IUT Nantes
Pôle Sciences et technologie
Nantes Université

R2.03 - Qualité de développement 1
Automatisation des tests

Gestion des dépendances : Injection de dépendance et Doublure de Test

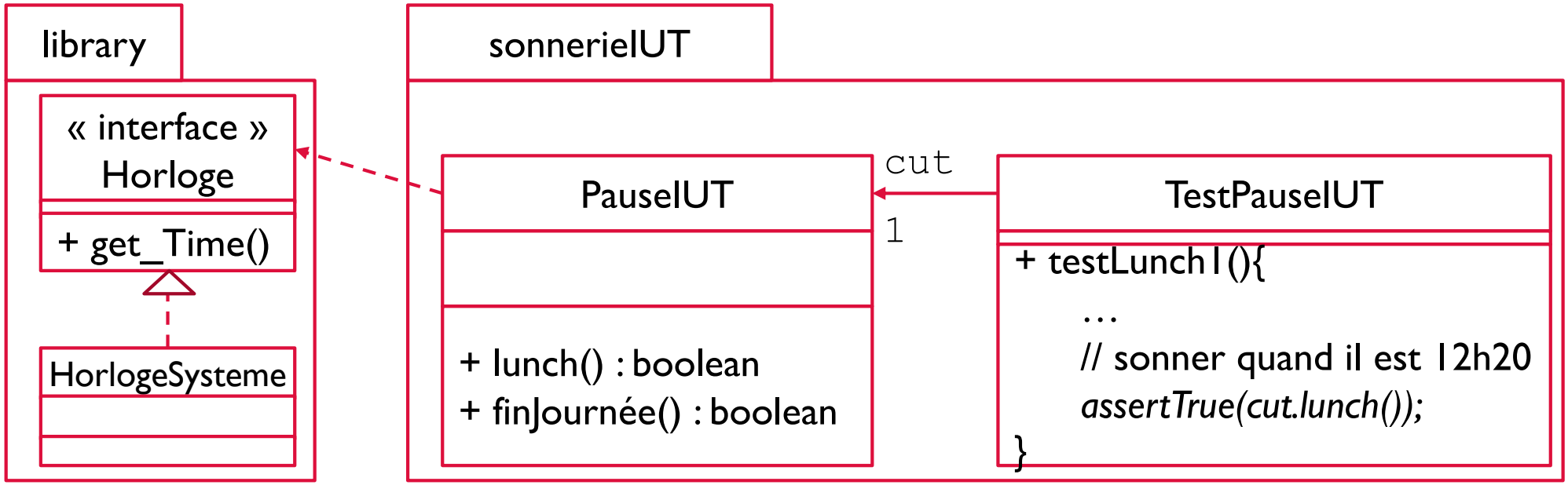
Jean-Marie Mottu – Gerson Sunyé

Problème de testabilité à gérer 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. impliquée 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 à gérer pendant les tests

