Initiation génie logiciel : Patrons de conception

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

23 octobre 2024

amU Faculté des sciences Aix Marseille Université

Section 1

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 2

Patron de conception *composite*

Exemple : forme géométrique

```
public class Triangle {
  Point2D point1, point2, point3;
}
public class Rectangle{
  Point2D leftTopCorner;
  double height, length;
}
public class Circle{
  Point2D center;
  double radius;
}
```

Méthode contains

```
boolean contains(Point2D point, Object object){
  return switch(object){
    case Rectangle r ->
          (point.x()-r.leftTopCorner.x()>=0)
          &&(point.x()-r.leftTopCorner.x()<=r.length)
          &&(point.y()-r.leftTopCorner.y()>=0)
          &&(point.y()-r.leftTopCorner.y()<=r.height);
    case Circle c ->
          point.distance(c.center) <= c.radius;
    case Triangle t -> ...
    default -> false;
  };
}
```

Est-ce que cette approche respecte les principes SOLID ?

Délégation au sein des classes (1/4)

```
public class Rectangle{
    Point2D leftTopCorner;
    double height, length;

    public boolean contains(Point2D point){
        return (point.x()-leftTopCorner.x()>=0)
                &&(point.x()-leftTopCorner.x()<=length)
                && (point.y()-leftTopCorner.y()>=0)
                &&(point.y()-leftTopCorner.y()<=height);
    }
}
```

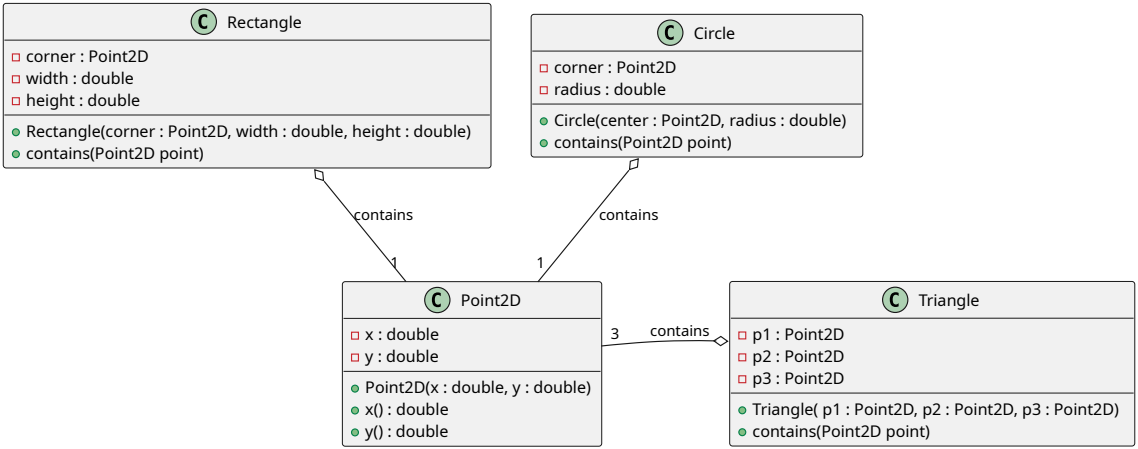
Délégation au sein des classes (2/4)

```
public class Triangle {
  Point2D p1, p2, p3;

  boolean contains(Point2D point) {
    double a = (p1.x()-p2.x()) * (p1.y() - point.y())
             - (p1.y()-p2.y()) * (p1.x() - point.x());
    double b = (p2.x()-p3.x()) * (p2.y() - point.y())
             - (p2.y()-p3.y()) * (p2.x() - point.x());
    double c = (p3.x()-p1.x()) * (p3.y() - point.y())
             - (p3.y()-p1.y()) * (p3.x() - point.x());
    return ((a <= 0) && (b <= 0) && (c <= 0))
           || ((a >= 0) && (b >= 0) && (c >= 0));
  }
}
```

Délégation au sein des classes (3/4)

```java
public class Circle {
    Point2D center;
    double radius;

    public boolean contains(Point2D point) {
        return point.distance(center) <= radius;
    }
}
```

Délégation au sein des classes (4/4)



Comment coder une union de formes ?

```
public class Union {
  List<Object> objects;
  public boolean contains(Point2D point){
    for(Object object : objects) {
      boolean isContained =
        switch (object) {
          case Rectangle r -> r.contains(point);
          case Circle c -> c.contains(point);
          case Triangle t -> t.contains(point);
          default -> false;
        };
      if (isContained) return true;
    }
    return false;
```

Interface Shape

Si on veut une liste qui puisse contenir à la fois des rectangles, des triangles et des cercles, il faut une interface :

```
public interface Shape {
  boolean contains(Point2D point);
}

public class Rectangle implements Shape{...}
public class Triangle implements Shape{...}
public class Circle implements Shape{...}
```

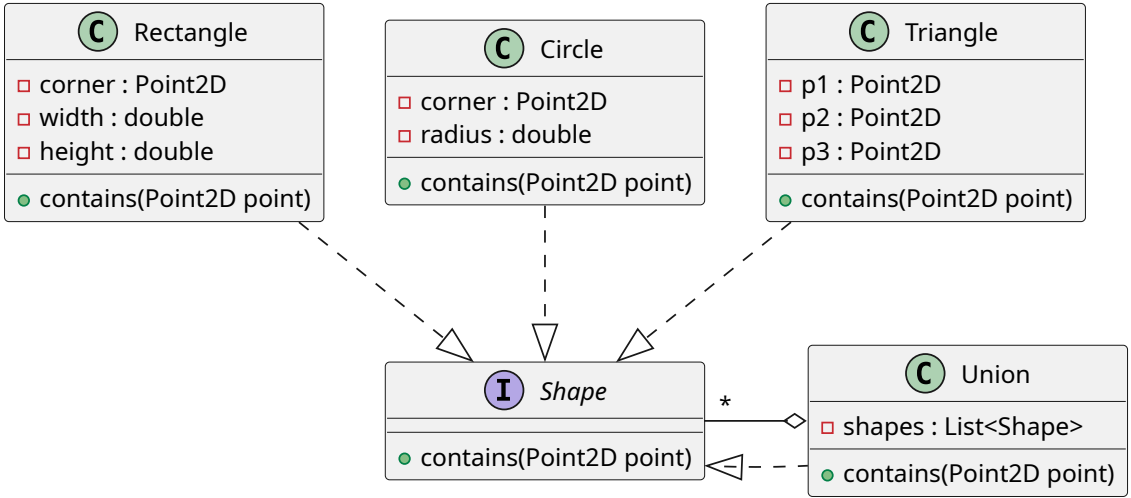
De cette manière on peut ajouter facilement une nouvelle forme : le principe SOLID OCP est respecté.

Union de forme

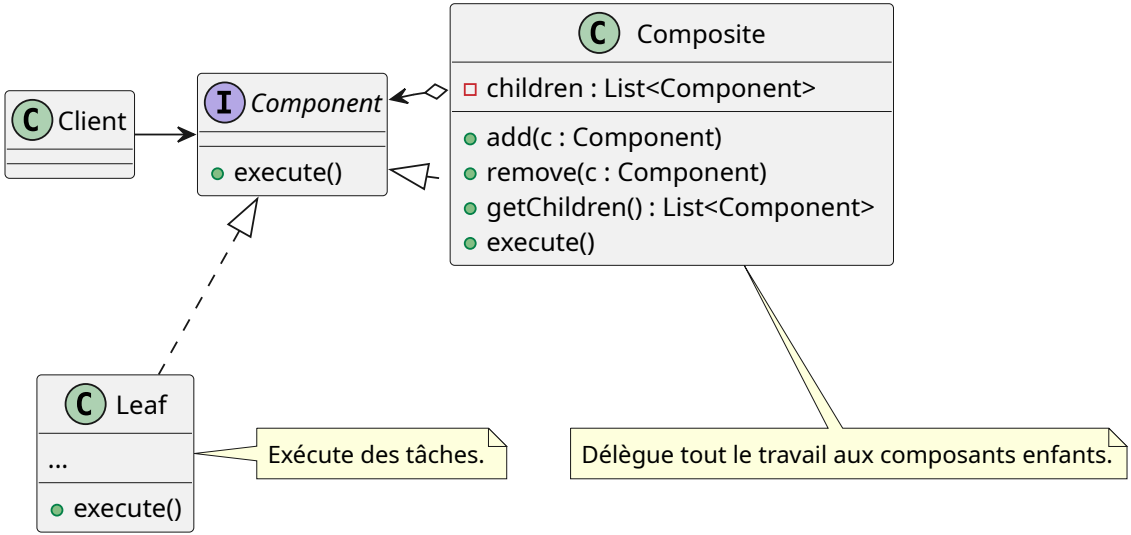
```
public class Union {
  List<Shape> shapes;
  public boolean contains(Point2D point){
    for(Shape shape : shapes)
      if(shape.contains(point))
        return true;
    return false;
  }
}
```

On peut remarquer que `Union` a une méthode `contains` et peut donc implémenter l'interface `Shape`.

Diagramme de classes



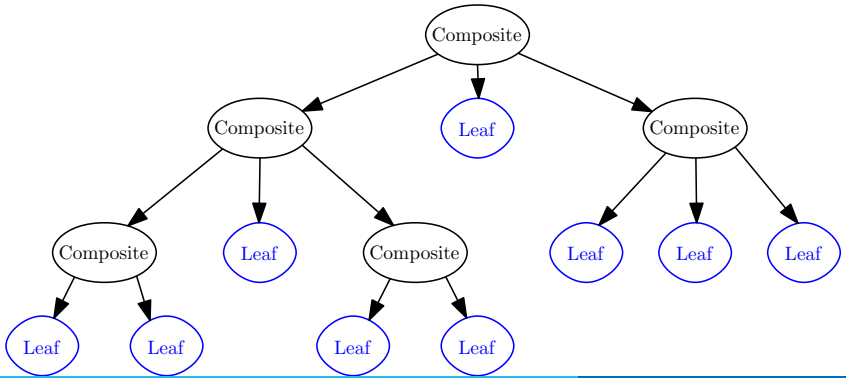
Patron de conception composite



Patron de conception composite

Intention

Permet d'organiser des objets en une structure d'arbre pour réaliser une tâche demandant une action de chaque objet.



Section 3

Patrons de conception *strategy* et *template method*

Classe ListSum

Supposons que nous ayons la classe suivante :

```
public class ListSum {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value;
    size++;
  }
  public int eval() {
    int result = 0;
    for (int i = 0; i < size; i++)
      result += list[i];
    return result;
  }
```

Classe ListProduct

Supposons que nous ayons aussi la classe suivante (très similaire) :

```
public class ListProduct {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) {
    list[size] = value;
    size++;
  }
  public int eval() {
    int result = 1;
    for (int i = 0; i < size; i++)
      result *= list[i];
    return result;
  }
```

Refactoring pour isoler la répétition

Les deux classes sont très similaires et il y a de la répétition de code.

Il est possible de *refactorer* (réécrire le code) les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListSum {
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
  private int neutral() { return 0; }
  private int compute(int a, int b) { return a+b; }
}
```

Refactoring pour isoler la répétition

Il est possible de *refactorer* les classes précédentes de façon à isoler les différences dans des méthodes :

```
public class ListProduct {
  public int eval() {
    int result = neutral();
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]);
    return result;
  }
  private int neutral() { return 1; }
  private int compute(int a, int b) { return a*b; }
}
```

Comment éviter la répétition ?

Après la *refactorisation* du code :

- seules les méthodes `neutral` et `compute` diffèrent
- il serait intéressant de pouvoir supprimer les duplications de code

Deux solutions :

- La délégation en utilisant une interface et l'agrégation $\rightarrow$ patron de conception **Stratégie**.
- L'extension et les classes abstraites $\rightarrow$ patron de conception **Patron de méthode**.

Solution Statégie : interface Operator

Nous allons externaliser les méthodes `neutral` et `compute` dans deux nouvelles classes. Elles vont implémenter l'interface `Operator` :

```
public interface Operator {
  public int neutral();
  public int compute(int a, int b);
}
public class Sum implements Operator {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
public class Product implements Operator {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

Délégation

Les classes `ListSum` et `ListProduct` sont fusionnées dans une unique classe `List` qui délègue les calculs à un objet qui implémente l'interface `Operator` :

```
public class List {
  /* attributs et méthode add */
  private Operator operator;
  public List(Operator operator){
    this.operator = operator;
  }
  public int eval() {
    int result = operator.neutral();
    for (int i = 0; i < size; i++)
      result = operator.compute(result, list[i]);
    return result;
  }
```

