retour de plusieurs valeurs consécutives

```
//you can use flexible argument matchers, e.g:
when(mock.someMethod(anyString())).thenReturn(10);
```

Arguments flexibles (Matchers)

Extraits Javadoc Mockito : https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html#when-T-
Pour plus de détails sur **ThenReturn** et **ThenThrown**, voir https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/stubbing/OngoingStubbing.html

Utiliser des ArgumentsMatchers pour plus de flexibilité ..

→ Mockito vérifie les valeurs des arguments via la méthode **equals()**

→ **Pour plus de flexibilité** (*c-a-d spécifier un appel à une méthode sans que les valeurs des paramètres n'aient vraiment d'importance*), il est possible d'utiliser des ***arguments matchers***

```
//stubbing using built-in anyInt() argument matcher
when(mockedList.get(anyInt())).thenReturn("element");
```

matcher

```
//following prints "element"
System.out.println(mockedList.get(999));
```

À l'utilisation, la doublure peut ainsi faire appel à get avec une valeur entière quelconque

Si vous décidez d'utiliser un matcher, **tous les arguments doivent être des matchers** !

```
verify(mock).someMethod(anyInt(), anyString(), eq("third argument"));
//above is correct - eq() is also an argument matcher
```

```
verify(mock).someMethod(anyInt(), anyString(), "third argument");
//above is incorrect - exception will be thrown because third argument is given without an argument matcher
```

Exemples extraits de https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html#3

Choisir le « bon » *ArgumentsMatchers* …

Les ***matchers*** souvent utilisés sont :

- **any, anyObject,**
- **anyBoolean, anyDouble, anyFloat, anyInt, anyString,…**
- **anyList, anyMap, anyCollection, anyCollectionOf, anySet(), anySetOf, …**
- **…**

org.mockito

Class ArgumentMatchers

java.lang.Object
org.mockito.ArgumentMatchers

Direct Known Subclasses:
Matchers, Mockito

Method and Description
`any()` Matches **anything**, including nulls and varargs.
`any(Class<T> type)` Matches any object of given type, excluding nulls.
`anyBoolean()` Any `boolean` or **non-null** `Boolean`
`anyByte()` Any `byte` or **non-null** `Byte`.
`anyChar()` Any `char` or **non-null** `Character`.
`anyCollection()` Any **non-null** `Collection`.
`anyCollectionOf(Class<T> clazz)` Deprecated. With Java 8 this method will be removed in Mockito 3.0.
`anyDouble()` Any `double` or **non-null** `Double`.
`anyFloat()` Any `float` or **non-null** `Float`.
`anyInt()` Any int or **non-null** `Integer`.
`anyIterable()` Any **non-null** `Iterable`.
`anyIterableOf(Class<T> clazz)` Deprecated. With Java 8 this method will be removed in Mockito 3.0.
`anyList()` Any **non-null** `List`.
`anyListOf(Class<T> clazz)` Deprecated. With Java 8 this method will be removed in Mockito 3.0. Thi
`anyLong()` Any `long` or **non-null** `Long`.
`anyMap()` Any **non-null** `Map`.
`anyMapOf(Class<K> keyClazz, Class<V> valueClazz)` Deprecated. With Java 8 this method will be removed in Mockito 3.0. Thi
`anyObject()` Deprecated. This will be removed in Mockito 3.0 (which will be java 8 on
`anySet()` Any **non-null** `Set`.
`anySetOf(Class<T> clazz)` Deprecated. With Java 8 this method will be removed in Mockito 3.0. Thi
`anyShort()` Any `short` or **non-null** `Short`.
`anyString()` Any **non-null** `String`
`anyVararg()` Deprecated. as of 2.1.0 use `any()`
`argThat(ArgumentMatcher<T> matcher)` Allows creating custom argument matchers.
`booleanThat(ArgumentMatcher<Boolean> matcher)` Allows creating custom `boolean` argument matchers.
`byteThat(ArgumentMatcher<Byte> matcher)` Allows creating custom `byte` argument matchers.
`charThat(ArgumentMatcher<Character> matcher)` Allows creating custom `char` argument matchers.
`contains(String substring)` `String` argument that contains the given substring.
`doubleThat(ArgumentMatcher<Double> matcher)` Allows creating custom `double` argument matchers.
`endsWith(String suffix)` `String` argument that ends with the given suffix.
`eq(boolean value)` `boolean` argument that is equal to the given value.
`eq(byte value)` `byte` argument that is equal to the given value.
`eq(char value)` `char` argument that is equal to the given value.
`eq(double value)` `double` argument that is equal to the given value.
`eq(float value)` `float` argument that is equal to the given value.
`eq(int value)` `int` argument that is equal to the given value.
`eq(long value)` `long` argument that is equal to the given value.
`eq(short value)` `short` argument that is equal to the given value.
`eq(T value)` Object argument that is equal to the given value.
`floatThat(ArgumentMatcher<Float> matcher)` Allows creating custom `float` argument matchers.
`intThat(ArgumentMatcher<Integer> matcher)` Allows creating custom `int` argument matchers.
`isA(Class<T> type)` `Object` argument that implements the given class.
`isNotNull()` Not `null` argument.

…etc…

Retrouvez tous les ArgumentsMatchers dans la *javadoc* :
https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/ArgumentMatchers.html

Spécifier le comportement d'une doublure : Stubber une méthode sans type de retour (void) (1/2)

doThrow

```
public static Stubber doThrow(Throwable... toBeThrown)
```

Use `doThrow()` when you want to stub the void method with an exception.

