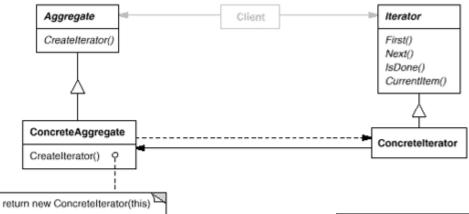
Résumé des patrons de conception vus en cours

Iterateur

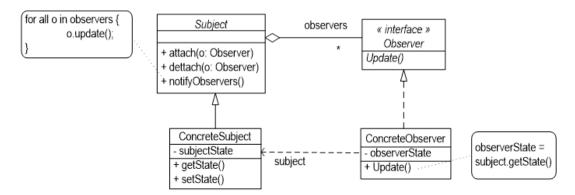


- Objet qui permet de parcourir et d'accéder aux éléments d'une collection d'objets sans exposer sa représentation interne (principe d'encapsulation)
- Il permet de séparer les responsabilités (principe de cohésion)

```
public class PancakeHouseMenu
   ArrayList <MenuItem> menuItems;
   public PancakeHouseMenu() {
       menuItems = new ArrayList<MenuItem>();
       addItem("K&B's Pancake Breakfast",
                "Pancakes with scrambled eggs, and toast",
       addItem("Regular Pancake Breakfast",
                "Pancakes with fried eggs, sausage",
               2.99);
       addItem("Blueberry Pancakes",
                "Pancakes made with fresh blueberries",
               3.49);
       addItem("Waffles",
                "Waffles, with your choice of blueberries or strawberries",
               3.59);
   public void addItem(String name, String description,
           boolean vegetarian, double price)
       MenuItem menuItem = new MenuItem(name, description, vegetarian, price);
       menuItems.add(menuItem);
   public Iterator createIterator() {
       return new PancakeHouseMenuIterator(menuItems);
   // other menu methods here
```

```
public interface Iterator {
                      boolean hasNext();
                      Object next();
public class PancakeHouseMenuIterator implements Iterator{
   ArrayList <MenuItem> items;
   int position = 0;
   public PancakeHouseMenuIterator(ArrayList <MenuItem> items) {
       this.items = items;
   public Object next() {
       MenuItem menuItem = (MenuItem)items.get(position);
       position = position + 1;
       return menuItem:
   public boolean hasNext() {
       if (position >= items.size()) {
           return false;
       } else {
```

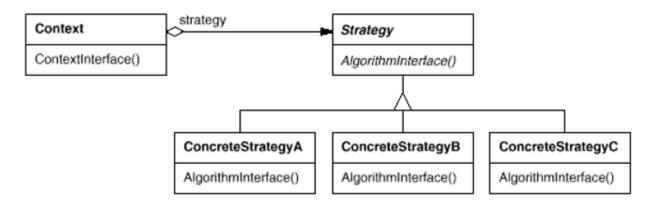
Observateur



- définit une dépendance un-àplusieurs entre objets de sorte que quand un change d'état, tous ses dépendants en sont informés et mis à jour
- minimise le couplage entre un objet et les objets qui dépendent de lui

```
public abstract class Subject {
                                                                                 public interface Observer {
      public abstract void attach(Observer o);
                                                                                     public void update(double s);
      public abstract void dettach(Observer o);
      public abstract void notifyObservers();
      public abstract double getSeuilMax();
public class Compte extends Subject{
  private double solde;
   private int numCpt;
   private final double SEUIL_MAX=10000;
   private final double SEUIL_MIN=-1000;
                                                                          public class Banque implements Observer {
   private ArrayList<Observer> liste;
                                                                              private Subject cpt;
                                                                              private double solde;
   public Compte(int num, double initSolde) {
       solde = initSolde;
                                                                              private double SEUIL MAX=10000;
       numCpt=num;
                                                                              public Banque(Subject c) {
       liste = new ArrayList<Observer>();
                                                                                 cpt = c;
                                                                                 c.attach(this);
   public void attach(Observer o) {
       liste.add(o);
                                                                              public void update(double solde) {
   public void dettach(Observer o) {
                                                                                 this.setSolde(solde):
      int i = liste.indexOf(o);
                                                                                 double seuil = cpt.getSeuilMax();
      if (i != 0)
                                                                                 if (solde >= seuil)
           liste.remove(i);
                                                                                     System.out.println("solde superieur au seuil max: " + solde);
   public double getSolde(){
                                                                                     System.out.println("solde inferieur au seuil min: " + solde);
       return solde;
   public double getSeuilMax(){
                                                                              public double getSolde() {
       return SEUIL_MAX;
                                                                                 return solde;
   public void seuilDepasse() {
       notifyObservers();
                                                                              public void setSolde(double solde) {
   public void notifyObservers() {
                                                                                  this.solde = solde;
       for (int i=0; i < liste.size(); i++) {
          Observer o = (Observer) liste.get(i);
           o.update(solde);
   public void deposer(double montant) {
       solde = solde + montant;
       System.out.println("depot de: " + montant);
       if (solde >= SEUIL MAX)
            seuilDepasse();
   public void retirer(double montant) {
       solde = solde - montant;
       System.out.println("retrait de: " + montant);
      if (solde <= SEUIL MIN)
            seuilDepasse();
```