Délégation

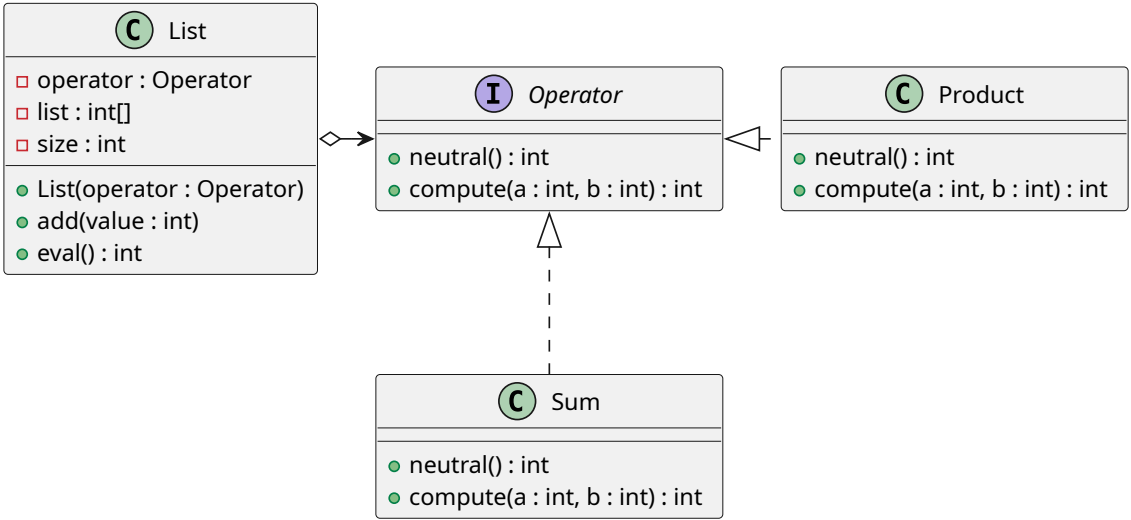
Utilisation des classes `ListSum` et `ListProduct` :

```
ListSum listSum = new ListSum(); listSum.add(2);
listSum.add(3); System.out.println(listSum.eval());
ListProduct listProduct = new ListProduct();
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

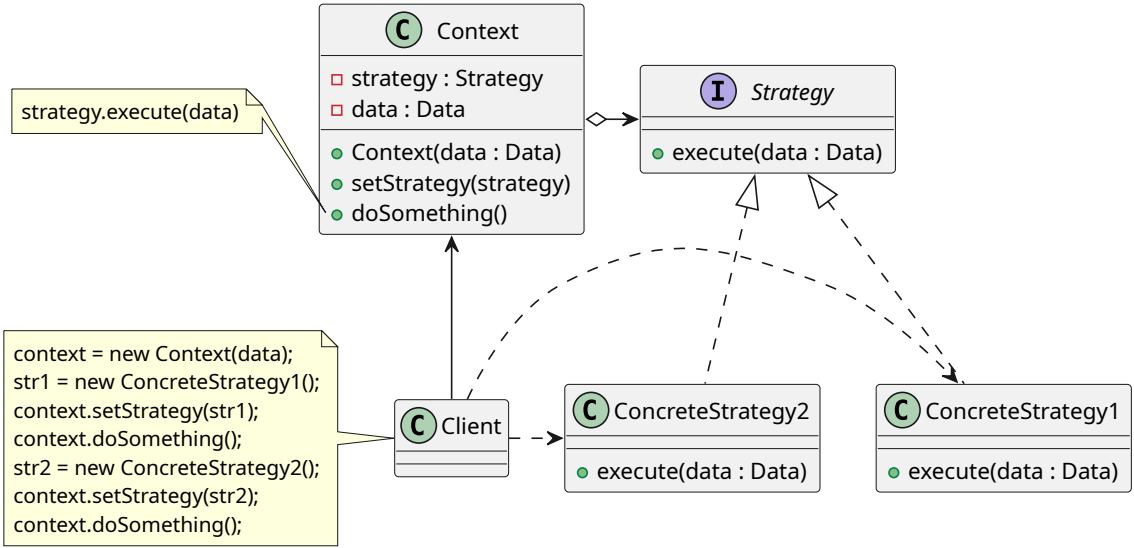
Utilisation après la *refactorisation* du code :

```
List listSum = new List(new Sum()); listSum.add(2);
listSum.add(3); System.out.println(listSum.eval());
List listProduct = new List(new Product());
listProduct.add(2); listProduct.add(3);
System.out.println(listProduct.eval());
```

Diagramme de la solution avec Stratégie



Patron de conception Stratégie



Patron de conception Stratégie

Intention

Définir une famille d'algorithmes, encapsuler chacun d'entre eux et les rendre interchangeables.

Analogie

Pour vous rendre à l'aéroport, vous pouvez prendre

- le bus,
- appeler un taxi ou
- enfourcher votre vélo.

Ce sont vos stratégies de transport et vous en choisissez une en fonction de vos besoins.

Quand utiliser le patron Stratégie ?

Cas d'utilisation

Une classe définit un comportement spécifique avec plusieurs manières de le réaliser.

Solution

- Séparer les différentes manières de réaliser le comportement de la classe en classes séparées appelées stratégies (partageant la même interface).
- La classe originale (le contexte) garde un attribut qui garde une référence vers une des stratégies.
- Plutôt que de s'occuper de la tâche, le contexte la délègue à l'objet stratégie associé.
- Le contexte n'a pas la responsabilité de la sélection de l'algorithme adapté, c'est le client qui lui envoie la stratégie.

Solution utilisant une classe abstraite

```
public abstract class List {
  private int[] list = new int[10];
  private int size = 0;
  public void add(int value) { list[size] = value; size++; }
  public int eval() {
    int result = neutral(); // utilisation d'une méthode abstraite
    for (int i = 0; i < size; i++)
      result = compute(result, list[i]); // idem
    return result;
  }
  public abstract int neutral(); // méthode abstraite
  public abstract int compute(int a, int b); // idem
}
```

Rappel : mot-clé `abstract`

Classe abstraite

- On peut mettre `abstract` devant le nom de la classe à sa définition pour signifier qu'une classe est abstraite.
- Une classe est abstraite si des méthodes ne sont pas implémentées.
  $\Rightarrow$ Classe abstraite = classe avec des méthodes abstraites
- Tout comme pour une interface, une classe abstraite n'est pas instanciable.

Méthode abstraite

- `abstract` devant le nom du type de retour de la méthode à sa définition pour signifier qu'une méthode est abstraite.
- Méthode abstraite = méthode sans code, juste la signature (type du retour et des paramètres) est définie

Classes abstraites et extension

Tout comme pour les interfaces, il n'est pas possible d'instancier une classe abstraite :

```
List list = new List(); // erreur
System.out.println(list.eval()) // car que faire ici ?
```

Nous allons étendre la classe `List` afin de récupérer les attributs et les méthodes de `List` et définir le code des méthodes abstraites :

```
public class ListSum extends List {
  public int neutral() { return 0; }
  public int compute(int a, int b) { return a+b; }
}
```

Classes abstraites et extension

La classe `ListSum` n'est plus abstraite, toutes ses méthodes sont définies :

- les méthodes `add` et `eval` sont définies dans `List` et `ListSum` hérite du code de ses méthodes.
- les méthodes `neutral` et `compute` qui étaient abstraites dans `List` sont définies dans `ListSum`.

On peut donc instancier la classe `ListSum` :

```
ListSum listSum = new ListSum();
listSum.add(3);
listSum.add(7);
System.out.println(listSum.eval());
```

Classes abstraites et extension

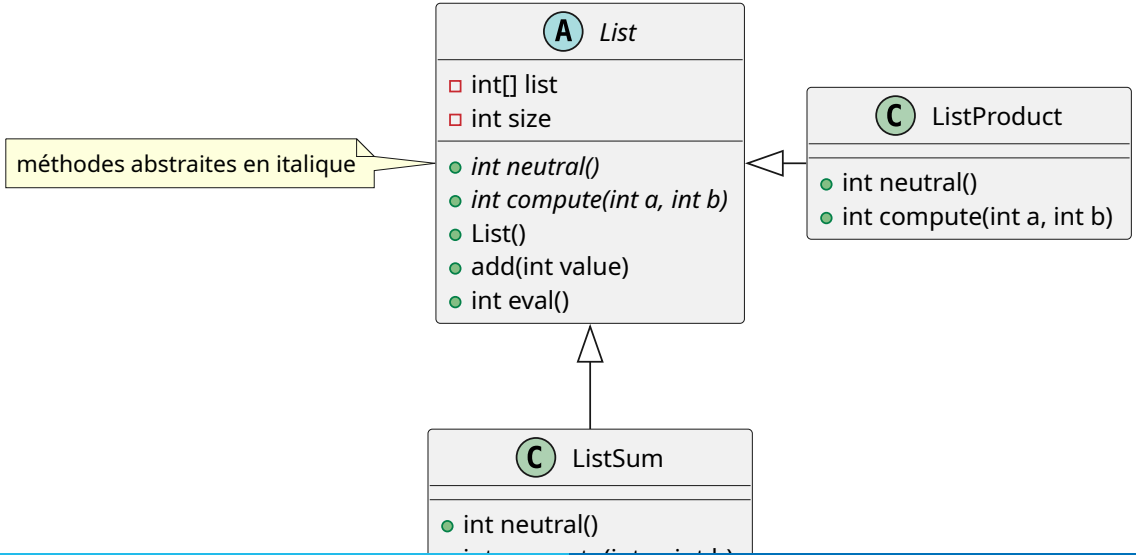
On peut procéder de manière similaire pour créer une classe `ListProduct`

```
public class ListProduct extends List {
  public int neutral() { return 1; }
  public int compute(int a, int b) { return a*b; }
}
```

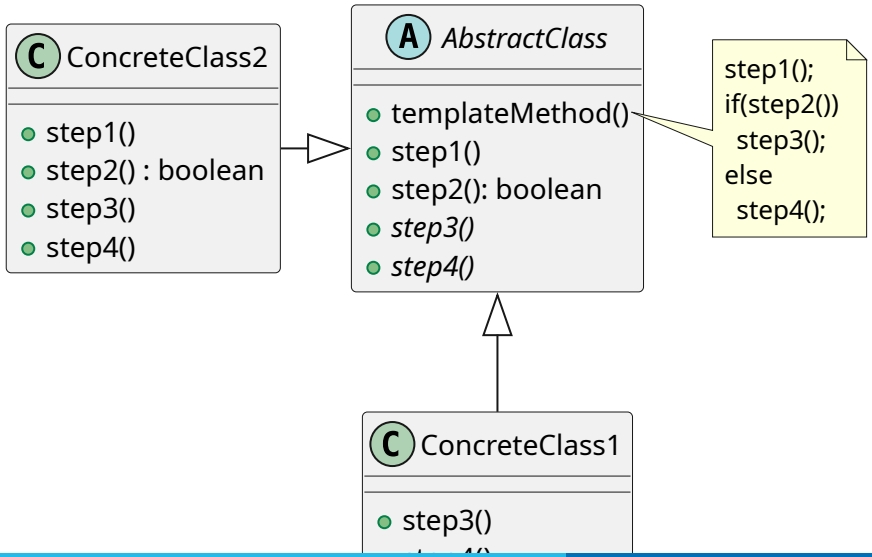
La classe `ListProduct` n'est plus abstraite, toutes ses méthodes sont définies :

```
ListProduct listProduct = new ListProduct();
listProduct.add(3);
listProduct.add(7);
System.out.println(listProduct.eval());
```

Diagramme de la solution avec extension



Patron de conception Patron de méthode



Patron de conception Patron de méthode

Intention

- Définir le squelette d'un algorithme dans une classe mère.
- Laisser les sous-classes redéfinir le code des étapes de l'algorithme sans changer sa structure.

Analogie

Les étapes pour construire une maison sont toujours les mêmes dans le même ordre :

1. poser les fondations,
2. poser la charpente,
3. monter les murs,
4. installer la plomberie pour l'eau et les câbles pour l'électricité.

Chaque étape de construction peut être légèrement modifiée pour différencier un peu la maison des autres.

Quand utiliser le patron de méthode ?

Cas d'utilisation

Plusieurs classes ont la même structure pour un algorithme avec le même enchainement d'étapes, mais une implémentation différente de ces étapes.

Solution

- Découper un algorithme en une série d'étapes et transformer ces étapes en méthodes potentiellement abstraites ;
- Créer une méthode socle exécutant l'algorithme avec des appels à ces méthodes ;
- Fournir une sous-classe concrète en implémentant toutes les étapes abstraites et redéfinissant certaines d'entre elles si besoin

Section 4

Patrons de conception *factory*

Factory

On considère la classe suivante :

```
public interface Button {
  public void draw();
}
```

Ainsi qu'une classe l'implémentant :

```
public class SimpleButton implements Button {
  public void draw() {
    System.out.println("Simple button.");
  }
}
```

La classe `SimpleButton` est instanciée dans de nombreuses autres classes.

Ajout d'un autre bouton