Stubbing voids requires different approach from `when(Object)` because the compiler does not like void methods inside brackets...

```
doThrow(new RuntimeException()).when(mock).someVoidMethod();
```

pour les méthodes sans *type de retour*
doXXX *suivi* de when

doAnswer

```
public static Stubber doAnswer(Answer answer)
```

Use `doAnswer()` when you want to stub a void method with generic `Answer`.

```
doAnswer(new Answer() {
    public Object answer(InvocationOnMock invocation) {
        Object[] args = invocation.getArguments();
        Mock mock = invocation.getMock();
        return null;
    }})
.when(mock).someMethod();
```

doNothing

```
public static Stubber doNothing()
```

Use `doNothing()` for setting void methods to do nothing. **Beware that void methods on mocks do nothing by default!**

```
doNothing().
doThrow(new RuntimeException())
.when(mock).someVoidMethod();

//does nothing the first time:
mock.someVoidMethod();

//throws RuntimeException the next time:
mock.someVoidMethod();
```

Pour plus de détails sur ces méthodes et leur utilisation, consulter la Javadoc de Mockito :
https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html

Spécifier le comportement d'une doublure : Stubber une méthode sans type de retour (`void`) (2/2)

doCallRealMethod

doCallRealMethod

```
public static Stubber doCallRealMethod()
```

Use `doCallRealMethod()` when you want to call the real implementation of a method.

```
Foo mock = mock(Foo.class);
doCallRealMethod().when(mock).someVoidMethod();

// this will call the real implementation of Foo.someVoidMethod()
mock.someVoidMethod();
```

doReturn

doReturn

```
public static Stubber doReturn(Object toBeReturned)
```

Use `doReturn()` in those rare occasions when you cannot use `when(Object)`.

Beware that `when(Object)` is always recommended for stubbing because it is argument type-safe and more readable (especially when stubbing consecutive calls)

Pour plus de détails sur ces méthodes et leur utilisation, consulter la Javadoc de Mockito :
https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html

Comment manipuler une doublure de test générée à l'aide de Mockito ?

Vérifier le« bon »comportement d'une doublure

verify

Isabelle BLASQUEZ

Vérification comportementale de la doublure : un exemple simple

```java
import org.junit.Test;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;
import static org.mockito.Mockito.verify;

public class TestMockitoVerification {

   @Test
   public void testVerification() {

      User user = mock(User.class);
      when(user.getLogin()).thenReturn("alice");


      System.out.println(user.getLogin());


      verify(user).getLogin();
    }
   }
```

Une fois créé…

… le mock va mémoriser toutes les interactions

… verify permet de vérifier qu'une méthode a bien été appelée
(les tests passent au VERT, l'assertion est réalisée au travers de verify)

Voir dans la Javadoc : https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html#1

Vérifier le nombre de fois où une méthode a été appelée...

```
LinkedList<String> mockedList = mock(LinkedList.class);

mockedList.add("twice");
mockedList.add("twice");

mockedList.add("three times");
mockedList.add("three times");
mockedList.add("three times");

//exact number of invocations verification
verify(mockedList, times(2)).add("twice");
verify(mockedList, times(3)).add("three times");

//verification using never(). never() is an alias to times(0)
verify(mockedList, never()).add("never happened");

//verification using atLeast()/atMost()
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("three times");
verify(mockedList, atMost(5)).add("three times");
```

appelée exactement *n* **fois** (*times*)

Jamais appelée (*never*)

appelée
au moins 1 fois (*atLeastOnce*)
au moins n fois (*atLeast*)
au plus **1 fois** (*atMost*)

Extrait simplifié dans la Javadoc : https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html#4

Comment manipuler une doublure de test générée à l'aide de Mockito ?

Espionner un objet réel

Isabelle BLASQUEZ

Espionner de vrais objet via Mockito à l'aide de Spy

A la différence du ***mock***, un espion (***spy***) permet de **doubler une *vraie* classe**
(et de *faire appel aux méthodes déjà implémentées*)

```
List spy = Mockito.spy(new LinkedList());

//optionally, you can stub out some methods:
when(spy.size()).thenReturn(100);

//using the spy calls *real* methods
spy.add("one");
spy.add("two");

//prints "one" - the first element of a list
System.out.println(spy.get(0));

//size() method was stubbed - 100 is printed
System.out.println(spy.size());

//optionally, you can verify
verify(spy).add("one");
verify(spy).add("two");
```

OU

```
@Spy
List list = new LinkedList();
```

Comme un mock, possibilité de stubber les méthodes ...

Si méthode non stubbée
(c-a-d comportement non spécifié)
spy appelle *vraie* méthode
≠
mock renverrait null

Comme un mock, possibilité de mettre en place une vérification comportementale

Extrait : https://static.javadoc.io/org.mockito/mockito-core/2.10.0/org/mockito/Mockito.html#13

Mockito : Conclusion

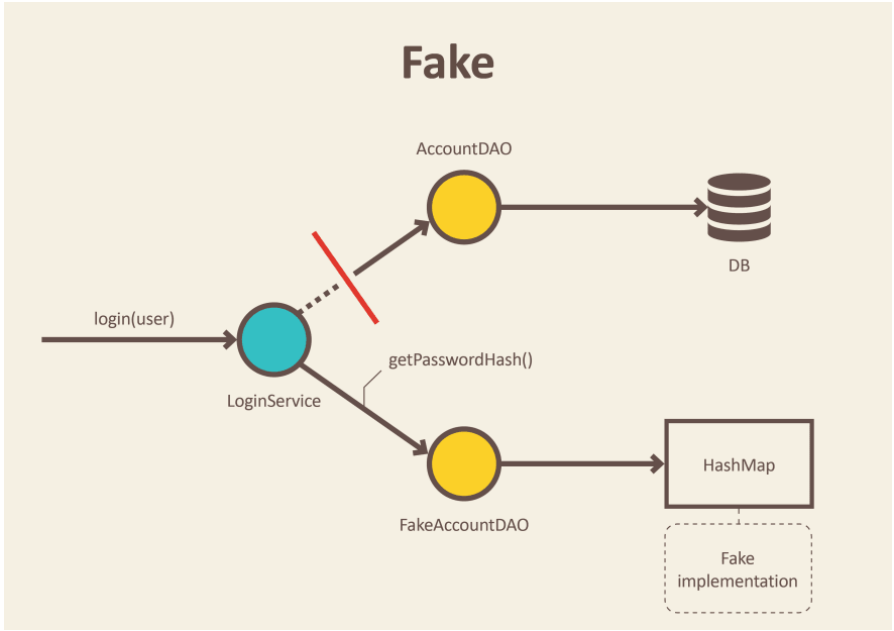
Mockito est un framework de doublures de test qui, associé à **JUnit**, permet :
→ une écriture rapide de tests unitaires,
→ de se focaliser sur le comportement de la méthode à tester en mockant les connexions aux bases de données, les appels aux web services …

