Initiation génie logiciel : Types paramétrés et patrons de conception

Arnaud Labourel (arnaud.labourel@univ-amu.fr)

1er octobre 2024

amU Faculté des sciences Aix Marseille Université

Section 1

Types paramétrés

Stack d'Object

Supposons que nous ayons la classe suivante :

```
public class Stack {
  private Object[] stack = new Object[100];
  private int size = 0;
  public void push(Object object) {
    stack[size] = object; size++;
  }
  public Object pop() {
    size--;
    Object object = stack[size];
    stack[size] = null; // Pour le Garbage Collector.
    return object;
  }
}
```

Problème de Stack d'Object

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();
String string = "truc";
stack.push(string);
string = (String) stack.pop();
// Transtypage obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();
Integer intValue = new Integer(2);
stack.push(intValue);
String string = (String) stack.pop();
// Erreur à l'exécution
```

La solution : types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();
String string = "truc";
stack.push(string); // Le paramètre doit être un String.
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe `Stack` qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypages automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Définition de classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {
    private Object[] stack = new Object[100];
    private int size = 0;
    public void push(T element) {
        stack[size] = element; size++;
    }
    public T pop() {
        size--;
        T element = (T) stack[size];
        stack[size] = null;
        return element;
    }
}
```

Emballage et déballage

Les types primitifs ne sont pas des classes :

Dans le cas d'un `int`, on doit utiliser la classe d'emballage (*wrapper class*) `Integer` :

Interdit : ~~`Stack<int> stack = new Stack<>();`~~

Autorisé :

```java
Stack<Integer> stack = new Stack<>();
int intValue = 2;
Integer integer = Integer.valueOf(intValue);
// → emballage du int dans un Integer.
stack.push(integer);
Integer otherInteger = stack.pop();
int otherIntValue = otherInteger.intValue();
// → déballage du int présent dans le Integer.
```

Types primitifs

type	classe d'emballage	taille	valeurs possibles
byte	Byte	8 bits	-128 à 127
short	Short	16 bits	-32768 à 32767
int	Integer	32 bits	-2^{31} à $2^{31}-1$
long	Long	64 bits	-2^{63} à $2^{63}-1$
float	Float	32 bits	
double	Double	64 bits	
char	Character	16 bits	caractère unicode
boolean	Boolean	non définie	`false` ou `true`

Classes d'emballage

La classe `Number` sert de base pour toutes les classes d'emballage.

Elle contient les méthodes suivantes :

- `public int intValue()`
- `public long longValue()`
- `public float floatValue()`
- `public double doubleValue()`
- `public byte byteValue()`
- `public short shortValue()`

Classes d'emballage

Les classes d'emballage étendent `Number` :

- `Byte` → `public static Byte valueOf(byte b)`
- `Short` → `public static Short valueOf(short s)`
- `Integer` → `public static Integer valueOf(int i)`
- `Long` → `public static Long valueOf(long l)`
- `Byte` → `public static Byte valueOf(byte b)`

Ils existent des constructeurs mais ils sont dépréciés (et donc pas à utiliser).

La classe Character

Les classes d'emballage ne contiennent pas que des méthodes liées aux instances :

- `public static Character valueOf(char c)` (emballage)
- `public char charValue()` (déballage)
- `public static boolean isLowerCase(char ch)`
- `public static boolean isUpperCase(char ch)`
- `public static boolean isDigit(char ch)`
- `public static boolean isLetter(char ch)`
- `public static boolean isLetterOrDigit(char ch)`
- `public static char toLowerCase(char ch)`
- `public static char toUpperCase(char ch)`
- `public static char toTitleCase(char ch)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
stack.push(intValue);
// → emballage automatique du int dans un Integer.
int otherIntValue = stack.pop();
// → déballage automatique du int.
```

Attention

Il est important de noter que des allocations sont effectuées lors des emballages sans que des `new` soient présents dans le code.

Plusieurs paramètres de types

```java
public class Pair<A, B> {
    public final A first;
    public final B second;
    public Pair(A first, B second) {
        this.first = first;
        this.second = second;
    }
    public String toString() {
        return "(" + first + ", " + second + ')';
    }
}
```

Utilisation:

```java
Pair<String,Integer> pair = new Pair<>("toto", 12);
```

Méthodes paramétrées

Il est possible de mettre de définir des paramètres de types pour une méthode.

Il faut les déclarer avant le type de retour.