```
public class ModernButton implements Button {
  public void draw() {
    System.out.println("Modern button.");
  }
}
```

Afin d'utiliser ce nouveau bouton, on doit modifier toutes les instanciations présentes dans notre code

$\rightarrow$ violation de OCP

Solution : patron de conception *factory*

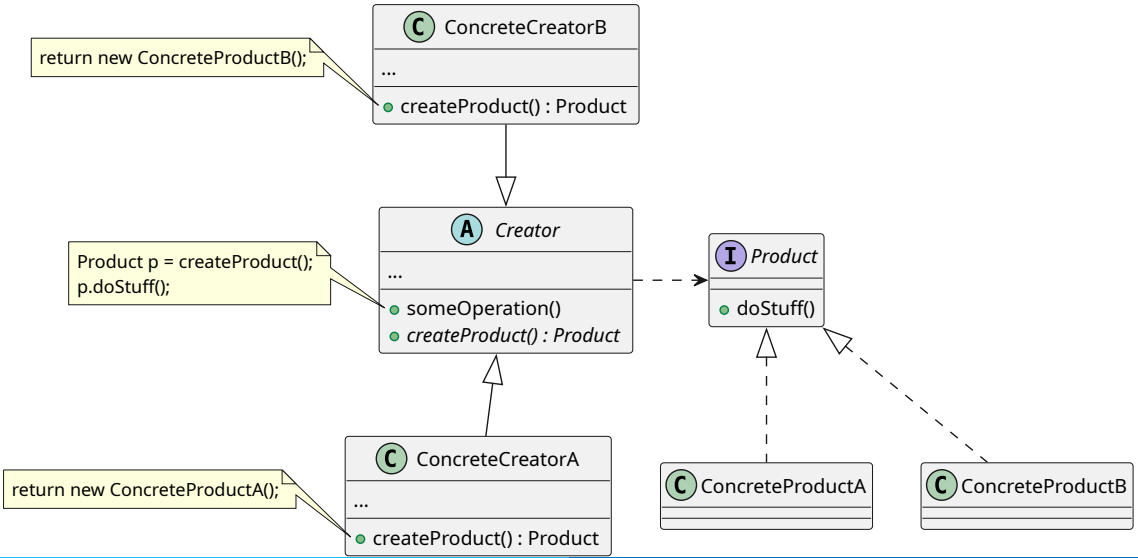
Une fabrique permet d'isoler la création des objets de leurs utilisations :

```
public class ButtonFactory {
  public Button createButton() {
    return new SimpleButton();
  }
}
```

Toutes les instanciations se font via cette classe. Les modifications nécessaires à l'utilisation de la classe `ModernButton` sont isolées :

```
public class ButtonFactory {
  public Button createButton() {
    return new ModernButton();
  }
}
```

Patron de méthode *factory*



Patron de conception *factory*

Intention de *factory*

Définit des méthodes pour créer des objets dans une classe mère, mais délègue le choix des types d'objets à créer aux sous-classes.

Analogie

Vous avez besoin de faire livrer des paquets soit par bateau ou bien par camion.
Les deux types de transports sont interchangeables, car ils rendent le même service de base.
On délègue la création des camions ou des navires à des classes filles (chacune ne construisant qu'un type de véhicule).
$\rightarrow$ il devient facile de passer d'un type de transport à l'autre

Section 5

Patrons de conception *abstract factory*

Fabrique de plusieurs objets

Imaginons que la fabrique fournisse différents éléments :

```
public class Factory {
  public Button createButton() { return new SimpleButton(); }
  public TextBox createTextBox() { return new SimpleTextBox(); }
  public List createList() { return new SimpleList(); }
}
```

Pour passer du style simple au moderne, il nous faut changer toutes les méthodes de la fabrique :

```
public class Factory {
  public Button createButton() { return new ModernButton(); }
  public TextBox createTextBox() { return new ModernTextBox(); }
  public List createList() { return new ModernList(); }
}
```

Patron de conception *abstract factory*

Le code suivant semble résoudre le problème :

```
public class Factory {
  private String type;
  public Factory(String type) { this.type = type; }
  public Button createButton() {
    if (type.equals("modern")) return new ModernButton();
    else return new SimpleButton();
  }
  public TextBox createTextBox() {
    if (type.equals("modern")) return new ModernTextBox();
    else return new SimpleTextBox();
  }
  /* Idem pour createList. */
}
```

Problème pour l'introduction de variante d'objet

Que se passe-t-il nous avons besoin d'introduire un nouveau type ?

```
public class Factory {
  private String type;
  public Factory(String type) { this.type = type; }
  public Button createButton() {
    if (type.equals("modern")) return new ModernButton();
    else if (type.equals("simple")) return new SimpleButton();
    else return new OtherButton();
  }
  /* Idem pour createTextBox et createList. */
}
```

Toutes les méthodes de la fabrique doivent être modifiées, ce qui est une violation de OCP.

Introduction d'une interface fabrique

Les fabriques abstraites permettent de corriger ce défaut :

```
public interface AbstractFactory {
  public Button createButton();
  public TextBox createTextBox();
  public List createList();
}
```

Les classes qui devront instancier des boutons, des boites de texte ou des listes auront à leur disposition une instance d'une classe qui implémente l'interface `ButtonFactory` :

```
AbstractFactory factory = new SimpleFactory();//new ModernFactory();
Button button = factory.createButton();
/* ... */
```

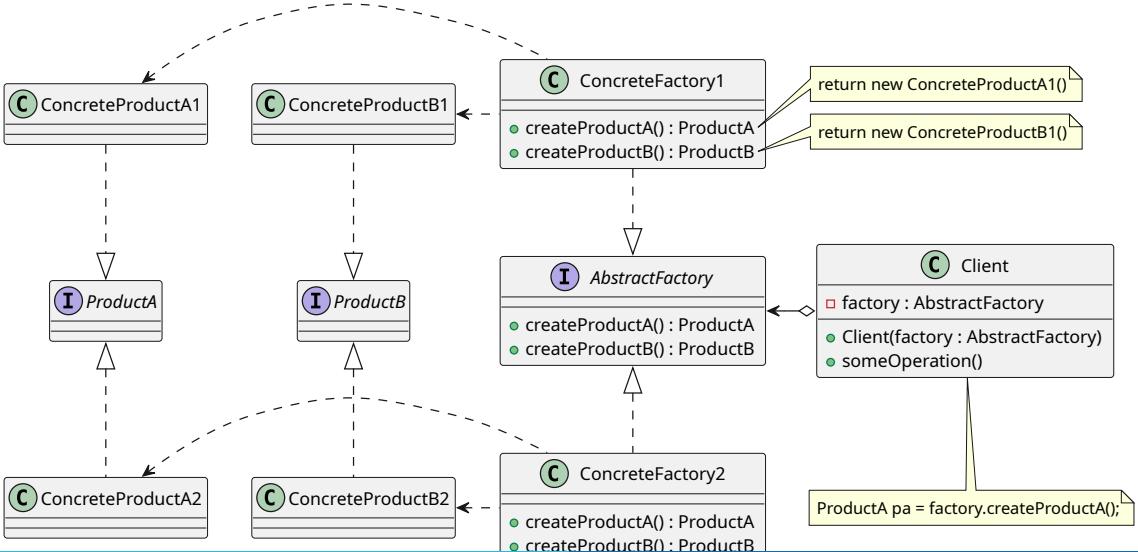
Implémentation de l'interface fabrique

On implémente ensuite une fabrique par type :

```
public class SimpleFactory implements AbstractFactory {
  public Button createButton() { return new SimpleButton(); }
  public TextBox createTextBox() { return new SimpleTextBox(); }
  public List createList() { return new SimpleList(); }
}
public class ModernFactory implements AbstractFactory {
  public Button createButton() { return new ModernButton(); }
  public TextBox createTextBox() { return new ModernTextBox(); }
  public List createList() { return new ModernList(); }
}
```

L'ajout d'un nouveau type s'effectue par l'ajout d'une nouvelle classe, ce qui respecte OCP.

Patron de conception *abstract factory*



Patron de conception *abstract factory*

Intention

Permet de créer des familles d'objets apparentés sans préciser leur classe concrète.

Utilisation du patron

Vous avez des variantes de familles de produits (Chaise, Sofa et Table) qui sont disponibles dans des variantes (Moderne ou Victorien).
La fabrique abstraite donne une interface pour créer tous les produits d'une famille.
Pour chaque variante de produits, on implémente l'interface avec la création des produits de la variante.

Quand utiliser *Abstract factory* ?

Quand on doit :

- manipuler des produits d'un même thème (garantie de compatibilité entre les objets) ;
- découpler le code client des produits concrets ;
- déléguer la création des objets à une autre classe (SRP) ;
- ajouter des variantes d'objet (OCP).

Section 6

Patron de conception *adapter*

Interface Pencil de dessin

Supposons que nous ayons la classe et l'interface suivantes :

```
public class Drawer {
  public void draw(Pencil pencil) {
    pencil.drawLine(0,0,10,10);
    pencil.drawCircle(5,5,5);
    pencil.drawLine(10,0,0,10);
  }
}
public interface Pencil {
  public void drawLine(int x1, int y1, int x2, int y2);
  public void drawCircle(int x, int y, int radius);
}
```

Classe de dessin `Crayon` non-compatible

Nous avons également la classe Crayon suivante :

```
public class Crayon {
  public void dessinerLigne(Point p1, Point p2) {
    /* ... */
  }
  public void dessinerCercle(Point center, int rayon) {
    /* ... */
  }
}
```

On souhaite utiliser la classe `Crayon` comme un `Pencil` pour qu'elle puisse être utilisée par la classe `Drawer`.

Solution : classe *adapter*

Pour ce faire, nous définissons l'adaptateur suivant :

```
public class CrayonAdapter implements Pencil {
  private Crayon crayon;
  public CrayonAdapter(Crayon c) { this.crayon = c; }
  public void drawLine(int x1, int y1, int x2, int y2) {
    crayon.dessinerLigne(new Point(x1, y1), new Point(x2, y2));
  }
  public void drawCircle(int x, int y, int radius) {
    crayon.dessinerCercle(new Point(x, y), radius);
  }
}
```

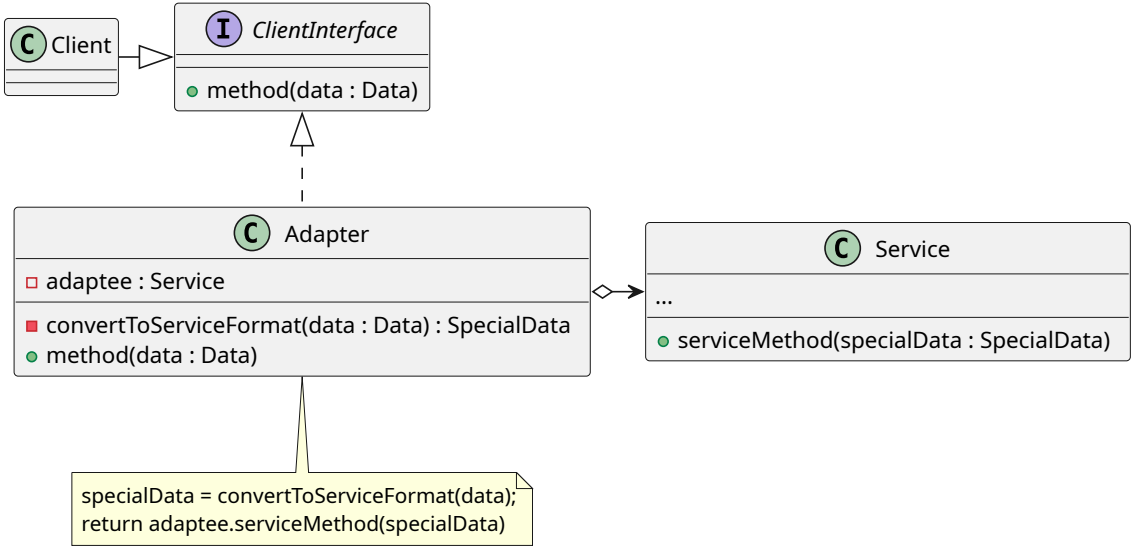
Solution

Cette nouvelle classe est utilisable de la façon suivante :

```
Pencil pencil = new CrayonAdapter(new Crayon());
Drawer drawer = new Drawer();
drawer.draw(pencil);
```

On n'a pas modifié les classes existantes (respect OCP).

Patron de conception *adapter*



Patron de conception *adapter*

Intention

Permet de faire collaborer des objets ayant des interfaces normalement incompatibles.

Utilisation du patron

Classe déjà écrite qui délègue certaines opérations à une autre classe. Changer la classe déléguée en utilisant une autre classe pas tout à fait compatible (opérations similaires, mais pas la même interface).
Solution :

1. Créer une interface pour les opérations déléguées
2. Créer un adaptateur qui convertit l'interface de votre nouvelle classe

Avantages et inconvénients

Avantages :

- Découpler l'interface ou le code de conversion des données, de la logique métier du programme (SRP).
- Permet de réutiliser du code existant sans le modifier (OCP)

Désavantage :

- Peut rajouter de la complexité inutile dans le code (parfois plus simple de recoder la classe service).

Section 7

Patron de conception *State*

Classe avec état

```
public class Writer {
  private int state = 0;
  public void write(char character) {
    switch (state) {
      case 0 -> {
        if (character=='{') state = 1;
        else System.out.print(character);
      }
      case 1 -> {
        if (character=='}') state = 0;
        else System.out.print(Character.toUpperCase(character));
      }
    }
  }
```

Utilisation de la classe

```
Writer writer = new Writer();
String string = "abc{def}ghi";
for (int index = 0; index < string.length(); index++)
writer.write(string.charAt(index));
```

Le code précédent génère la sortie suivante : `abcDEFghi`

Notez que si nous souhaitons ajouter de nouveaux états, nous devons modifier les méthodes de la classe Writer

$\rightarrow$ violation d'OCP

De plus, le fait qu'une classe implémente un grand nombre d'états peut être considéré comme une violation de SRP.

Solution

Pour corriger ces défauts, nous déclarons la classe et l'interface suivantes :

