Initiation génie logiciel : introduction

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

10 septembre 2024

amU Faculté des sciences Aix Marseille Université

Section 1

Bienvenue dans le cours d'initiation Génie logiciel

Organisation du cours

Volume horaire

- 6 séances de 1h30 de cours (9h)
- 3 séances de 2h de TD (6h)
- 6 séances de 2h et 1 séance de 3h de TP (15h)

Évaluation

- un examen sur papier en janvier (50% de la note)
- un TP noté sur machine (25% de la note)
- un rendu de projet (25% de la note)

Max avec la note d'examen terminal

Objectifs du cours

- Utiliser la programmation objet et Java pour organiser un projet d'envergure.
- Être capable de travailler sur des diagrammes de conception (diagramme de classes UML).
- Utiliser des patrons de conception.
- Respecter les principes SOLID.
- Obtenir un code de qualité :
  - permettant des évolutions faciles et
  - étant facilement testable

Conception

Nous allons voir comment atteindre ces objectifs en utilisant la programmation orienté objet en :

- Séparant et découplant les parties des projets en paquets/classes ;
- Limitant et localisant les modifications lors des évolutions ;
- Faisant en sorte d'écrire du code facilement réutilisable.

Objectif

Écrire du code durable : modifiable et pas jetable

S'initier à toutes les étapes du développement de logiciels

1. **Analyser** les besoins
2. **Spécifier** les comportements du programme
3. **Choisir** et éventuellement **concevoir** les solutions techniques
4. **Implémenter** le programme (coder)
5. **Vérifier** que le programme a le comportement spécifié (tester)
6. **Déployer** le programme dans son environnement, **fournir** une documentation dans le cas de bibliothèque
7. **Maintenir** le programme (corriger les bugs, ajouter des fonctionnalités)

Apprendre à programmer proprement

Un programme propre :

- respecte les attentes des utilisateurs
- est fiable
- peut évoluer facilement/rapidement
- est compréhensible par tous

Points importants

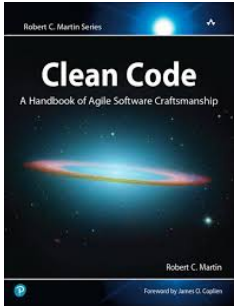
- Nommage des éléments du code : donner l'intention avec le nom
- Fonctions courtes
- Éviter commentaire inutile (réécrire le code plutôt que le commenter)
- Tests unitaires

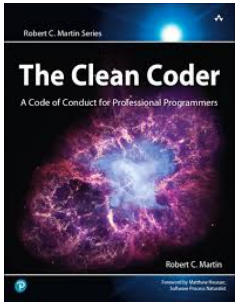
Comment programmer proprement ?

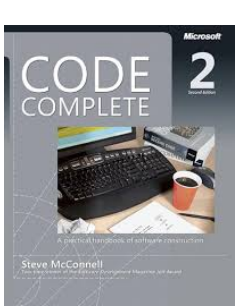
Pour programmer proprement dans un langage objet, il faut :

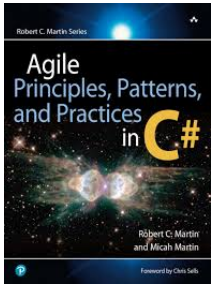
- écrire du code lisible (par un autre humain)
- relire et améliorer le code

Guides de bonnes pratiques :









Exemple de code mal écrit

```java
public class Rec {
  int l, L;

  public Rec(int l, int L) {
    this.l = l;
    this.L = L;
  }

  public static int compte(List<Rec> r){
    for(int i = 0, int s = 0; i < r.size(); i++)
      if(r.get(i).l == r.get(i).L)
        s++;
    return s;
  }
```

Après un petit peu de *refactoring*

```java
public class Rectangle {
  int width, height;
  public Rectangle(int width, int height) {
    this.width = width;
    this.height = height;
  }
  private boolean isSquare(){ return width == height; }
  static int countSquares(List<Rectangle> rectangles){
    int squareCount = 0;
    for(Rectangle rectangle : rectangles)
      if(rectangle.isSquare())
        squareCount++;
    return squareCount;
  }
```

Les cinq principes (pour créer du code) SOLID

- **Single Responsibility Principle (SRP)** : Une classe ne doit avoir qu'une seule responsabilité
- **Open/Closed Principle (OCP)** : Programme ouvert pour l'extension, fermé à la modification
- **Liskov Substitution Principle (LSP)** : Les sous-types doivent être substituables par leurs types de base
- **Interface Segregation Principle (ISP)** : Éviter les interfaces qui contiennent beaucoup de méthodes
- **Dependency Inversion Principle (DIP)** :
  - Les modules d'un programme doivent être indépendants
  - Les modules doivent dépendre d'abstractions

Patrons de conception (*design patterns*)

Les patrons de conception donnent des solutions (sous forme de schémas à personnaliser) pour résoudre un problème récurrent de conception logicielle.