```
public class Pair<A, B> {
  public static <C,D> Pair<C,D> makePair(C first, D second) {
    return new Pair<C,D>(first, second);
  }
}
```

Utilisation:

```
Pair<String,Integer> pair = Pair.<String,Integer>makePair("toto",12);
System.out.println(pair); // affiche (toto, 12)
```

Le *diamond* <>

Depuis Java 7, lors de l'appel d'un constructeur, il est possible de laisser les arguments de type vide (*diamond* <>) du moment que le compilateur peut déterminer (inférer) le type ou les types à partir du contexte.

```
Box<Integer> integerBox = new Box<>();
public static <C, D> Pair<C,D> makePair(C first, D second) {
  return new Pair<>(first, second);
}
```

De même pour une méthode paramétrée en ne mettant même pas le *diamond*.

```
Pair<String,Integer> pair = Pair.makePair("tot",12);
```

Convention de nommage pour les types paramétrés

Par convention, les noms des paramètres de type sont des lettres majuscules.

C'est contraire bonnes pratiques visant à donner des noms détaillés aux éléments du code, mais indispensable pour différencier d'un coup d'œil les paramètres de type des classes.

Les noms les plus utilisés sont :

- `E` : *Element* (très utilisé pour les `Collections` de Java)
- `K` : *Key*
- `N` : *Number*
- `T` : *Type*
- `V` : *Value*

Section 2

Types paramétrés (notions avancées)

Condition sur les paramètres – Problématique

```
public interface Comparable<T> {
  public int compareTo(T element);
}

class Greatest {
  private String element;
  public void add(String element) {
    if (this.element==null || this.element.compareTo(element)<0)
      this.element = element;
  }
  public String get() { return element; }
}
```

Comment rendre la classe `Greatest` générique ?

Condition sur les paramètres

```
class Greatest<T extends Comparable<T>> {
  private T element;
  public void add(T element) {
  if (this.element==null || element.compareTo(this.element)>0)
    this.element = element;
  }
  public T get() {
    return element;
  }
}
```

Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public T get() { return element; }
}
class Card implements Comparable<Card> { /* ... */ }
class PrettyCard extends Card { /* ... */ }
```

Il n'est pas possible d'écrire les lignes suivantes, car `PrettyCard` n'implémente pas l'interface `Comparable<PrettyCard>` :

```
Greatest<PrettyCard> greatest = new Greatest<PrettyCard>();
greatest.add(new PrettyCard(Card.diamond, 7));
```

Syntaxe ? super

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public T get() { return element; }
}

class Card implements Comparable<Card> { /* ... */ }
class PrettyCard extends Card { /* ... */ }
```

Il est possible d'écrire les lignes suivantes car `PrettyCard` implémente l'interface `Comparable<Card>` et `Card super PrettyCard`:

```
Greatest<PrettyCard> greatest = new Greatest<PrettyCard>();
greatest.add(new PrettyCard(Card.diamond, 7));
```

Wildcard ?

Dans un paramètre de généricité, le symbole ? (appelé *wildcard*) dénote une variable de type anonyme.

On peut la contraindre avec les mot-clés `super` et `extends`.

Exemples :

- `List<?>` : une liste de type quelconque.
- `List<? extends Shape>` : une liste d'instances d'une sous-classe de `Shape`.
- `List<? super Disc>` : une liste d'instances d'une classe ancêtre de `Disc`.
- `E extends Comparable<? super E>` : un type `E` implémentant l'interface `Comparable<P>` pour `P` ancêtre de `E`.

? extends – Problématique

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public void addAll(List<T> list) {
    for (T element : list) add(element);
  }
}
```

Il n'est pas possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();
Greatest<Card> greatest = new Greatest<Card>();
greatest.addAll(list);
```

? extends

Supposons que nous ayons les classes suivantes :

```
class Greatest<T extends Comparable<? super T>> {
  /* ... */
  public void add(T element) { /* ... */ }
  public void addAll(List<? extends T> list) {
    for (T element : list) add(element);
  }
}
```

Il est maintenant possible d'écrire les lignes suivantes :

