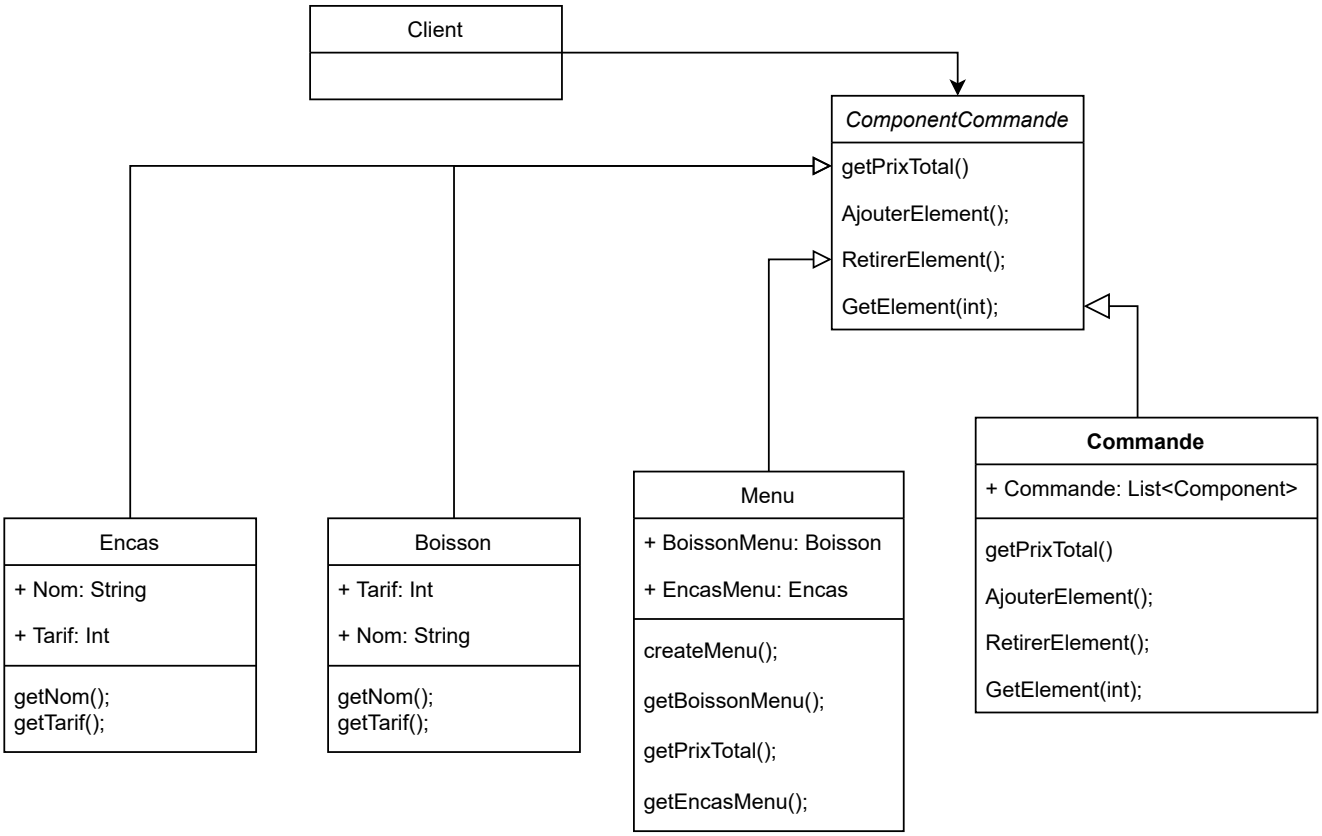
TP5 - Composite / Décoration

#DesignPattern

Composite

Le composite permet de composer des objets dans des stuctures arborescentes pour représenter des hiérarchies composants/composés. Permet à un Client de traiter les objets composés de la même manière qu'un objet individuel. Cela consiste à définir une classe abstraite qui spécifie le comportement commun à tous les composés et composants. Une relation de composition lie un composé à tout objet du type de la classe abstraite.




```
// La classe abstraite qui va modéliser l'objet le plus général que
l'on va composer.

public abstract class ComponentCommande {
    public abstract double getPrix();

    public ComponentCommande getElement(int number) {
        return this;
    }
```

```
    public void ajouterElement(ComponentCommande component) {
    }

    public void retirerElement(ComponentCommande component) {
    }
}

/*****************************************************************************



// La commande est notre compositve de ComponentCommande mais elle
l'étend également.

import java.util.ArrayList;
import java.util.List;

public class Commande extends ComponentCommande {
    List<ComponentCommande> myList;

    public Commande() {
        this.myList = new ArrayList<ComponentCommande>();
    }

    @Override
    public double getPrix() {
        int total = 0;
        for (ComponentCommande componentCommande : myList) {
            total += componentCommande.getPrix();
        }
        return (total);
    }

    @Override
    public String toString() {
        String output = "";
        for (int i = 0; i < myList.size(); i++) {
            output += myList.get(i).toString() + "€ \\n";
        }
        output += "Total : " + String.valueOf(this.getPrix()) + "€";
        return output;
```

```
    }

    @Override
    public void ajouterElement(ComponentCommande component) {
        myList.add(component);
    }

    @Override
    public void retirerElement(ComponentCommande component) {

        myList.remove(component);
    }

    @Override
    public ComponentCommande getElement(int number) {
        return (myList.get(number));
    }
}

/*********************************      Encas
*********************************/
// Dans la commande on peut rajouter différents éléments concrets qui
vont étendre
// ComponentCommande et ainsi devenir composable. e.g : Menu, Encas,
Boisson

public class Encas extends ComponentCommande {
    private int prix;
    private String nom;

