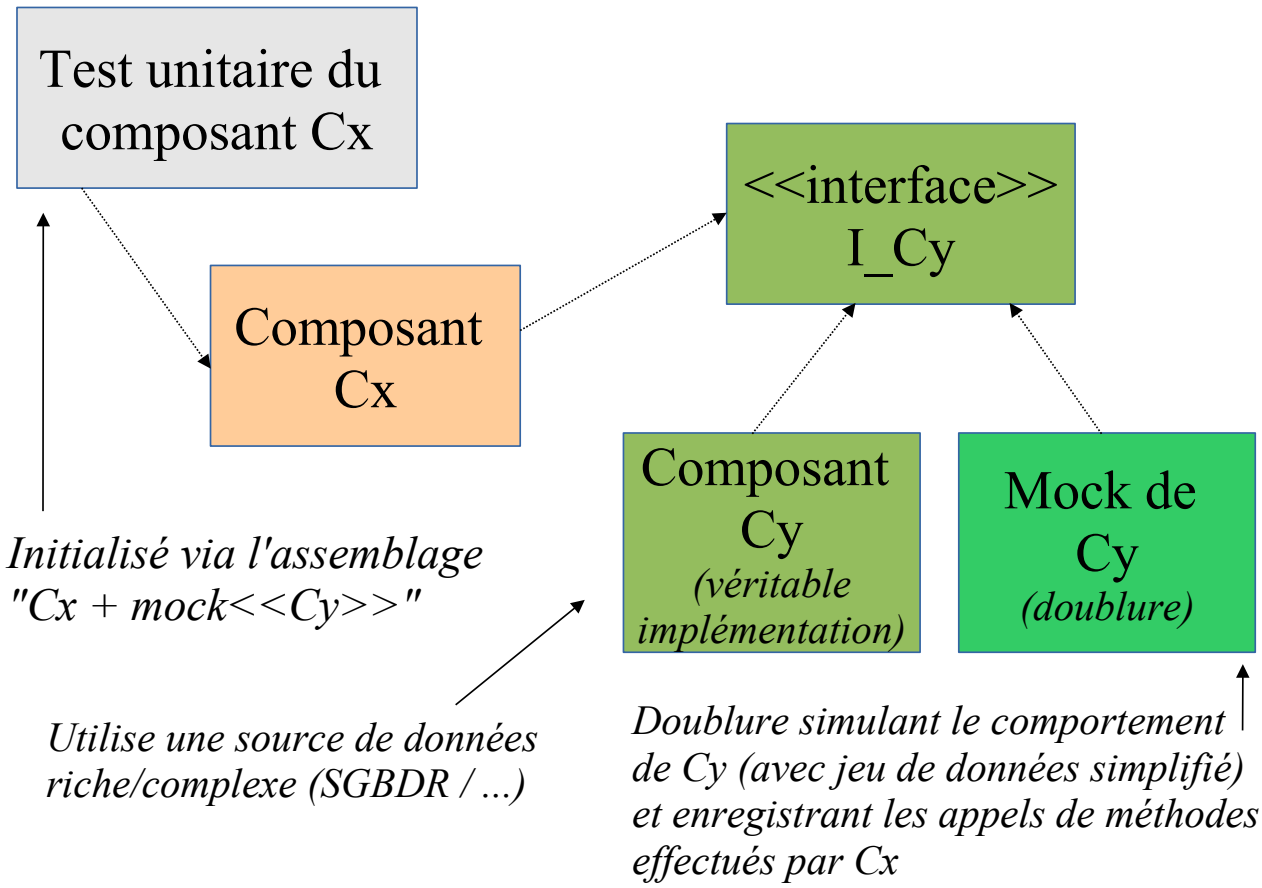
