Les tests unitaires automatisés

Programmation des tests unitaires

- La programmation des tests et celle de l'application ont des objectifs différents :
  - Code de l'application : il doit implémenter des fonctionnalités qui répondent aux besoins des utilisateurs, il doit être optimisé (performance, sécurité, réactivité, gestion des ressources, etc.), maintenable et évolutif.
  - Code des tests : il doit vérifier que l'application fonctionne comme prévu, il doit être simple et lisible afin que d'être facilement compréhensible, et il doit s'exécuter rapidement.
- Par conséquent, les bonnes pratiques de programmation diffèrent selon que l'on programme l'application ou les tests. Dans les tests :
  - L'**encapsulation** (contrôle de l'accès aux membres de la classe) est moins critique
  - L'**optimisation** du code (sauf sa rapidité d'exécution) n'est pas une priorité, la rapidité, la simplicité et la lisibilité sont prioritaires
  - La **duplication de code** peut être tolérée si elle favorise la lisibilité et l'isolation des tests
  - Les **commentaires** sont moins nécessaires si les tests sont auto-documentés par des noms de méthodes descriptifs et des assertions claires
  - L'**héritage** et le **polymorphisme** ne sont généralement pas des préoccupations importantes
- Parmi les bonnes pratiques communes, on retrouve : respecter les conventions de nommage et utiliser un nommage explicite et approprié, concevoir des objets immuables autant que possible, utiliser l'abstraction pour partager du code, gérer convenablement les exceptions, utiliser les classes prédéfinies de l'API, documenter, utiliser des outils.

Les tests unitaires : comment ?

- **Assurer une couverture des fonctionnalités maximale :** représentativité des données, cas moyens et cas limites, exceptions (voir diapositives 10 à 18)

  Assurer une couverture du code maximale (voir diapositives 19 à 22)

- **Eviter la redondance des cas de test** qui augmente le coût de maintenance des tests, augmente leur temps d'exécution et diminue leur lisibilité
- **Faible longueur des méthodes du code source**, de façon à limiter la complexité des tests structurels
- **Le plus simple est toujours le mieux.**
- **Isolation maximale** de l'environnement extérieur (voir diapositives 29 à 36)
- **Les tests unitaires doivent être séparés des tests d'intégration.**

Les tests unitaires : quand ?

- L'écriture du code et l'écriture des tests sont liées. **L'écriture des tests devrait se faire a priori** (avant ou pendant l'écriture de la fonctionnalité) **car l'écriture des tests a posteriori a un coût plus important** :
  - elle augmente le risque de partager du code non testé, donc bogué,
  - elle nécessite souvent de refactoriser le code de l'application et le coût de refactoring sera plus lourd si le code est déjà partagé et utilisé,
  - elle a un coût supplémentaire de remémoration du contexte des fonctionnalités, avec risque d'oubli de cas limites.

- Pour livrer un patch de production en urgence, il faut choisir entre ignorer les tests et livrer une version potentiellement buggée, ou faire un build du patch complet avec les tests mais augmenter les délais. **Quelle stratégie adopter en cas de date limite de développement intenable ?**
  - Coder la fonctionnalité
  - Tester rapidement à la main ou écrire un minimum de tests (critiques) automatisés
  - Livrer
  - Ecrire les tests unitaires complémentaires après la mise en production, donc a posteriori :(

Les tests unitaires : combien ?

- **Un temps d'exécution des tests trop long peut empêcher leur maintenance et leur utilisation systématique** et être ainsi la cause d'une augmentation de la fréquence des régressions. Quelles sont les solutions ?
  - Etude de la pertinence des tests : duplication de code ? redondances ? tests inutiles ? tests coûteux ?
  - Découpage des tests pour un lancement différencié
  - Optimisation des tests : complexité algorithmique, factorisation, parallélisme

- **Trop de tests tuent les tests** : Les tests unitaires ont un coût d'écriture et un coût de maintenance. Des coûts de maintenance des tests qui deviennent supérieurs à ceux du code ou du code qui devient trop compliqué à cause des tests sont les symptômes de mauvais tests.

- **Etudier le coût du compromis** (coût de mise en place par rapport au gain) : quelle est l'augmentation de la qualité de l'application, de la qualité du code, de la documentation, de la productivité par rapport à l'augmentation du temps de développement et de maintenance ?

Pertinence, indépendance et isolation

Les trois caractéristiques essentielles des tests unitaires sont :

- **Pertinence** : test d'un comportement à la fois, tout en couvrant complètement les fonctionnalités

  Un cas de test est plus facile à comprendre s'il se concentre sur un seul aspect du comportement. La couverture complète des fonctionnalités est assurée par l'ensemble des cas de test du jeu de test. Les tests doivent être conçus par une appoche à la fois fonctionnelle et structurelle.

- **Indépendance** : indépendance des tests entre eux

  L'état laissé par un test ne doit pas affecter un autre test, les tests doivent pouvoir s'exécuter dans n'importe quel ordre.

  L'annotation @BeforeEach est utilisée en JUnit pour garantir l'indépendance des méthodes de test, elle permet de réinitialiser l'état entre les tests de sorte que chaque test ne dépende pas des résultats d'un test précédent.

- **Isolation :** indépendance des composants de l'application entre eux

  Le test est réalisé de façon indépendante des autres composants du programme.

  Cela implique souvent de simuler les dépendances externes (appels à une base de données, services externes, interfaces réseau) en utilisant des objets factices (mocks). L'objectif est d'isoler l'unité de code à tester sans dépendre d'éléments extérieurs et assurer ainsi la fiablilité des tests.

La sélection de jeux de test pertinents

- Problématique

  impossibilité d'exécuter un programme sur toutes ses entrées possibles

  => nécessité de sélectionner des jeux de tests pertinents

- Les cas de tests doivent couvrir toutes les types de situations possibles :
  - les **cas nominaux** représentent les scénarios d'utilisation standard et attendue de l'unité de code, ils sont faciles à imaginer,
  - les **cas alternatifs** couvrent les scénarios imprévus ou exceptionnels : comportement inattendu de l'utilisateur, erreur de l'utilisateur, problèmes réseau, ressource ou service indisponible, valeurs hors limites, etc.

- Les cas de test doivent permettre de communiquer efficacement sur le fonctionnement du code test.

- Les cas de test choisis doivent assurer la **couverture fonctionnelle des exigences** et la **couverture structurelle du code à tester**.