```
public class WriterContext {
  private WriterState state = new WriterState0();
  public void write(char character) {
    state.write(this, character);
  }
  public void changeState(WriterState state) {
    this.state = state;
  }
}
public interface WriterState {
  public void write(WriterContext context, char character);
}
```

État

Nous pouvons maintenant définir les différents états :

```
public class WriterStateIdentity implements WriterState {
  public void write(WriterContext context, char c) {
    if (c=='{') {context.changeState(new WriterStateUpper());}
    else System.out.print(c);
  }
}
public class WriterStateUpperCase implements WriterState {
  public void write(WriterContext context, char c) {
    if (c=='}') {context.changeState(new WriterStateIdentity());}
    else System.out.print(Character.toUpperCase(c));
  }
}
```

Utilisation

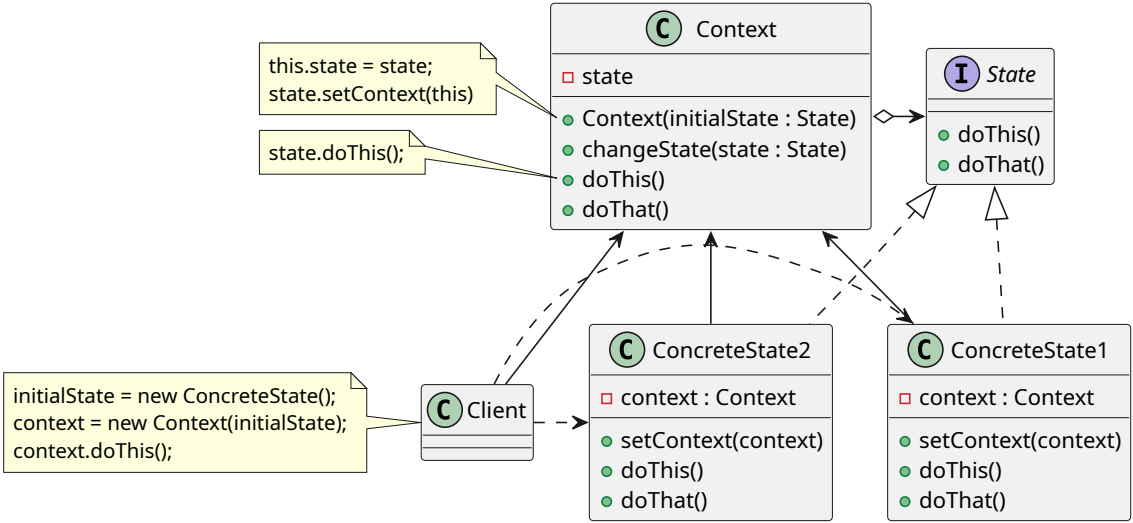
Un exemple d'utilisation de la classe précédente :

```
WriterContext writer = new WriterContext();
String string = "abc{def}ghi";
for (int index = 0; index < string.length(); index++)
writer.write(string.charAt(index));
```

Ce code génère la sortie suivante : `abcDEFghi`

Notez que l'ajout d'un nouvel état s'effectue en ajoutant une nouvelle classe qui implémente l'interface `WriterState` (respect d'OCP).

Diagramme du patron *state*



Patron de conception *state*

Intention

Permet de modifier le comportement d'un objet lorsque son état interne change.

Solution

Implémenter un automate fini en créant :

1. une interface état contenant les méthodes que doit faire chaque état ;
2. pour chaque état possible une classe pour cet état qui gère les opérations et les transitions ;
3. une classe contexte contenant un état et déléguant les opérations à l'état courant.

Avantages et inconvénients

Avantages :

- Organiser le code des différents états dans des classes séparées (SRP).
- Ajouter de nouveaux états sans modifier les classes état ou le contexte existants (OCP).
- Simplifiez le code du contexte en éliminant les gros blocs conditionnels de l'automate.

Désavantages :

- Créer de la complexité inutile (patron surdimensionné pour un automate ayant peu d'états et de transitions)

Section 8

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

méthodes \ classes	Rectangle	Circle	Polygon	...
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>" ;
  }
}
```

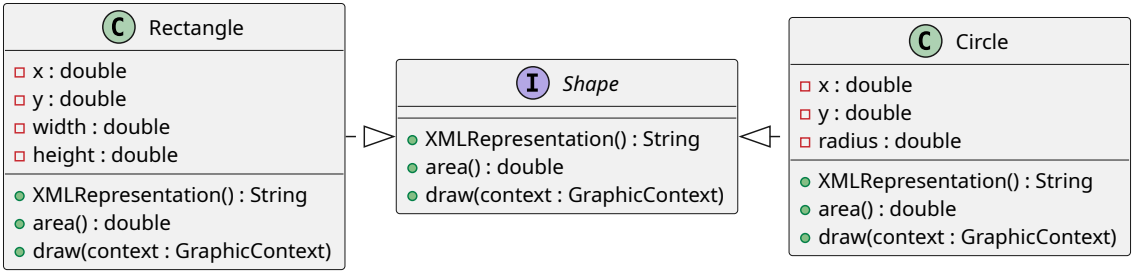
Implémentation du visiteur AreaVisitor (1/2)

```
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 */
}
```

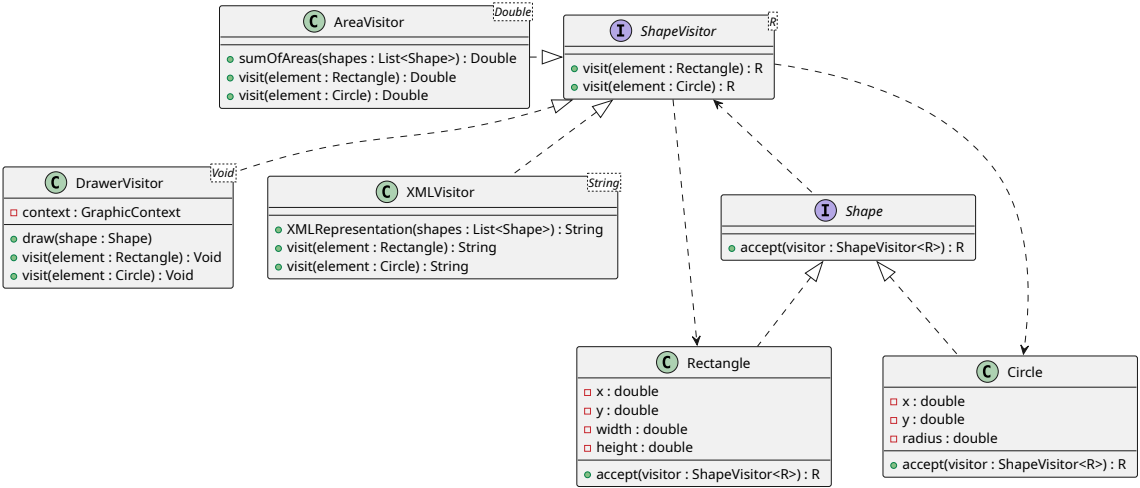
Implémentation du visiteur AreaVisitor (2/2)

```
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)

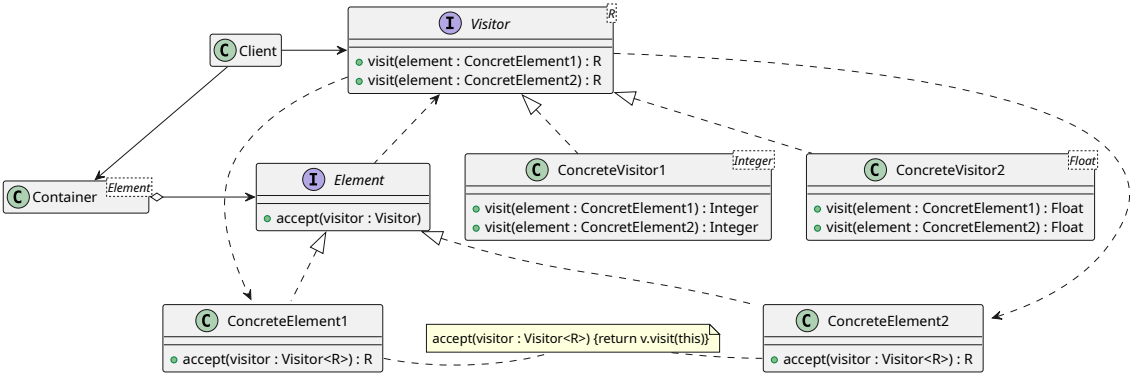
- Simplicité lors de l'ajout d'une nouvelle forme
- Couplage dessin/forme
- Une seule façon de dessiner par forme
- Méthodes de dessin éclatées dans de nombreuses classes

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

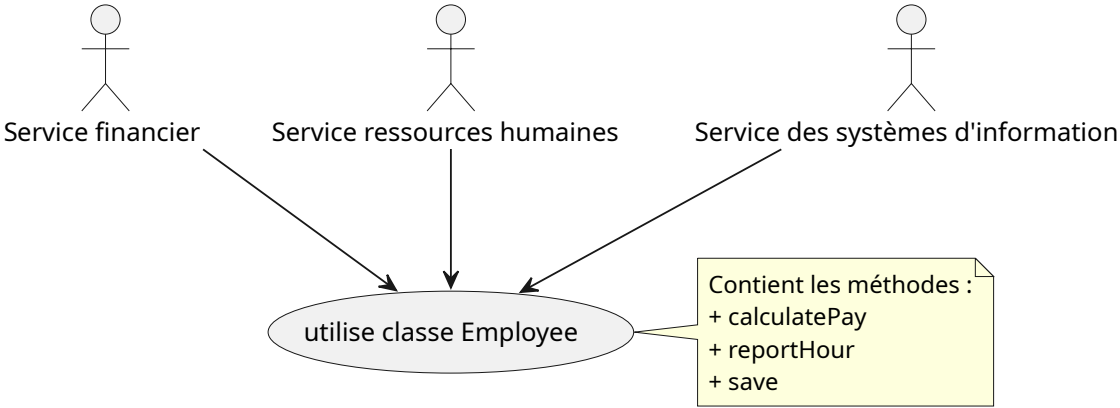
Désavantages :

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

Section 9

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.

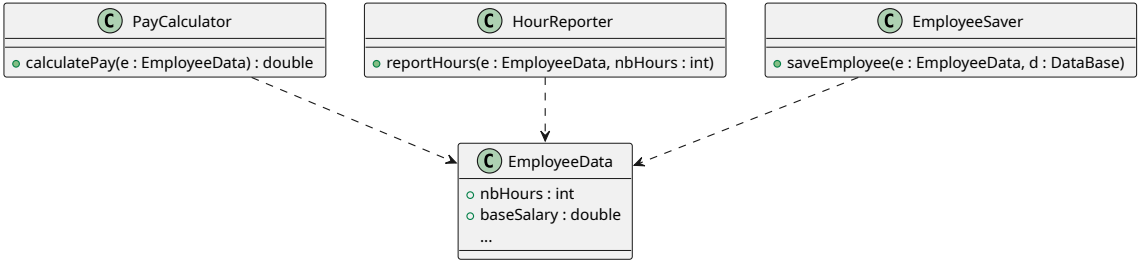
`regularHours` est utilisé par :

- `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.

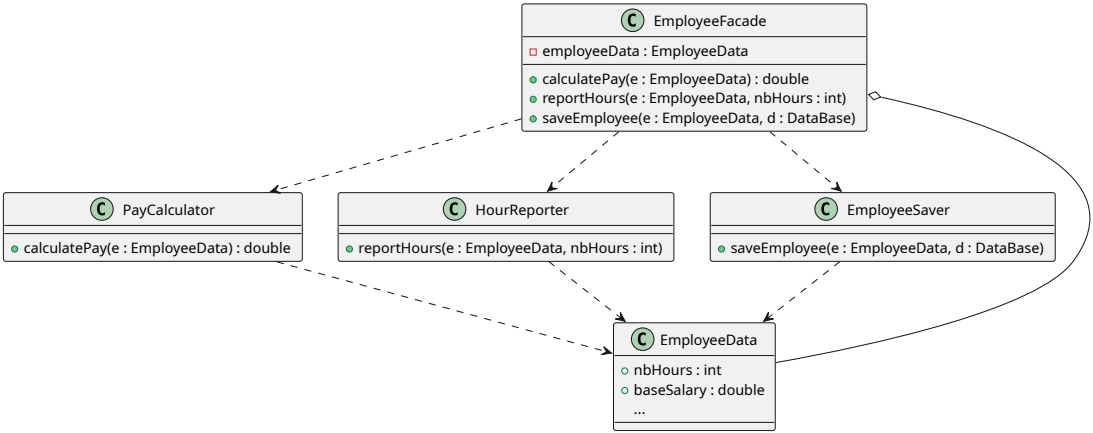
Solution : séparer les données des méthodes

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

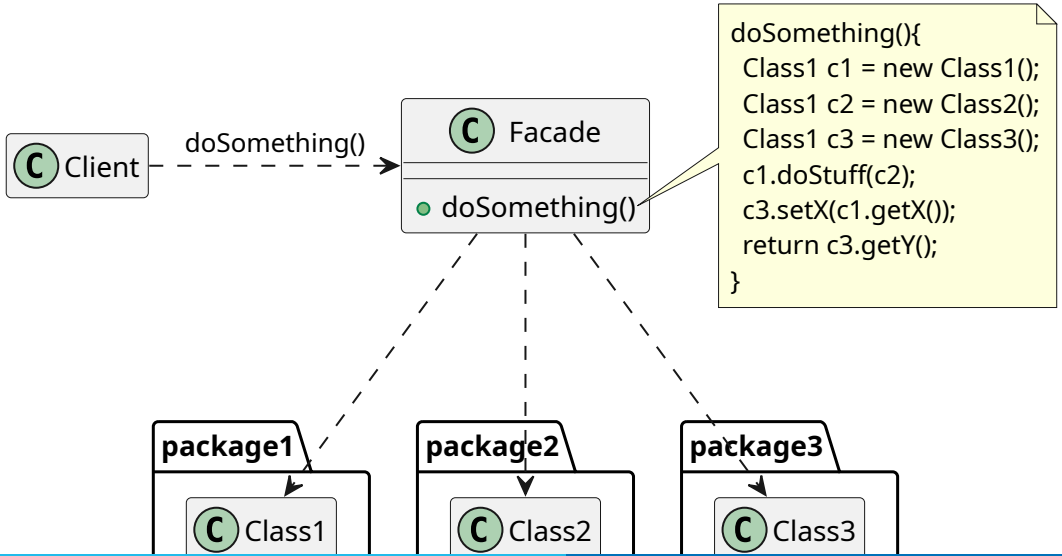


Solution : utiliser le patron de conception *facade*

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.

Section 10

Patron de conception *Proxy*

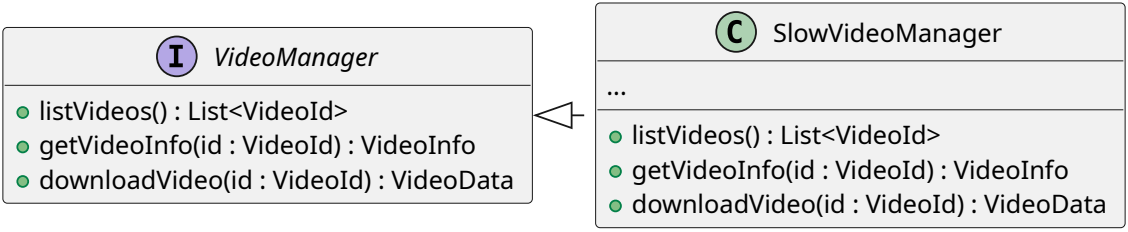
Service de vidéos

On suppose qu'on a accès à une classe permettant d'accéder à des vidéos.

Problème : le service est lent et on aimerait mettre en cache si possible les réponses aux requêtes.

Introduction d'une interface

On commence par introduire une interface pour le service de vidéo :



On va rajouter une classe de procuration implémentant cette interface et déléguant le téléchargement à l'outil de téléchargement original.

La classe de procuration garde la trace de tous les fichiers téléchargés et retourne les données du cache lorsque l'application fait plusieurs fois appel à la même vidéo.

Classe *proxy* CachedVideoManager