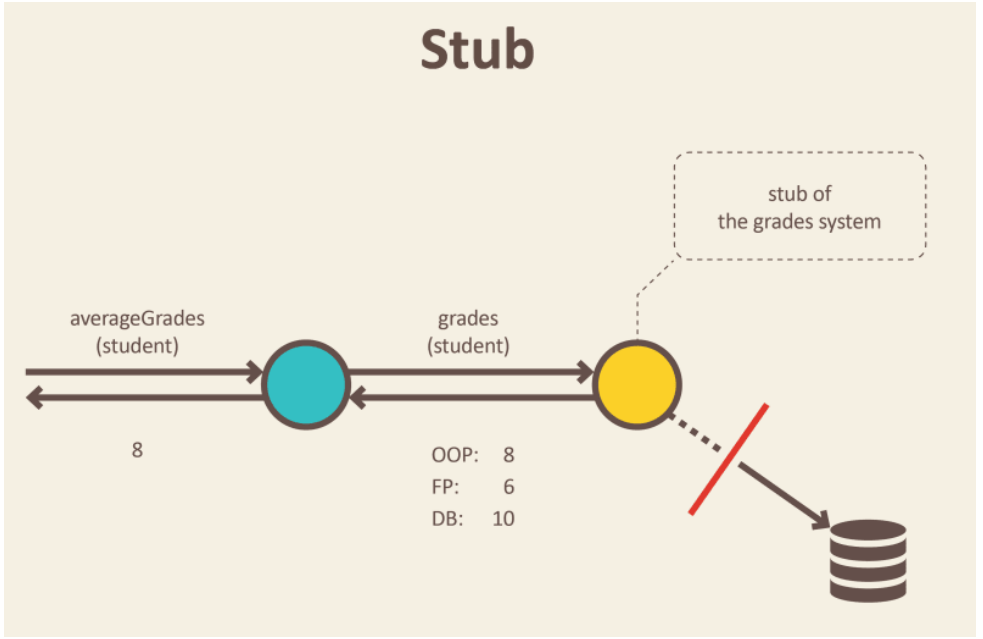
✓ **Limites :** Mockito ne permet pas de *bouchonner* :
→ les classes finales,
→ les `enums`,
→ les méthodes privées, `final`, `static` (alternative possible avec PowerMock voir ici)
→ les méthodes `hashCode()` et `equals()` (utilisées en interne par Mockito notamment pour les matchers)

✓ **A noter aussi que … :**
→ Un comportement de mock non exécuté ne provoque pas d'erreur
→ **`Verify`** : si la vérification échoue, le test échoue.

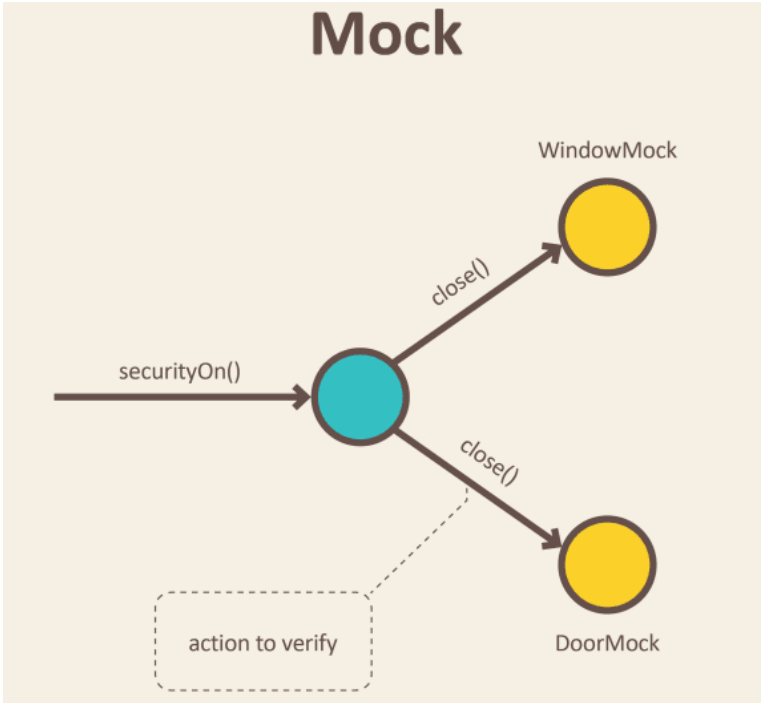
Les objets *mockés* doivent être maintenus au fur et à mesure des évolutions du code à tester.

