**Qu’est-ce qu’un Design Pattern :**

Un **Design Pattern** est la réponse à des problèmes récurrents rencontrés en informatique et surtout lors de la construction d’objets complexes.

Ces **Design Pattern** sont répertoriés en 3 types (création, structure et comportement).

Ils sont là pour répondre à des problèmes liés à la création d'objets complexes par exemple, utilisable dans des cas précis et réutilisable à l’infini.

Les Design Pattern ont étaient inventés par le GOF (Gang of Four), 4 architectes qui ont regroupés tous les **design Pattern** dans un diagramme dit de GOF.

**Le Design Pattern Builder :**

Ce Design Pattern permet de créer un objet complexe à l'aide d'objets simples et d'une approche étape par étape.

C'est un pattern de création qui propose la meilleure pour créer des objets complexes.

C’est la classe Builder qui permet de créer l’objet en question. Il faut savoir que cette classe est complétement indépendante des autres.

**Exemple de cas de restauration rapide**

Nous avons envisagé un cas de restauration rapide :

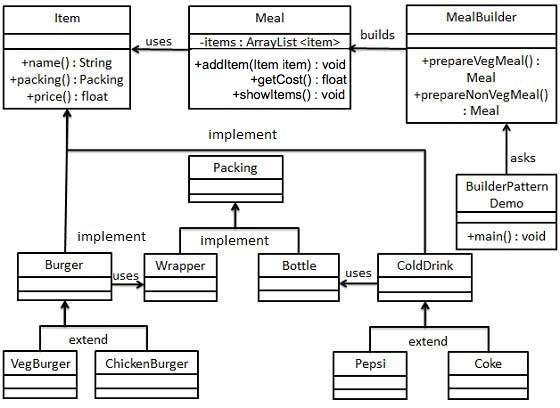
Un repas typique se compose d’un burger et d’une boisson fraîche. Le Burger peut être un Burger Végétarien ou un Burger no végétarien et disposera d'un emballage papier dit Wrapper. Une boisson fraîche accompagnera le reste du menu (Pepsi ou Coca) qui sera également emballé dans un gobelet.

Nous allons créer une interface Item représentant des produits alimentaires, tels que les Burgers et les boissons fraîches, ainsi que des classes concrètes implémentant l’interface Item car le Burger serait emballé dans un "Wrapper" et la boisson fraîche serait emballé dans une "Bottle".

Nous créons ensuite une classe Meal ayant une ArrayList d’Items et un MealBuilder pour construire différents types d’objets Meal en combinant Item. BuilderPatternDemo, notre classe Main de démonstration utilisera MealBuilder pour créer un repas.

Implémentation :

* Présentation du diagramme de classes de notre exemple :



**Étape n°1 :**

Créer une interface Article représentant un aliment.

public interface Item {

public String name();

public Packing packing();

public float price();

}

Créer une interface Emballage représentant un emballage.

public interface Packing {

public String pack();

}

**Étape n°2 :**

Créez des classes concrètes implémentant l'interface Packing.

Créez une classe Wrapper (l'emballage des burgers)

public class Wrapper implements Packing {

@Override

public String pack() {

return "Wrapper";

}

}

Créez une classe Bottle (l'emballage des boissons fraîches)

public class Bottle implements Packing {

@Override

public String pack() {

return "Bottle";

}

}

**Étape n°3 :**

Créez des classes abstraites implémentant l'interface d'élément fournissant les fonctionnalités par défaut.

public abstract class Burger implements Item {

@Override

public Packing packing() {

return new Wrapper();

}

@Override

public abstract float price();

}

ColdDrink.java

public abstract class ColdDrink implements Item {

@Override

public Packing packing() {

return new Bottle();

}

@Override

public abstract float price();

}

**Étape n°4 :**

Créez des classes héritant des classes ColdDrink (qui contient Pepsi et Coke) et Burger qui contient (VegBurger et ChickenBurger).

