

Testabilité – Dépendance – Doublure

Jean-Marie Mottu

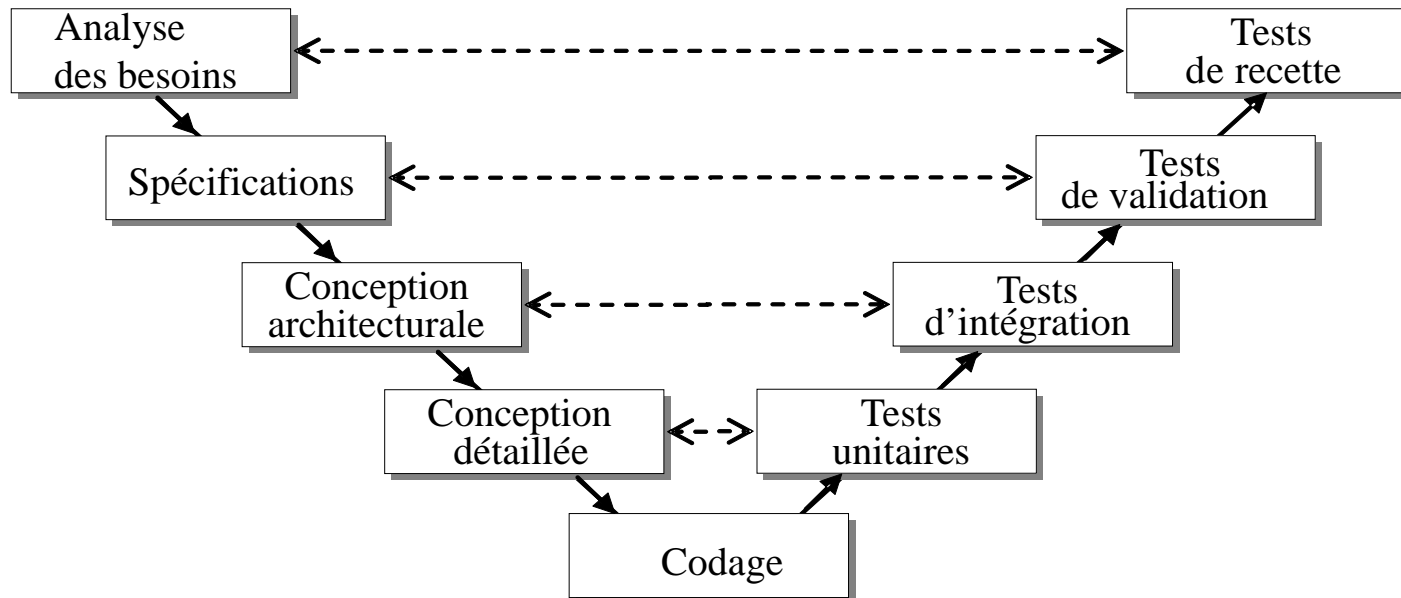
Testabilité

Jean-Marie Mottu – Gerson Sunye
Université de Nantes

Introduction à la Testabilité

► Objectif

- Prendre conscience qu'il ne suffit pas de décider de tester pour
 - Tester facilement
 - Trouver toutes les erreurs



2 principales heuristiques de la Testabilité

► Observabilité

- Qu'est-ce qu'il est possible d'observer dans le logiciel ?
 - Peut-on observer une seule/toutes les partie(s) du logiciel ?
 - Classes, propriétés, variables
 - Peut-on observer l'environnement ?
 - Le temps qui s'écoule, l'interaction avec d'autres logiciels

► Contrôlabilité

- Qu'est-ce qu'il est possible de manipuler ?
 - Logiciel sous test, l'environnement
- Peut-on mettre le système dans l'état voulu ?
 - Faire des tests sur un système en marche ?