Inspiré de : https://www.docdoku.com/blog/2015/03/23/tests-unitaires-avec-mockito/


Fake
AccountDAO
DB
login(user)
getPasswordHash()
LoginService
HashMap
FakeAccountDAO
Fake
implementation

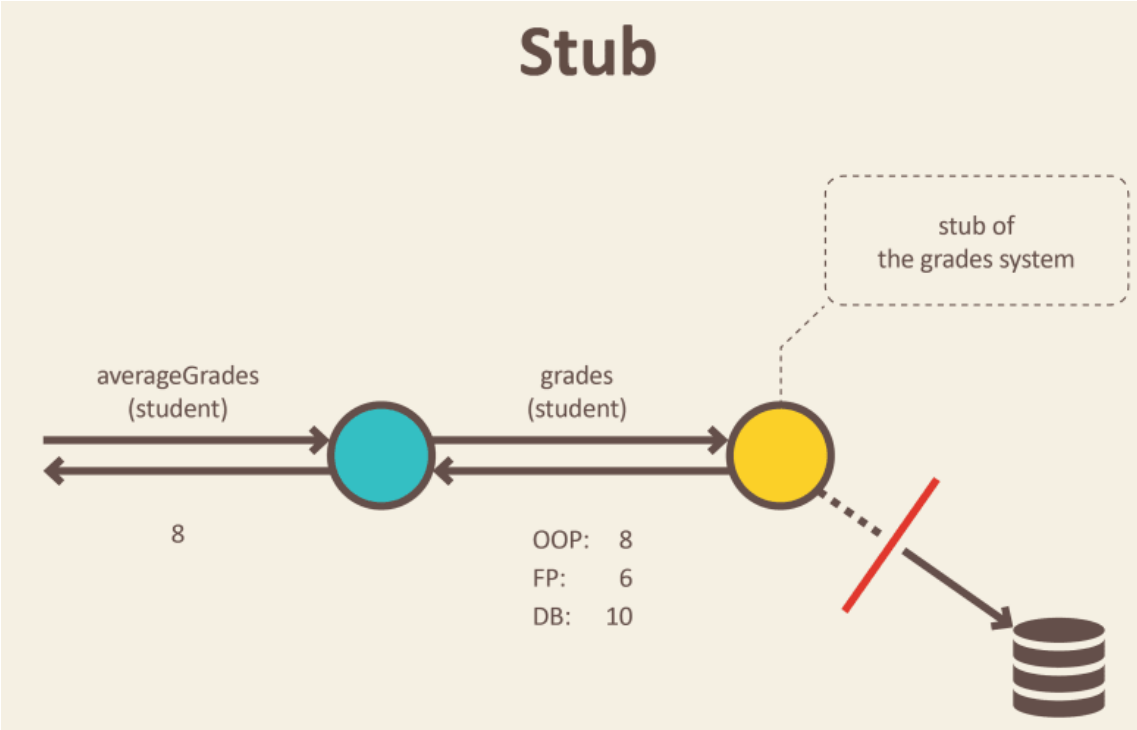

Stub
stub of
the grades system
averageGrades
(student)
grades
(student)
8
OOP: 8
FP: 6
DB: 10

Exemples


Mock
WindowMock
close()
securityOn()
close()
action to verify
DoorMock

Exemple de Stub ...

Stub is an object that holds predefined data and uses it to answer calls during tests. It is used when we cannot or don't want to involve objects that would answer with real data or have undesirable side effects.




```java
public class GradesService {

    private final Gradebook gradebook;

    public GradesService(Gradebook gradebook) {
        this.gradebook = gradebook;
    }

    Double averageGrades(Student student) {
        return average(gradebook.gradesFor(student));
    }
}
```

```java
public class GradesServiceTest {

    private Student student;
    private Gradebook gradebook;

    @Before
    public void setUp() throws Exception {
        gradebook = mock(Gradebook.class);
        student = new Student();
    }

    @Test
    public void calculates_grades_average_for_student() {
        when(gradebook.gradesFor(student)).thenReturn(grades(8, 6, 10)); //stubbing gradebook

        double averageGrades = new GradesService(gradebook).averageGrades(student);

        assertThat(averageGrades).isEqualTo(8.0);
    }
}
```

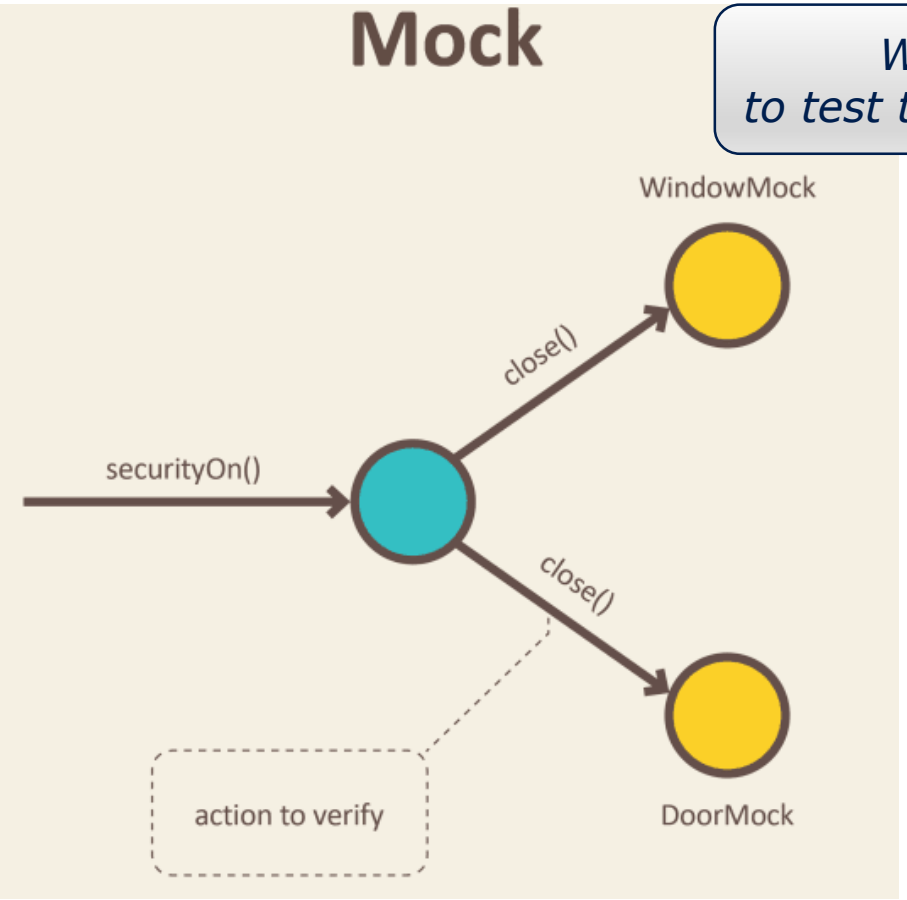
define just enough data to test average calculation algorithm.

