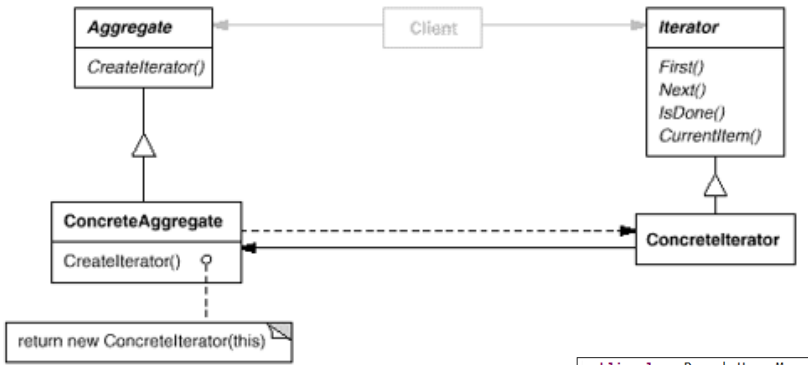
Résumé des patrons de conception vus en cours

Iterateur



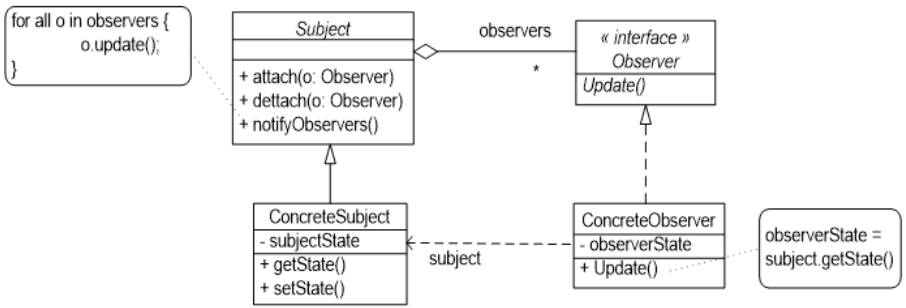
- Objet qui permet de parcourir et d'accéder aux éléments d'une collection d'objets sans exposer sa représentation interne (principe d'encapsulation)
- Il permet de séparer les responsabilités (principe de cohésion)

```java
public class PancakeHouseMenu {
    ArrayList <MenuItem> menuItems;
    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();
        addItem("K&B's Pancake Breakfast",
                "Pancakes with scrambled eggs, and toast",
                true,
                2.99);
        addItem("Regular Pancake Breakfast",
                "Pancakes with fried eggs, sausage",
                false,
                2.99);
        addItem("Blueberry Pancakes",
                "Pancakes made with fresh blueberries",
                true,
                3.49);
        addItem("Waffles",
                "Waffles, with your choice of blueberries or strawberries",
                true,
                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
}
```

```java
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

```java
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 {
            return true;
        }
    }
}
```

Observateur



- définit une dépendance un-à-plusieurs 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

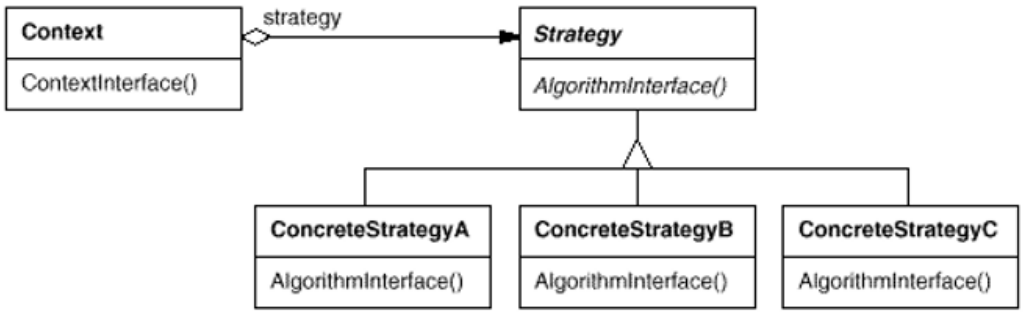
```java
public abstract class Subject {
    public abstract void attach(Observer o);
    public abstract void dettach(Observer o);
    public abstract void notifyObservers();
    public abstract double getSeuilMax();
}
```

```java
public interface Observer {
    public void update(double s);
}
```

```java
public class Compte extends Subject{
    private double solde;
    private int numCpt;
    private final double SEUIL_MAX=10000;
    private final double SEUIL_MIN=-1000;
    private ArrayList<Observer> liste;

    public Compte(int num, double initSolde) {
        solde = initSolde;
        numCpt=num;
        liste = new ArrayList<Observer>();
    }
    public void attach(Observer o) {
        liste.add(o);
    }
    public void dettach(Observer o) {
        int i = liste.indexOf(o);
        if (i != 0)
            liste.remove(i);
    }
    public double getSolde(){
        return solde;
    }
    public double getSeuilMax(){
        return SEUIL_MAX;
    }
    public void seuilDepasse() {
        notifyObservers();
    }
    public void notifyObservers() {
        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();
    }
}
```