Jeu de test et cas de test

- Un **cas de test** = données d'entrée + résultats attendus + objectif de test

Un cas de test pour la recherche du minimum d'une liste d'entiers

Objectif	Entrée	Sortie
Trouver le minimum lorsqu'il est placé au milieu d'une liste non vide sans doublons	[4, 1, 8, 12]	1

- Un **jeu de test** = ensemble de cas de test
  - Objectif du jeu de test : détecter **un maximum de défauts avec un minimum de tests**
  - Nombre de cas de test : déterminé selon les critères de **couverture** fonctionnelle et structurelle
  - Caractéristiques du jeu de test
    - praticable : qui peut être exécuté en temps fini et raisonnable
    - non redondant : plusieurs cas de test ne doivent pas cibler la même faute
    - représentatif : susceptible de trouver un grand nombre de fautes

Exemple de jeu de test

- Soit la méthode à tester **static int [] triSansDoublons(int [] tab)** qui trie le tableau tab donné en paramètre sans conserver les doublons.

Un jeu de test pour triSansDoublons formé de 6 cas de test

Objectif	Entrée	Sortie
Tester avec un tableau vide	tab=[]	[]
Tester avec un tableau d'un seul élément	tab = [10]	[10]
Tester avec un tableau où tous les éléments sont identiques	tab=[3,3,3]	[3]
Tester avec un tableau déjà trié	tab=[1,2,2,3]	[1,2,3]
Tester avec un tableau non vide non trié sans doublons et avec des valeurs négatives	tab = [1,0,-2,-5]	[-5,-2,0,1]
Tester avec un tableau non vide non trié avec doublons consécutifs et non consécutifs	tab = [1,20,8,4,4,8,0,8,-4]	[-4,0,1,4,8,20]

Couverture fonctionnelle des exigences

La méthode est basée sur un partitionnement par équivalence et une analyse des valeurs aux limites :

- Définir les **classes d'équivalence fonctionnelle** sur les domaines de valeur des entrées
- S'il y a plusieurs entrées, faire le **produit cartésien des classes** obtenues.
- Choisir **une valeur dans chaque classe**, cette valeur est alors suffisante pour tester le comportement associé à la classe
- Choisir des **valeurs aux limites**

Couverture fonctionnelle des exigences

Partitionnement en classes d'équivalence

- Déterminer le domaine des valeurs d'entrée à partir des interfaces (spécification et programme).
- Partitionner le domaine en classes d'équivalence qui correspondent aux différents objectifs de test, de telle sorte que :
  - toutes les entrées d'une même classe d'équivalence génèrent le même comportement (c'est-à-dire détectent le même défaut)
  - les objectifs de test couvrent les exigences fonctionnelles

Exemple :

2 classes d'équivalence pour l'entrée *x* pour le calcul de *racine carrée de l'entier x*

- $x < 0$ classe d'équivalence pour les valeurs de x invalides (résultat indéfini)
- $x \geq 0$ classe d'équivalence pour les valeurs de x valides (résultat racine carrée de x)

Les entrées d'un jeu de test pourrait être : {-1,2}

Couverture fonctionnelle des exigences

- Lorsqu'il y a plusieurs entrées, il faut considérer le **produit cartésien des classes** d'équivalence obtenues.

  Exemple :

  Soit le calcul de la *valeur absolue du produit de deux entiers a et b*. Le résultat dépend du signe du produit, qui dépend lui-même des signes de a et de b.

  - 2 classes d'équivalence pour a : a < 0 et a ≥ 0
  - 2 classes d'équivalence pour b : b < 0 et b ≥ 0
  - => 4 = 2x2 classes pour le produit cartésien

	a	b
classe 1	a < 0	b < 0
classe 2	a < 0	b ≥ 0
classe 3	a ≥ 0	b < 0
classe 4	a ≥ 0	b ≥ 0

Couverture fonctionnelle des exigences

- Choisir ensuite **une valeur dans chaque classe d'équivalence fonctionnelle**, cette valeur est alors suffisante pour tester le comportement associé à la classe.

	a		b		résultat attendu
classe 1	a < 0	***-4***	b < 0	***-2***	***8***
classe 2	a < 0	***-1***	b ≥ 0	***5***	***5***
classe 3	a ≥ 0	***3***	b < 0	***-3***	***9***
classe 4	a ≥ 0	***2***	b ≥ 0	***4***	***8***

4 classes
= 4 cas de test

Couverture fonctionnelle des exigences

- Ajouter des **tests aux limites**

  On ajoute des valeurs limites, c'est-à-dire des valeurs aux bornes ou hors bornes de chaque classe : condition de boucle, valeurs très grandes, valeurs non valides, présence de doublons, absence de valeurs.

  Exemple :

  Pour le calcul de la *valeur absolue du produit de deux entiers a et b*, on doit aussi considérer le cas où a et b sont nuls et le cas, s'il est plausible, où le résultat dépasse la capacité des entiers (Integer.MAX_VALUE et Integer.MIN_VALUE).

	a		b		attendu
classe 5	a = 0	***0***	b > 0	***5***	***0***
classe 6	a < 0	***-15***	b = 0	***0***	***0***
classe 7	a = 0	***0***	b = 0	***0***	***0***
classe 8	a max	***Integer.MAX_VALUE***	b > 0	***2***	***dépassement de capacité***
classe 9	a < 0	***-1***	b min	***Integer.MIN_VALUE***	***Integer.MAX_VALUE***
classe 10	a > 0	***1***	b max	***Integer.MAX_VALUE***	***Integer.MAX_VALUE***

Couverture fonctionnelle des exigences

10 cas de test fonctionnels au total

pour le calcul de la valeur absolue du produit de deux entiers a et b

	a		b		attendu
classe 1	a < 0	***-4***	b < 0	***-2***	***8***
classe 2	a < 0	***-1***	b ≥ 0	***5***	***5***
classe 3	a ≥ 0	***3***	b < 0	***-3***	***9***
classe 4	a ≥ 0	***2***	b ≥ 0	***4***	***8***
classe 5	a = 0	***0***	b > 0	***5***	***0***
classe 6	a < 0	***-15***	b = 0	***0***	***0***
classe 7	a = 0	***0***	b = 0	***0***	***0***
classe 8	a max	***Integer.MAX_VALUE***	b > 0	***2***	***dépassement***
classe 9	a < 0	***-1***	b min	***Integer.MIN_VALUE***	***Integer.MAX_VALUE***
classe 10	a > 0	***1***	b max	***Integer.MAX_VALUE***	***Integer.MAX_VALUE***

Puisque le produit ne dépend pas de l'ordre des opérandes, on pourrait limiter ce jeu de test à 8 cas en considérant que :

- les classes 2 et 3 sont équivalentes = cas où un entier est positif et l'autre négatif
- les classes 5 et 6 sont équivalentes = cas où un seul des deux entiers est nul

Problématique de la couverture fonctionnelle

- Le nombre de cas de test fonctionnels est très élevé, surtout lorsqu'on veut garantir la robustesse du code en ce qui concerne la gestion des valeurs limites.

