**Exercice UML**

Supposons que nous avons une application de vente de café. Il y a plusieurs types de café (Espresso, Cappuccino, etc.) et nous pouvons ajouter différents condiments (lait, sucre, etc.).

Le diagramme UML serait donc comme suit:



* Component: déclare l'interface pour les objets de l'opération qui peut avoir des comportements additionnels.
* ConcreteComponent: définit un objet à qui des responsabilités supplémentaires peuvent être ajoutées.
* Decorator: maintient une référence à un objet Component et définit une interface qui se conforme à l'interface de Component.
* ConcreteDecorator: ajoute des responsabilités à l'objet Component.

**Implémentation en C#**

csharp

// "Component"

public abstract class Coffee

{

public abstract double GetCost(); // Returns the cost of the coffee

public abstract string GetDescription(); // Returns the ingredients of the coffee

}

// "ConcreteComponent"

public class SimpleCoffee : Coffee

{

public override double GetCost()

{

return 1;

}

public override string GetDescription()

{

return "Simple coffee";

}

}

// "Decorator"

public abstract class CoffeeDecorator : Coffee

{

protected readonly Coffee \_decoratedCoffee;

public CoffeeDecorator(Coffee decoratedCoffee)

{

\_decoratedCoffee = decoratedCoffee;

}

}

// "ConcreteDecorator"

public class WithMilk : CoffeeDecorator

{

public WithMilk(Coffee decoratedCoffee)

: base(decoratedCoffee)

{

}

public override double GetCost() // Overriding methods defined in the abstract superclass

{

return \_decoratedCoffee.GetCost() + 0.5;

}

public override string GetDescription()

{

return \_decoratedCoffee.GetDescription() + ", with milk";

}

}

public class WithSugar : CoffeeDecorator

{

public WithSugar(Coffee decoratedCoffee)

: base(decoratedCoffee)

{

}

public override double GetCost()

{

return \_decoratedCoffee.GetCost() + 0.2;

}

public override string GetDescription()

{

return \_decoratedCoffee.GetDescription() + ", with sugar";

}

}

Vous pouvez tester le code comme suit:

csharp

Coffee simpleCoffee = new SimpleCoffee();

Console.WriteLine($"{simpleCoffee.GetDescription()} : ${simpleCoffee.GetCost()}");

Coffee milkCoffee = new WithMilk(simpleCoffee);

Console.WriteLine($"{milkCoffee.GetDescription()} : ${milkCoffee.GetCost()}");

Coffee sugarCoffee = new WithSugar(milkCoffee);

Console.WriteLine($"{sugarCoffee.GetDescription()} : ${sugarCoffee.GetCost()}");

Dans ce code, SimpleCoffee, WithMilk, et WithSugar sont des "ConcreteComponents". Le pattern Décorateur permet d'ajouter dynamiquement des comportements ou des responsabilités à un objet, par opposition à l'héritage qui est statique.