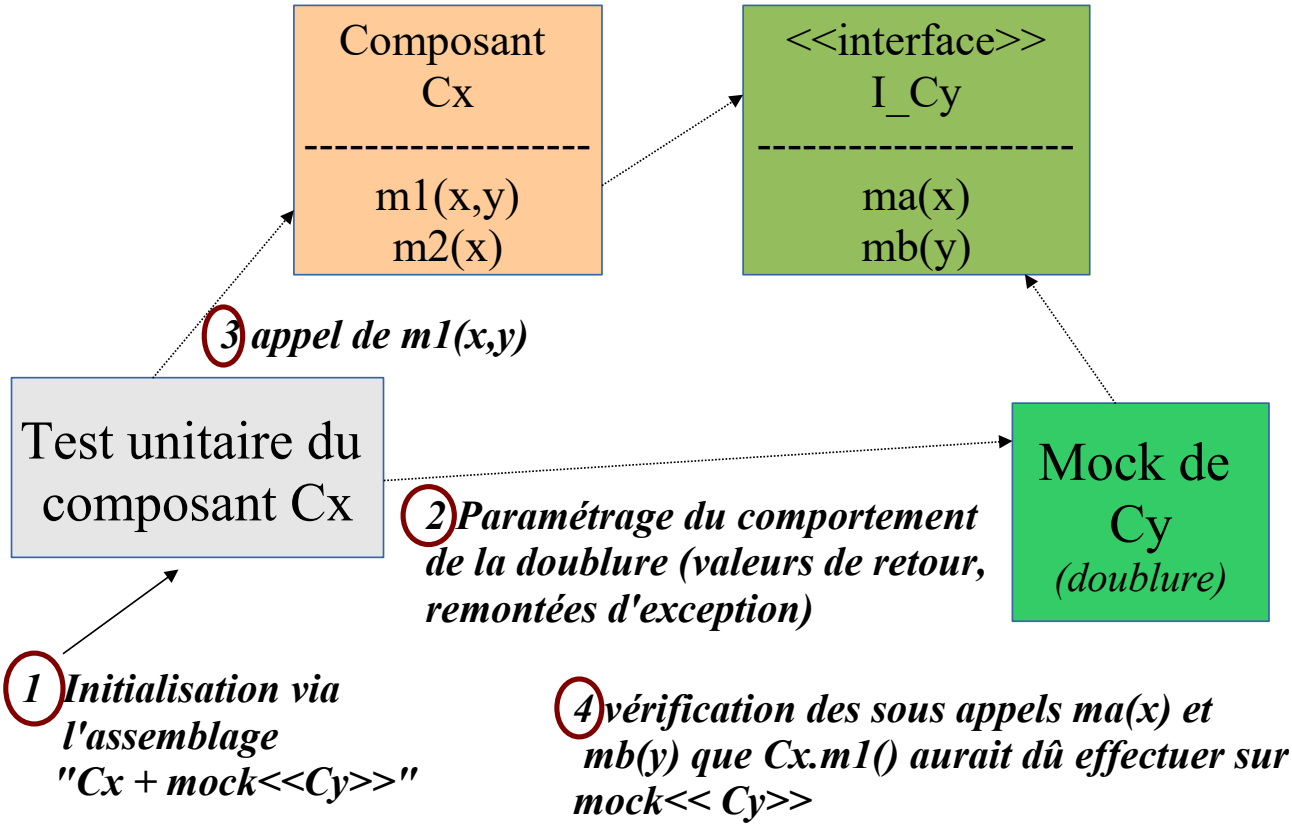
1. Tests de composants et mocks