- De plus, à mesure que le nombre d'entrées augmente, le risque d'explosion combinatoire lié au produit cartésien des classes d'équivalence des entrées entraîne un nombre exponentiel de cas de test.

  Exemple : Avec 3 classes d'équivalence par entrée

  - pour 2 entrées, il y a 3 x 3 = 9 classes pour le produit cartésien
  - pour 3 entrées, il y a 3 x 3 x 3 = 27 classes pour le produit cartésien
  - pour 4 entrées, il y a 3 x 3 x 3 x 3 = 81 classes pour le produit cartésien

  auxquelles il faut rajouter les classes résultant de l'analyse des valeurs limites !

- Pour réduire le nombre de cas de tests fonctionnels tout en assurant une couverture adéquate, il est possible de :
  - combiner les classes d'équivalence de chaque entrée par paires (**test par paires**) plutôt qu'en utilisant le produit cartésien
  - prioritiser les cas de test et ne conserver que les tests les plus critiques ou susceptibles de provoquer des erreurs.

Problématique de la couverture fonctionnelle

- Les tests par paires reposent sur l'observation que la plupart des défauts sont causés par l'interaction d'au plus deux entrées, donc tester toutes les paires de valeurs est suffisant. Le test par paires réduit considérablement le nombre de tests nécessaires par rapport au nombre de combinaisons obtenu par le produit cartésien des classe d'équivalence. Le calcul du nombre minimal de cas de test par paires est complexe et dépend de l'algorithme utilisé.

  Exemple

  - Avec 3 entrées dont chacune a 2 ou 3 valeurs/classes possibles : x = {1,2}, y = {A,B,C}, z = {3,4}, on a 12 = 2x3x2 combinaisons possibles obtenues par produit cartésien (1A3 1A4 1B3 1B4 1C3 1C4 2A3 2A4 2B3 2B4 2C3 2C4) alors que 6 combinaisons suffisent pour couvrir tous les couples avec un test par paires

	1A	1B	1C	2A	2B	2C	A3	A4	B3	B4	C3	C4	13	14	23	24
1A3	X						X						X			
1B4		X								X				X		
1C3			X								X		X			
2A4				X				X								X
2B3					X				X						X	
2C4						X						X				X

  - Avec 3 entrées de 3 valeurs/classes chacune, on a 27 = 3x3x3 combinaisons possibles obtenues par produit cartésien alors que 9 combinaisons suffisent pour couvrir tous les couples avec un test par paires.

Entrées du test fonctionnel

- Dans le cas de la POO, les entrées prises en compte pour les tests d'une méthode non statique ne se limitent pas uniquement à ses paramètres mais dépendent également de l'état de l'instance.

  Exemple :

  Il faut **deux entrées (solde et montant)** pour tester la méthode retirer de CompteBancaire :

```
public class CompteBancaire {
    private double solde;
    public CompteBancaire( double soldeInitial) { this.solde = soldeInitial; }
    public boolean retirer ( double montant) {
        if ( montant > solde ) { // solde insuffisant
            return false;
        }
        solde -= montant;
        return true;
    }
}
```

Couverture structurelle du code

- La couverture de code est une mesure utilisée pour déterminer quelles parties du code à tester ont été exécutées lors des tests.
- La vérification de la couverture du code peut être assistée par des outils tels que **EclEmma** (ou JaCoCo).
- Il existe différents critères de couverture de code
  - Couverture des appels de fonctions/méthodes : Chaque fonction/méthode doit être appelée au moins une fois.
  - Couverture des instructions : Chaque instruction doit être exécutée au moins une fois.
  - Couverture des branches : Chaque branche doit être traversée au moins une fois, donc pour chaque condition on teste le cas où la condition est vraie et où elle est fausse. Ce critère est plus fort que la couverture des instructions.

EclEmma

  - Couverture des conditions (et des branches) : Chaque sous-expression d'une condition composée doit être évaluée à la fois comme vrai et fausse. Ce critère est plus fort que la couverture des branches, mais ne garantit pas que toutes les combinaisons de sous-conditions ont été testées.
  - Couverture des boucles : chaque boucle doit être exécutée, si possible, 0 fois, 1 fois et plusieurs fois

Couverture structurelle du code

```
public static int bizarre( int a, int b) {
    int c = 0 ;
    if ( a > 0 && b > 0 )
        c = 1 ;
    return c ;
}
```

- 1 cas de test suffit pour couvrir toutes les instructions :
  cas où a > 0 et b > 0 (a=1, b=2) resultat attendu : 1

- 2 cas de test sont nécessaires pour couvrir toutes les branches :
  cas où condition vraie (a > 0 et b > 0) (a=1, b=2) resultat attendu : 1
  cas où condition fausse (par exemple a <= 0 et b > 0) (a=0, b=2) resultat attendu : 0

- 3 cas de test sont nécessaires pour couvrir toutes les conditions et les branches :
  cas où a > 0 et b > 0 (a=1, b=2) resultat attendu : 1
  cas où a <= 0 et b > 0 (a=0, b=2) resultat attendu : 0
  cas où a > 0 et b < 0 (a=1, b=-2) resultat attendu : 0
  Le cas où a < 0 && b < 0 est inutile, puisque les autres cas permettent déjà d'avoir la condition (a > 0 && b > 0) vraie et fausse, (a > 0) vrai et faux et (b > 0) vrai et faux.

Couverture structurelle du code

```
public static boolean contient( int[] tab,
            int nbElements, int element) {
   int i = 0 ;
   while ( i < nbElements) {
      if ( tab[i] == element ) {
         return true ;
      }
      i++ ;
   }
   return false ;
}
```

Méthode qui renvoie vrai si l'entier *element* est présent parmi les *nbElements* premiers éléments du tableau d'entiers *tab,* et faux sinon.