Stratégie



- Permet de séparer un objet de ses comportements/algorithmes en encapsulant ces derniers dans des objets à part.
- Les comportements/algorithmes sont interchangeables
- Plus de flexibilité quant à l'ajout de nouveaux comportements

```
public class Graphe {
        private StrategieDessin strat;
                                                                                                                       public abstract class StrategieDessin {
        public Graphe() {
                                                                                                                           public abstract void dessiner(int [][] m);
            strat = new LinePlot();
        public void tracerGraphe(int [][] m) {
            strat.dessiner(m);
        public void setStrategie(StrategieDessin s) 
                                                                                                                                   public class LinePlot extends StrategieDessin{
                                                           public class BarPlot extends StrategieDessin {
            strat=s;
                                                                                                                                       public void dessiner(int [][] m) {
                                                               public void dessiner(int [][] m) {
                                                                                                                                           System.out.println(" graphe dessiner avec des lignes!");
                                                                       System.out.println(" graphe dessiner avec des bars!");
```

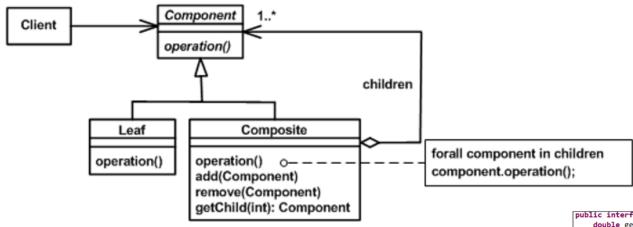
```
public class testerGraphe {

public static void main(String[] args) {
    int [][] mat = new int[][] { { 1, 2 }, { 3, 4, 5 } };

Graphe g= new Graphe();
    g.tracerGraphe(mat);
    g.setStrategie(new BarPlot());
    g.tracerGraphe(mat);
}

}
```

Composite



- permet de composer des objets composite à partir d'autres objets primitifs et/ou composites
- Permet de traiter tous les objets de la même manière (qu'il soit primitifs ou composite)

```
public class Bundle implements LineItem {
        private ArrayList<LineItem> items;
        public Bundle() {
            setItems(new ArrayList<LineItem>());
        public void add(LineItem item) {
            getItems().add(item);
        public double getPrice() {
            double sum = 0;
            for (Iterator <LineItem> i=getItems().iterator();i.hasNext();) {
                LineItem 1 = i.next();
                sum += l.getPrice();
            return sum;
        public String getDescription() {
            String description = "Le Bundle contient:";
            for (int i=0;i<getItems().size();i++) {</pre>
                LineItem 1 = getItems().get(i);
                if (i>0) description += ",";
                description += 1.toString();
            return description;
        public ArrayList<LineItem> getItems() {
            return items;
        public void setItems(ArrayList<LineItem> items) {
            this.items = items;
```