```
public class CachedVideoManager implements VideoManager {
  private SlowVideoManager service;
  private Map<VideoID,VideoData> videoDataCache;
  private Map<VideoID,VideoInfo> videoInfoCache;
  private List<VideoId> listeCache;

  public CachedVideoManager(){
    service = new SlowVideoManager(/* ... */);
    videoDataCache = new HashMap<>();
    videoInfoCache = new HashMap<>();
  }
}
```

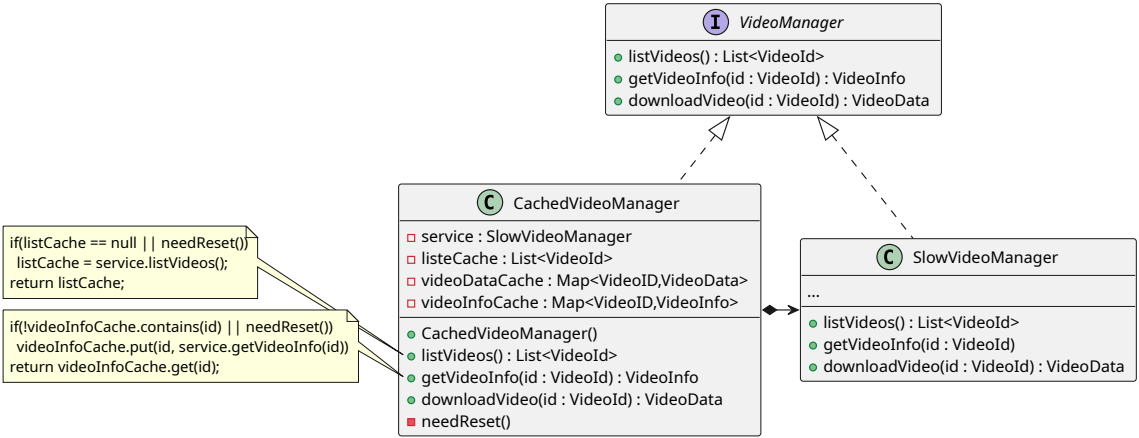
Classe *proxy* CachedVideoManager

```
public class CachedVideoManager implements VideoManager {
  private boolean needReset(){
    /* test if a reset is needed */
  }
  public List<VideoId> listVideos(){
    if(listCache == null || needReset())
      listCache = service.listVideos();
    return listCache;
  }
}
```

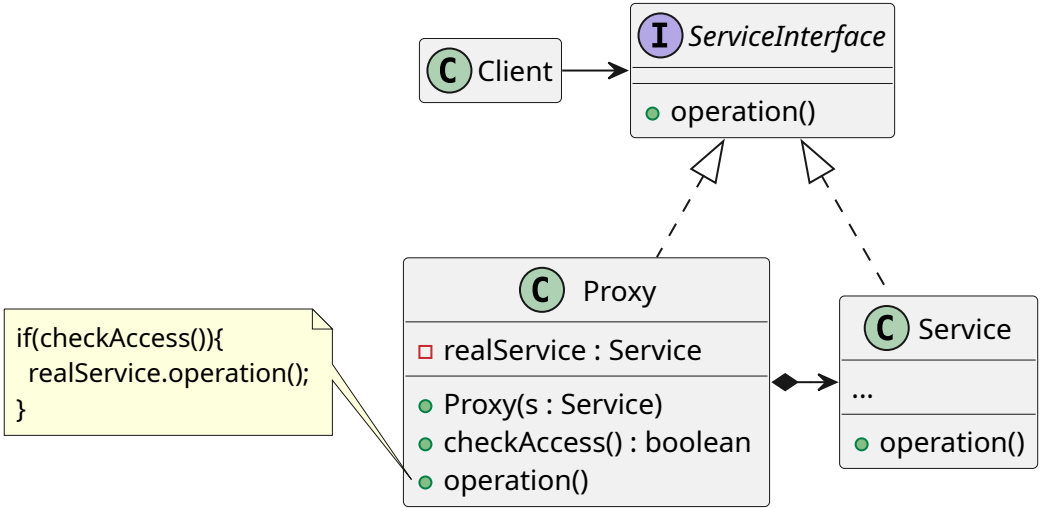
Classe *proxy* CachedVideoManager

```
public class CachedVideoManager implements VideoManager {
  public VideoInfo getVideoInfo(VideoId id){
    if(!videoInfoCache.contains(id) || needReset())
      videoInfoCache.put(id, service.getVideoInfo(id))
    return videoInfoCache.get(id);
  }
  public VideoData getVideoData(VideoId id){
    if(!videoDataCache.contains(id) || needReset())
      videoDataCache.put(id, service.getDataInfo(id))
    return videoDataCache.get(id);
  }
}
```

Classe *proxy* CachedVideoManager



Patron de conception *proxy*



Patron de conception *proxy*

Intention

Permet d'utiliser un substitut pour un objet donnant le contrôle sur l'objet original.

Avantages :

- Contrôler l'objet du service sans que le client ne s'en aperçoive.
- Ajout possible de procuration sans toucher au service (OCP)
- Gestion du cycle de vie de l'objet du service originel

Désavantages :

- Code plus complexe, car introduction de classes supplémentaires
- Intermédiaire pouvant ralentir les requêtes.

Section 11

Patron de conception *Decorator*

Pile

Supposons que nous avons la classe suivante :

```
public class ArrayStack {
  private int[] elements = new int[10];
  private int size = 0;

  public void push(int value) {
    elements[size] = value;
    size++;
  }
  public int pop() {
    size--; elements[size]= null;
    return elements[size];
  }
}
```

Pile avec *log*

Nous souhaitons ajouter des logs (`print`) pour débugger notre programme :

```
public class ArrayStack {
  private int[] elements = new int[10];
  private int size = 0;
  public void push(int value) {
    System.out.println("push("+value+")");
    elements[size] = value; size++;
  }
  public int pop() {
    System.out.println("pop()");
    size--; elements[size]= null;
    return elements[size];
  }
}
```

Modification de Stack

Cette modification a été réalisée en modifiant une classe existante (violation OCP). De plus, une nouvelle modification est nécessaire pour retirer les logs.

Définissons l'interface suivante :

```
public interface Stack {
  public void push(int value);
  public int pop();
}
```

Faisons en sorte que `ArrayStack` implémente cette interface :

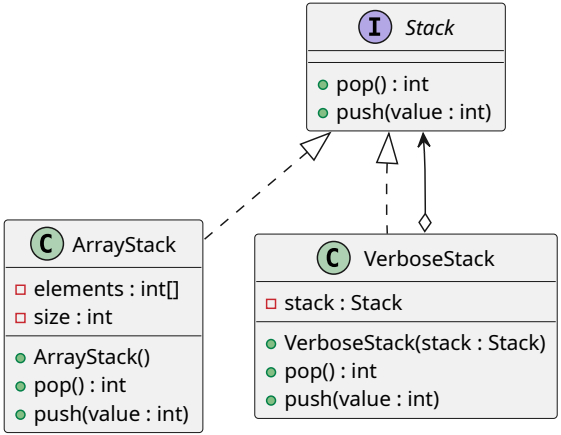
```
public class ArrayStack implements Stack {
  /* ... */
}
```

Classe VerboseStack décorant Stack

Il suffit alors de définir une classe *decorator* `VerboseStack` implémentant `Stack` :

```
public class VerboseStack implements Stack {
  private Stack stack;
  public VerboseStack(Stack stack) { this.stack = stack; }
  public void push(int value) {
    System.out.println("push("+value+")");
    stack.push(value);
  }
  public int pop() {
    System.out.println("pop()");
    return stack.pop();
  }
}
```

Principe de VerboseStack



`VerboseStack` :

- implémente `Stack`
- délègue une partie du comportement des méthodes de `Stack` à une autre instance implémentant `Stack`
- rajoute un comportement (*print*) aux méthodes de `Stack` (décoration)

Exemple d'utilisation

Supposons que nous ayons le code suivant :

```
Stack stack = new ArrayStack(10);
stack.push(2); stack.pop();
```

Il est très facile d'introduire le décorateur :

```
Stack stack = new ArrayStack(10);
stack = new VerboseStack(stack);
stack.push(2);
stack.pop();
```

Ce code produit la sortie suivante :

```
push(2)
pop()
```

Nouvelle classe de décoration

Nous définissons un nouveau décorateur :

```
public class ParenthesesStack implements Stack {
  private Stack stack;
  private String parentheses;
  public VerboseStack(Stack stack) {
    this.stack = stack; parentheses = "";
  }
  public void push(int value) {
    parentheses += "(";
    stack.push(value);
  }
  public int pop() { parentheses += ")"; return stack.pop(); }
  public String getParentheses() { return parentheses; }
}
```

Exemple d'utilisation

Un exemple d'utilisateur du décorateur précédent :

```
Stack stack = new ArrayStack(10);
Stack parenthesesStack = new ParenthesesStack(stack);
stack.push(2); stack.push(3); stack.pop();
stack.push(1); stack.pop(); stack.pop();
System.out.println(parenthesesStack.getParentheses());
```

Ce code produit la sortie suivante :

```
(()())
```

Plusieurs décorateurs

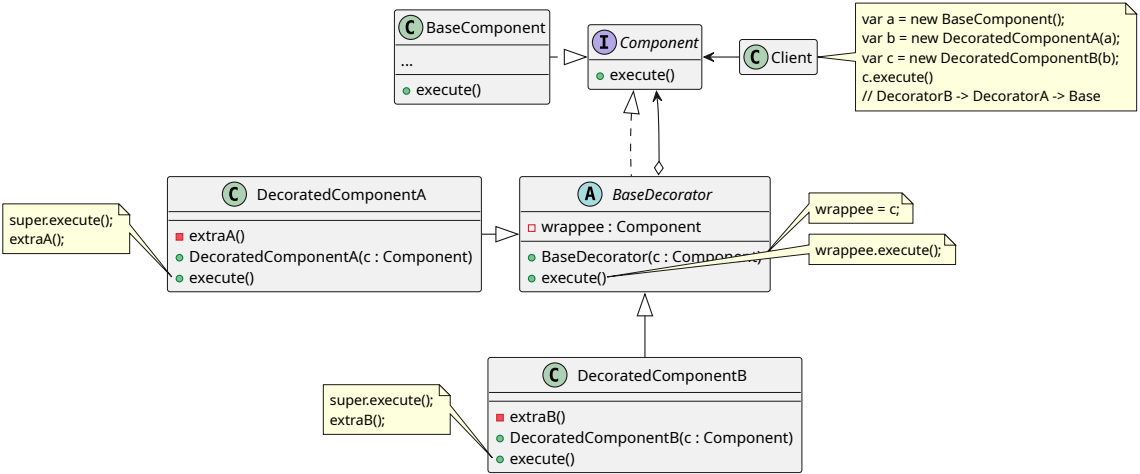
Il est possible d'utiliser plusieurs décorateurs simultanément :