```
List<PrettyCard> list = new ArrayList<PrettyCard>();
Greatest<Card> greatest = new Greatest<Card>();
greatest.addAll(list);
```

Méthodes paramétrées et conditions sur les types

```
class Tools {
  static <T extends Comparable<T>> boolean isSorted(T[] array) {
    for (int i = 0; i < array.length-1; i++)
      if (array[i].compareTo(array[i+1]) > 0)
        return false;
    return true;
  }
}
```

Exemple :

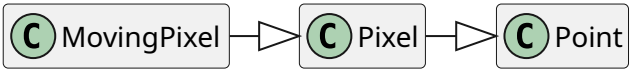
```
String[] array = { "ezjf", "aaz", "zz" };
System.out.println(Tools.isSorted(array));
```

Méthodes paramétrées et conditions sur les types

Méthode pour copier une liste `src` vers une autre liste `dest` :

```
static <T> void copy(List<? super T> dest, List<? extends T> src)
```

On suppose qu'on a une classe `MovingPixel` qui étend `Pixel` qui elle-même étend `Point`.



On peut écrire :

```
List<MovingPixel> src = new ArrayList<>();
List<Point> dest = new ArrayList<>();
Collections.<Pixel>copy(dest, src);
```

Utilisation `extends` et `super`

Lorsqu'on a une collection d'objets de type `T` :

- En entrée/écriture, on veut donner des objets qui ont au moins tous les services des objets de type `T`.

  On doit donc donner des objets dont la classe étend `T` : `? extends T`

- En sortie/lecture, on veut récupérer des objets qui ont au plus tous les services des objets de type `T`.

  On doit donc récupérer des objets qui sont étendu par la classe `T` : `? super T`

Section 3

Patrons de conception

Listes des patrons de conception

~~Patrons faits~~

- Patrons de création
  - ~~Fabrique~~ (*factory*)
  - ~~Fabrique abstraite~~ (*abstract factory*)
  - Monteur (*builder*)
  - Prototype (*prototype*)
  - Singleton (*singleton*)
- Patrons structurels
  - ~~Adaptateur~~ (*adapter*)
  - Pont (*bridge*)
  - ~~Composite~~ (*composite*)
  - Décorateur (*decorator*)
  - Façade (*facade*)
  - Poids mouche (*flyweight*)
  - Procuration (*proxy*)
- Patrons comportementaux
  - Chaîne de responsabilité (*chain of responsibility*)
  - Commande (*command*)
  - Itérateur (*iterator*)
  - Médiateur (*mediator*)
  - Mémento (*memento*)
  - Observateur (*observer*)
  - ~~État~~ (*state*)
  - ~~Stratégie~~ (*strategy*)
  - ~~Patron de méthode~~ (*template method*)
  - Visiteur (*visitor*)

Section 4

Patron de conception *Visitor*

Interface Shape

Considérons l'interface suivante :

```
public interface Shape {
  void draw(GraphicsContext context);
  String XMLRepresentation();
  double area();
}
```

Quel principe SOLID est violé par cette interface ?

Implémentation d'un rectangle

```
public class  Rectangle implements Shape {
  public double x, y, w, h;
  public Rectangle(double x, double y, double w, double h) {
    this.x = x; this.y = y; this.w = w; this.h = h;
  }
  public void draw(GraphicsContext context) {
    context.strokeRect(x, y, h, w);
  }
  public String XMLRepresentation() {
    return "<Rectangle><x>" + x + "</x>" + ... + "</Rectangle>" ;
  }
  public double area(){
    return w*h;
  }
```

Implémentation d'un cercle

```
public class Circle implements Shape {
  public final double x, y, radius;
  public Circle(double x, double y, double radius) {
    this.x = x; this.y = y; this.radius = radius;
  }
  public void draw(GraphicsContext context) {
    context.strokeOval(x-radius, y - radius, 2*radius, 2*radius);
  }
  public double area(){
    return Math.pow(circle.radius,2) * Math.PI;
  }
  public String XMLRepresentation() {
    return "<Circle><x>" + ... + "</radius></Circle>" ;
  }
```

Classes et méthodes

		classes			
		Rectangle	Circle	Polygon	···
méthodes	draw()				
	XMLRepresentation()				
	area()				
	···				

Dans le tableau ci-dessus, on a :

- Une classe par colonne
- Une méthode par ligne
- SRP violé, car plusieurs responsabilités dans chaque classe

Solution

Définir une classe par ligne en utilisant le patron de conception *Visitor*

Interfaces pour *Visitor*

Création des deux interfaces qui permettent de coder *Visitor* :