Instead of calling database from Gradebook store to get real students grades, we preconfigure stub with grades that will be returned

Extrait de https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da repris dans https://dev.to/milipski/test-doubles---fakes-mocks-and-stubs

Exemple de Mock ...

Mocks are objects that register calls they receive. In test assertion we can verify on Mocks that all expected actions were performed.



We don't want to close real doors to test that security method is working, right?

```java
public class SecurityCentral {

    private final Window window;
    private final Door door;

    public SecurityCentral(Window window, Door door) {
        this.window = window;
        this.door = door;
    }

    void securityOn() {
        window.close();
        door.close();
    }
}
```

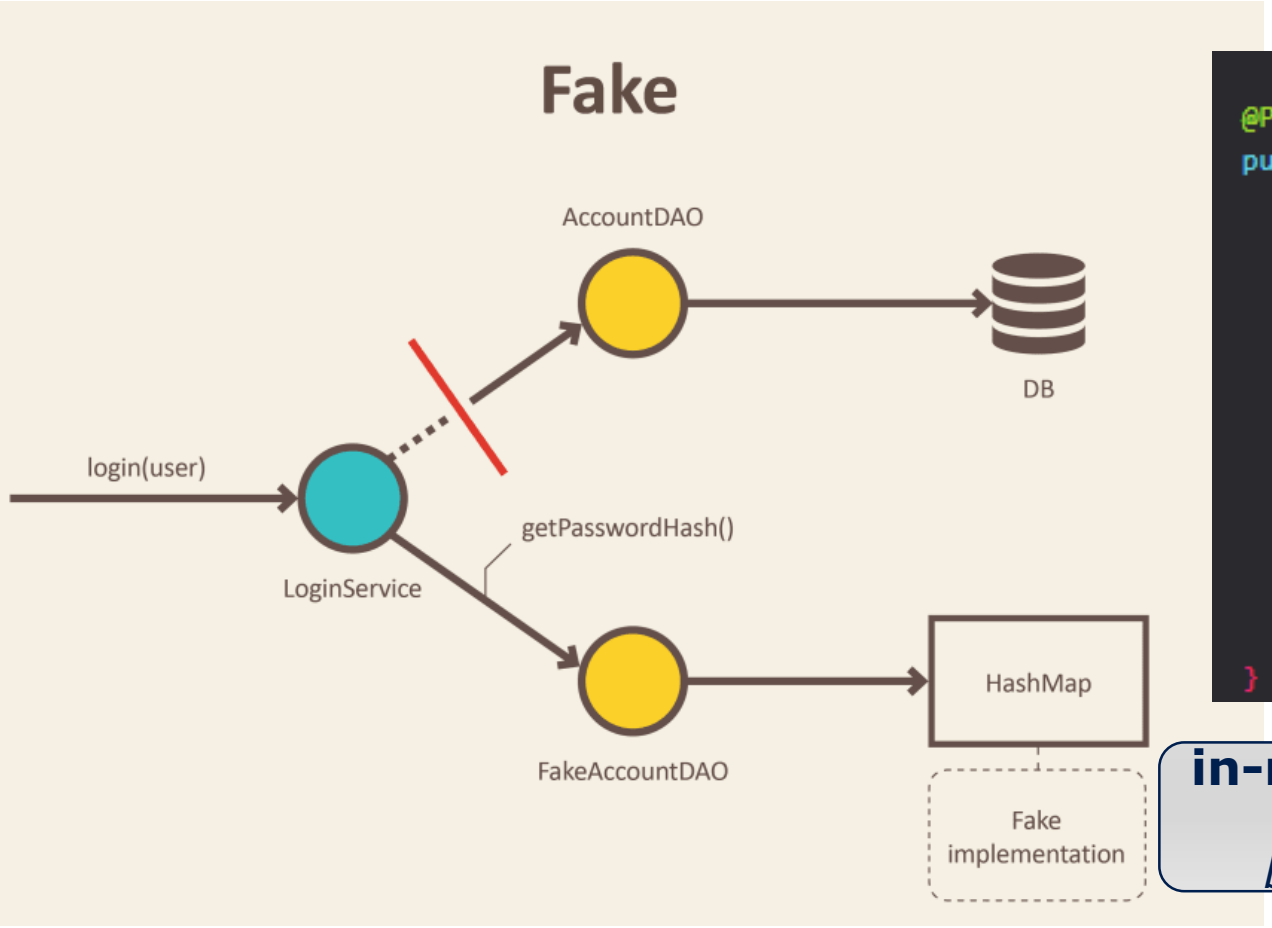
```java
public class SecurityCentralTest {

    Window windowMock = mock(Window.class);
    Door doorMock = mock(Door.class);

    @Test
    public void enabling_security_locks_windows_and_doors() {
        SecurityCentral securityCentral = new SecurityCentral(windowMock, doorMock);

        securityCentral.securityOn();

        verify(doorMock).close();
        verify(windowMock).close();
    }
}
```

*This is **responsibility of `Door` and `Window` alone to close itself** when they get proper signal. We can test it independently in different unit test.*

Extrait de https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da repris dans https://dev.to/milipski/test-doubles---fakes-mocks-and-stubs

Exemple de Fake ...