Trois groupes principaux de patrons :

- de création qui fournissent des mécanismes de création d'objets ;
- structurels qui expliquent comment assembler des objets et des classes en de plus grandes structures ;
- comportementaux qui mettent en place une communication efficace et répartissent les responsabilités entre les objets.

Listes des patrons de conception (1/2)

- 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*)

Listes des patrons de conception (2/2)

- 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

Exemple d'utilisation d'un patron de conception

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);
    }
}
```

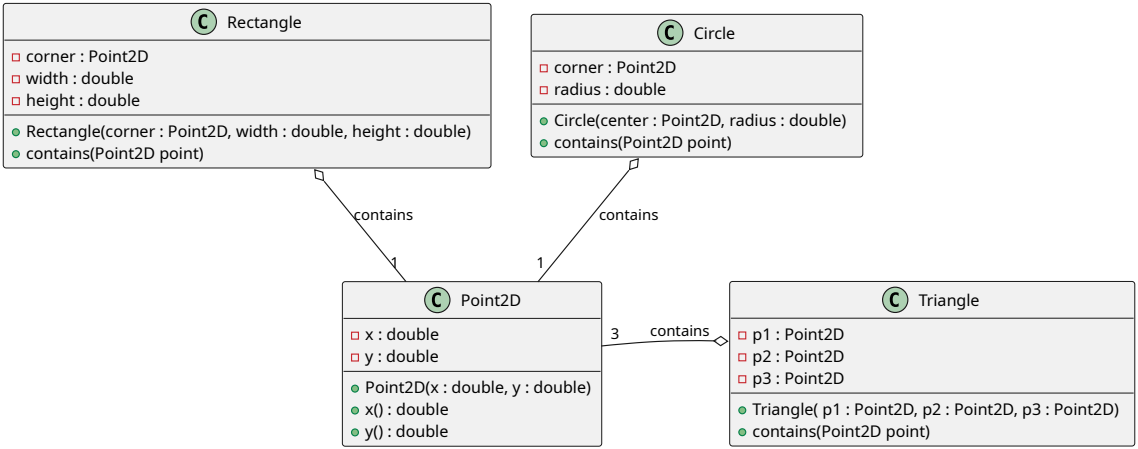
Délégation au sein des classes (2/4)