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

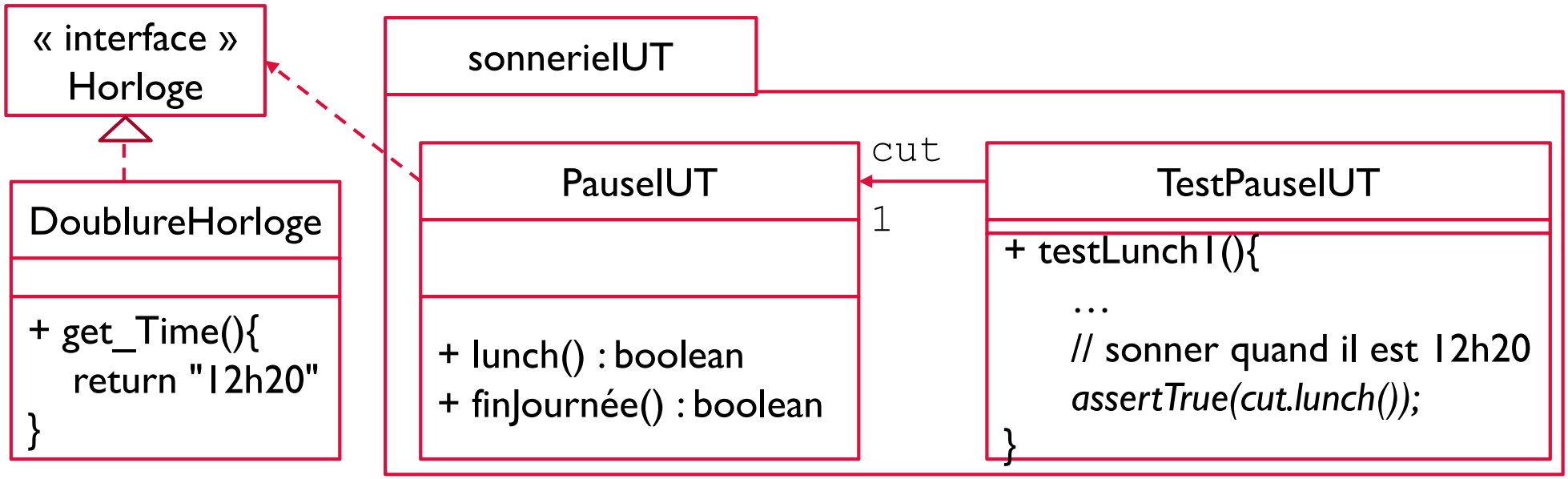


Solution : doublure de test

- 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 nécessaire pour tester est implémenté.

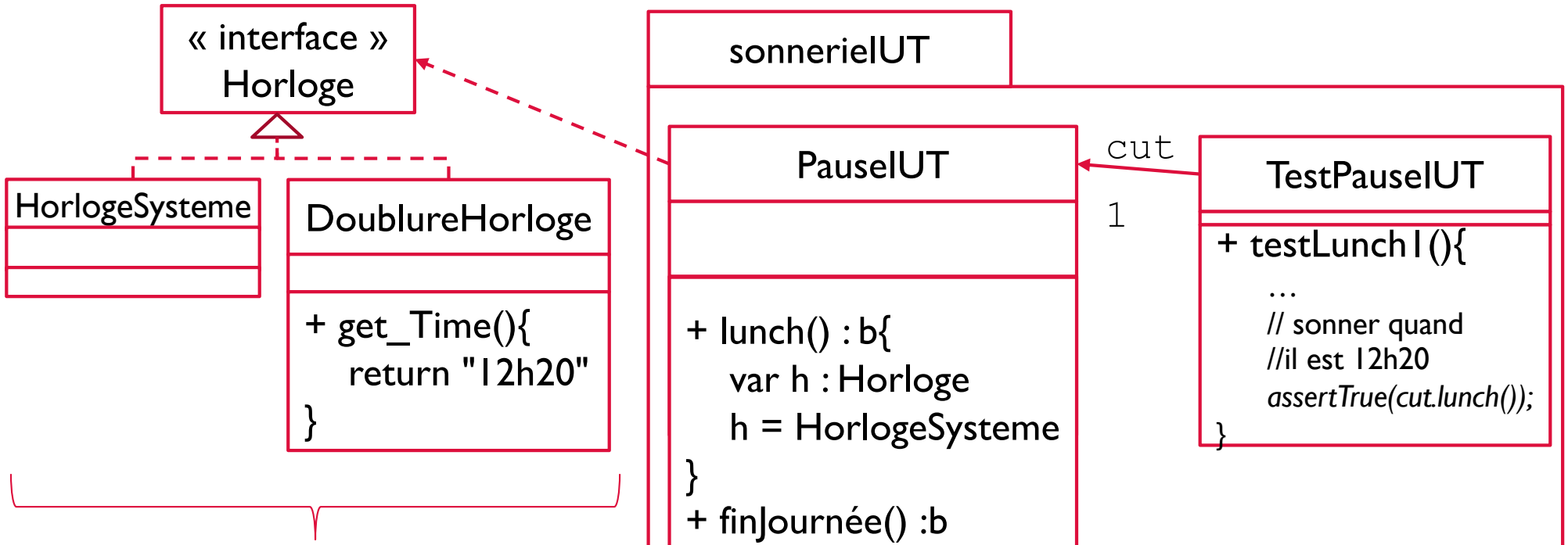
Solution : doublure de test

- 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 nécessaire pour tester est implémenté.