Soient les 3 cas de test suivants :

tab	nbElements	element	résultat attendu	objectif
[]	0	8	false	tableau vide
[5, 9, 2, 4]	3	1	false	élément absent
[6, 4, 5, 7, 8]	5	4	true	élément présent

couverture des instructions : oui

couverture des conditions (et branches) : oui

couverture de la boucle : oui (0 et plusieurs fois)

On pourrait rajouter le cas de test ([5],1,5) pour obtenir une seule fois l'exécution de la boucle

Mais ces tests sont-ils suffisants ? Et si nbElements > tab.length ?

Couverture structurelle du code

- Autres exemples :

```
y++ ;
if ( x==y && z>y )
   x-- ;
```

couverture des instructions => (x=2,y=1,z=4)
couverture des branches => + (x=3,y=2,z=2)
couverture des conditions et branches => + (x=2,y=2,z=2)

données en entrée	(x==y)*	(z>y)*	(x==y && z>y)*
(x=2,y=1,z=4)	vrai	vrai	vrai
(x=3,y=2,z=2)	vrai	faux	faux
(x=2,y=2,z=2)	faux	faux	faux

* après exécution de y++;

```
y++ ;
if ( x==y && z>y && z>0 )
   x-- ;
```

couverture des branches => (x=2,y=1,z=4) + (x=3,y=2,z=2)
couverture des sous-conditions => (x=4,y=4,z=-2)

données en entrée	(x==y)*	(z>y)*	(z>0)	(x==y && z>y && z>0)*
(x=2,y=1,z=4)	vrai	vrai	vrai	vrai
(x=3,y=2,z=2)	vrai	faux	vrai	faux
(x=4,y=4,z=-2)	faux	faux	faux	faux

* après exécution de y++;

Un jeu de test pertinent

Exercice :

Donnez un jeu de test qui assure la couverture fonctionnelle et structurelle de la méthode puissance.

```
public static double puissance( int a, int b) {
        int x = abs(b);
        double y = 1.0;
        while ( x != 0 ) {
                y *= a ;
                x-- ;
        }
        if ( b < 0)
             y = 1.0 / y;
        return y;
}
```

La méthode calcule a^b

Un jeu de test pertinent

```
public static double puissance(
                              int a, int b) {
        int x = abs(b);
        double y = 1.0;
        while ( x != 0 ) {
                y *= a ;
                x-- ;
        }
        if ( b < 0)
                y = 1.0 / y;
        return y;
}
```

Couverture structurelle

- Le cas de test (a=4,b=-2) couvre toutes les instructions. Pour ce cas de test, la boucle est exécutée 2 fois et la condition (b < 0) est vraie.
- Comme le jeu de test réduit au cas précédent (a=4,b=-2) ne couvre pas le cas où la condition (b < 0) est fausse , il faut lui ajouter un cas de test, par exemple (a=5,b=1).
- Comme le jeu de test formé des cas (a=4,b=-2) et (a=5,b=1) ne couvre pas le cas où la boucle n'est pas exécutée, il faut lui ajouter un autre cas de test, par exemple (a=12,b=0).

8 cas de test

données d'entrée	résultat attendu	objectif du cas de test
(a=4,b=-2)	4^{-2} = 0,0625	couverture des instructions + a>0&b<0
(a=5,b=1)	5^{1} = 5.0	couverture des conditions + a>0&b>0
(a=12,b=0)	12^{0} = 1.0	couverture des boucles + b=0
(a=-4,b=3)	$(-4)^{3}$ = -64.0	a<0&b>0
(a=-4,b=-1)	$(-4)^{-1}$ = -0.25	a<0&b<0
(a=0,b=10)	0^{10} = 0.0	a=0&b>0
(a=0,b=-8)	0^{-8} = Infinity	a=0&b<0
(a=0,b=0)	0^{0} = 1.0	a=0&b=0

Couverture fonctionnelle

(la méthode calcule a^b)

- 2 classes d'équivalence pour a : a ≥ 0, a < 0
- 2 classes d'équivalence pour b : b ≥ 0, b < 0
- donc 4 classes d'équivalence pour (a,b) : a≥0 et b>0, a≥0 et b<0, a<0 et b≥0, a<0 et b<0
- 5 cas limites : a=0 et b>0, a=0 et b<0, a=0 et b=0, a!=0 et b=0, b=1

Retour sur les assertions JUnit5

Pour écrire les test unitaires avec Junit5, vous devez utiliser les assertions de manière appropriée et choisir l'assertion la plus adaptée à chaque situation.

assertEquals ou assertTrue

- Si vous connaissez la valeur exacte que la variable ou la méthode doit retourner, vous devez vérifier que cette valeur est correcte avec assertEquals plutôt qu'avec assertTrue :

```
assertEquals( 3, x)                    plutôt que assertTrue ( x == 3) ;
assertEquals( "[4]", x.toString() )    plutôt que assertTrue ( x.toString().equals("[4]")) ;
```

- En effet, en cas d'erreur le message d'erreur sera plus explicite :

```
Expected 3 but was 2                   au lieu de : Expected true but was false
Expected "[4]" but was "[]"            au lieu de : Expected true but was false
```

- assertEquals(a, b) est sécurisé pour les valeurs nulles, assertTrue(a.equals(b)) non.

assertEquals et assertNotEquals

- Une fois que vous avez vérifié que la valeur est correcte avec assertEquals, vérifier ensuite qu'elle n'est pas incorrecte avec assertNotEquals est inutile car cela n'apporte aucune information supplémentaire

```
assertEquals( 5, resultat) ;
assertNotEquals( 4, resultat) ; // inutile : si le résultat vaut 5, il est évident qu'il ne vaut pas 4
```

Retour sur les assertions JUnit5

assertEquals ou assertIterableEquals

- assertIterableEquals vérifie que deux collections (ou itérables) contiennent les mêmes éléments dans le même ordre, assertEquals vérifie que deux objets sont equals

```
ArrayList<Integer> l1 = new ArrayList<>(Arrays.asList(1,2,3));
LinkedList<Integer> l2 = new LinkedList<>(Arrays.asList(1,2,3));
TreeSet<Integer> e = new TreeSet<>(Arrays.asList(1,2,3));
assertEquals( l1, l2); // réussit car l1 et l2 sont des List
assertIterableEquals( l1, l2);
assertNotEquals( l1, e); // réussit car l1 est une ArrayList et e est un TreeSet
assertIterableEquals( l1, e);
```