```
public interface Shape {
  <R> R accept(ShapeVisitor<R> visitor);
}

public interface ShapeVisitor<R> {
  R visit(Rectangle rectangle);
  R visit(Circle circle);
}
```

Nouvelle classe `Circle`

Simplification des classes et implémentation de la méthode `accept` :

```
public class Circle implements Shape {
  public final double x, y, radius;
  public Circle(double x, double y, double radius) {
    this.x = x;
    this.y = y;
    this.radius = radius;
  }

  public <R> R accept(ShapeVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

Nouvelle classe Rectangle

Simplification des classes et implémentation de la méthode accept :

```
public class Rectangle implements Shape {
  public final double x, y, w, h;
  public Rectangle(double x, double y, double w, double h) {
    this.x = x; this.y = y; this.w = w; this.h = h;
  }

  public <R> R accept(ShapeVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

Implémentation du visiteur DrawerVisitor (1/2)

```
public class DrawerVisitor implements ShapeVisitor<Void> {
  private GraphicsContext context;
  public DrawerVisitor(GraphicsContext context) {
    this.context = context;
  }
  public void draw(List<Shape> shapes) {
    for (Shape shape : shapes)
      shape.accept(this);
  }
  /* Voir slide suivant pour la suite */
}
```

Implémentation du visiteur DrawerVisitor (2/2)

```
public class DrawerVisitor implements DrawableVisitor {
  public Void visit(Rectangle rectangle) {
    context.strokeRect(rectangle.x, rectangle.y,
        rectangle.h, rectangle.w);
    return null;
  }
  public Void visit(Circle circle) {
    context.strokeOval(circle.x - circle.radius,
        circle.y - circle.radius, circle.radius*2, circle.radius*2);
    return null;
  }
}
```

Implémentation du visiteur XMLVisitor (1/2)

```
public class XMLVisitor implements ShapeVisitor<String> {
  public String XMLRepresentation(List<Shape> shapes) {
    StringBuilder builder = new StringBuilder( "<Shapes>" );
    for (Shape shape : shapes)
      builder.append(shape.accept(this));
    builder.append("</Shapes>");
    return builder.toString();
  }
  /* Voir slide suivant pour la suite */
}
```

Implémentation du visiteur XMLVisitor (2/2)

```
public class XMLVisitor implements ShapeVisitor<String> {
  // suite du slide précédent
  public String visit(Rectangle rectangle) {
    return "<Rectangle><x>" + x + "</x><y>" + y + "</y>" +
          "<width>" + w + "</width><height>" + h + "</height>"
          + "</Rectangle>" ;
  }
  public String visit(Circle circle) {
    return "<Circle><x>" + x + "</x><y>" + y + "</y>" +
          + "<radius>" + radius + "</radius></Circle>" ;
  }
}
```

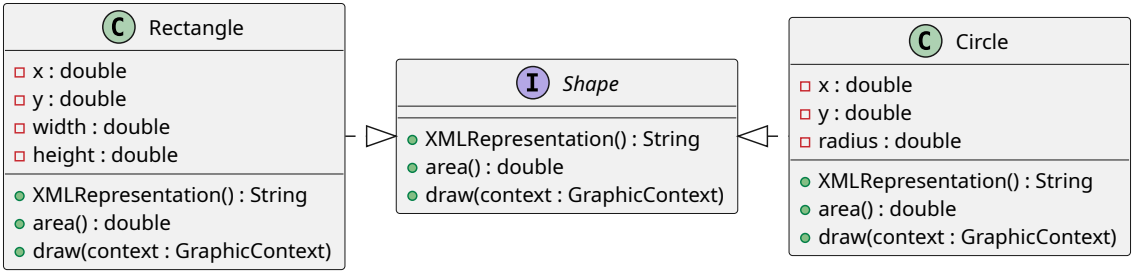
Implémentation du visiteur AreaVisitor (1/2)

