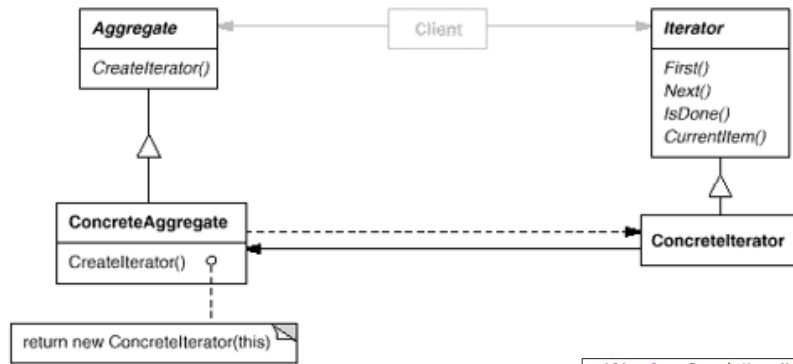


Résumé des patrons de conception vus en cours

Iterateur



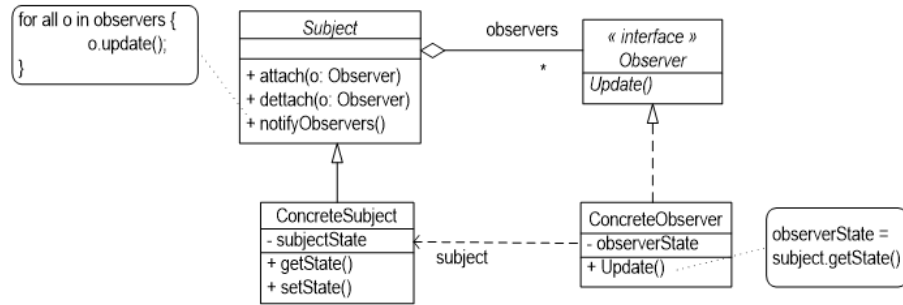
- Objet qui permet de parcourir et d'accéder aux éléments d'une collection d'objets sans exposer sa représentation interne (principe d'encapsulation)
- Il permet de séparer les responsabilités (principe de cohésion)

```
public class PancakeHouseMenu {
    ArrayList<MenuItem> menuItems;
    public PancakeHouseMenu() {
        menuItems = new ArrayList<MenuItem>();
        addItem("K&B's Pancake Breakfast",
            "Pancakes with scrambled eggs, and toast",
            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
}
```

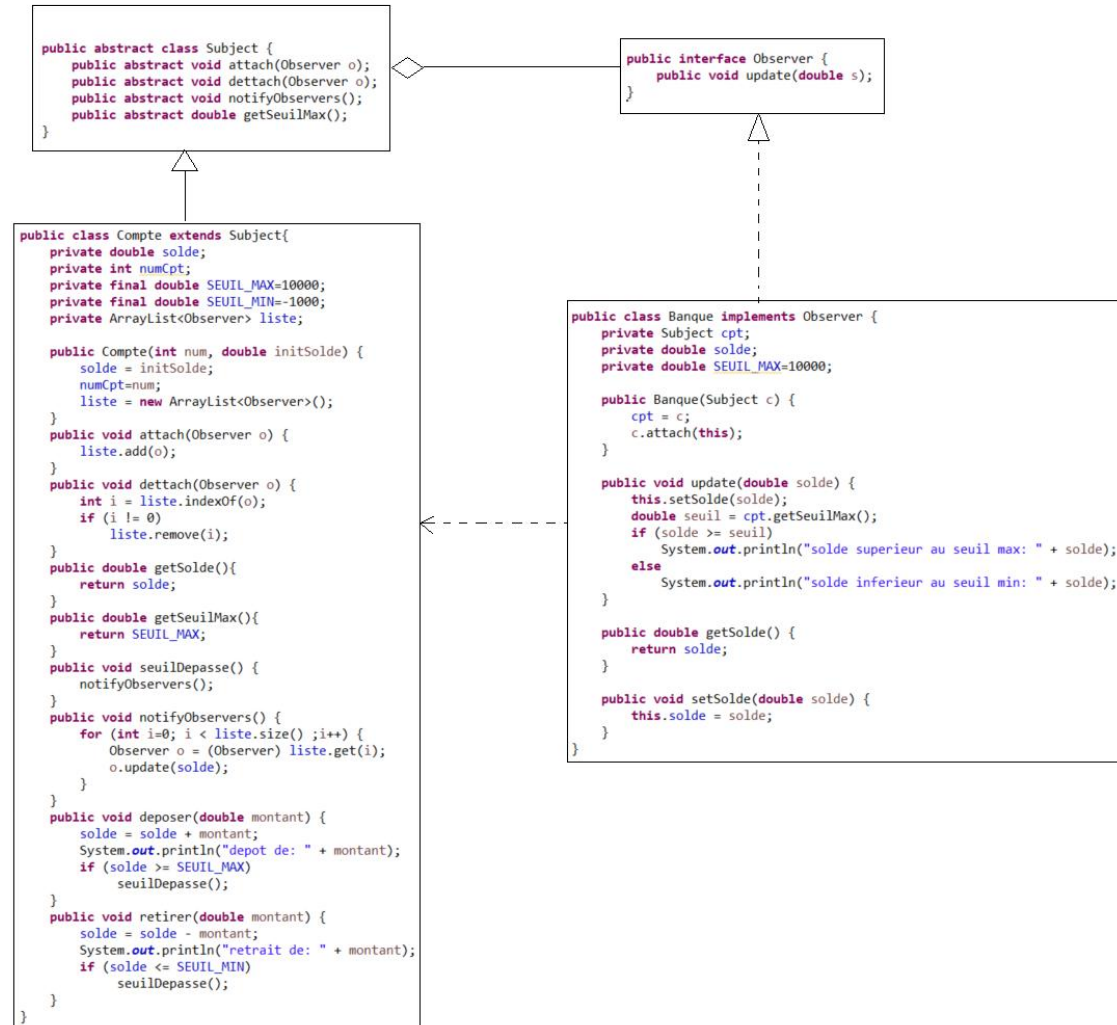
```
public interface Iterator {
    boolean hasNext();
    Object next();
}
```