```
Stack stack = new ArrayStack(10);
stack = new VerboseStack(stack);
Stack parenthesesStack = new ParenthesesStack(stack);
stack = parenthesesStack;
stack.push(2); stack.push(3); stack.pop();
System.out.println(parenthesesStack.getParentheses());
```

Ce code produit la sortie suivante :

```
push(2)
push(3)
pop()
(()
```

Patron de conception *decorator*



Patron de conception *decorator*

Intention

Permet d'affecter dynamiquement de nouveaux comportements à des objets en les plaçant dans des emballeurs qui implémentent ces comportements.

Avantages :

- Séparer le code d'une classe monolithique en plusieurs classes (SRP).
- Gestion dynamique (à l'exécution) des comportements des objets.
- Permet de chainer des modifications (décorations) de comportement.

Désavantages :

- Retirer un emballeur spécifique de la pile n'est pas chose aisée.
- Position dans le chaînage influant le comportement.
- Code de mise en place un peu complexe.

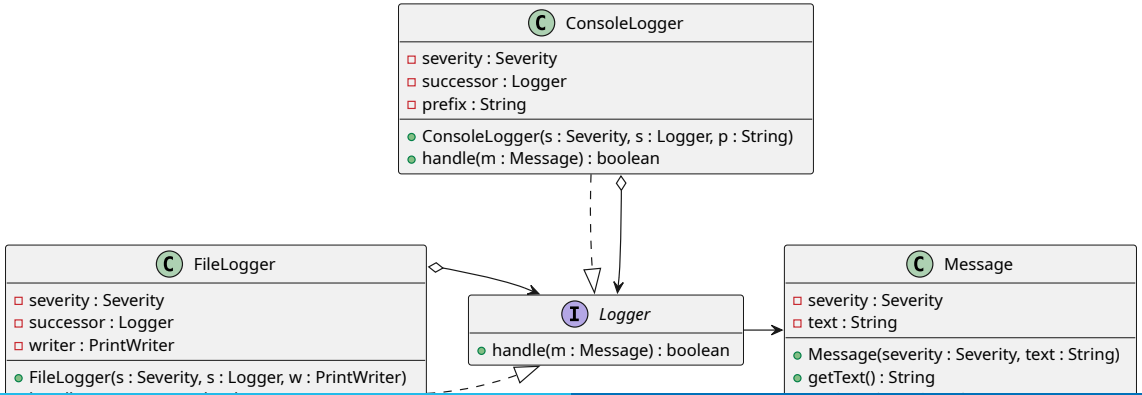
Section 12

Patron de conception *Chain of responsability*

Gestion de log de messages

On souhaite faire une gestion de *log* avec :

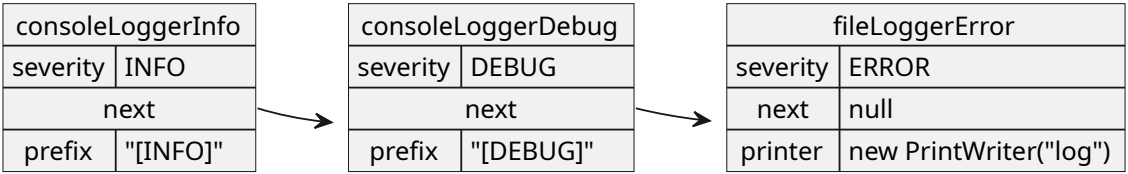
- Plusieurs niveaux de Log : `INFO`, `DEBUG`, `ERROR`
- Plusieurs façons d'écrire les messages : console ou fichier



Classe Message

```
public class Message {
  public enum Severity { INFO, DEBUG, ERROR }
  public final Severity severity;
  public final String text;

  public String getText() { return text; }
  public Severity getSeverity() { return severity; }

  public Message(Severity severity, String message) {
    this.severity = severity;
    this.text = message;
  }
}
```

Utilisation



Exemple d'utilisation :

```
Logger logger = new FileLogger(Message.Severity.ERROR,
                               new PrintWriter("log"), null);
logger = new ConsoleLogger(Message.Severity.DEBUG,"[DEBUG]", logger);
logger = new ConsoleLogger(Message.Severity.INFO,"[INFO]", logger);

logger.handle(new Message(Message.Severity.ERROR,"error"));
logger.handle(new Message(Message.Severity.DEBUG,"debug"));
logger.handle(new Message(Message.Severity.INFO,"info"));
```

Classe abstraite AbstractLogger

```
public abstract class AbstractLogger implements Logger {
  public Message.Severity severity; private Logger successor;
  public Logger(Logger successor, Message.Severity severity) {
    this.successor = successor; this.severity = severity;
  }
  public boolean handle(Message message) {
    if (acceptMessage(message)) return true;
    if (successor!=null) return successor.handle(message);
    return false;
  }
  public abstract boolean acceptMessage(Message message);
  protected boolean mayAcceptMessage(){
    return message.severity == severity;
  }
```

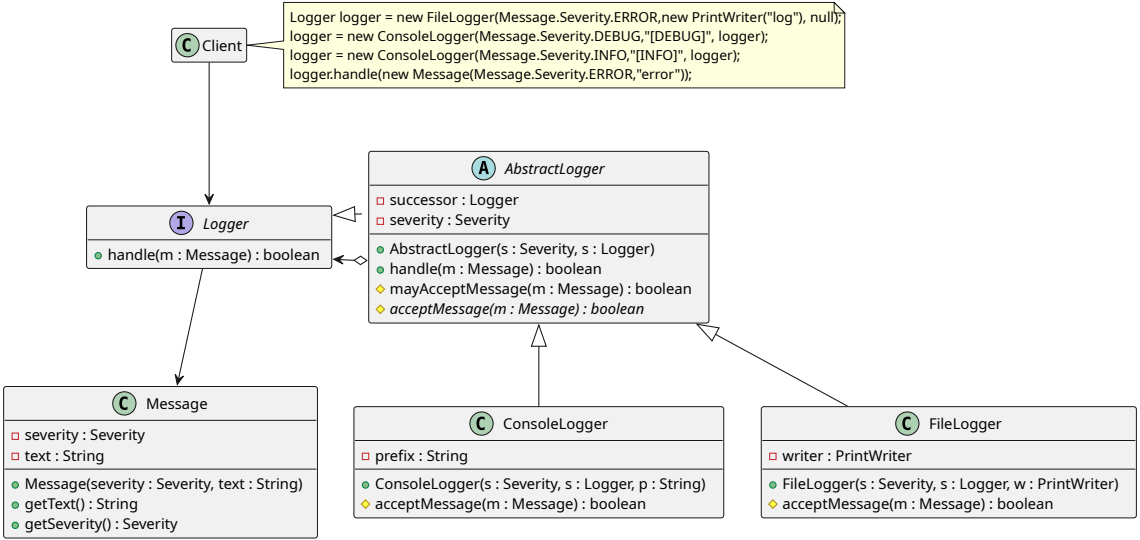
Classe FileLogger

```
public class FileLogger extends AbstractLogger {
  public final PrintWriter writer;
  public FileLogger(Message.Severity severity, PrintWriter writer,
                    Logger successor) {
    super(successor, severity); this.writer = writer;
  }
  protected boolean acceptMessage(Message message) {
    if (mayAcceptMessage()) {
      writer.println(message.text);
      return true;
    }
    return false;
  }
}
```

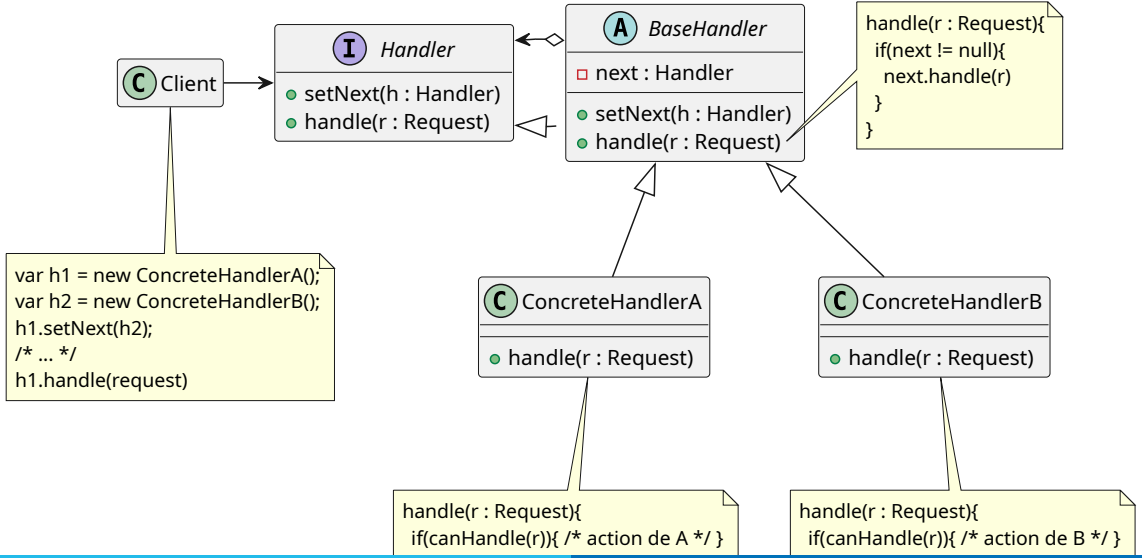
Classe ConsoleLogger

```
public class ConsoleLogger extends AbstractLogger {
  private final String prefix;
  public ConsoleLogger(Message.Severity severity, String prefix,
                       Logger successor) {
    super(successor, severity); this.prefix = prefix;
  }
  protected boolean acceptMessage(Message message) {
    if (mayAcceptMessage()) {
      System.out.println(prefix+" "+message.text);
      return true;
    }
    return false;
  }
}
```

Diagramme de classe final



Patron de conception *Chain of responsability*



Patron de conception *Chain of responsibility*

Intention

Permet de faire circuler des demandes dans une chaîne de *handlers*. Lorsqu'un *handler* reçoit une demande, il décide de la traiter ou de l'envoyer au *handler* suivant de la chaîne.

Avantages :

- Possibilité de contrôler de l'ordre des traitements des demandes.
- Séparer les classes appelant les traitements des classes les implémentant (SRP).
- Facile de rajouter de nouveaux *handler* sans modifier les classes existantes (OCP).

Désavantages :

- Pas de garantie de traitement : les demandes acceptées par aucun *handler* ne sont pas traitées.

Section 13

Patron de conception *observer*

Exemple d'utilisation

On cherche à concevoir un éditeur collaboratif en ligne permettant l'édition de fichier.

On souhaite avoir les deux fonctionnalités suivantes :

- La possibilité pour les utilisateurs du service de s'inscrire/se désinscrire à un système de notification permettant de recevoir des emails lors des changements sur les fichiers.
- Un système de *log* enregistrant les changements effectués sur les fichiers dans un fichier de *log*.

La solution

Le patron de conception *observer* :

- un `EventManager` gérant les listes d'objets voulant être notifiés.
- des `EventListener` attendant d'être notifiés.

enum EventType et classe EventManager

Un enum EventType pour définir les types d'événement (sauvegarde et ouverture de fichier)

```
public enum EventType{ SAVE, OPEN }
```

Une classe EventManager notifiant des EventListener :

```
public class EventManager {
  Map<FileOperation,List<EventListener>> listeners = new HashMap<>();
  public EventManager(EventType... eventTypes) {
    for (EventType eventType : eventTypes)
      this.listeners.put(eventType, new ArrayList<>());
  }
  // suite au transparent suivant
}
```

Classe EventManager

```
public class EventManager {
  public void subscribe(EventType e, EventListener listener) {
    listeners.get(e).add(listener);
  }
  public void unsubscribe(EventType e, EventListener listener) {
    listeners.get(e).remove(listener);
  }
  public void notify(EventType e, File file) {
    List<EventListener> users = listeners.get(e);
    for (EventListener listener : users) {
      listener.update(eventType, file);
    }
  }
}
```

Classe Editor (1/2)

Éditeur concret permettant d'ouvrir et sauvegarder des fichiers.

```
public class Editor {
  public final EventManager events;
  private File file;
  public Editor() {
    this.events = new EventManager(EventType.OPEN, EventType.SAVE);
  }
  public void openFile(String filePath) {
    this.file = new File(filePath);
    events.notify(EventType.OPEN, file);
  }
  // suite au transparent suivant
}
```

Classe Editor (2/2)

```
public class Editor {
  // suite du transparent précédent
  public void saveFile() throws Exception {
    if (this.file != null) {
      events.notify(EventType.SAVE, file);
    } else {
      throw new Exception("Please open a file first.");
    }
  }
}
```

Interface d'écoute EventListener

Un objet voulant être notifié devra implémenter l'interface suivante :

```
public interface EventListener {
    void update(EventType eventType, File file);
}
```

C'est à l'objet observant de définir l'action à accomplir lorsqu'un événement se produit.

Classe EmailNotificationListener

Il est possible de créer une classe envoyant un email à chaque événement pour lequel il est notifié.