1.1. principes et intérêts des "mocks"

Positionnement des "mocks"



intérêts des "mocks"



1.2. Mockito (as stub)

Mockito (une des API java pour "mocks")

Mockito est une technologie java assez populaire pour mettre en œuvre des "Mocks" (simulacres).

Dépendance maven nécessaire:

```
<dependency>
		<groupId>org.mockito</groupId>
		<artifactId>mockito-core</artifactId>
		<!-- <artifactId>mockito-all</artifactId> -->
		<version>1.10.19</version>
</dependency>
```

Initialisation d'un "mock" au sein d'un test unitaire (1/2)

Solution1 : via @RunWith et @Mock :

```
import org.junit.runner.RunWith;            import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class UserLoginMockTest {
  @Mock
   private static UserLogin userLogin;
….
```

Solution2 : via @Mock et MockitoAnnotations.initMocks():

```
import org.mockito.Mock;          import org.mockito.MockitoAnnotations;

public class UserLoginMockTest {
   @Mock
   private  UserLogin userLogin;

   @Before
      public void init() /* or setUp or ... */ {
      MockitoAnnotations.initMocks(this);
   } …
```

Initialisation d'un "mock" au sein d'un test unitaire (2/2)

Solution3 : sans annotation , avec un appel explicite à Mockito.mock() :

```
import org.mockito.Mockito;

public class UserLoginMockTest {
   private static UserLogin userLogin;

   @BeforeClass
   public static void init(){
      userLogin = Mockito.mock(UserLogin.class);
   }
….
```

Avec (en exemple) l'interface "UserLogin" suivante :

```
public interface UserLogin {
   public boolean verifyLogin(String username,String password);
   public String goodPasswordForUser(String username);
   public void setSize(int size);
   public int getSize();
   public String getAuteur();
   public double getDoubleValue(double x); //return x * 2
}
```

Classe java en exemple pour illustrer le comportement de Mockito

Classe d'implémentation basique de l'interface "UserLogin" :

```
public class UserLoginImpl implements UserLogin {
	private int size=10;  //par défaut

	public boolean verifyLogin(String username, String password) {
		boolean res=false;
		if(password !=null && password.equals("pwd_"+username))
			return true;
		return res;
	}

	public String goodPasswordForUser(String username) {
			return "pwd_"+username;
	}
	public void setSize(int size) { this.size=size;}
	public int getSize() {  return size ; }

	public String getAuteur() {return "didier";  }

	public double getDoubleValue(double x) {  return 2 *x;}
}
```

Comportement par défaut d'un mock (géré par Mockito)

Avec aussi bien
userLogin = Mockito.mock(UserLogin.class); **//interface**
que
userLogin = Mockito.mock(UserLoginImpl.class); **//classe d'implémentation**
tout appel de méthode sur l'objet "mock" géré par mockito retourne une valeur par défaut de type "null" , false , 0 ou 0.0 selon le type de retour :

```
public void displayReturnValues(){
    boolean pwdOk= userLogin.verifyLogin("toto", "pwd_toto");
    System.out.println("pwdOk="+pwdOk);

    String goodPwd= userLogin.goodPasswordForUser("toto");
    System.out.println("goodPwd="+goodPwd);

    String auteur = userLogin.getAuteur();
    System.out.println("auteur="+auteur);

    int taille = userLogin.getSize();
    System.out.println("taille="+taille);

    double  val = userLogin.getDoubleValue(3.2);
    System.out.println("val="+val);
 }
```

Comportement vraie classe:

pwdOk=*true*
goodPwd=*pwd_toto*
auteur=*didier*
taille=*10*
val=*6.4*

Comportement du mock:

pwdOk=**false**
goodPwd=**null**
auteur=**null**
taille=**0**
val=**0.0**

Préciser (forcer) une valeur de retour via Mockito :

```
Mockito.when(userLogin.getSize()).thenReturn(5);
```

int taille = userLogin.getSize();
System.out.println("taille="+taille); → affiche toujours taille=5
userLogin.setSize(20); taille = userLogin.getSize();
System.out.println("taille="+taille); → affiche toujours taille=5 (et pas 20 !)

On peut forcer un **retour d'exception** selon par exemple certaines valeurs en entrée :

```
Mockito.doThrow(new IllegalArgumentException())
        .when(userLogin).setSize(Mockito.eq(-1));
```

Lorsque l'on "mock" une classe (et pas une interface), on peut explicitement demander à Mockito de rétablir le comportement de la véritable classe d'implémentation sur certaines méthodes :

```
Mockito.when(userLogin.getSize()).thenCallRealMethod();
Mockito.doCallRealMethod().when(userLogin).setSize(Mockito.anyInt());
// il existe aussi  Mockito.anyString() , ...
```

userLogin.setSize(20) ; taille = userLogin.getSize();
System.out.println("taille="+taille); → affiche taille=20