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

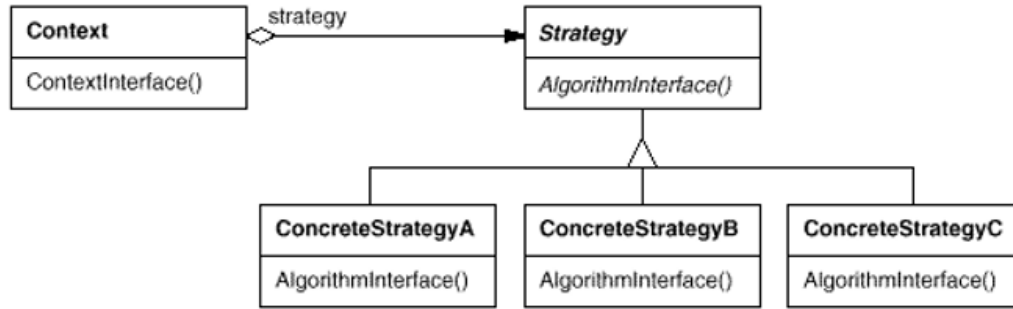
Observateur



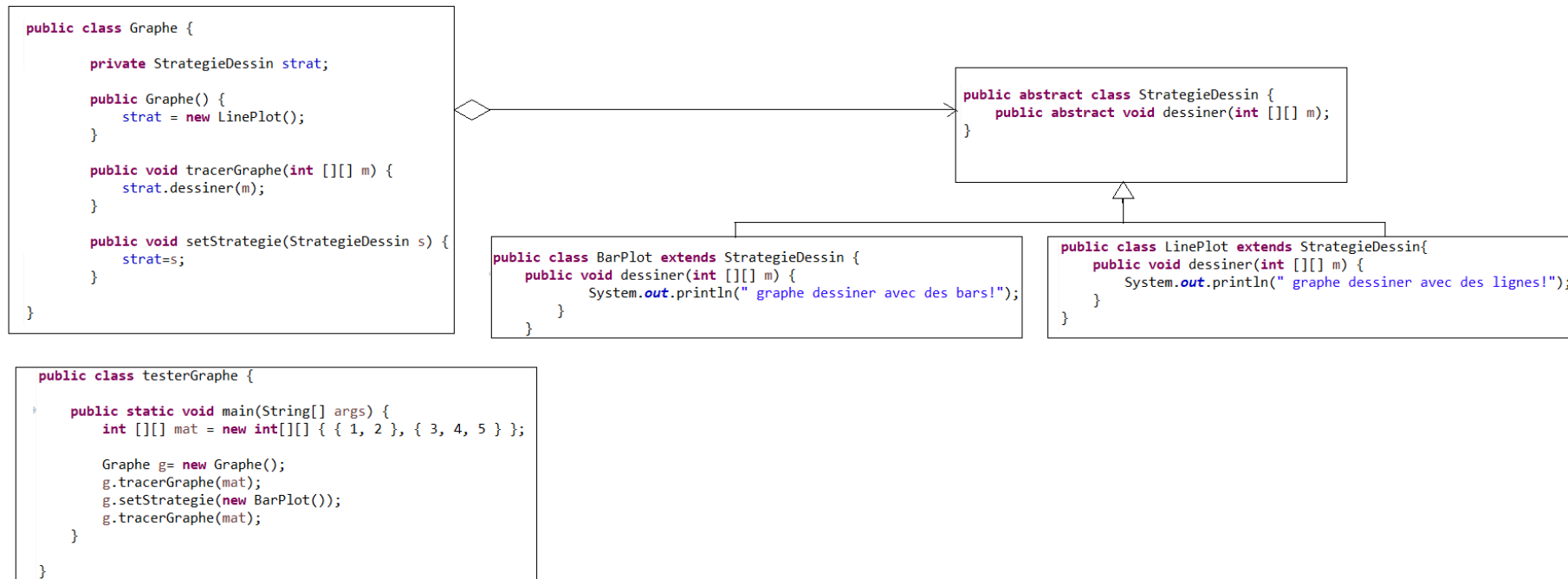
- définit une dépendance un-à-plusieurs entre objets de sorte que quand un change d'état, tous ses dépendants en sont informés et mis à jour
- minimise le couplage entre un objet et les objets qui dépendent de lui