Comment gérer la substitution de la dépendance ?

- Règle : le testeur ne peut pas modifier (temporairement) le code sous test



Le testeur peut créer une doublure mais pas modifier la classe actuelle, ni sa liaison

On a un problème de testabilité (de contrôlabilité) avec une Horloge pas substituable

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 fournie depuis l'extérieur
  - Basiquement en paramètre du construteur
  - En limitant les instanciations dans les méthodes (new…)

Injection de dépendance : contre-exemple

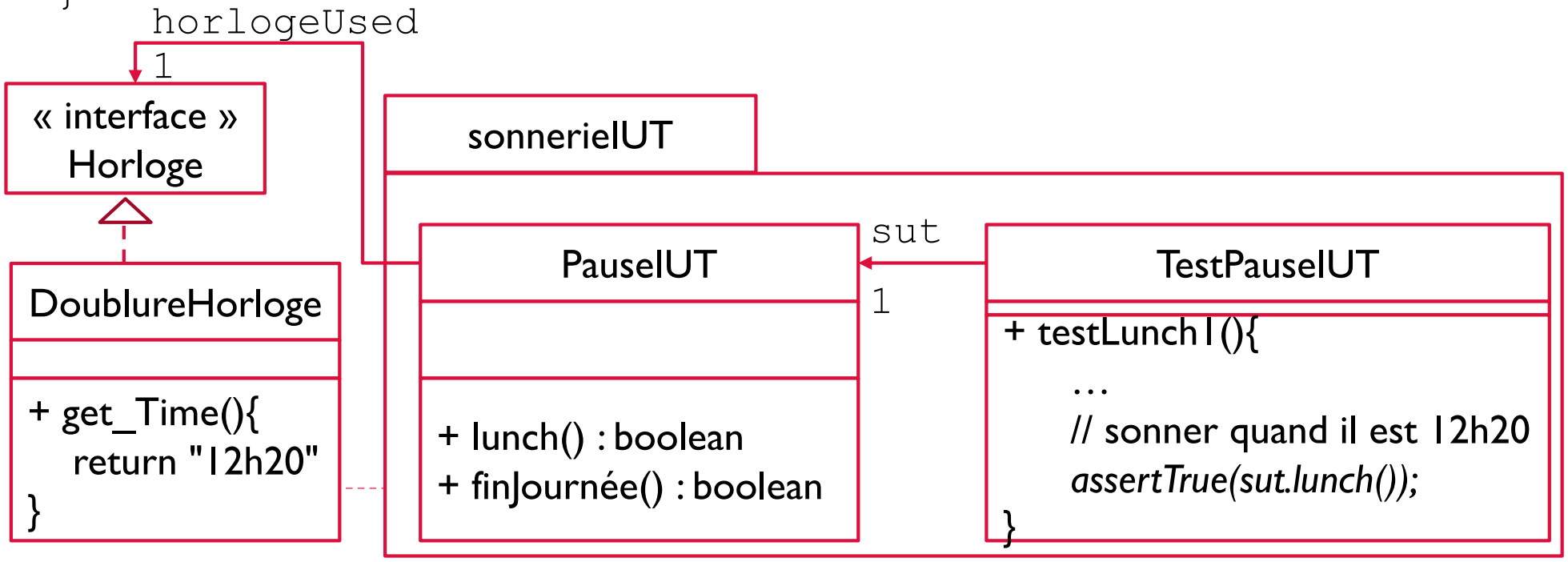
```
interface Horloge {
	int getHour();
	int getMin();
}

public class PauseIUT{

	PauseIUT(){   }

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

Injection de dépendance de classe : bonne pratique

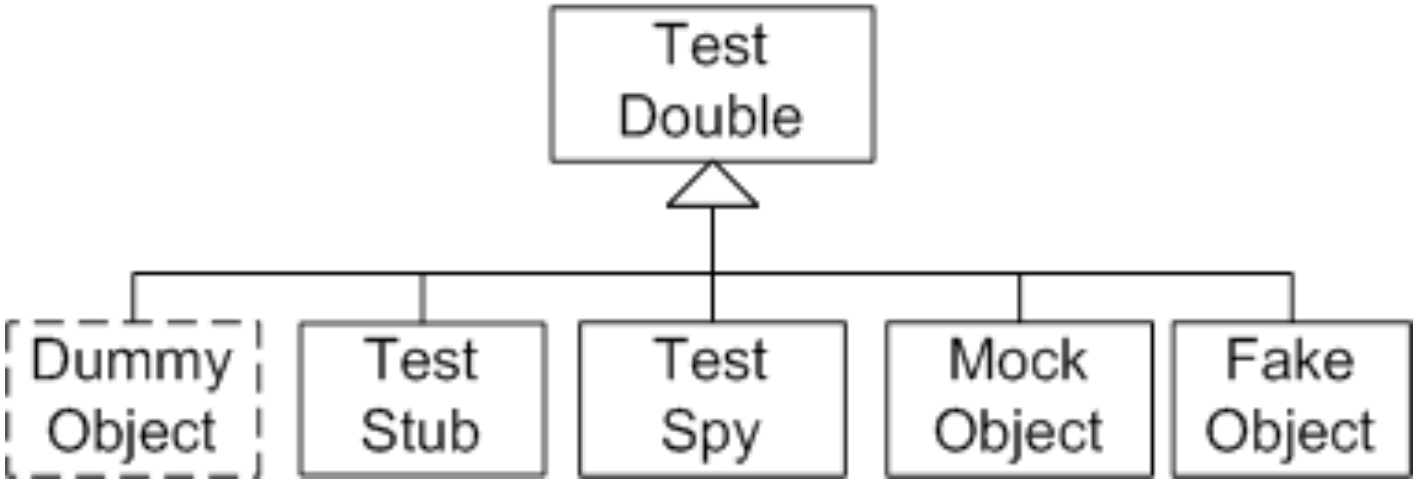
```
interface Horloge {
	int getHour();
	int getMin();
}