```java
public class Banque implements Observer {
    private Subject cpt;
    private double solde;
    private double SEUIL_MAX=10000;

    public Banque(Subject c) {
        cpt = c;
        c.attach(this);
    }

    public void update(double solde) {
        this.setSolde(solde);
        double seuil = cpt.getSeuilMax();
        if (solde >= seuil)
            System.out.println("solde superieur au seuil max: " + solde);
        else
            System.out.println("solde inferieur au seuil min: " + solde);
    }

    public double getSolde() {
        return solde;
    }

    public void setSolde(double solde) {
        this.solde = solde;
    }
}
```

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

```java
public class Graphe {

        private StrategieDessin strat;

        public Graphe() {
            strat = new LinePlot();
        }

        public void tracerGraphe(int [][] m) {
            strat.dessiner(m);
        }

        public void setStrategie(StrategieDessin s) {
            strat=s;
        }

}
```

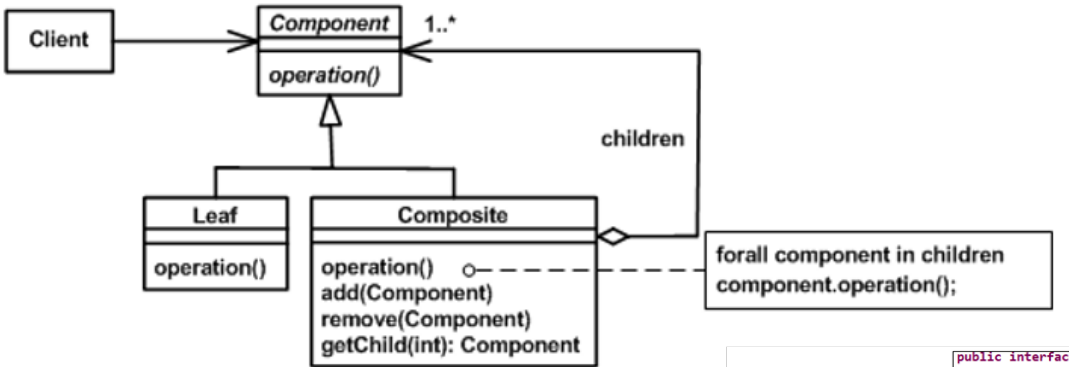
```java
public abstract class StrategieDessin {
    public abstract void dessiner(int [][] m);
}
```

```java
public class BarPlot extends StrategieDessin {
    public void dessiner(int [][] m) {
            System.out.println(" graphe dessiner avec des bars!");
        }
    }
```

```java
public class LinePlot extends StrategieDessin{
    public void dessiner(int [][] m) {
        System.out.println(" graphe dessiner avec des lignes!");
    }
}
```

```java
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)

```java
public interface LineItem {
    double getPrice();
    String getDescription();
}
```

```java
public class Product implements LineItem{
    private double price;
    private String description;

    public Product(String description, double price) {
        this.description =description;
        this.price = price;
    }

    @Override
    public double getPrice() {
        // TODO Auto-generated method stub
        return this.price;
    }

    public String getDescription() {
        return this.description;
    }
}
```

```java
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 l = i.next();
            sum += l.getPrice();
        }
        return sum;
    }
    public String getDescription() {
        String description = "Le Bundle contient:";
        for (int i=0;i<getItems().size();i++) {
            LineItem l = getItems().get(i);
            if (i>0) description += ",";
            description += l.toString();
        }
        return description;
    }
    public ArrayList<LineItem> getItems() {
        return items;
    }
    public void setItems(ArrayList<LineItem> items) {
        this.items = items;
    }
}
```

Médiateur

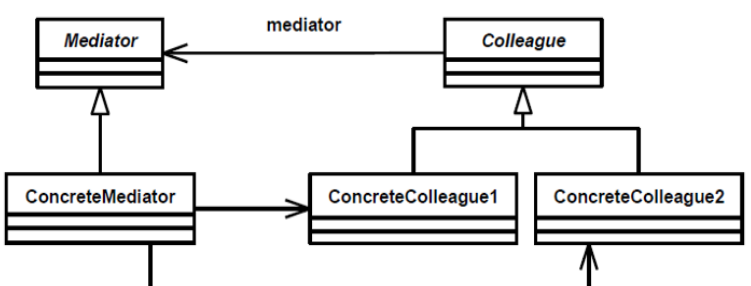
```java
/**
 * Mediator abstrait
 */
public abstract class AbstractMediator {
  private HashMap<String, AbstractCollegue> collegues = new HashMap<String, AbstractCollegue>();

  public void register(String id, AbstractCollegue collegue) {
    if (!this.collegues.containsKey(id)) {
      this.collegues.put(id, collegue);
    }
    collegue.setMediator(this);
  }

  public void send(String id, Object msg) {
    AbstractCollegue c = this.collegues.get(id);
    if (c != null) {
      c.receive(msg);
    }
  }
}
```

```java
/**
 * Participants abstraits
 */
public abstract class AbstractCollegue {
  private AbstractMediator mediator;

  public void setMediator(AbstractMediator mediator) {
    this.mediator = mediator;
  }

  public void send(String id, Object msg) {
    this.mediator.send(id, msg);
  }

  // Callback
  public abstract void receive(Object msg);
}
```

```java
/**
 * Chambre de chat (ConcreteMediator)
 */
public class Chatroom extends AbstractMediator {

  public Chatroom() {
  }

  public void send(String id, Object msg) {
    System.out.println("Message pour : " + id);
    super.send(id, msg);
  }
}
```