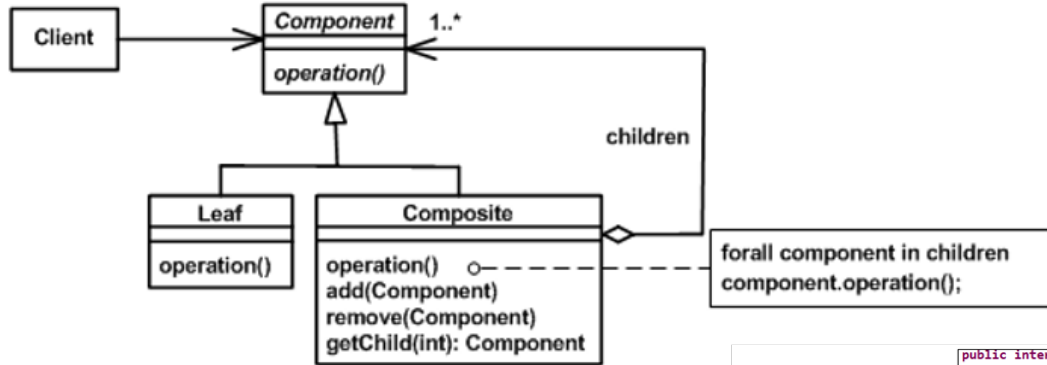
Stratégie



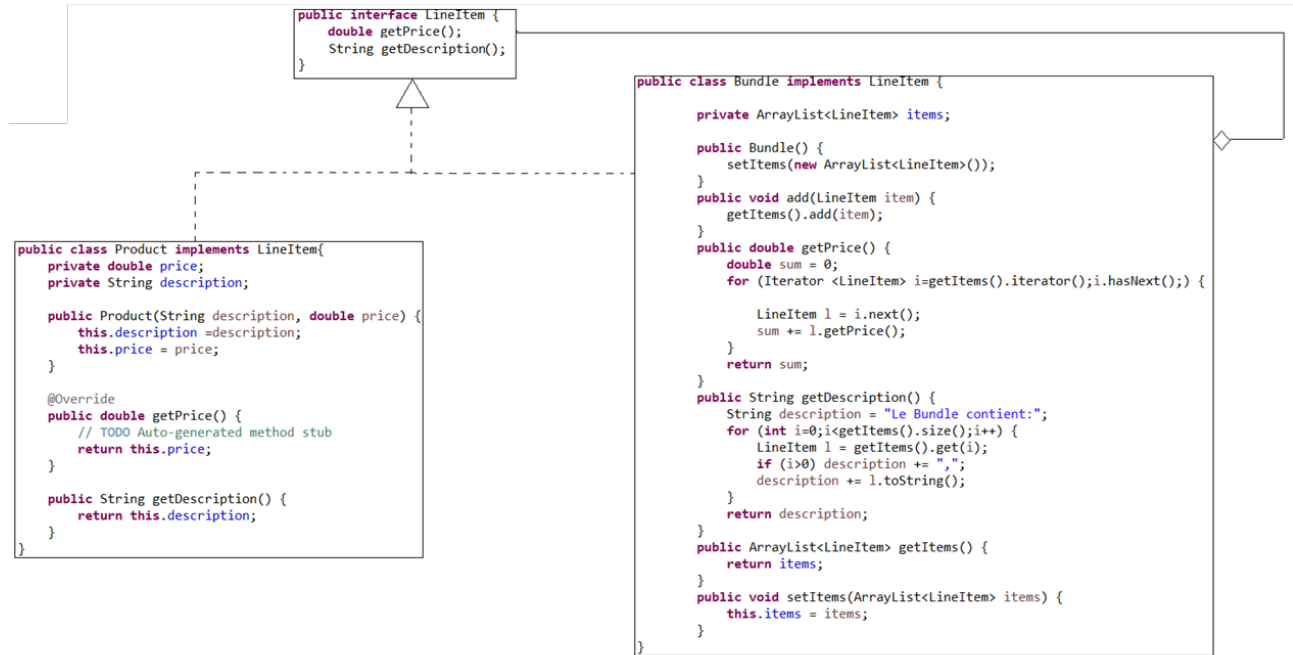
- Permet de séparer un objet de ses comportements/algorithmes en encapsulant ces derniers dans des objets à part.
- Les comportements/algorithmes sont interchangeables
- Plus de flexibilité quant à l'ajout de nouveaux comportements