Médiateur

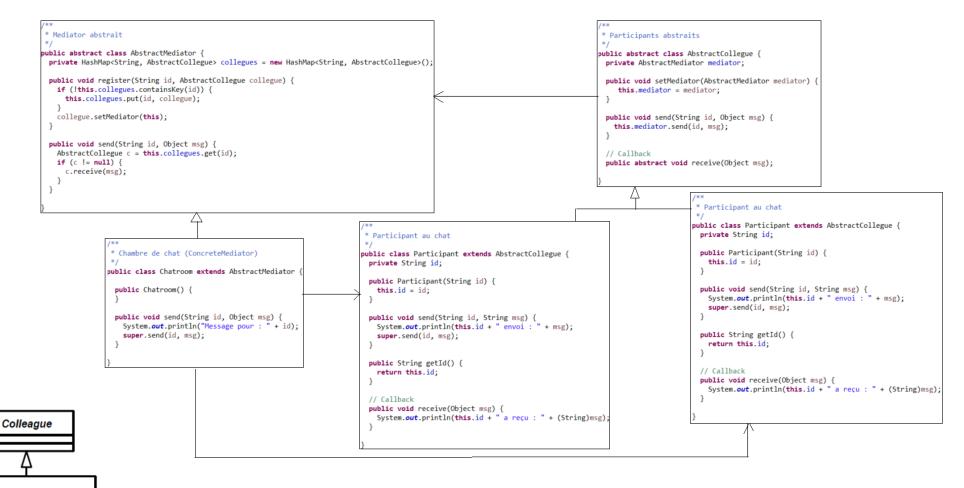
mediator

ConcreteColleague1

ConcreteColleague2

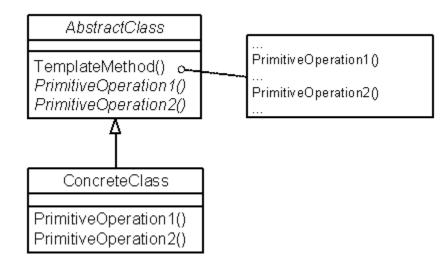
Mediator

ConcreteMediator



- permet l'interaction entre des objets sans que ces derniers ne maintiennent des références directes les uns vers les autres ancapsule les détails des communications netre les objets dans
- encapsule les détails des communications netre les objets dans une classe centrale
- Assure le découplage entre les objets

Méthode Template

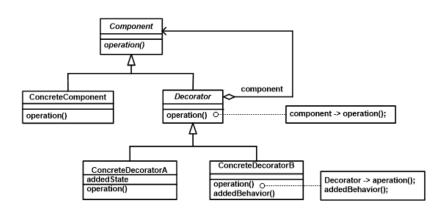


- Définit le squelette d'un algorithme dans une méthode modèle (méthode Template).
- Permet de supprimer le code dupliqué dans des sous-classes en l'implémentant dans la super classe et en permettant aux sous classes d'implémenter certaines étapes (sans changer la structure de l'algorithme).

```
Public abstract class Boisson {
     final void preparer() {
          bouillir eau();
         infuser();
         verser tasse();
         ajouter condiments();
    abstract void infuser();
    abstract void ajouter condiments();
     public void bouillir eau() {
          System.out.println("L'eau est
            entrain de bouillir!");
    public void verser tasse() {
          System.out.println("Je vous sers
            dans une tasse");
```

```
public class The extends Boisson {
     public void infuser() {
           System.out.println("Le thé est
             entrain de tromper dans l'eau!");
     public void ajouter condiments() {
           System.out.println("J'ajoute du
             citron!"):
public class Cafe extends Boisson {
   public void infuser() {
          System.out.println("Le café est entrain
            d'infuser!");
   public void ajouter condiments() {
          System.out.println("J'ajoute du lait et
            du sucre!");
```

Décorateur



- permet d'ajouter dynamiquement une fonctionnalité à un objet.
- c'est une alternative flexible et souple pour la dérivation pour étendre les fonctionnalités (on a ajouté un comportement par composition et non par héritage)
- c'est le décorateur qui s'occupe de décorer
 l'objet
- l'objet décoré est utilisé de la même façon que l'objet non décoré.

```
public abstract class Beverage {
                                                          String description;
                                                          public String getDescription() {
                                                              return description;
                                                          public abstract double cost();
public class Espresso extends Beverage{
                                              public class HouseBlend extends Beverage{
    public Espresso() {
                                                  public HouseBlend() {
        description ="Espresso Coffee";
                                                      description ="House Blend Coffee";
                                                                                                    public abstract class CondimentDecorator extends Beverage{
                                                                                                        public abstract String getDescription();
                                                  public double cost() {
    public double cost() {
        return 1.99;
                                                      return 0.89;
                                                                                                         public class Whip extends CondimentDecorator{
                                                 public class Moka extends CondimentDecorator{
                                                                                                             Beverage beverage;
                                                     Beverage beverage:
                                                     public Moka(Beverage beverage) {
                                                                                                             public Whip(Beverage beverage) {
                                                         this.beverage = beverage;
                                                                                                                 this.beverage = beverage;
                                                     public String getDescription() {
                                                                                                             public String getDescription() {
                                                                                                                 return beverage.description + ", Whip";
                                                         return beverage.description + ", Moka";
                                                     public double cost() {
                                                                                                             public double cost() {
                                                          return .20 + beverage.cost();
                                                                                                                 return .10 + beverage.cost();
```

```
public class StarBuzzCoffee {

public static void main(String[] args) {
    Beverage beverage = new Espresso();
    System.out.println(beverage.getDescription() + " $ " + beverage.cost());

    Beverage boisson = new HouseBlend();
    System.out.println(boisson.getDescription() + " $ " + boisson.cost());

    boisson = new Moka(boisson);
    System.out.println(boisson.getDescription() + " $ " + boisson.cost());

    boisson = new Moka(boisson);
    System.out.println(boisson.getDescription() + " $ " + boisson.cost());

    boisson = new Whip(boisson);
    System.out.println(boisson.getDescription() + " $ " + boisson.cost());
}
```