Heuristiques secondaires de la Testabilité

▶ Disponibilité

- ▶ Logiciel disponible (boite noire et blanche)
- ▶ Le logiciel doit être suffisamment développé pour exécuter les tests
- ▶ La spécification doit être explicite et disponible

▶ Stabilité

- ▶ Les éléments testés sont modifiés en fournissant un suivi

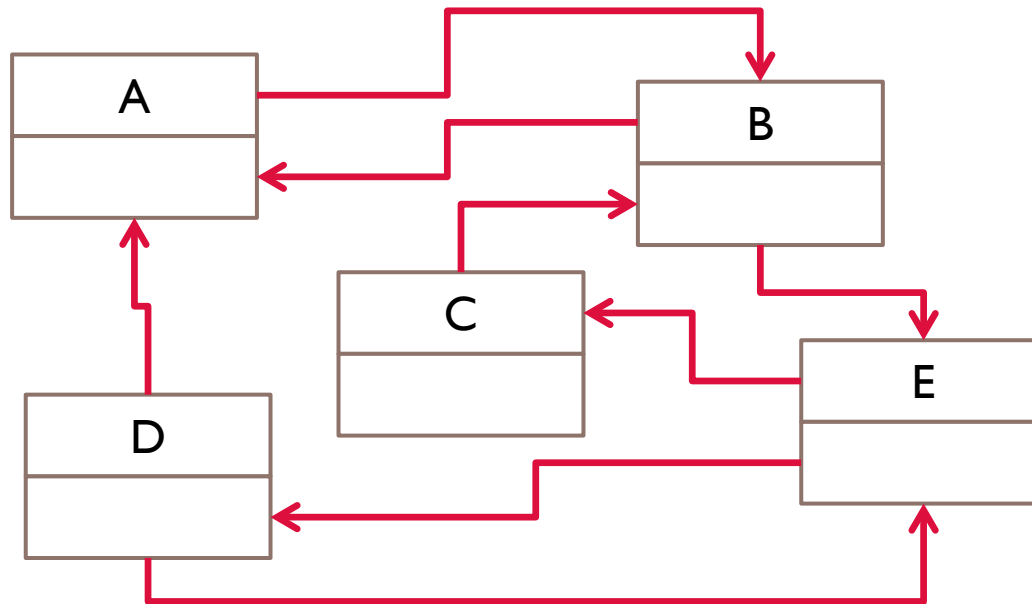
Testabilité

- ▶ La testabilité s'évalue et complète les tests
- ▶ La testabilité est l'activité qui permet d'évaluer la capacité à tester un système selon plusieurs points de vue:
 - ▶ La capacité d'un logiciel à être testé
 - ▶ (observabilité, contrôlabilité, disponibilité, stabilité)
 - ▶ La capacité des tests à faire des vérifications pertinentes
 - ▶ (observabilité, contrôlabilité, stabilité)
 - ▶ La capacité d'une spécification à formuler des exigences vérifiables
 - ▶ (disponibilité, stabilité, observabilité)

Testabilité du point de vue design du logiciel

► Exemples de choix du design pouvant être préjudiciables à la testabilité

- Les variables privées d'une classe ne sont pas observables
- Peu de typage, typage dynamique
- Les interdépendances entre classes provoquent des interblocages



Comment anticiper/résoudre les problèmes de testabilité ?

- ▶ Dès la spécification, conception
 - ▶ Considérer les besoins de contrôlabilité, d'observabilité
- ▶ Comment faire quand on ne peut pas contrôler certaines parties du code ?
 - ▶ Le temps
 - ▶ La communication avec l'extérieur du programme
 - ▶ D'autres programmes
 - ▶ Des liaisons vers le monde physique, nettement moins contrôlable
 - Contrôle impossible
 - Contrôle possible mais avec des coûts/temps non raisonnables

Améliorer la testabilité dès la conception

- ▶ **Améliorer la structure des packages**
 - ▶ Diminuer les interdépendances entre classes.
- ▶ **Limiter la complexité des classes et des méthodes**
 - ▶ De trop nombreuses imbrications de boucles et de conditionnelles font exploser la combinatoire pour résoudre la sensibilisation des chemins du graphe de flot de contrôle
 - ▶ En diminuant le nombre de chemin d'une méthode, on augmente sa testabilité [Nejmeh 1988]
 - ▶ L'analyse du flot de donnée est aussi important
 - ▶ Rapport entre la définition et l'utilisation des variables

Améliorer la testabilité dès la conception

- ▶ Permettre l'observation intermédiaire d'état
 - ▶ Un programme dont l'état interne est important est moins testable qu'un autre n'ayant que des entrées et des sorties
 - ▶ L'état interne est-il masqué ?
 - ▶ L'état interne est-il observable en continue ?
 - ▶ On peut ajouter des observateurs
 - ▶ Des contraintes embarquées
 - ▶ Important pour le test mais aussi la localisation d'erreur