```
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{...}
```

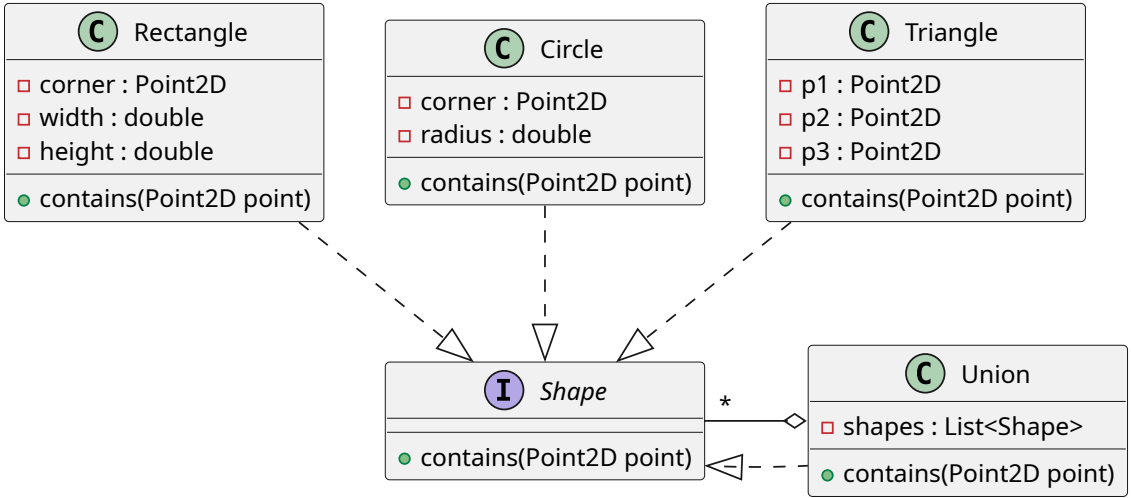
De cette manière on peut ajouter facilement une nouvelle forme : le principe SOLID OCP est respecté.

Union de forme

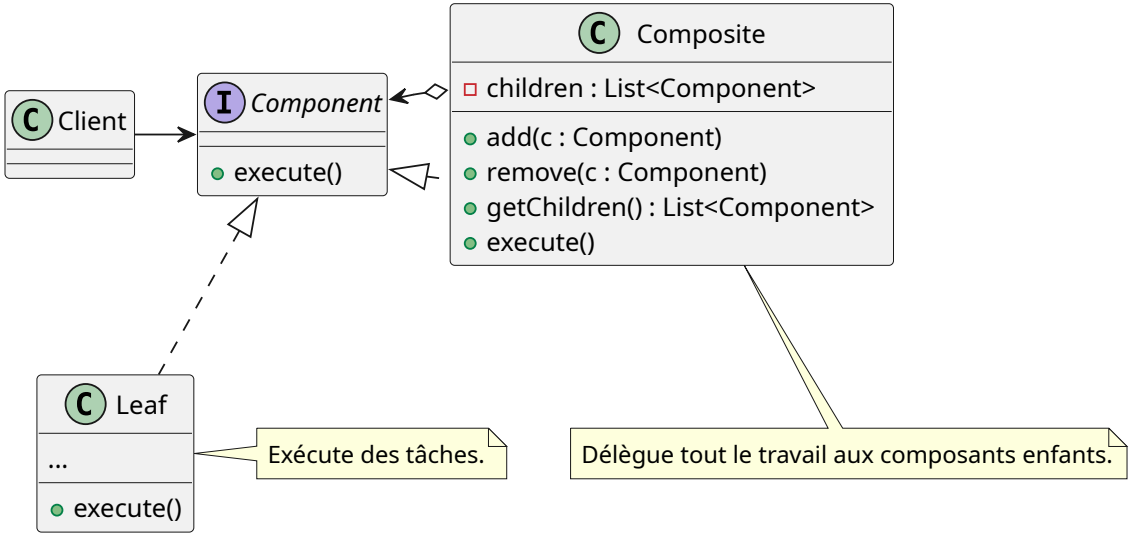
```
public class Union {
  List<Shape> shapes;
  public boolean contains(Point2D point){
    for(Shape shape : shapes)
      if(shape.contains(point))
        return true;
    return false;
  }
}
```

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

Diagramme de classes



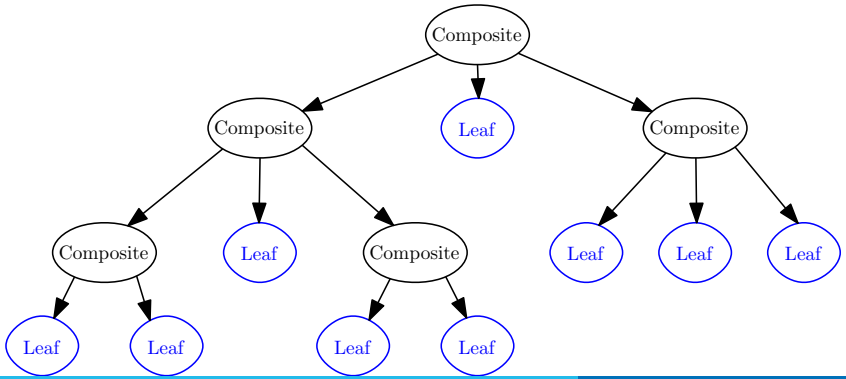
Patron de conception composite



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

Composition, agrégation et délégation

Exemple d'une classe `Point`

On considère la classe `Point` suivante :

```
public class Point {
    public final double x, y;
    public Point(double x, double y){this.x = x;
                                     this.y = y;}
    public double distanceTo(Point p){
        double dx = this.x - p.x;
        double dy = this.y - p.y;
        return Math.hypot(dx, dy);
    }
    public double getX(){ return x; }
    public double getY(){ return y; }
}
```

Réutilisation de Point : composition

Afin d'implémenter ses services, une instance peut créer des instances d'autres classes et conserver leurs références dans ses attributs.