    public Encas(String nom, int prix) {
        this.setNom(nom);
        this.prix = prix;
    }

    @Override
    public double getPrix() {
        return this.prix;
    }

    @Override
    public String toString() {
```

```
        return "- " + this.getNom() + " : " + this.getPrix();
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {

        this.nom = nom;
    }
}

/*********************************        Menu
*******************************/

public class Menu extends ComponentCommande {
    private Boisson boissonMenu;
    private Encas encasMenu;

    public Menu(Boisson maboisson, Encas monencas) {
        this.boissonMenu = maboisson;
        this.encasMenu = monencas;
    }

    public Boisson getBoissonMenu() {
        return boissonMenu;
    }

    public Encas getEncasMenu() {
        return encasMenu;
    }

    @Override
    public double getPrix() {

        return ((this.getEncasMenu().getPrix() +
this.getBoissonMenu().getPrix()) * 0.9);
    }

    @Override
```

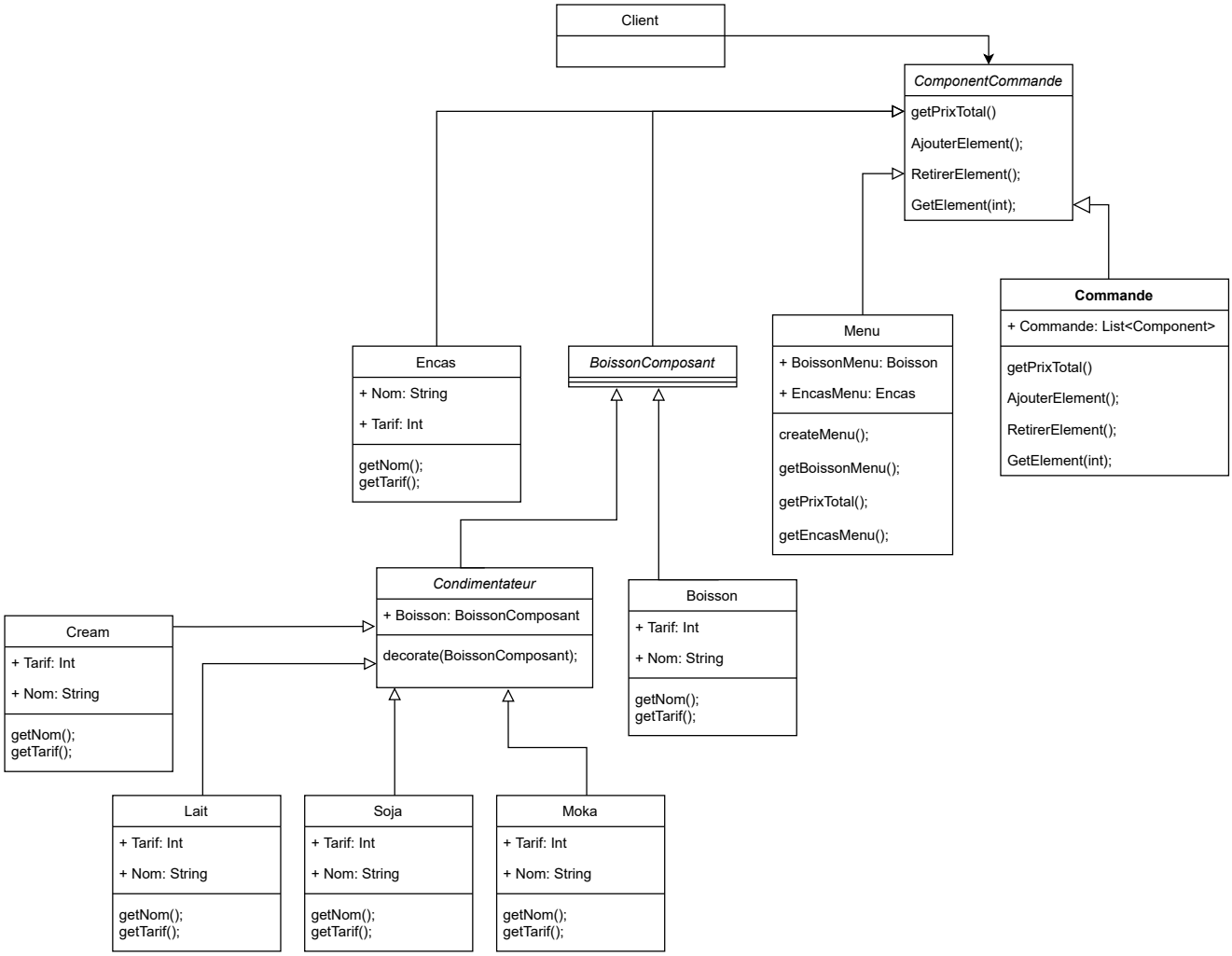
```
    public String toString() {

        return "- menu ( " + this.getBoissonMenu().getNom() + " + " +
this.getEncasMenu().getNom() + " ) : "
                + this.getPrix();
    }
}

/*********************************       Boisson

*********************************/

public class Boisson extends ComponentCommande {
    private int prix;
    private String nom;
    private boolean moka;
    private boolean soja;
    private boolean lait;
    private boolean cream;


    public Boisson(String nom, int prix) {
        this.setNom(nom);
        this.prix = prix;
    }



    @Override
    public double getPrix() {
        return this.prix;
    }

    @Override
    public String toString() {

        return "- " + this.getNom() + " : " + this.getPrix();
    }

    public String getNom() {
        return nom;
    }
```

```
    public void setNom(String nom) {
        this.nom = nom;
    }
}

/*********************************************************************

// Rajoutons un client qui va instancier une commande et ajouter des

ElementCommande
// à cette dernière

public class Client {

    public static void main(String[] args) {
        ComponentCommande maCommande = new Commande();
        maCommande.ajouterElement(new Encas("KitKat", 2));
        maCommande.ajouterElement(new Boisson("CocaCola",2));
        maCommande.ajouterElement(new Menu(new Boisson("Monster", 3),
new Encas("Bueno", 1)));
        System.out.println(maCommande.toString());
    }
}
```