Gestion des dépendances : Injection de dépendance et Doublure de Test

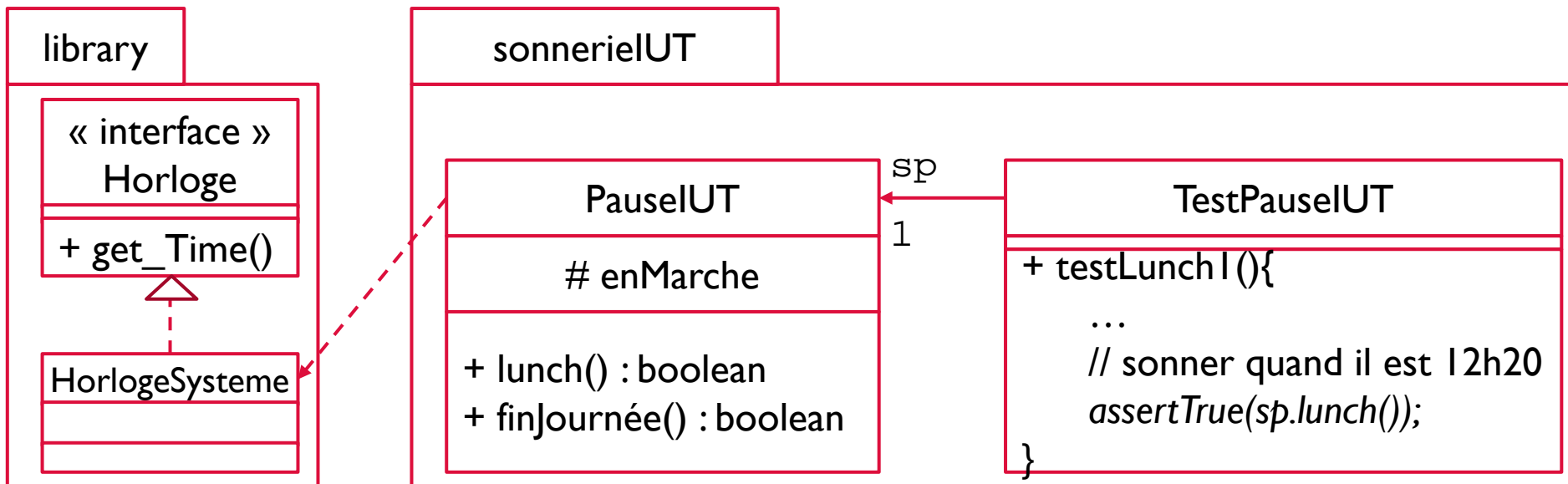
Jean-Marie Mottu

Problème de testabilité pendant les tests:

- ▶ Comment tester efficacement une unité qui dépend d'une autre qui :
 - ▶ n'existe pas encore
 - ▶ n'est pas contrôlable
 - ▶ Le temps, l'aléatoire, l'indéterminisme
 - ▶ Des liaisons vers le monde physique
 - ▶ n'est pas fiable
 - ▶ e.g. impliqué dans des cycles d'interdépendances
 - ▶ est difficilement configurable
 - ▶ a des effets de bords
 - ▶ est lente ou coûteuse
 - ▶ Des conditions extrêmes ou exceptionnelles

Problème de testabilité pendant les tests:

- ▶ Comment tester efficacement une unité qui dépend d'une autre qui :
 - ▶ n'existe pas encore
 - ▶ n'est pas contrôlable