```
public class Circle {
    private Point center;
    private double radius;
    public Circle(double x, double y, double radius){
        this.center = new Point(x, y);
        this.radius = radius;
    }
    public double getCenterX(){ return center.getX(); }
    public double getCenterY(){ return center.getY(); }
    public double getRadius(){ return radius; }
}
```

Définition composition

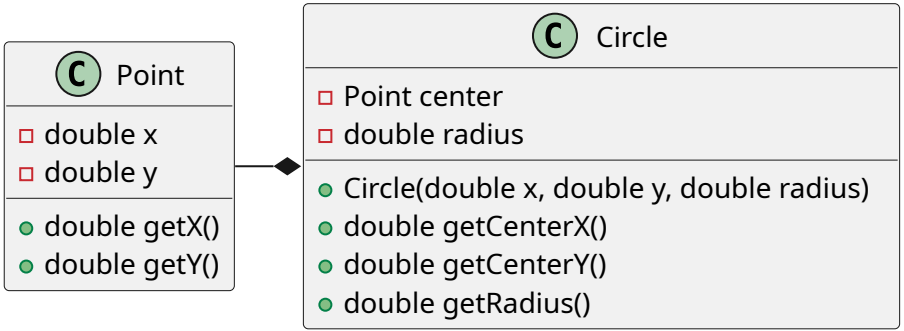
Composition d'une classe `Partie` dans une classe `Classe`

- Une (ou plus) instance(s) de `Partie` est stockée dans une instance de `Classe`
- Chaque instance de `Partie` appartient à une seule instance de `Classe`
- Chaque instance de `Partie` a sa durée de vie (construction) gérée par l'instance de `Classe` qui la contient
- Les instances de `Partie` n'ont pas connaissance de l'existence des instances de `Classe`

Exemple: Pièces d'une maison

Une maison contient une ou plusieurs pièces. La durée de vie d'une pièce est contrôlée par la maison, car elle n'existerait pas sans la maison.

Diagramme composition



Variante de `Circle` : agrégation

Une instance peut simplement posséder des références vers des instances d'une autre classe :

```
public class Circle {
    private Point center, point;
    public Circle(Point center, Point point){
        this.center = center;
        this.point = point;
    }
    public Point getCenter(){ return center; }
    public double getRadius(){
        double dx = center.x - point.x;
        double dy = center.y - point.y;
        return Math.hypot(dx, dy);
    }
```

Définition agrégation

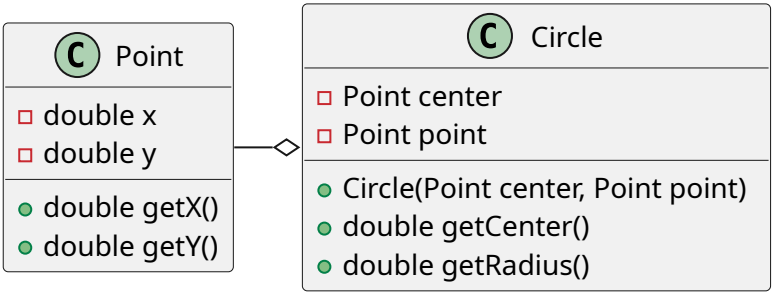
Agrégation d'une classe `Partie` dans une classe `Classe`

- une (ou plus) instance(s) de `Partie` est stockée dans une instance de `Classe`
- Chaque instance de `Partie` peut appartenir à plusieurs instances de `Classe`
- La durée de vie des instances de `Partie` n'est pas gérée par les instances de `Classe` qui la contient
- Les instances de `Partie` n'ont pas connaissance de l'existence des instances de `Classe`

Exemple: Bloc de construction d'une maison jouet

On a une maison jouet construite à partir de blocs. Vous pouvez la démonter, mais les blocs resteront.

Diagramme agrégation



Délégation

Délégation du calcul de la distance à l'instance `center` de la classe `Point` :