- En cas d'échec assertIterableEquals fournit un message d'erreur détaillé en précisant l'élément différent avec son indice :

```
ArrayList<Integer> l1 = new ArrayList<>(Arrays.asList(1,2,3,4));
ArrayList<Integer> l2 = new ArrayList<>(Arrays.asList(1,3,2,5));
assertIterableEquals(l1, l2);
=> AssertionFailedError: iterable contents differ at index [1], expected: <2> but was: <3>
assertEquals(l1,l2) ; => AssertionFailedError: expected: <[1, 2, 3, 4]> but was: <[1, 3, 2, 5]>
```

Retour sur les assertions JUnit5

assertEquals ou assertIterableEquals

- Notez que assertEquals et assertIterableEquals vérifient equals en profondeur

```
List<List<Integer>> ll1 = new ArrayList<>( Arrays.asList( Arrays.asList(1,2,3), Arrays.asList(4,5)));
List<List<Integer>> ll2 = new LinkedList<>( Arrays.asList(Arrays.asList(1,2,3),Arrays.asList(4,5)));
LinkedHashSet<List<Integer>> el2 =
            new LinkedHashSet<>(Arrays.asList(Arrays.asList(1,2,3),Arrays.asList(4,5)));
assertEquals( ll1, ll2);          assertIterableEquals( ll1, ll2);
assertNotEquals( ll1, el2);       assertIterableEquals( ll1, el2);
```

assertDoesNotThrow()

pour vérifier seulement que l'exécution ne plante pas, lorsque le résultat n'est pas prévisible avec certitude ou n'est pas important, ou bien pour tester une méthode qui ne retourne rien (void).

Retour sur les assertions JUnit5

assertEquals ou assertArrayEquals

- Notez que assertEquals et assertArrayEquals vérifient equals en profondeur

```
/* tableaux imbriqués */
Object[] tab1 = new Object[] { "un", null, new Integer[] {2,3}};
Object[] tab2 = new Object[] { "un", null, new Integer[] {2,3} };
assertFalse(Arrays.equals(tab1, tab2));
assertTrue(Arrays.deepEquals(tab1, tab2));
assertArrayEquals(tab1, tab2);
/* tableaux multidimensionnels */
int[][] tab3 = { { 1, 2, 3 }, { 4, 5, 6, 9 }, { 7 } };
int[][] tab4 = { { 1, 2, 3 }, { 4, 5, 6, 9 }, { 7 } };
assertFalse(Arrays.equals(tab3, tab4));
assertTrue(Arrays.deepEquals(tab3, tab4));
assertArrayEquals(tab3, tab4);
class A {
    private final int a;
    public A(int n) { a = n; }
    @Override public boolean equals(Object o) { return o instanceof A && a == ((A)o).a; }
}
A [][] tab5 = { { new A(1), new A(2) }, { new A(3) } };
A [][] tab6 = { { new A(1), new A(2) }, { new A(3) } };
assertFalse(Arrays.equals(tab5, tab6));
assertTrue(Arrays.deepEquals(tab5, tab6));
assertArrayEquals(tab5, tab6);
```

Tests unitaires en isolation

- **Problème**

  Comment tester unitairement une classe qui dépend d'autres classes ?

- **Solution**

  Supprimer la dépendance et utiliser des doublures (mocks).

- **Dépendance entre classes**

  Une classe A dépend d'une classe B si dans la définition de A apparaît l'identificateur B, autrement dit si pour créer un A, on a besoin d'un B, ou encore si une des conditions suivantes est vérifiée :

  - A possède un attribut de type B
  - une méthode de A appelle une méthode de B
  - A est de type B
  - A dépend de C qui dépend de B

Si la classe A dépend de la classe B, tout changement dans B peut impacter A.

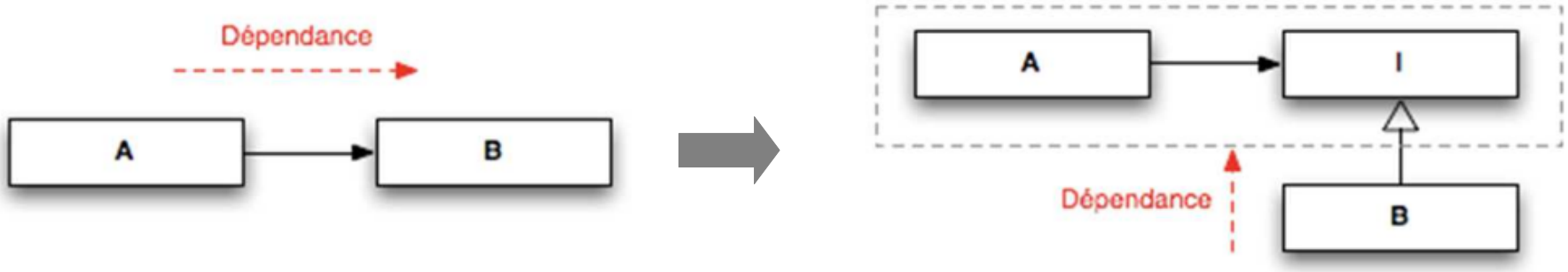
Réalisation des tests unitaires en isolation

Soit une classe A qui dépend d'une classe B. Pour pouvoir tester A indépendamment de B, il faut d'abord inverser la dépendance

- **Inversion de dépendances**

Technique utilisée pour réduire le couplage entre les classes A et B en introduisant des abstractions : on fait dépendre A d'une interface I de B qui contient toutes les méthodes que A peut appeler sur B, on indique que B implémente cette abstraction et on remplace dans A toutes les références au type B par des références à l'interface I.

A l'exécution, le code va créer une instance de A en lui fournissant une instance de B qui satisfait le contrat attendu donné par I, donc A ne dépendra plus de B qu'à l'exécution.



Réalisation des tests unitaires en isolation

Exemple d'inversion des dépendances

```
public class B { ...
      public int mB() { ... }
}
```

```
public class A {
      private B b;
      public A () {
            this.b = new B();
      }
      public int mA() {
            int k = b.mB(); ...
      }
}
```

A dépend de la classe B :
si B change, A doit être modifiée.

```
public interface I { int mB(); }
```

```
public class B implements I { ... public int mB() { ... } }
```

```
public class A {
      private I b;
      public A (I b) { this.b = b; }
      public int mA() { int k = b.mB(); ... }
}
```

```
public class Main {
      public static void main( String [] args) {
            I instance ;
            if (condition) instance  = new B() ;
            else instance = new C() ;
            A a = new A(instance) ;
      }
}
```

A dépend d'une implémentation I :
A peut être instanciée avec n'importe quelle implémentation de I, et ce choix peut être fait dynamiquement, à l'exécution, et non à la compilation.