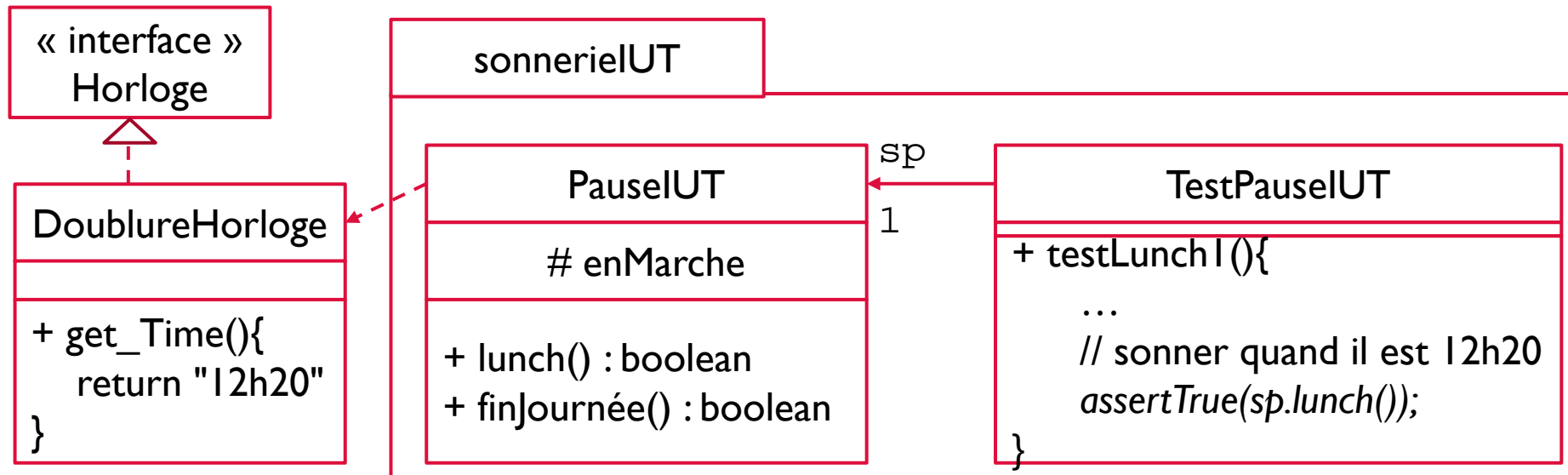


Solution

- ▶ Remplacer l'unité dont dépend l'unité sous test par un équivalent spécifique au test : sa “doublure”
- ▶ Différentes variantes de doublures sont possibles.
- ▶ Indépendamment de la variante, seul le comportement attendu par un test est implémenté.

Solution

- ▶ Remplacer l'unité dont dépend l'unité sous test par un équivalent spécifique au test : sa “doublure”
- ▶ Différentes variantes de doublures sont possibles.
- ▶ Indépendamment de la variante, seul le comportement attendu par un test est implémenté.



Injection de dépendance

- ▶ Créer une doublure nécessite de pouvoir la substituer à la classe originale.
- ▶ Le principe est de ne pas permettre à une méthode de gérer ses dépendances
 - ▶ Sinon le testeur devrait modifier son code pour utiliser la doublure
- ▶ Les dépendances d'une classe doivent lui être fournies depuis l'extérieur
 - ▶ Basiquement en paramètre du constructeur
 - ▶ En limitant les instantiations dans les méthodes (new...)

Injection de dépendance : contre-exemple

```
interface Horloge {  
    int getHour();  
    int getMin();  
}  
  
public class PauseIUT{  
  
    PauseIUT() {  
  
        boolean lunch() {  
            // dépendance pas substituable  
            Horloge horlogeActual = new HorlogeSysteme();  
            return horlogeActual.getHour() == 12  
                &&  
                horlogeActual.getMin() == 20;  
        }  
    }  
}
```