```
public class AreaVisitor implements ShapeVisitor<Double> {
  public double sumOfArea(List<Shape> shapes) {
    double sum = 0;
    for (Shape shape : shapes)
      sum += shape.accept(this);
    return sum;
  }
  public double area(Shape shape) {
    return shape.accept(this);
  }
  /* Voir slide suivant pour la suite */
}
```

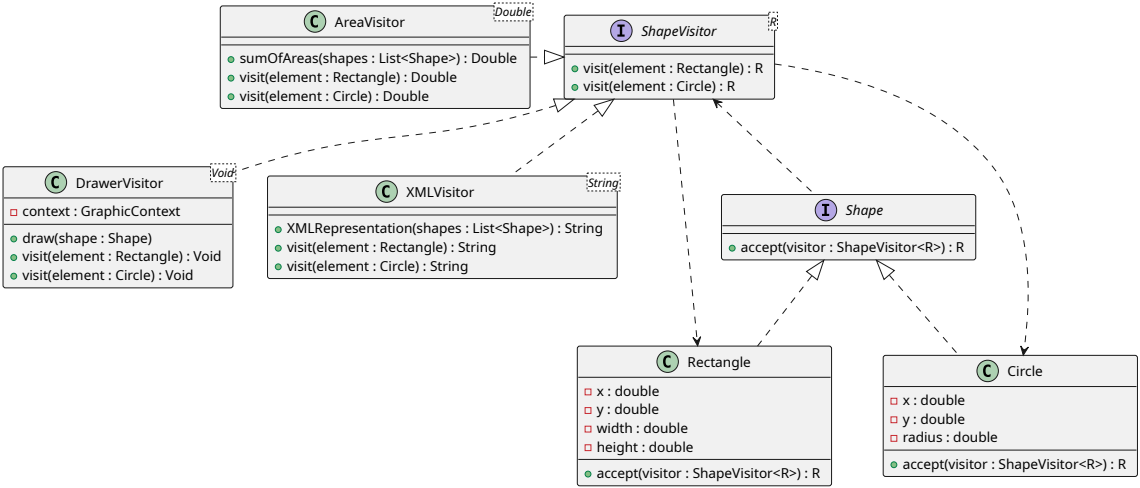
Implémentation du visiteur AreaVisitor (2/2)