```
public class Circle {
    private Point center, point;

    public Circle(Point center, Point point){
        this.center = center;
        this.point = point;
    }
    public Point getCenter(){ return center; }
    public double getRadius(){
        return center.distanceTo(point);
    }
}
```

Définition délégation

Délégation d'une opération d'une classe `Classe` à une instance d'une classe `Delegate`

Une instance de `Classe` confie la résolution d'une de ses opérations (méthodes) à une instance de `Delegate`.
L'objet endossant cette responsabilité est nommé le *delegate* (ou délégué). Le rôle d'un *delegate* est de définir du comportement laissé à sa charge.

Exemple : confier le fait de faire un café à un stagiaire

Votre patron vous a demandé de lui faire un café, vous l'avez plutôt fait faire par un stagiaire.

Section 4

Classes abstraites et extension

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 → patron de conception **Stratégie**.
- L'extension et les classes abstraites → 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

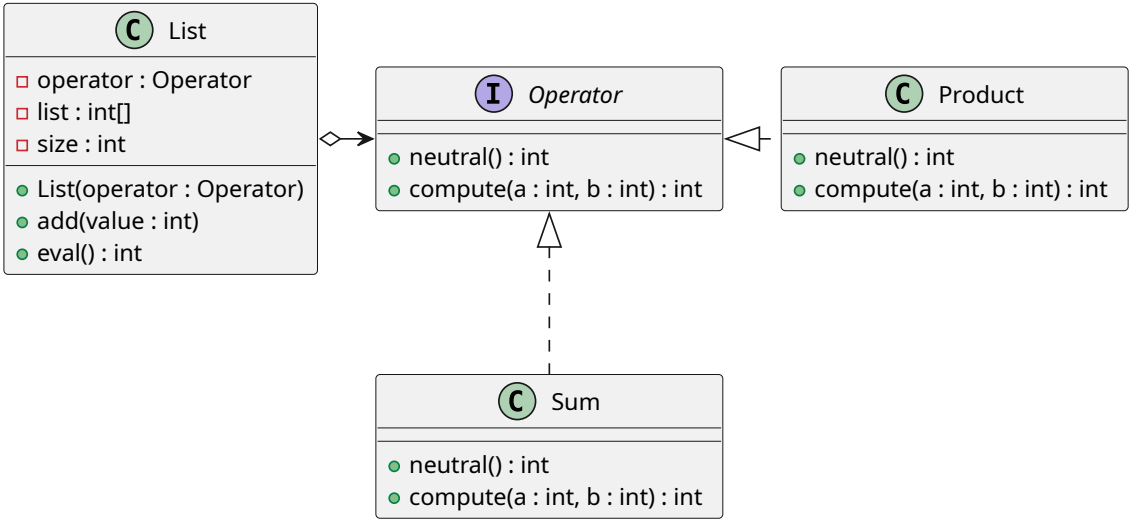
Utilisation des classes `ListSum` et `ListProduct` :

```
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());
```

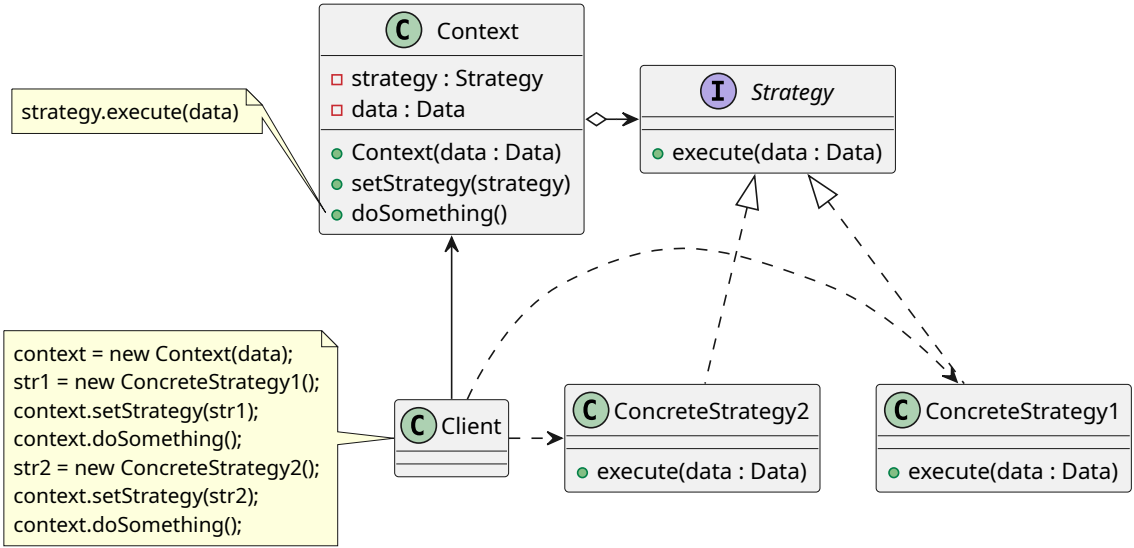
Utilisation après la *refactorisation* du code :