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)



Médiateur

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

    public void register(String id, AbstractColleague colleague) {
        if (!this.colleagues.containsKey(id)) {
            this.colleagues.put(id, colleague);
        }
        colleague.setMediator(this);
    }

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

```
/**
 * Participants abstraits
 */
public abstract class AbstractColleague {
    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);
}
```

```
/**
 * 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);
    }
}
```

```
/**
 * Participant au chat
 */
public class Participant extends AbstractColleague {
    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);
    }
}
```

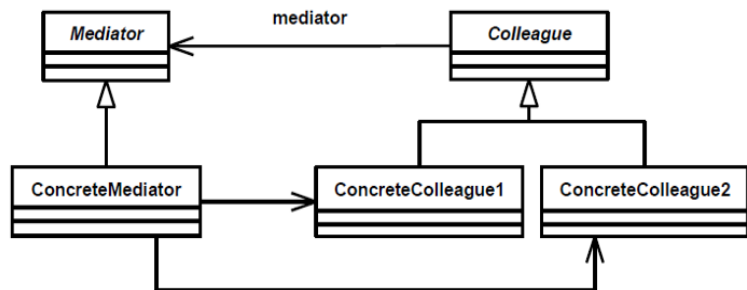
```
/**
 * Participant au chat
 */
public class Participant extends AbstractColleague {
    private String id;

    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);
    }

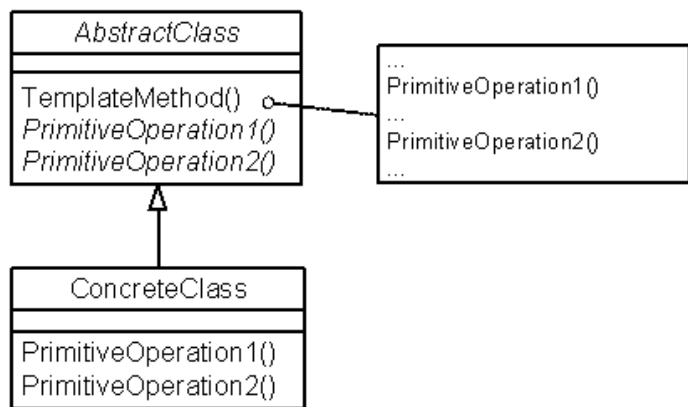
    public String getId() {
        return this.id;
    }

    // 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 entre 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();

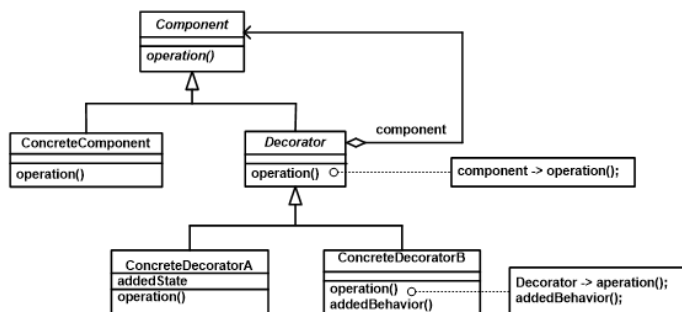
    public void bouillir_eau() {
        System.out.println("L'eau est
        entrain de bouillir!");
    }

    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.