Fakes are objects that have working implementations, but not same as production one. Usually they take some shortcut and have simplified version of production code.




```java
@Profile("transient")
public class FakeAccountRepository implements AccountRepository {

    Map<User, Account> accounts = new HashMap<>();

    public FakeAccountRepository() {
        this.accounts.put(new User("john@bmail.com"), new UserAccount());
        this.accounts.put(new User("boby@bmail.com"), new AdminAccount());
    }

    String getPasswordHash(User user) {
        return accounts.get(user).getPasswordHash();
    }
}
```

in-memory implementation of Data Access Object

This fake implementation will not engage database, but will use a simple collection to store data (HashMap)

This allows to **do integration test of services** without starting up a database and **performing time consuming requests**

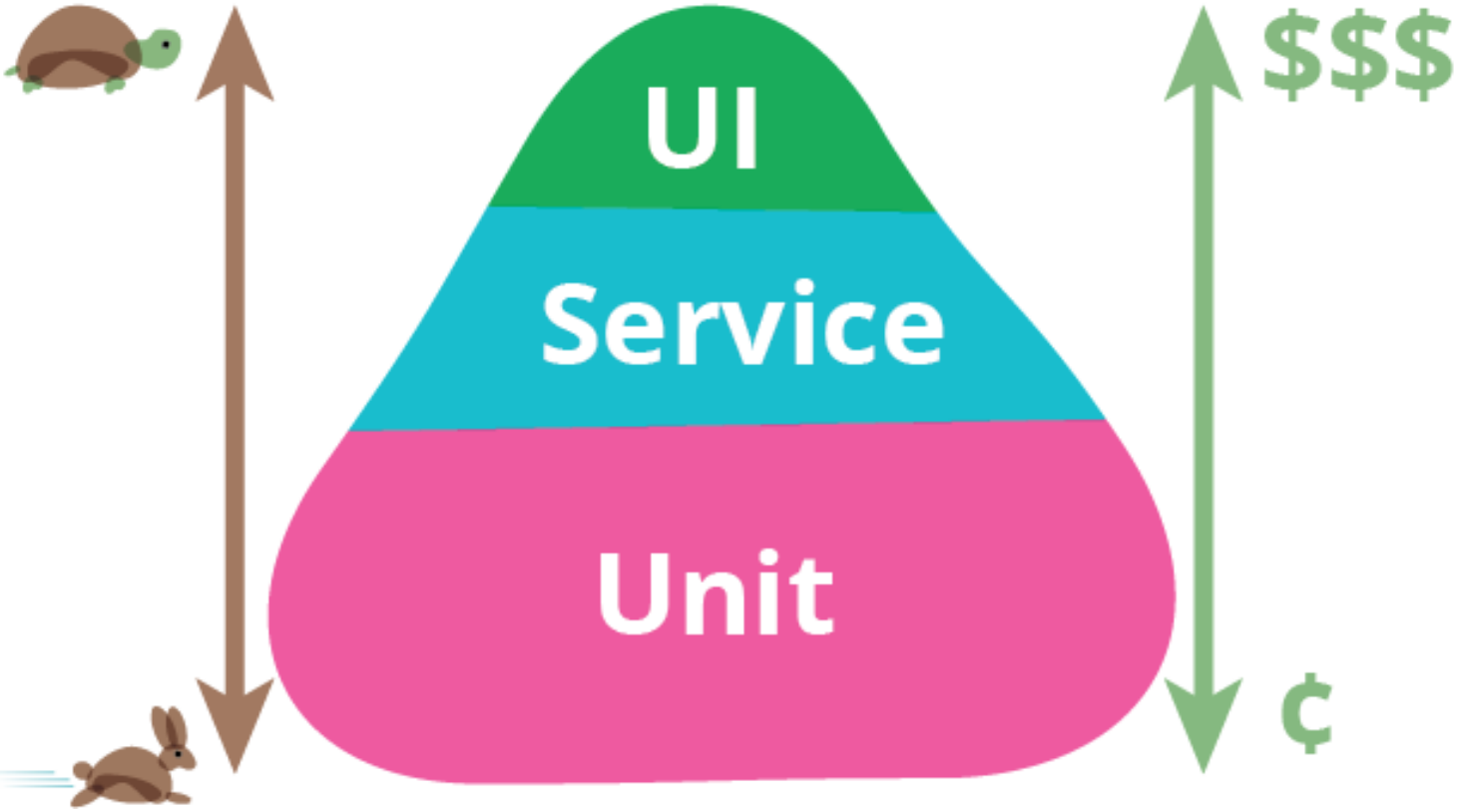
Extrait de https://blog.pragmatists.com/test-doubles-fakes-mocks-and-stubs-1a7491dfa3da repris dans https://dev.to/milipski/test-doubles---fakes-mocks-and-stubs

Mais au fait,

d'où vient ce *besoin* de doublures ?

Des tests d'intégration aux doublures …

Stratégie d'automatisation optimale des tests d'après Mike Cohn (dans un développement agile)



Tests d'Interface Utilisateur
(mise en place et maintenance couteuse)

Tests Services
(tests d'acceptation, indépendants de l'interface Utilisateur)

Tests Unitaires
(prépondérants !!!)

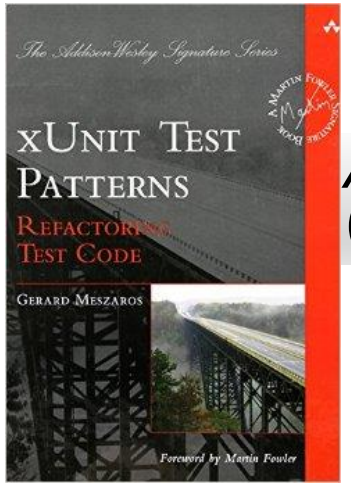
Image extraite de : https://martinfowler.com/bliki/TestPyramid.html représentant la Pyramide de Mike Cohn (initialement présentée dans **Succeeding with Agile**)

Isabelle BLASQUEZ

Qu'est-ce qu'un test unitaire ?