Réalisation des tests unitaires en isolation

Une fois la dépendance inversée, il faut injecter dans A une doublure de B, plutôt qu'un objet B.

- **Injection de dépendances**

  Technique qui permet de fournir les dépendances de B à la classe A au lieu de les créer directement dans A : la création de l'objet B est faite à l'extérieur de A et l'instance de B est injectée (par constructeur, par "setter", ou par interface) dans la classe A ; il y a séparation de la création de B et de l'utilisation de B. C'est une forme d'inversion de contrôle : A ne contrôle plus l'instanciation de B.

- **Création d'une doublure de B**

  L'utilisation d'une doublure de B permet de se concentrer sur le test de A sans avoir à se préoccuper de B. En effet, il peut être très difficile de trouver un bug qui pourrait provenir du code de A aussi bien que du code de B. Le but n'est pas de tester B qui fera elle-même l'objet de tests unitaires spécifiques.

  La doublure de B implémente l'interface I (ajoutée pour l'inversion de dépendances) et simule le comportement des objets B pour la durée des tests de façon maîtrisée (en particulier sans bug). **C'est une doublure de B est réellement injectée dans A.**

Tests unitaires en isolation : exemple

- Exemple : Tester unitairement une classe A qui dépend d'une classe B

```
public class A {
      private B b;
      public A () {
            this.b = new B();
      }
      public int mA() {
            int k = b.mB(); ...
      }
}
```

```
public class B { ...
      public int mB() { ... }
}
```

Les bugs de mB peuvent générer des bugs dans mA.

```
public class A {
      private I b;
      public A (I b) { this.b = b; }
      public mA() { b.mB(); }
}
```

inversion des dépendances

```
public interface I { void mB(); }
```

```
public class B implements I { public int mB() { ... } }
```

```
public class DoublureB implements I {
      public int mB() { return 10 ; }
}
```

création d'une doublure

```
public class Test {
      public static void main(String [ ] arg) {
            DoublureB doublure = new DoublureB();
            A a = new A( doublure);
            // si 165 est le résultat de mA quand k vaut 10
            assert a.mA() == 165 ;
      }
}
```

injection d'une doublure

Les tests de mA de A sont devenus indépendants des éventuels bugs de mB de B.

Tests unitaires en isolation

- Le principe de limiter les responsabilités d'un objet, de façon à faciliter la réutilisation, implique qu'un objet a souvent besoin d'autres objets pour réaliser ses tâches, qui ont eux aussi des dépendances vers d'autres objets. **Les dépendances entre objets forment ainsi un graphe complexe** qui rend les tests difficiles et empêche l'isolation des tests.
- Seules les dépendances directes ont besoin d'être remplacées par des doublures pour les tests unitaires.
- **Pour être testable, le programme doit être conçu avec des objets faiblement couplés** : le code ne doit pas avoir trop de dépendances et les dépendances doivent pouvoir être injectées. Penser aux tests avant de coder favorise ainsi une bonne conception.
- Le terme générique de **doublure** désigne un objet utilisé à la place d'un vrai objet pour réaliser des tests unitaires ou remplacer un objet qui n'est pas encore programmé (phase de codage).
- Les doublures peuvent être codées manuellement ou générées dynamiquement par un environnement à partir d'une spécification de leur comportement. L'intérêt d'un environnement de création de doublures est qu'il rend la création de doublures rapide et fiable. **Mockito** est un exemple d'environnement, il peut être couplé avec JUnit.

Exemples de doublures pour le test unitaire

```java
public interface CompteBancaire {
        void depot(double montant);
        void retrait(double montant);
        double solde();
}
```

```java
public class CompteDepot implements CompteBancaire {
        /* code de CompteDeDepot */
}
```

```java
public class GestionnaireDeCompte {
    private CompteBancaire leCompte;
    private LocalDate dateDO;
    public GestionnaireDeCompte(CompteBancaire c) {
        this.leCompte = c;  this.dateDO = null; }
    public void deposer(double montant) {
      leCompte.depot( montant); }
    public boolean retirer(double montant) {
      if ( leCompte.solde() < montant ) return false;
      leCompte.retrait( montant);  return true; }
    public double consulter() { return leCompte.solde(); }
    public LocalDate dateDerniereOp() { return dateDO; }
    public void miseAJourDateDerniereOp() {
      dateDO = LocalDate.now(); }
}
```

Test de la méthode miseAJourDateDerniereOp de GestionnaireDeCompte en utilisant une doublure de type Fantome pour simuler un CompteBancaire :

```java
class GestionnaireDeCompteTest {
   class Fantome implements CompteBancaire {
      @Override public void depot(double montant) {
          throw new UnsupportedOperationException();  }
      @Override public void retrait(double montant) {
          throw new UnsupportedOperationException(); }
      @Override public double solde() {
          throw new UnsupportedOperationException(); }
   }
   @Test
   void testDateAvecUnFantome() {
    // La methode miseAJourDateDerniereOp n'utilise pas
    // leCompte. Il faut utiliser un fantome pour la tester.
      Fantome doublure = new Fantome();
      GestionnaireDeCompte g = new
                          GestionnaireDeCompte(doublure);
      assertNull( g.dateDerniereOp());
      g.miseAJourDateDerniereOp());
      assertEquals(LocalDate.now(), g.dateDerniereOp());
   }
}
```

Exemples de doublures pour le test unitaire

```
class Bouchon implements CompteBancaire {
	private String operation = "";
	public int nbRetraits = 0;
	@Override
	public void depot(double montant) {
		operation = "depot " + montant;
	}
	@Override
	public void retrait(double montant) {
		operation = "retrait " + montant;
		nbRetraits++;
	}
	public double solde() {
		operation = "solde";
		return 100.0;
	}
	public String operation() {
		String resultat = operation;
		operation = "";
		return resultat;
	}
}
```

Test des méthodes deposer, retirer et consulter de GestionnaireDeCompte en utilisant une doublure de type bouchon pour simuler un CompteBancaire :

```
@Test
void testDeposerRetirerConsulterAvecUnBouchon() {
	// Les methodes deposer, retirer et solde utilisent leCompte.
	// Pour les tester, on utilise un bouchon avec un
	/ /comportement simplifie pour leCompte.
	Bouchon doublure = new Bouchon();
	GestionnaireDeCompte g =
			new GestionnaireDeCompte(doublure);

	g.deposer( 10000.0);
	assertEquals( "depot 10000.0", doublure.operation());

	assertTrue( g.retirer(10.0));
	assertEquals( "retrait 10.0", doublure.operation());

	assertFalse( g.retirer(500.0));
	assertEquals( "solde", doublure.operation());

	assertEquals( 100.0, g.consulter());
	assertEquals( "solde", doublure.operation());

	assertEquals( 1, doublure.nbRetraits);
}
```

Environnement de création de doublures

- Il existe plusieurs API permettant de créer et utiliser des doublures de test : EasyMock, MockMaker, **Mockito**, PowerMockito, etc.