```
public class EmailNotificationListener implements EventListener {
  private String email;
  public EmailNotificationListener(String email) {
    this.email = email;
  }
  public void update(EventType eventType, File file) {
    System.out.println("Email to " + email
        + ": Someone has performed " + eventType
        + " operation with the following file: "
        + file.getName());
  }
}
```

Classe LogOpenListener

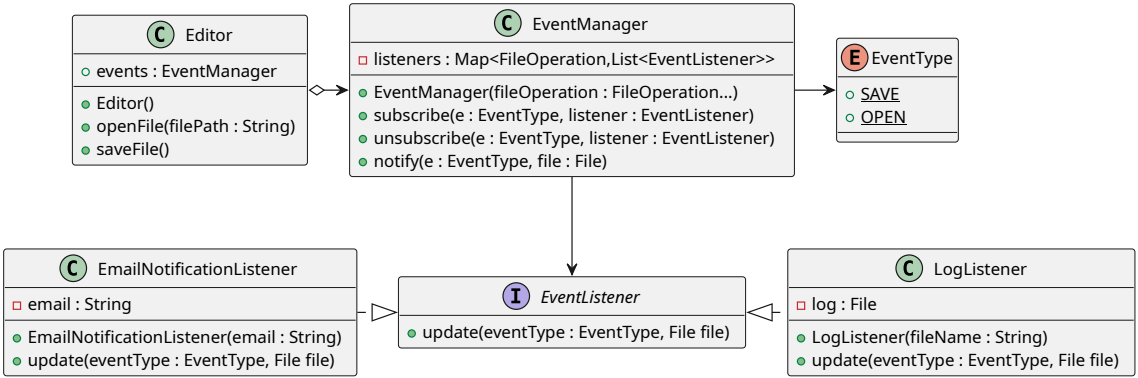
Il est possible de créer une classe écrivant une ligne de log dans un fichier pour chaque événement pour lequel il est notifié.

```
public class LogListener implements EventListener {
  private File log;
  public LogListener(String fileName) {
    this.log = new File(fileName);
  }
  public void update(String eventType, File file) {
    System.out.println("Save to log " + log
      + ": Someone has performed " + eventType
      + " operation with the following file: " + file.getName());
  }
}
```

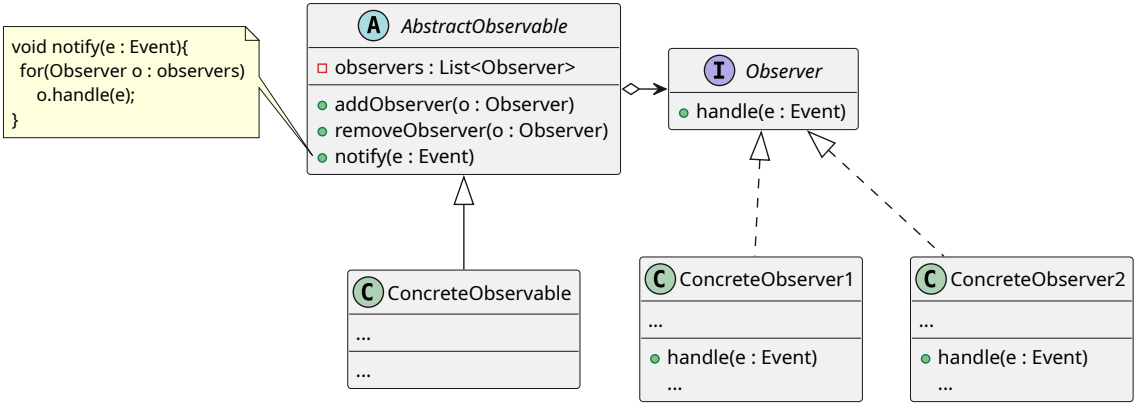
Classe Demo : *log* des *open* et email des *save*

```
public class Demo {
  public static void main(String[] args) {
    Editor editor = new Editor();
    editor.events.subscribe(EventType.OPEN,
      new LogListener("/path/to/log/file.txt"));
    editor.events.subscribe(EventType.SAVE,
      new EmailNotificationListener("admin@example.com"));
    try {
      editor.openFile("test.txt");
      editor.saveFile();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Diagramme de la solution



Patron de conception *observer*



Patron de conception *observer*

Intention

Mécanisme de souscription pour envoyer des notifications à plusieurs objets observateurs lorsqu'une opération est effectuée sur un objet observé.

Avantages :

- Facile de rajouter des classes d'observateurs (OCP).
- Permet d'établir des relations à l'exécution.

Désavantage :

- pas de réel contrôle sur l'ordre de notification des observateurs.

Section 14

Patron de conception *command*

Classe Calculator

On a implémenté la classe suivante qui simule une calculatrice avec des opérations de base :

```
public class Calculator {
  private double value;
  public void setValue(double value) { this.value = value; }
  public double value() { return value; }
  public void add(double value) { this.value += value; }
  public void sub(double value) { this.value -= value; }
  public void divide(double value) { this.value /= value; }
  public void multiply(double value) { this.value *= value; }
}
```

Gestion des *undo* via un `Invoker`

On souhaite pouvoir écrire le code suivant :

```
Calculator calculator = new Calculator();
Invoker invoker = new Invoker(calculator);
invoker.add(new SetCommand(10));
invoker.add(new AddCommand(20));
invoker.add(new DivideCommand(3));
System.out.println(calculator.value()); // 10.0
invoker.undo();
System.out.println(calculator.value()); // 30.0
invoker.redo();
System.out.println(calculator.value()); // 10.0
invoker.undo(); invoker.add(new DivideCommand(2));
System.out.println(calculator.value()); // 15.0
```

Classe Invoker

Exécution des commandes ajoutées :

```
public class Invoker {
  private List<Command> commands;
  private Calculator calculator;
  public Invoker(Calculator calculator) {
    this.calculator = calculator;
    this.commands = new List<>();
  }
  public void add(Command command) {
    command.execute(calculator);
    commands.add(command);
  }
  /* ... */
}
```

Classe Invoker avec *undo* (1/3)

Nous ajoutons la possibilité d'annuler une commande :

```
public class Invoker {
  private List<Command> commands;
  private int undoDepth;
  public Invoker(Calculator calculator) {
    this.undoDepth = 0; /* ... */
  }
  public void undo() {
    if (undoDepth >= commands.size()) return;
    int position = commands.size() - undoDepth - 1;
    Command command = commands.get(position);
    command.unexecute(calculator);
    undoDepth++;
  }
```

Classe Invoker avec *undo* (2/3)

Nous supprimons l'historique si une commande est ajoutée :

```
public class Invoker {
  /* ...*/
  public void add(Command command) {
    clearHistory();
    command.execute(calculator); commands.add(command);
  }
  private void clearHistory() {
    while (undoDepth > 0){
      commands.remove(commands.size()-1);
      undoDepth--;
    }
  }
}
```

Classe Invoker avec *undo* (3/3)

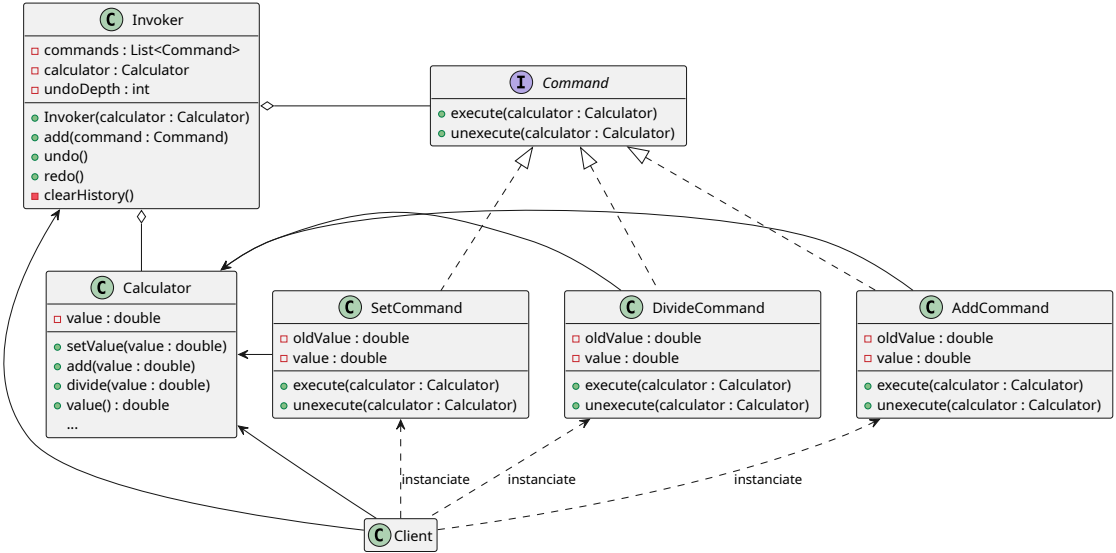
Nous ajoutons la possibilité d'exécuter à nouveau une commande annulée :

```
public class Invoker {
  /* ...*/
  public void redo() {
    if (undoDepth == 0) return;
    int position = commands.size() - undoDepth;
    Command command = commands.get(position);
    command.execute(calculator);
    undoDepth--;
  }
}
```

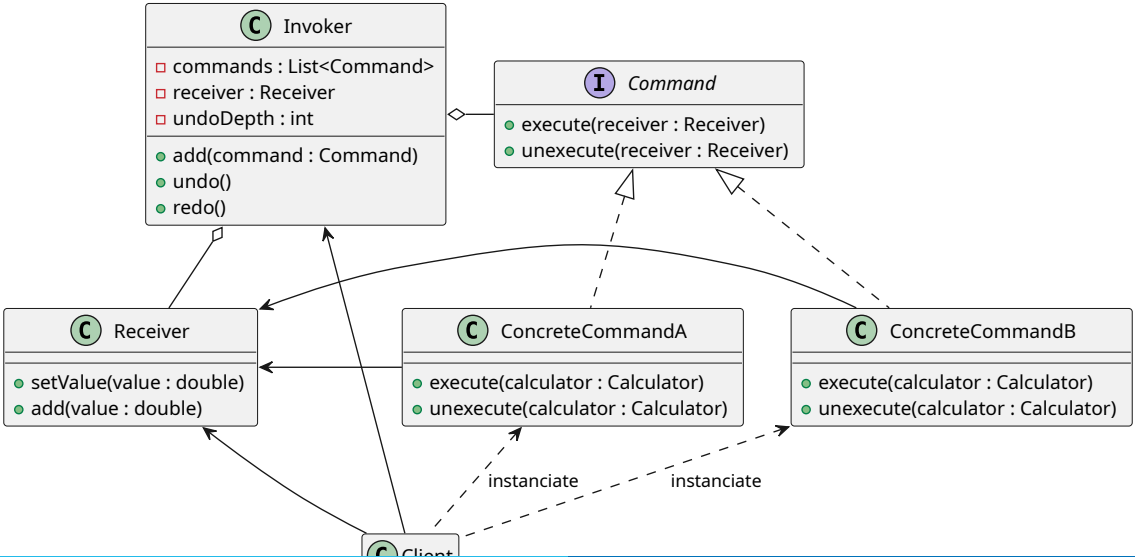
Classe DivideCommand

```
public class DivideCommand implements Command {
  private double oldValue;
  private final double value;
  public DivideCommand(double value) { this.value = value; }
  public void execute(Calculator calculator) {
    oldValue = calculator.value();
    calculator.divide(value);
  }
  public void unexecute(Calculator calculator) {
    calculator.setValue(oldValue);
  }
}
```

Diagramme de la solution



Patron de conception *command*



Patron de conception *command*

Intention

Transformer des actions en des objets autonomes qui contiennent tous les détails des actions. Permet de planifier leur exécution et d'avoir un mécanisme d'annulation

Avantages :

- Découpler les classes qui appellent des traitements, de celles qui les exécutent (SRP)
- Ajout facile de nouvelles commandes (OCP)
- Mise en place *undo*/*redo*

Désavantages :

- Code plus complexe par l'ajout d'une couche entre les demandeurs et les récepteurs.

Section 15

Patron de conception *builder*

Classe User avec beaucoup d'attributs

```
public class User {
  private final String firstName;
  private final String lastName;
  private final int age;
  private final String phone;
  private final String address;
  public User(String firstName, String lastName,
  int age, String phone, String address) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.age = age;
    /* ... */
  }
}
```

Classe User avec beaucoup de constructeurs

```java
public class User {
  /* ... */
  public User(String firstName, String lastName) {
    this(firstName, lastName, 0);
  }
  public User(String firstName, String lastName, int age) {
    this(firstName, lastName, age, "");
  }
  public User(String firstName, String lastName,
              int age, String phone) {
    this(firstName, lastName, age, phone, "");
  }
}
```

Problématique

Dans le code donné précédemment, on a :

- beaucoup de constructeurs différents ;
- un nombre de constructeurs risque d'augmenter ;

On a donc une violation d'OCP (et de SRP).