```
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

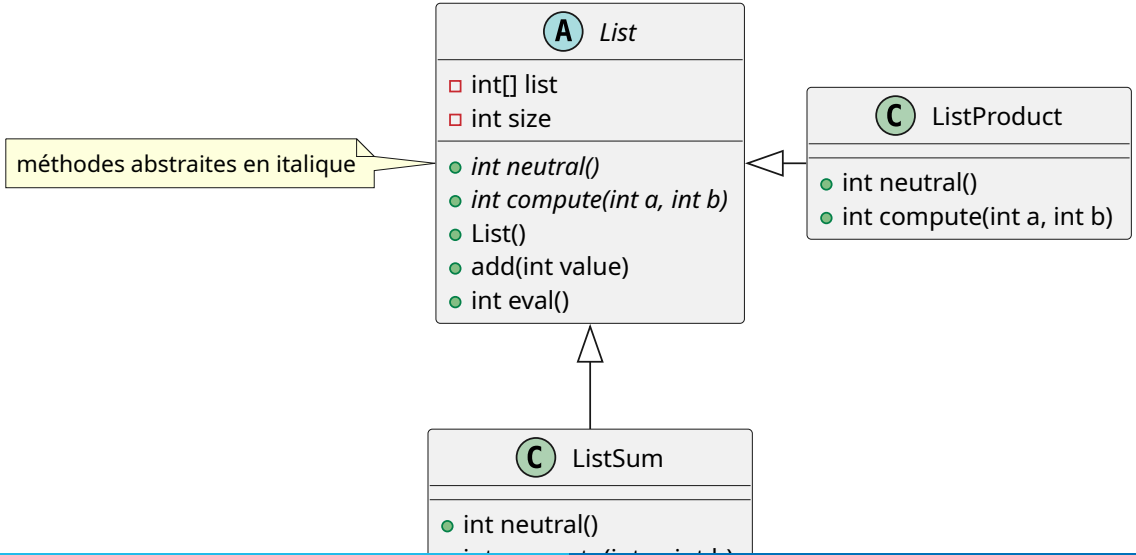
On peut procéder de manière similaire pour créer une classe `ListProduct`

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

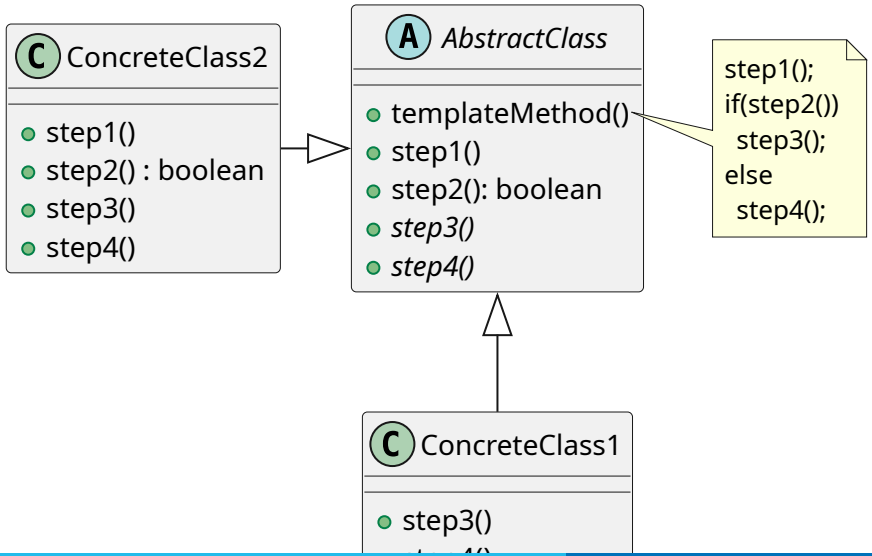
La classe `ListProduct` n'est plus abstraite, toutes ses méthodes sont définies :

```
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