1.3. Mockito (as spy)

Comportement par défaut d'un mock initialisé via Mockito.spy()

userLogin = **Mockito.spy**(*UserLogin.class*); //**interface**
==> même comportement qu'après une initialisation via Mockito.mock() mais avec enregistrement des appels effectués pour d'ultérieures vérifications.

```
userLogin = Mockito.spy(new UserLoginImpl()); //classe d'implémntation
```

On obtient alors un **comportement normal** (identique à la classe d'origine) **sur toutes les méthodes sauf sur celles où l'on demande explicitement à redéfinir le comportement** :

```
Mockito.when(userLogin.getSize()).thenReturn(5);
```

Résultats (par défaut) depuis code précédent :
pwdOk=true
goodPwd=pwd_toto
auteur=didier
taille=**5** *(à la place de taille = 20)*
val=6.4

On peut désactiver le comportement d'un setter (ou d'une méthode en void) :

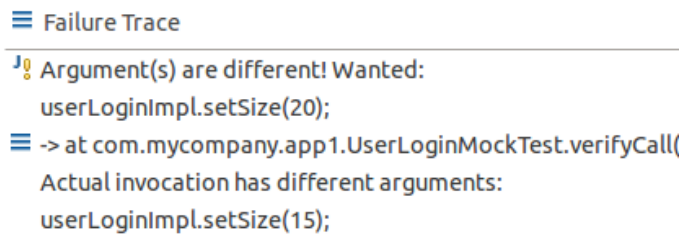
```
Mockito.doNothing().when(userLogin).setSize(Mockito.anyInt());
```

Vérification des appels effectués sur un mock (spy) :

Mockito.spy(...) porte bien son nom lorsque l'on sait que l'**on peut demander à Mockito d'espionner les appels effectués sur un "mock" et vérifier si certaines méthodes ont bien été appelées (avec certaines valeurs attendues de paramètres en entrée)** :

```
@Test
   public void verifyCall(){
      userLogin.setSize(15);  //userLogin.setSize(20);
       //appel habituellement indirect effectué depuis
      // le code caché d'un composant à tester
      // vers le mock (en mode "spy") d'un sous composant
      Mockito.verify(userLogin).setSize(Mockito.eq(20));
   }
```

Avec setSize(15)
au lieu de setSize(20)



Quelques exemples de vérifications via Mockito

```
// vérifie que la méthode m1 a été appelée sur obj,
// avec une String strictement égale à "s1" :
Mockito.verify(obj).m1(Mockito.eq("s1"));
// note : ici, le matcher n'est pas indispensable, la ligne suivante est équivalente :
Mockito.verify(obj).m1("s1");

// vérifie que la méthode m2 n'a jamais été appelée sur l'objet obj :
Mockito.verify(obj, Mockito.never()).m2();

// vérifie que la méthode m3 a été appelée exactement 2 fois sur l'objet obj :
Mockito.verify(obj, Mockito.times(2)).m3();

// idem avec un nombre minimum et maximum d'appels :
Mockito.verify(obj, Mockito.atLeast(3)).m3();
Mockito.verify(obj, Mockito.atMost(10)).m3();

// vérifie que la méthode m4 a été appelée sur obj,
// avec un objet similaire à celui passé en argument :
Mockito.verify(obj).m4(Mockito.refEq(obj2));
```

Quelques "matchers" pour vérifier ou paramétrer les valeurs des paramètres :

Mockito.**eq**(...)	Ègal à ...
Mockito.**refEq**(obj2)	Égal à cet objet
Mockito.**anyString**() , **anyInt**() , **anyFloat**() , ….	Chaîne quelconque , entier quelconque, ..
Mockito.**anyObject**()	Objet quelconque
Mockito.**any**(Class<T> c)	Objet d'un certain type
Mockito.**anyList**()	Toute implémentation de List
Mockito.**argThat**(new MyMatcher())	Vérifiant matcher spécifique
...	

On peut définir de nouveaux "matcher" via des classes qui héritent de **ArgumentMatcher**<T>