public class PauseIUT{
	// dépendance substituable
	Horloge horlogeUsed;
	PauseIUT(Horloge horlogeParam){
		horlogeUsed = horlogeParam;
	}

	boolean lunch(){
		return horlogeUsed.getHour() == 12
			&&
			horlogeUsed.getMin() == 20;
	}
}
```

Injection de dépendance de classe : bonne pratique

```
public class testPauseIUT{
    boolean testlunch1(){
        Horloge hDoublure = new HorlogeDoublure()
        // HorlogeDoublure renvoie tout le temps 12h20
        PauseIUT cut = new PauseIUT(hDoublure)
        assertTrue(cut.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 *Mockk* 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 déclarer explicitement « open »
    - Discutable avec les développeurs comme un problème de testabilité
- Alternative en Kotlin : Mockk

IUT Nantes
Pôle Sciences et technologie

Nantes Université

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){mockPauseIUT.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 maitrisé à une méthode.
- Provoque le comportement `returns` ou `throws` :

```
every { mockHorloge.getHour() } returns 12
every {mockPauseIUT.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 {mockPauseIUT.getPause(more(18), any()) } throws IllegalArgumentException
every {mockPauseIUT.getPause(less(8), anyInt) } throws IllegalArgumentException
every {mockPauseIUT.getPause(9, 20) } returns 10 //durée pause
every {mockPauseIUT.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 {
  mockPauseIUT.getPause(9, 20)
  mockPauseIUT.getPause(10, 50)
  mockPauseIUT.getPause(12, 20)
}
```

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