Appelez les méthodes abstraites des classes précédemment créent puis les compléter dans les 4 classes ci-dessous en y ajoutant le nom du produit à retourner ainsi que son prix unitaire pour calculer le prix total du menu.

public class VegBurger extends Burger {

@Override

public float price() {

return 25.0f;

}

@Override

public String name() {

return "Veg Burger";

}

}

public class ChickenBurger extends Burger {

@Override

public float price() {

return 50.5f;

}

@Override

public String name() {

return "Chicken Burger";

}

}

public class Coke extends ColdDrink {

@Override

public float price() {

return 30.0f;

}

@Override

public String name() {

return "Coke";

}

}

public class Pepsi extends ColdDrink {

@Override

public float price() {

return 35.0f;

}

@Override

public String name() {

return "Pepsi";

}

}

**Étape n°5 :**

Créez une classe de repas "meal" ayant les objets d’Item définis ci-dessus.

Créez une ArrayList ou tous nos Items se trouveront dans cette collection.

Ajout de 3 méthodes :

- addItem pour ajouter tous e que l'on veut dans notre menu (pepsi, ChickenBurger,...).

- getCost pour calculer le prix total du menu en parcourant

tous les items ajoutés et déterminer le prix à payer par le client.

- Affichage des éléments de la commande (l'item, le packaging, le prix de chaque article)

import java.util.ArrayList;

import java.util.List;

public class Meal {

private List<Item> items = new ArrayList<Item>();

public void addItem(Item item){

items.add(item);

}

public float getCost(){

float cost = 0.0f;

for (Item item : items) {

cost += item.price();

}

return cost;

}

public void showItems(){

for (Item item : items) {

System.out.print("Item : " + item.name());

System.out.print(", Packing : " + item.packing().pack());

System.out.println(", Price : " + item.price());

}

}

}

**Étape n°6 :**

Créez une classe MealBuilder, la classe de générateur chargée de créer les objets Meal.

La classe qui va nous permettre de créer des repas comme on le souhaite et ou on peut y ajouter ce que l'on veut.

public class MealBuilder {

public Meal prepareVegMeal (){

Meal meal = new Meal();

meal.addItem(new VegBurger());

meal.addItem(new Coke());

return meal;

}

public Meal prepareNonVegMeal (){

Meal meal = new Meal();

meal.addItem(new ChickenBurger());

meal.addItem(new Pepsi());

return meal;

} }

**Étape n°7 :**

Création de la classe Main afin de tester le Design Pattern Builder et déterminer si cela fonctionne.

On lui demande d'afficher tous les items compris dans le repas Végétarien créaient dans la classe MealBuilder.

Mais aussi d'afficher le cout total de ce repas.

La même chose est demandée pour le repas non Végétarien

public class BuilderPatternDemo {

public static void main(String[] args) {

MealBuilder mealBuilder = new MealBuilder();

Meal vegMeal = mealBuilder.prepareVegMeal();

System.out.println("Veg Meal");

vegMeal.showItems();

System.out.println("Total Cost: " + vegMeal.getCost());

Meal nonVegMeal = mealBuilder.prepareNonVegMeal();

System.out.println("\n\nNon-Veg Meal");

nonVegMeal.showItems();

System.out.println("Total Cost: " + nonVegMeal.getCost());

}

}

**Étape n°8 :**

Si tout fonctionne comme prévu vous devriez trouver les résultats suivants qui s'afficheront dans votre console. Sinon reprenez l'étape n°4 et/ou n°5.

Veg Meal

Item : Veg Burger, Packing : Wrapper, Price : 25.0

Item : Coke, Packing : Bottle, Price : 30.0

Total Cost: 55.0

Non-Veg Meal

Item : Chicken Burger, Packing : Wrapper, Price : 50.5

Item : Pepsi, Packing : Bottle, Price : 35.0

Total Cost: 85.5