```
public class AreaVisitor implements ShapeVisitor<Double> {
  // suite du slide précédent
  public Double visit(Rectangle rectangle) {
    return rectangle.w * rectangle.h;
  }
  public Double visit(Circle circle) {
    return Math.pow(circle.radius,2) * Math.PI;
  }
}
```

Organisation initiale du code : solution 1



Utilisation du patron de conception *Visitor* : solution 2



Comparaison des deux solutions pour le dessin

Première solution (méthode `draw` dans chaque classe)

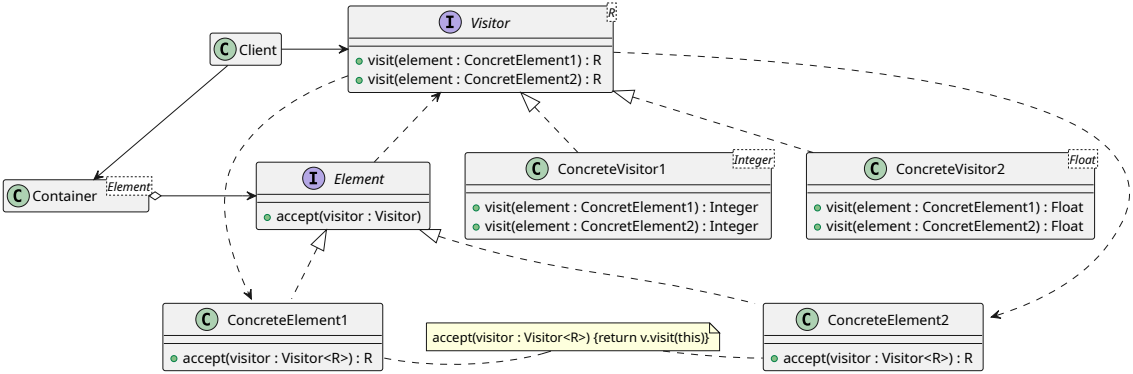
- Simplicité lors de l'ajout d'une nouvelle forme
- Couplage dessin/forme
- Une seule façon de dessiner par forme
- Méthodes de dessin éclatées dans de nombreuses classes

Deuxième solution (visiteur `DrawerVisitor`)

- Les méthodes de dessin sont regroupées dans une classe
- Possibilité d'avoir plusieurs façons de dessiner les formes
- Nombreuses classes à modifier lors de l'ajout d'une forme

$\Rightarrow$ Le choix de l'implémentation d'une fonctionnalité doit dépendre des probables évolutions de l'application.

Patron de conception *Visitor*



Patron de conception *Visitor*

Intention

Séparer les algorithmes des classes sur lesquels ils opèrent.

Avantages :

- ajout facile d'un nouveau comportement (OCP)
- extraction d'un comportement hors de la classe (SRP)
- Cohérence du comportement entre différents objets

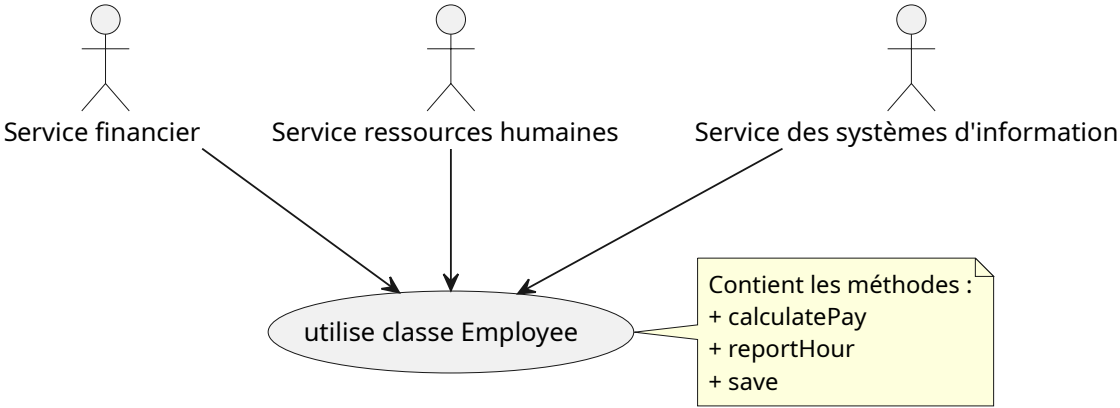
Désavantages :

- Ajout d'une classe difficile
- Accès aux données internes des objets souvent requis par le visiteur

Section 5

Patron de conception *Facade*

Exemple : système de paiement de salaire



Violation SRP

La classe `Employee` dépend de trois acteurs différents car :