Injection de dépendance : bonne pratique

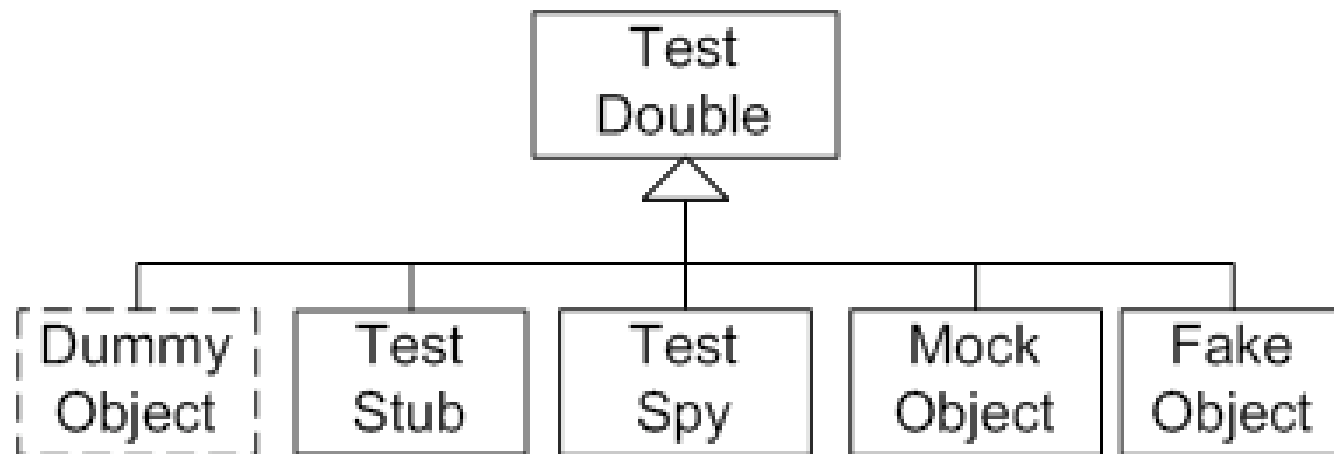
```
interface Horloge {  
    int getHour();  
    int getMin();  
}  
  
public class PauseIUT{  
    // dépendance substituable  
    Horloge horlogeUsed;  
    PauseIUT(Horloge horlogeParam){  
        horlogeUsed = horlogeParam;  
    }  
  
    boolean lunch(){  
        return horlogeUsed.getHour() == 12  
            &&  
            horlogeUsed.getMin() == 20;  
    }  
}
```

Injection de dépendance : bonne pratique

```
public class testPauseIUT{

    boolean testlunch1(){
        Horloge hDoublure = new HorlogeDoublure()
        // HorlogeDoublure renvoie tout le temps
        // 12h20
        PauseIUT pi = new PauseIUT(hDoublure)
        assertTrue(pi.lunch, "A 12h20 c'est la pause")
    }
}
```

Doublures de test [Meszaros]



[Meszaros] <http://xunitpatterns.com/Test%20Double.html>

Doublures de test

en	Fr	description
Stub	Bouchon	Classe codée à la main. Fournit des réponses pré-définies aux appels faits durant le test.
Mock	Simulacre	Les attentes sont adaptées avant l'exécution de la méthode de test.
Spy	Espion	La vérification se fait après l'exécution de la méthode de test.
Dummy	Fantôme	Objets vides, utilisés simplement pour remplir des listes de paramètres.
Fake	Substitut	Objets qui implémentent le même comportement que l'original, mais de façon plus simple.

<http://tinyurl.com/testdoubles>



Bouchons

- Un Bouchon (Stub) est une classe codée à la main, qui fournit des réponses pré-définies aux appels faits durant le test.

```
public class HorlogeStub implements Horloge {  
    public int getHour(){  
        return 12;  
    }  
  
    public int getMin(){  
        return 20;  
    }  
}
```

Simulacres

- Un Simulacre (Mock) est une doublure configurable où les attentes sont adaptées avant l'exécution de la méthode de test (qu'on code ici à la main, on verra l'outil Mockito plus loin).

```
public class HorlogeMock implements Horloge {  
    private int hour;  
    private int min;  
  
    public HorlogeMock(int hourParam, int minParam){  
        this.hour = hourParam;  
        this.min = minParam;  
    }  
  
    public int getHour(){  
        return this.hour;  
    }  
    // etc.  
}
```

Espions

- Un Espion (Spy) est une doublure qui permet la vérification après l'exécution de la méthode de test.

```
public class HorlogeSpy implements Horloge {  
    private Horloge spied;  
  
    public HorlogeSpy(Horloge actual) {  
        this.spied = actual;  
    }  
  
    public List<int> listHours = new LinkedList<int>();  
    public int getHour() {  
        int hourConsulted = spied.getHour();  
        this.listHours.add(hourConsulted);  
        return hourConsulted;  
    }  
  
    // etc.  
}
```


Fantômes

- ▶ Un fantôme (dummy) est un objet vide, utilisé simplement pour remplir des listes de paramètres.

```
Horloge dummy = new Horloge() {  
    public int getHour(){  
        return 0;  
    }  
  
    public int getMin(){  
        return 0;  
    }  
}
```

Substituts

- ▶ Un substitut (Fake) est une doublure qui met en œuvre le même comportement que l'original, mais de façon différente (e.g. un ancien code, la version d'une autre plateforme).
- ▶ Par exemple, avec un code « patrimonial » :

```
@Deprecated
public class LegacyHorloge implements Horloge {
    public int getHour(){
        //previous implementation code that still works.
        // (...)
    }

    public int getMin(){
        //previous implementation code that still works.
        // (...)
    }
}
```