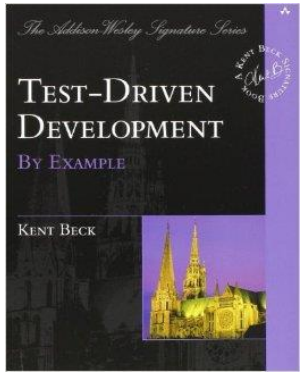
… Autant de définitions que d'auteurs ….

Procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme (appelée « unité » ou « module »)
(Wikipedia : https://fr.wikipedia.org/wiki/Test_unitaire)



A test that verifies the behavior of some small part of the overall system.
(Meszaros)

Test à petite échelle.
(Kent Beck)



Test qui vérifie un comportement d'un morceau de code (Antoine Vernois)

Test Unitaire

(Vérification au plus proche du code)

Rappel

Un test unitaire *(appelé aussi test de composant) est un test qui permet de tester de manière isolée une unité logicielle ou un groupe d'unités* **(IEEE, 1990)**.

Outil : ***Framework xUnit pour faciliter la création et l'exécution des tests***

Les tests unitaires sont des ***tests dits en isolation*** *car les composants sont testés individuellement les uns des autres, et ce dans n'importe quel ordre.*

Image : http://www.beilers.com/2010/03/unit-testing-not-required/

Mais ...

2 tests unitaires : ✓

0 test d'integration : ✗

Rappel

Lorsque l'architecture logicielle se complexifie, les tests unitaires ne sont plus suffisants,
des **tests d'intégration** sont nécessaires
pour **vérifier les interactions entre différents composants** (testés unitairement).

Extraits : http://keynote.hoa-project.net/ForumPHP12/
https://twitter.com/ThePracticalDev/status/687672086152753152

Test d'Intégration

(Vérification des connexions d'interface des composants)

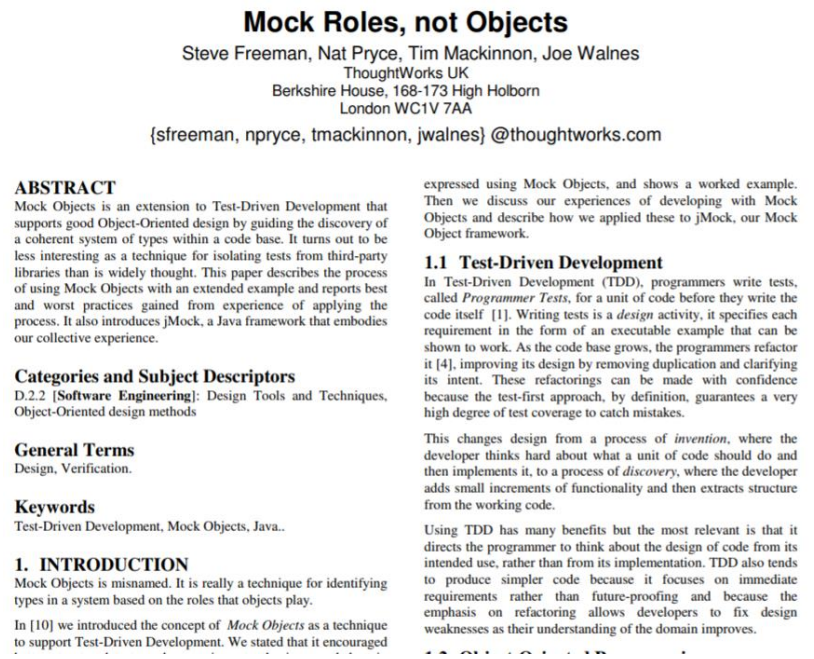
Rappel

Un test d'intégration *est un test dans lequel les composants logiciels, les composants matériels ou les deux sont combinés pour tester et évaluer leurs interactions* **(IEEE, 1990)**.

Extrait : http://keynote.hoa-project.net/ForumPHP12/

Des tests d'Intégration aux doublures de test

Doublure de tests : technique proposée par la communauté agile pour permettre aux **tests d'intégration de devenir indépendants** et d'être joués comme les tests unitaires.



Mock Roles, not Objects

Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes
ThoughtWorks UK
Berkshire House, 168-173 High Holborn
London WC1V 7AA
{sfreeman, npryce, tmackinnon, jwalnes} @thoughtworks.com

ABSTRACT
Mock Objects is an extension to Test-Driven Development that supports good Object-Oriented design by guiding the discovery of a coherent system of types within a code base. It turns out to be less interesting as a technique for isolating tests from third-party libraries than is widely thought. This paper describes the process of using Mock Objects with an extended example and reports best and worst practices gained from experience of applying the process. It also introduces jMock, a Java framework that embodies our collective experience.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques, Object-Oriented design methods

General Terms
Design, Verification.

Keywords
Test-Driven Development, Mock Objects, Java..

1. INTRODUCTION
Mock Objects is misnamed. It is really a technique for identifying types in a system based on the roles that objects play.

In [10] we introduced the concept of *Mock Objects* as a technique to support Test-Driven Development. We stated that it encouraged expressed using Mock Objects, and shows a worked example. Then we discuss our experiences of developing with Mock Objects and describe how we applied these to jMock, our Mock Object framework.

1.1 Test-Driven Development
In Test-Driven Development (TDD), programmers write tests, called *Programmer Tests*, for a unit of code before they write the code itself [1]. Writing tests is a *design* activity, it specifies each requirement in the form of an executable example that can be shown to work. As the code base grows, the programmers refactor it [4], improving its design by removing duplication and clarifying its intent. These refactorings can be made with confidence because the test-first approach, by definition, guarantees a very high degree of test coverage to catch mistakes.

This changes design from a process of *invention*, where the developer thinks hard about what a unit of code should do and then implements it, to a process of *discovery*, where the developer adds small increments of functionality and then extracts structure from the working code.