- ComponentCommande déclare une classe abstraite des objets de la composition et implémente un comportement par défaut commun à toutes les classes.
- Boisson, Menu et Encas sont les objets feuilles (i.e. sans enfant) dans la composition et définit le comportement des objets primitifs.
- Commande définit le comportement des objets composés, stocke les objets enfants et implémente les opérations nécéssaires à leur gestion.
- Client quand à lui manipule les objets de la composition à travers la classe abstraite ComponentCommande.

Décorateur

Le décorateur ressemble au composite mais à une intention bien différente. Il permet de se focaliser sur l'ajout dynamique de responsabilités / propriétés à l'objet sans hériter. Généralement un décorateur étend un composite.




```
// on va créer une class abstraite BoissonComposant qui va étendre
ComponentCommande

public abstract class BoissonComposant extends ComposantCommande {
}

// puis un décorateur de boisson qui étend cette classe
public abstract class Condimentateur extends BoissonComposant {

    protected BoissonComposant decorated;

    public Condimentateur(BoissonComposant b) {
        this.decorated = b;
    }

}

/**************************************************************************

// on réutilise la structure de fichiuer de l'exercice précédent.
```

```java
public class Boisson extends BoissonComposant {
    private int prix;
    private String nom;
    private boolean moka;
    private boolean soja;
    private boolean lait;
    private boolean cream;


    public Boisson(String nom, int prix) {
        this.setNom(nom);
        this.prix = prix;
    }

    @Override
    public double getPrix() {
        return this.prix;
    }

    @Override
    public String toString() {

        return "- " + this.getNom() + " : " + this.getPrix();
    }

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }
}

/*************************************************************************


// on va désormais ajouter les élements de décoration
public class Milk extends Condimentateur{
```

```
    public Milk(BoissonComposant b) {
        super(b);
    }

    public double getPrice() {
        return this.decorated.getPrice() + 0.2;
    }

    public String toString() {
        return this.decorated + ", laité";
    }
}

// et cela pour plusieurs éléments, Sucre, Soja etc.

/*************************************************************************



// notre nouveau Client lui, va utiliser le condimentateur pour
construire des

// produits décorés

public class Client {

    public static void main(String[] args) {
        ComponentCommande maCommande = new Commande();
        maCommande.add(new Milk(new Coffee()));
        System.out.println(">>> Commande avec condiments <<<");
        maCommande.getPrix();
}
```

- BoissonComposant défini l'interface (ou la classe abstraite) des objets qui peuvent recevoir dynamiquement des responsabilités supplémentaires.
- Boisson définit un objet auquel des responsabilités supplémentaires peuvent être rattachées.
- Condimentateur gère une référence à l'objet de type BoissonComponent (l'objet décoré) et définit une interface conforme à ce dernier.
- Lait ajoute des responsabilité au composant décoré.