- la méthode `calculatePay` qui calcule le salaire dépend du service financier ;
- la méthode `reportHours` qui permet à l'employé d'indiquer ces horaires de travail dépend du service des ressources humaines ;
- la méthode `save` qui permet de sauvegarder l'employé dans la base de donnée dépend du service des systèmes d'information.

Deux problèmes :

- *couplages* de fonctionnalité pouvant entraîner des bugs
- *merges* à gérer dans le cas où deux acteurs font un changement en même temps

Exemple de problème

Méthode `regularHours` présente dans `Employee` pour calculer les heures normales de travail de l'employé.

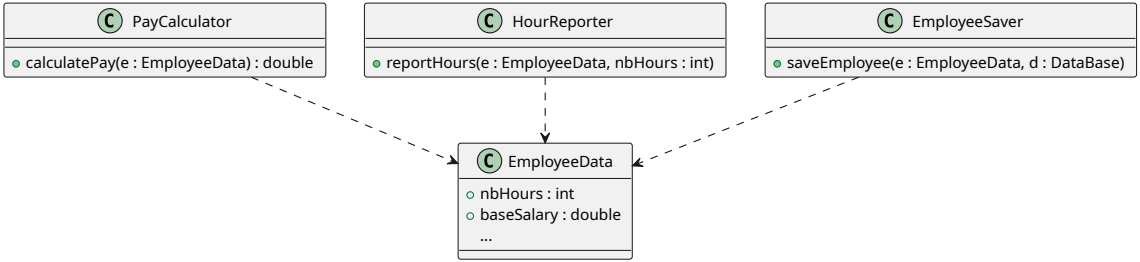
`regularHours` est utilisé par :

- `calculatePay` qui dépend du service financier ;
- `reportHours` qui dépend du service des ressources humaines.

Un service peut modifier la méthode `regularHours` sans voir qu'elle est utilisé par un autre service et donc créer un bug pour l'autre service.

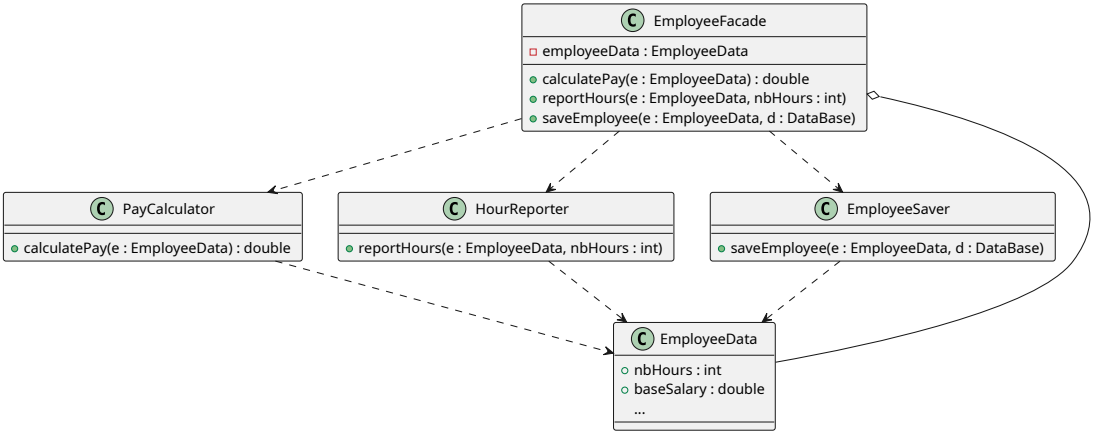
Solution : séparer les données des méthodes

Une solution est de séparer les données des fonctions en créant trois classes distinctes dont chacune accède aux données de l'employé.

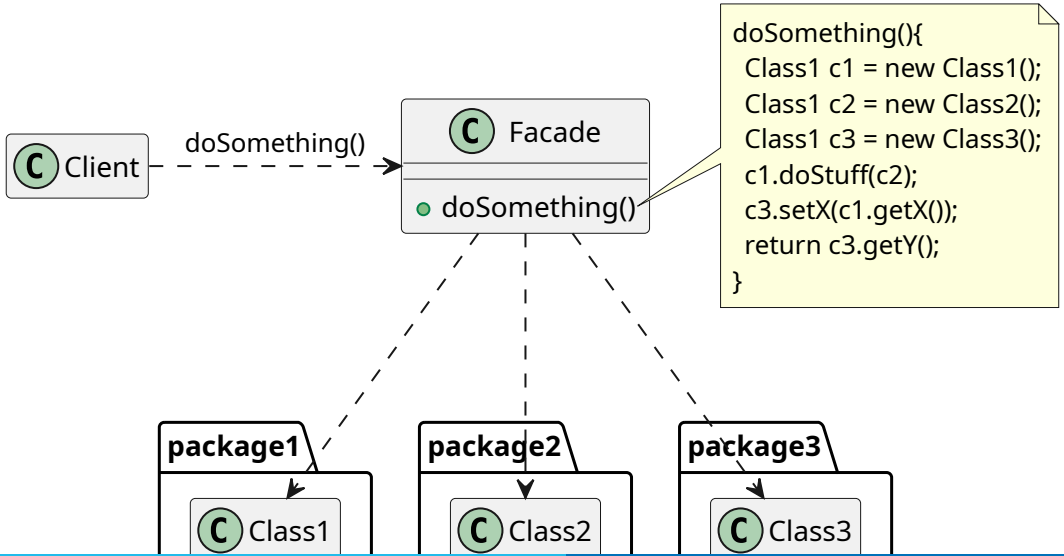


Solution : utiliser le patron de conception *facade*

Si on veut avoir à nouveau qu'un seul objet correspondant à un employé, on peut utiliser le patron de conception *facade*.



Patron de conception *facade*



Patron de conception *facade*

Intention

Donner une interface offrant un accès simplifié à un ensemble complexe de classes.

Avantage :

- Vous pouvez isoler votre code de la complexité d'un sous-système.

Désavantage :

- Une façade peut devenir un objet omniscient couplé à toutes les classes d'une application.