Outillage de Mock

▶ Java

- ▶ Mockito
- ▶ EasyMock
- ▶ PowerMock
- ▶ Jmockit

▶ Kotlin déclare les classes « final » par défaut,

- ▶ Cela empêche un outil comme Mockito de fonctionner :
il y a des astuces comme déclaré explicitement « open »
 - ▶ Discutable avec les développeurs comme un problème de testabilité

▶ Alternative en Kotlin : Mockk

Mockk

Principes

- ▶ **Générateur de doublures.**
 - ▶ Plus précisément, de simulacres et de classes-espionnes.

- ▶ **Etapes d'utilisation:**
 1. Création des doublures.
 2. Description du comportement attendu.
 3. Utilisation des doublures dans les tests
 1. Pour fournir des réponses prédéfinies nécessaires aux tests,
 2. Pour vérifier que l'interaction avec les doublures est correcte.

Exemple

```
import io.mockk.*
```

```
//Création de la doublure grâce à la méthode mockk()
```

```
var mockHorloge = mockk<HorlogeSysteme>()
```

```
//Description d'un comportement attendu qui renvoie 12h20
```

```
every {mockHorloge.getHour() } returns 12
```

```
every {mockHorloge.getMin() } returns 20
```

```
// Utilisation pour fournir une réponse prédéfinie
```

```
println(mockHorloge.getHour()) //12
```

```
println(mockHorloge.getMin()) //20
```

```
// Utilisation du nombre d'interaction avec la doublure
```

```
verify(atMost = 1){mockPauseUT.getPause(12,20)}
```

```
    //vérifie que getPause(12,20) n'a été appelé
```

```
    qu'une fois sur la doublure (et qu'on n'a pas
```

```
    dérangé trop de fois le prof pour aller déjeuner)
```



Création de doublures

- ▶ La méthode `mockk()` permet la création de doublures de classes et d'interfaces

```
var mockHorloge = mockk<HorlogeSysteme>()
```

- ▶ Lors de la création de la doublure, un comportement par défaut est défini : lève une exception
 - ▶ On peut demander à renvoyer des valeurs par défaut comme Mockito

```
var mockHorloge = mockk<HorlogeSysteme>(relaxed = true)  
println(mockHorloge.getHour()); //renvoie 0
```

- ▶ Si on veut seulement « espionner » le comportement de la classe dépendance sans la doubler, on peut seulement créer un espion et lui appliquer des `verify` :

```
var spyHorloge = spyk<HorlogeSysteme>()
```

La méthode `every{ }`

- ▶ Méthode utilisée pour associer un comportement maîtrisé à une méthode.

- ▶ Provoque le comportement `returns` ou `throws` :

`every { mockHorloge.getHour() } returns 12`

`every { mockPauseUT.getPause(22,30) } throws IllegalArgumentException`

- ▶ Peut provoquer des suites de retours

`every { mockHorloge.getHour() } returns 12 andThen 17`

`every { mockHorloge.getMin() } returnsMany listOf (20, 50)`

- ▶ Paramétrable

`every { mockPauseUT.getPause(more(18), any()) } throws IllegalArgumentException`

`every { mockPauseUT.getPause(less(8), anyInt) } throws IllegalArgumentException`

`every { mockPauseUT.getPause(9, 20) } returns 10 //durée pause`

`every { mockPauseUT.getPause(12,20) } returns 70 /durée pause`

La méthode verify

- Les interactions réalisées avec une doublure sont enregistrées et peuvent être vérifiées à posteriori

```
verify{mock.getHour()} // appelée au moins 1 fois  
verify(exactly = 5){mock.getHour()} // en précisant combien  
verify(atLeast = 1) {mock.getHour()} // au moins 1  
verify(atMost = 7) {mock.getHour()} // au plus 7
```

```
verify{mock wasNot Called}  
verifySequence {  
    mockPauseUT.getPause(9, 20)  
    mockPauseUT.getPause(10, 50)  
    mockPauseUT.getPause(12, 20)  
}
```

```
verifyOrder {  
    mockPauseUT.getPause(9, 20)  
    mockPauseUT.getPause(12, 20)  
}
```