- Les doublures sont des objets factices qui simulent le comportement des objets réels afin de faciliter l'écriture des tests en isolation.

Mockito

- Site originel : https://site.mockito.org/
- Utilisation
  - Télécharger la bibliothèque mockito-all-x.x.x.jar à partir du site originel ou du site https://code.google.com/archive/p/mockito/downloads
  - Ajouter la bibliothèque au BuildPath du projet Eclipse
  - Pour gérer les mocks, il suffit d'utiliser essentiellement des méthodes statiques de la classe org.mockito.Mockito. Ainsi, Mockito permet en particulier de créer une doublure d'une classe concrète ou d'une interface en appelant la méthode **mock** sur la classe/interface.
- Limitations de Mockito
  - nécessite une version de Java supérieure ou égale à 1.5
  - impossible de créer une doublure pour une classe ou une méthode finale
  - impossible de créer une doublure pour une classe anonyme
  - impossible de créer une doublure d'une méthode statique ou privée
  - impossible de créer des doublures des méthodes equals() et hashcode() (car Mockito les utilise en interne, par exemple pour vérifier les valeurs en arguments avec equals())

Exemple de création d'une doublure avec Mockito

```
public class MaClasse {
   private Environnement env;
   public MaClasse(Environnement e) {
        env = e;
   }
   public Environnement environnement() {
        return env;
   }
}
```

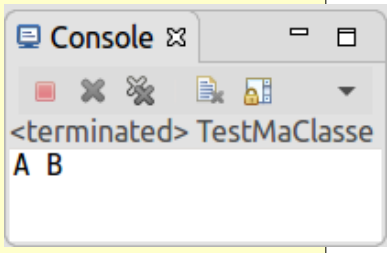
```
public class Environnement
        implements Iterable<Forme> {
   public int largeur() { ... }
   public int hauteur() { ... }
   public Iterator<Forme> iterator() { … }
}
```

```
public class Forme {
   String nom;
   public Forme( String n) { nom = n; }
   @Override
   public String toString() { return nom; }
}
```

```
public class TestMaClasse {
   /* Creation d'une doublure de l'environnement */
   private static final Environnement doublure
                                        = mock(Environnement.class);

   private static void init() {
      final int largeur = 300;
      final int hauteur = 200;
      final List<Forme> formes = new ArrayList<Forme>();
      formes.add(new Forme("A")); formes.add(new Forme("B"));

      /* Définition du comportement du mock de l'environnement */
      when(doublure.largeur()).thenReturn(largeur);
      when(doublure.hauteur()).thenReturn(hauteur);
      when(doublure.iterator()).thenAnswer(
         new Answer<Iterator<Forme>>() {
            @Override
            public Iterator<Forme> answer(InvocationOnMock
                                        invocation) throws Throwable {
               return formes.iterator();
      } });
   }
   public static void main(String[] args) {
      init();
      MaClasse c = new MaClasse(doublure);
      for ( Forme f : c.environnement())
         System.out.print( f+ " ");
   }
}
```



Doublure d'une classe concrète

- Dès qu'un mock est créé, il est utilisable avec un comportement par défaut : ses méthodes sans valeur de retour ont un corps vide, celles avec valeur de retour ont une valeur par défaut (null, 0 ou false).
- Les méthodes when() et thenReturn() précisent le comportement du mock en écrasant le comportement par défaut.
- Le code ci-contre de MonTest.test1() crée une doublure d'objet de type Personne dont le comportement initial de la méthode bonjour() est de toujours renvoyer la même valeur "Bonjour".

```java
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class MonTest {
	private Personne p;
	@Test
	void test1() {
		/* Creation de la doublure et tests */
		p = mock(Personne.class);
		assertNull( p.bonjour());
		p.changeBonjour("Bonjour");
		assertNull( p.bonjour());

		/* Definition du comportement de la doublure et tests */
		when(p.bonjour()).thenReturn("Bonjour");
		p.changeBonjour("Hey");
		assertEquals( "Bonjour", p.bonjour());

		/* Restauration du comportement normal et tests*/
		when(p.bonjour()).thenCallRealMethod();
		assertNull( p.bonjour());
		doCallRealMethod().when(p).changeBonjour(anyString());
		p.changeBonjour("Hey");
		assertEquals( "Hey", p.bonjour());
	}
}
```

```java
public class Personne {
	private String bonjour;
	public Personne() {
		bonjour = null;
	}
	public String bonjour() {
		return bonjour;
	}
	void changeBonjour(String s) {
		bonjour = s;
	}
}
```

La syntaxe when ne doit pas être utilisée avec les méthodes void.

Doublure d'une méthode void

- Il existe plusieurs solutions pour faire une doublure d'une méthode void :
  - doNothing()
  - doAnswer()
  - doThrow()
  - doCallRealMethod()

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

public class MonTest {
	private Personne p;
	@Test
	void test1() {
		p = mock(Personne.class);
		doNothing().when(p).changeBonjour(anyString()); // comportement par défaut

		compteur = 0;
		doAnswer( new Answer<Void>() {
			public Void answer(InvocationOnMock invocation) {
				if (invocation.getArguments()[0].equals("Hey"))
					compteur++;
				return null;
			}
		}).when(p).changeBonjour(anyString());

		doThrow(new NullPointerException()).when(p).changeBonjour(null);

		p.changeBonjour("Hey");
		assertEquals(1, compteur);
		p.changeBonjour("Salut");
		assertEquals(1, compteur);
		assertThrows( NullPointerException.class, () -> p.changeBonjour(null));
	}
}
```

Doublure d'une interface

- Création d'une doublure d'objet qui implémente l'interface IPersonne et dont le comportement de bonjour() est de toujours renvoyer "Bonjour" et le comportement de changeBonjour(null) est de renvoyer une NullPointerException.