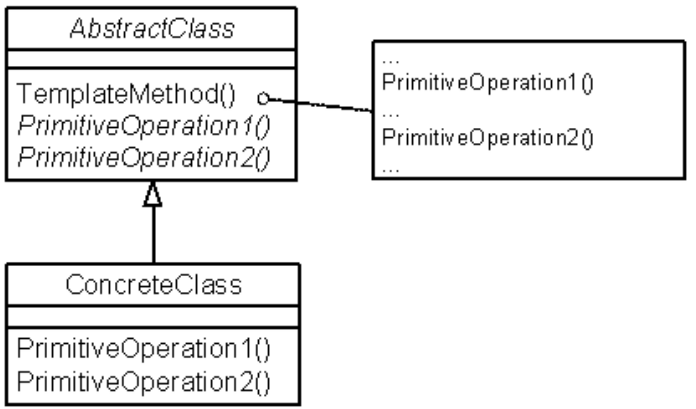
```java
/**
 * Participant au chat
 */
public class Participant extends AbstractCollegue {
  private String id;

  public Participant(String id) {
    this.id = id;
  }

  public void send(String id, String msg) {
    System.out.println(this.id + " envoi : " + msg);
    super.send(id, msg);
  }

  public String getId() {
    return this.id;
  }

  // Callback
  public void receive(Object msg) {
    System.out.println(this.id + " a reçu : " + (String)msg);
  }
}
```

```java
/**
 * Participant au chat
 */
public class Participant extends AbstractCollegue {
  private String id;

  public Participant(String id) {
    this.id = id;
  }

  public void send(String id, String msg) {
    System.out.println(this.id + " envoi : " + msg);
    super.send(id, msg);
  }

  public String getId() {
    return this.id;
  }

  // Callback
  public void receive(Object msg) {
    System.out.println(this.id + " a reçu : " + (String)msg);
  }
}
```



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

Méthode Template



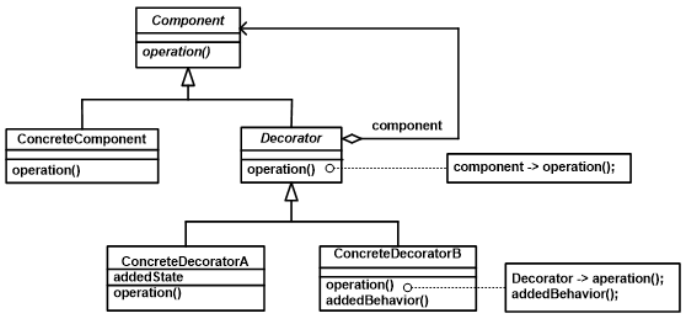
- Définit le squelette d'un algorithme dans une méthode modèle (méthode Template).
- Permet de supprimer le code dupliqué dans des sous-classes en l'implémentant dans la super classe et en permettant aux sous classes d'implémenter certaines étapes (sans changer la structure de l'algorithme).

```
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.

```java
public abstract class Beverage {
    String description;

    public String getDescription() {
        return description;
    }
    public abstract double cost();
}
```

```java
public class Espresso extends Beverage{
    public Espresso() {
        description ="Espresso Coffee";
    }
    public double cost() {
        return 1.99;
    }
}
```

```java
public class HouseBlend extends Beverage{
    public HouseBlend() {
        description ="House Blend Coffee";
    }
    public double cost() {
        return 0.89;
    }
}
```

```java
public abstract class CondimentDecorator extends Beverage{
    public abstract String getDescription();
}
```

```java
public class Moka extends CondimentDecorator{

    Beverage beverage;
    public Moka(Beverage beverage) {
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.description + ", Moka";
    }
    public double cost() {
        return .20 + beverage.cost();
    }

}
```

```java
public class Whip extends CondimentDecorator{

    Beverage beverage;
    public Whip(Beverage beverage) {
        this.beverage = beverage;
    }
    public String getDescription() {
        return beverage.description + ", Whip";
    }
    public double cost() {
        return .10 + beverage.cost();
    }

}
```

```java
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());

    }
}
```