Using TDD has many benefits but the most relevant is that it directs the programmer to think about the design of code from its intended use, rather than from its implementation. TDD also tends to produce simpler code because it focuses on immediate requirements rather than future-proofing and because the emphasis on refactoring allows developers to fix design weaknesses as their understanding of the domain improves.

Les doublures permettent de **vérifier le comportement** d'un système sous test
sans disposer de tous ses objets réels.

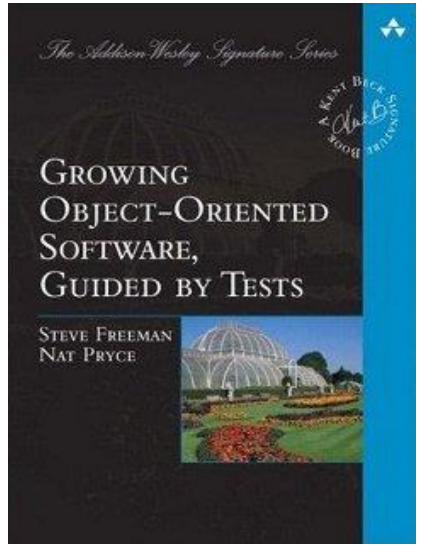
Grâce aux doublures, les tests unitaires et les tests d'intégration s'écrivent de la même façon avec les mêmes outils (framework xUnit ou autres) bien qu'ils abordent des points de vue différents sur le code en production.

Article de référence des doublures : http://www.jmock.org/oopsla2004.pdf

Vers une nouvelle approche du développement dirigé par les tests

Grâce aux doublures, la *rapidité* (**F**ast), l'*isolation* (**I**ndependant), et la *répétition* (***R***epeatable) des tests sont facilitées.

Une (nouvelle) approche de développement dirigé par les tests est dès lors envisageable (approche dite « TDD mockiste »)



Livre de référence : http://www.growing-object-oriented-software.com

Annexe

Générer des doublures dans différents langages …

Java

Il en existe plusieurs mais le plus utilisé est Mockito [archive]. D'autres solutions sont plus abouties comme Mockachino [archive] ou JMockit [archive] ;

C++

Google propose le Google C++ Mocking Framework [archive] ;

Groovy

Gmock [archive] ;

Ruby

mocha [archive]

JavaScript

SinonJS [archive]

Extrait : https://fr.wikibooks.org/wiki/Introduction_au_test_logiciel/Doublures_de_test

→ **Mocking in Python :**
https://semaphoreci.com/community/tutorials/getting-started-with-mocking-in-python

→ **Mocking in Ruby** :
https://semaphoreci.com/community/tutorials/mocking-with-rspec-doubles-and-expectations

→ **Mocking in Javascript** :
https://semaphoreci.com/community/tutorials/best-practices-for-spies-stubs-and-mocks-in-sinon-js

Doublures : Définitions (1/3)

The generic term he uses is a Test Double (think stunt double). Test Double is a generic term for any case where you replace a production object for testing purposes. There are various kinds of double that Gerard lists:

- **Dummy** objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Fake** objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an InMemoryTestDatabase is a good example).
- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
- **Spies** are stubs that also record some information based on how they were called. One form of this might be an email service that records how many messages it was sent.
- **Mocks** are pre-programmed with expectations which form a specification of the calls they are expected to receive. They can throw an exception if they receive a call they don't expect and are checked during verification to ensure they got all the calls they were expecting.

Définitions sur les différents types de doublure proposées par Matin Fowler sur :
https://martinfowler.com/bliki/TestDouble.html

Doublures : Définitions (2/3)

Fantôme (dummy) : doublure de test la plus simple.
Ne contient ***aucune implémentation***. Dans le code d'un test, il est utilisé comme paramètre d'appel à une méthode.

Bouchon (stub) : fait l'objet d'une ***implémentation minimale*** afin de fournir des réponses prédéfinies lorsque l'objet sous test a besoin de ses services.

Espion (spy) : similaire au bouchon, mais il ***enregistre en plus les paramètres de ses appels*** pour un traitement ultérieur.

Objet factice (mock) : objet simulé dont le ***comportement est décrit spécifiquement pour un test unitaire dans un test unitaire***.
De plus, il a la capacité à vérifier la validité et l'enchaînement d'appels de méthode sur l'objet simulé par un langage d'expectations.

Imposteur (fake) : contient une implémentation alternative opérationnelle.
Il est utilisé pour simplifier une dépendance, par exemple une base de données en mémoire au lieu d'une base de données réelle.

Extrait de « Les tests dans le développement logiciel, du cycle en V aux méthodes agiles »
I. Blasquez, H.Leblanc, C.Percebois (TSI 2017 à paraître)

Doublures : Définitions (3/3)

Les termes de "Stub" et "Mock" sont aujourd'hui utilisés de manière assez courante mais, faute de véritable référence, tout le monde n'utilise pas ces mots avec la même signification.

Une affirmation assez courante, consiste à dire "les stubs c'est pour faire des vérifications d'état, les mocks c'est pour faire des vérifications de comportement".
Je préfère, de loin, les définitions proposées par **Gerard Meszaros** dans ses **xUnit Patterns** :

Stub : doublure qui remplace un objet réel **en fournissant des entrées indirectes** à l'objet à tester

Mock : doublure qui remplace un objet réel en **vérifiant les sorties indirectes** de l'objet à tester

Spy : c'est une doublure qui remplace un objet réel en **enregistrant les sorties indirectes** de l'objet à tester pour **vérification ultérieure** par le test

*Quand on parle des doublures (objets qui remplacent les objets réels lors des tests) les caractéristiques à identifier (celles qui auront le plus d'impact sur la stratégie d'implémentation du test) sont la mise en œuvre **des entrées indirectes** et **des sorties indirectes**.*

Extrait (et voir les exemples proposés) : http://agilitateur.azeau.com/post/2011/04/12/Pour-en-finir-avec-les-Stubs-et-les-Mocks