Solution (patron de conception *builder*) :

- Séparer la construction de l'objet et sa représentation

Avantages :

- Proposer une interface pratique pour construire les objets ;
- Contrôler les étapes de création de l'objet ;

Classe UserBuilder : gestion champs obligatoires

On crée une classe UserBuilder pour construire des User :

```
public class UserBuilder {
  private final String firstName;
  private final String lastName;
  private int age = 0;
  private String phone = "";
  private String address = "";

  public UserBuilder(String firstName, String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

Setters pour les champs facultatifs

```
public class UserBuilder { // suite du slide précédent
  public UserBuilder setAge(int age) {
    this.age = age;
    return this;
  }
  public UserBuilder setPhone(String phone) {
    this.phone = phone;
    return this;
  }
  public UserBuilder setAddress(String address) {
    this.address = address;
    return this;
  }
}
```

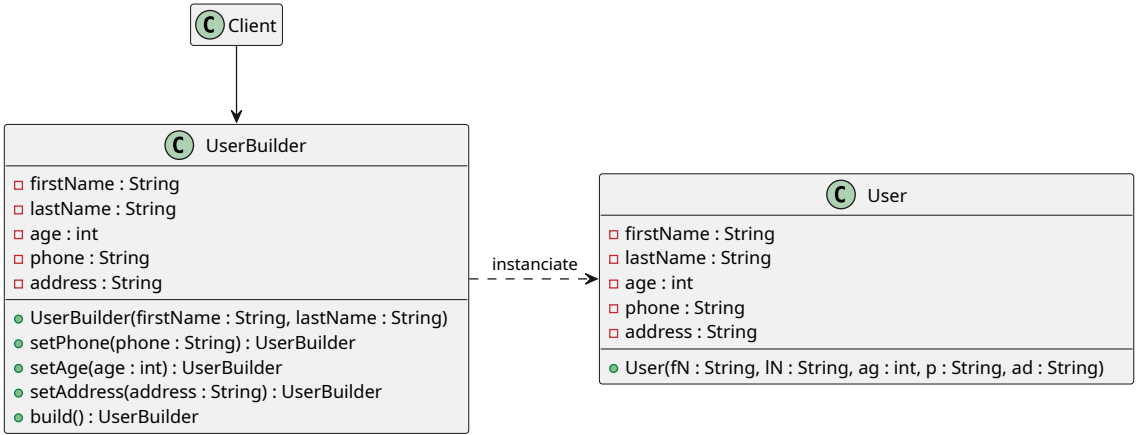
Méthode de construction de l'objet

```
public class UserBuilder {
  public User build() {
    /* Possibles vérifications avant de construire. */
    return new User(firstName, lastName, age, phone, address);
  }
}
```

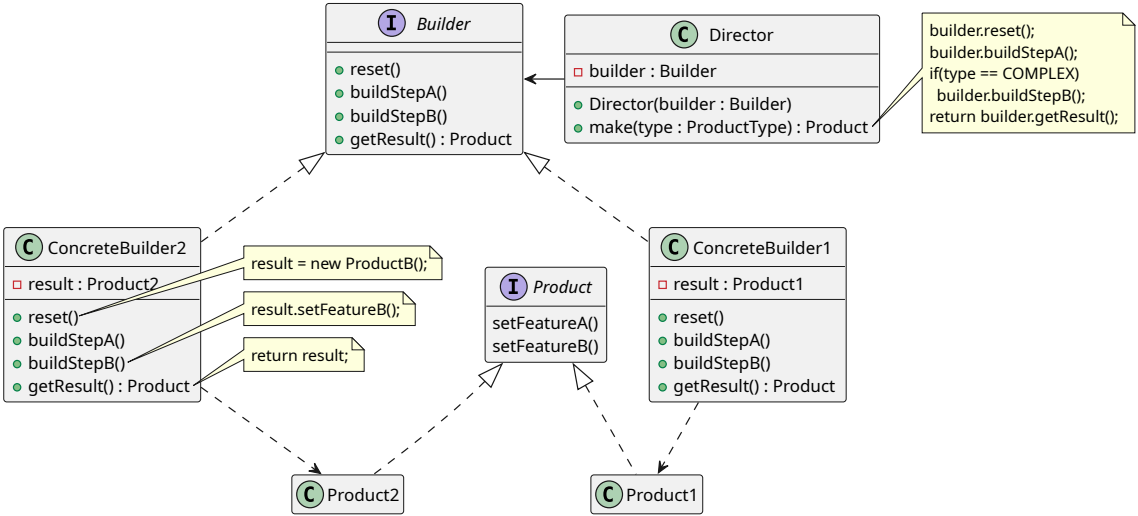
On peut construire l'objet après avoir construit le `Builder` puis fixer certains attributs :

```
User user = new UserBuilder("machin", "truc")
              .setAge(25)
              .setAddress("Rue de machin truc")
              .build();
```

Diagramme de la solution



Patron de conception *builder*



Patron de conception *command*

Intention

Permettre de construire des objets complexes étape par étape.

Avantages :

- Réutilisation du même code de construction lorsque vous construisez différentes représentations des produits
- Découpler le code complexe de la construction et de l'utilisation de l'objet (SRP)

Désavantages :

- Nécessité de créer beaucoup nouvelles classes, ce qui accroit la complexité générale du code.

Section 16

Résumé des patrons de conception

Patrons de création

Fournissent des mécanismes de création d'objets qui augmentent la flexibilité et la réutilisation du code.

- *factory* : classe mère déléguant aux classes filles la construction d'objets
- *abstract factory* : interface pour créer des familles d'objets sans préciser les classes concrètes
- *builder* : construire des objets complexes étapes par étapes
- *prototype* : créer des copies d'objets sans connaître leur classe
- *singleton* : garantir que l'instance d'une classe n'existe qu'en un seul exemplaire, tout en fournissant un point d'accès global à cette instance.

Patrons structurels

Guide l'assemblage d'objets et de classes en de plus grandes structures tout en gardant celles-ci flexibles et efficaces.

- *composite* : agencer les objets en une structure arborescente.
- *adapter* : faire collaborer des objets ayant des interfaces normalement incompatibles.
- *decorator* : affecter des nouveaux comportements en les plaçant dans des emballeurs.
- *facade* : procure une interface pour un accès simplifié à un ensemble de classes.
- *bridge* : permet de découper une classe monolithique en deux hiérarchies : abstraction et plateforme.
- *flyweight* : stocker de nombreux objets en mutualisant leurs données.
- *proxy* : fournit un substitut à un objet permettant de contrôler les opérations sur l'objet original.

Patrons comportementaux

S'occupent des algorithmes et de la répartition des responsabilités entre les objets.

- *strategy* : définir une famille d'algorithme et les rendre leurs objets interchangeables
- *template method* : mettre le squelette d'un algorithme dans une classe mère en permettant aux classes filles de redéfinir l'implémentation de certaines étapes.
- *state* : faire dépendre le comportement d'un objet de son état
- *chain of responsibility* : faire circuler une demande dans une chaîne de *handler*
- *command* : donner une classe à des actions
- *visitor* : Séparer les algorithmes des objets sur lesquels ils opèrent.
- *observer* : permettre la souscription à des événements concernant un objet observé
- *mediator* : forcer la collaboration entre objets via un seul objet
- *memento* : sauvegarder l'état d'un objet
- *iterator* : parcourir des éléments d'une collection

Composite

- *Quand?* Il permet de modéliser une structure arborescente (notion de nœuds et feuilles)
- *Pourquoi?* C'est une structure qui permet d'ajouter de nouveaux types de nœuds sans difficulté
- *Comment?* On utilise une interface commune à tous les nœuds. Cela permet à chaque nœud d'avoir juste une liste de nœud comme fils.
- *Exemples d'utilisation :* langages à balises, arbre d'analyse de langages formels, systèmes de fichiers, composants graphiques dans JavaFX, ...

Template Method

- *Quand?* Plusieurs classes similaires partagent du code
- *Pourquoi?* Pour éviter de la redondance de code. Cela permet d'ajouter des classes similaires à moindre coût.
- *Comment?* On utilise une classe abstraite qui va contenir le code commun et des méthodes abstraites vont contenir le code spécifique.
- *Exemples d'utilisation :* `ListSum` et `ListProduct`, ...

Strategy

- *Quand?* Plusieurs classes similaires partagent du code
- *Pourquoi?* Pour éviter la redondance de code. Pour séparer les responsabilités dans deux classes différentes.
- *Comment?* On délègue à un attribut de la classe une part des méthodes de cette classe, ces méthodes forme une interface qu'implémentera l'attribut.
- *Exemples d'utilisation :* `ListSum` et `ListProduct` $\rightarrow$ `List`

Visitor

- *Quand?* Vous voulez ajouter un service à un ensemble de classes qui partagent une interface.
- *Pourquoi?* Pour ne pas (trop) modifier ces classes et cette interface.
- *Comment?* On rend ces classes visitable (`accept` à ajouter dans l'interface des éléments) et on implémente le nouveau service dans une classe `Visitor` qui a une méthode `visit` pour chaque classe implémentant l'interface
- *Exemples d'utilisation :* Calcul d'un salaire pour des personnes occupant différentes fonctions, ajout d'un service de serialisation, ajout d'un service de visualisation.

State

- *Quand?* Vous avez plusieurs états dans votre système en fonction des interactions précédente. Votre système est dynamique.
- *Pourquoi?* Cela vous permet de mieux partitionner les comportements en fonction de l'état et de rajouter de nouveaux états facilement.
- *Comment?* En déléguant à un attribut (`state`) l'exécution du code de votre système. Cet état implémente une interface et pourra être remplacé par un autre état au cours de l'exécution.
- *Exemples d'utilisation :* Gestion d'un système avec interface utilisateur, protocole réseau à état, ...

Factory

- *Quand?* Vous remplacerez potentiellement une classe par une autre qui rend le même service.
- *Pourquoi?* Pour éviter de devoir modifier tous les appels aux constructeurs lorsque vous ferez ce changement.
- *Comment?* En créant une méthode `create` pour chacun des objets que vous voulez créer dans une classe `Factory` et en remplaçant tous les appels à un constructeur par un appel à la méthode `create` correspondante.
- *Exemples d'utilisation :* `ButtonFactory`

Abstract Factory

- *Quand?* Vous voulez pouvoir passer une `Factory` à une autre facilement
- *Pourquoi?* Cela vous évite de devoir modifier des `Factory` déjà faîtes.
- *Comment?* En créant une interface `AbstractFactory` qui contient tous les `create` nécessaire et chaque `Factory` concrète implémentera cette interface
- *Exemples d'utilisation :* `GUIFactory`

Adapter

- *Quand?* Vous avez une classe qui fournit sensiblement les services demandés, mais qui n'implémente pas l'interface que vous voulez.
- *Pourquoi?* Pour éviter de modifier la classe existante et l'interface existante.
- *Comment?* En créant une nouvelle classe qui a en attribut la classe existante et qui implémente l'interface voulue et adapte ses méthodes en appelant les méthodes de la classe en attribut

Decorator

- *Quand?* Vous voulez customiser les services d'un ensemble de classes qui partagent une interface.
- *Pourquoi?* De manière à avoir des variations très modulaires de classes
- *Comment?* En utilisant une classe abstraite qui implémente cette interface et qui a en attribut un élément de cette classe.
- *Exemples d'utilisation :* verbose, debug

Observer

- *Quand?* L'appel de méthodes sur un objet va générer des modifications dans un certain nombre d'objets d'autres classes.
- *Pourquoi?* On veut déléguer à ces autres classes la façon de gérer ces modifications et garder une liberté sur les objets affectés.
- *Comment?* On utilise une classe abstraite qui va gérer une liste d'observer et leur transmettre les modifications de l'objet.
- *Exemples d'utilisation :* éléments d'interface graphique qui observent des données pour se mettre à jour si elles changent.

Chain of responsability

- *Quand?* Vous voulez faire un `switch` sur des valeurs ou bien que vous voulez faire des `if then else` en série
- *Pourquoi?* Pour permettre une gestion modulaire de ce `switch`.
- *Comment?* En utilisant une classe abstraite qui gère le passage d'un `handler` à un autre.
- *Exemples d'utilisation :* Message d'erreurs, gestion de cas compliqués.

Proxy

- *Quand?* Quand vous voulez contrôler les méthodes utilisables pour une classe et/ou sa création.
- *Pourquoi?* Vous ne voulez pas modifier la classe existante et utiliser le code existant autant que possible.
- *Comment?* Vous ajoutez une classe qui a pour attribut un objet de la classe initiale et chaque méthode de cette classe va potentiellement contrôler l'accès de la méthode de l'objet originel ou bien créer l'objet originel seulement lorsqu'il est utile.
- *Exemples d'utilisation :* éditeur de texte, calculatrice.

Command

- *Quand?* Vous souhaitez pouvoir revenir en arrière sur des opérations.
- *Pourquoi?* Pour ne pas modifier le code des opérations.
- *Comment?* En ajoutant une classe qui va retenir l'ensemble des opérations faîtes dans une liste et qui va permettre de défaire ces opérations.
- *Exemples d'utilisation :* éditeur de texte, calculatrice.

Builder

- *Quand?* Vous créez un objet complexe avec beaucoup d'attributs.
- *Pourquoi?* Pour décharger la classe de la responsabilité de création de cet objet qui n'est pas en tant que tel un service de cette classe.
- *Comment?* En créant une classe qui initialise les champs de l'objet (set) et qui une fois que les champs sont remplis crée l'objet (build).
- *Exemples d'utilisation :* Personne, objets complexes à construire