```java
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.function.Executable;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class MonTest {
	private IPersonne p;
	@Test
	void test1() {
		/* Creation de la doublure */
		p = mock(IPersonne.class);

		/* Definition du comportement de la doublure */
		when(p.bonjour()).thenReturn("Bonjour");
		doThrow(new NullPointerException()).when(p).changeBonjour(null);

		/* Tests */
		p.changeBonjour("Hey");
		assertEquals( "Bonjour", p.bonjour());

		assertThrows( NullPointerException.class, () -> p.changeBonjour(null));
	}
}
```

```java
public interface IPersonne {
	String bonjour() ;
	void changeBonjour(String s) ;
}
```

Vérification d'appels

La méthode statique verify() permet de vérifier qu'une méthode a été appelée.

```java
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.function.Executable;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class MonTest {
	private IPersonne p;
	@Test
	void test1() {
		/* Creation de la doublure */
		p = mock(IPersonne.class);

		/* Definition du comportement de la doublure */
		when(p.bonjour()).thenReturn("Bonjour");
		doThrow(new NullPointerException()).when(p).changeBonjour(null);

		/* Tests */
		verify(p, never()).changeBonjour("Hey");
		p.changeBonjour("Hey");
		verify(p, never()).changeBonjour("Ciao");
		verify(p).changeBonjour("Hey");
	}
}
```

```java
public interface IPersonne {
	String bonjour() ;
	void changeBonjour(String s) ;
}
```

Espions

La méthode spy() permet de créer des espions. Un espion est une doublure partielle d'un objet dont le comportement est défini par les méthodes réelles de l'objet sauf pour celles qui ont été surchargées par l'espion.

```java
import org.junit.jupiter.api.*;
import org.junit.jupiter.api.function.Executable;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

public class MonTest {
	private Personne p;
	@Test
	void test1() {
		/* Creation de l'espion */
		p = spy(new Personne());

		/* Definition du comportement différencié de l'espion */
		when(p.taille()).thenReturn(100);

		/* Tests */
		p.changeBonjour("Hello");
		assertEquals( "Hello", p.bonjour());
		assertEquals( 100, p.taille());
	}
}
```

```java
public class Personne {
	private String bonjour;
	public Personne() {
		bonjour = null;
	}
	public String bonjour() {
		return bonjour;
	}
	void changeBonjour(String s) {
		bonjour = s;
	}
	public int taille() {
		return bonjour==null?
		0 : bonjour.length(); }
	}
}
```

Annotations @Mock et @Spy

Plutôt que d'utiliser les méthodes mock() et spy(), il est possible d'utiliser les annotations @Mock et @Spy.

```
import org.junit.jupiter.api.* ;
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assertions.*;

import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import org.mockito.Spy;
import static org.mockito.Mockito.*;

import java.util.*;

public class MonTest {

	@Mock
	List<String> doublure;

	@Spy
	List<String> espion = new ArrayList<>();

	@BeforeEach
	void init() {
		MockitoAnnotations.initMocks(this);
	}

	@Test
	void test1() {
		doublure.add("un");
		verify(doublure).add("un");
		assertEquals(0, doublure.size());
		when(doublure.size()).thenReturn(100);
		assertEquals(100, doublure.size());
	}

	@Test
	void test2() {
		espion.add("un");
		espion.add("deux");
		verify(espion).add("un");
		verify(espion).add("deux");
		assertEquals(2, espion.size());
		when(espion.size()).thenReturn(100);
		assertEquals(100, espion.size());
		doReturn(500).when(espion).size();
		assertEquals(500, espion.size());
	}
}
```

- *when(x.faire()).thenReturn(100);*
  fait un appel réel à faire() si x espion
  ne fait pas d'appel réel si x doublure
- *doReturn(100).when(x).faire();*
  ne fait pas d'appel réel
  syntaxe à préférer pour les espions

avec une doublure, les 2 syntaxes se valent

Capture des arguments

ArgumentCaptor permet de capturer les arguments des méthodes « mockées ». Il peut être utilisé avec ou sans l'annotation @Captor.

```
import org.junit.jupiter.api.*;
import org.mockito.ArgumentCaptor;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

import java.util.List;

public class MonTest {

	@Test
	void test1() {
		List doublure = mock(List.class);
		ArgumentCaptor<String> capteur =
			ArgumentCaptor.forClass(String.class);
		doublure.add("un");
		doublure.add("deux");
		verify(doublure,
			times(2)).add(capteur.capture());
		assertEquals("[un, deux]",
			capteur.getAllValues().toString());
	}
}
```

```
import org.junit.jupiter.api.*;
import org.mockito.ArgumentCaptor;
import org.mockito.Captor;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

import java.util.List;

public class MonTest {
	@Mock
	List<String> doublure;
	@Captor
	ArgumentCaptor<String> capteur;
	@BeforeEach
	void init() {
		MockitoAnnotations.initMocks(this);
	}
	@Test
	void test1() {
		doublure.add("un");
		doublure.add("deux");
		verify(doublure,
			times(2)).add(capteur.capture());
		assertEquals("[un, deux]",
			capteur.getAllValues().toString());
	}
}
```

Annotation @InjectMocks

- @InjectMocks crée une instance de la classe et injecte dans cette instance tous les mocks créés avec les annotations @Mock et @Spy.

- Dans le code ci-contre, la classe A a besoin des classes B et C pour calculer sa valeur. Mockito crée une doublure de B, une doublure de C et une instance de A, et il met les doublures dans l'instance de A.

```
public class C {
        private String nom;
        public C(String s) { nom = s; }
        public int taille() { return nom.length(); }
}
```

```
public class B {
        private int valeur;
        public B(int v) { valeur = v; }
        public int valeur() { return valeur; }
}
```

```
public class A {
        private B b;
        private C c;
        public A(B b1, C c1) { b = b1; c = c1; }
        public int valeur() { return b.valeur() + c.taille(); }
}
```

```
import org.junit.jupiter.api.*;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

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

import java.util.List;

public class MonTest {
        @Mock
        B doublureDeB;
        @Mock
        C doublureDeC;
        @InjectMocks
        A a;
        @BeforeEach
        void init() {
          MockitoAnnotations.initMocks(this);
        }

        @Test
        void test1() {
                when(doublureDeB.valeur()).thenReturn(6);
                when(doublureDeC.taille()).thenReturn(4);
                assertEquals(10, a.valeur());
        }
}
```