Design Patterns

Builder Pattern

ebru@hacettepe.edu.tr

ebruakcapinarsezer@gmail.com

http://yunus.hacettepe.edu.tr/~ebru/

@ebru176

Aralık 2017

Type & intent

- One of the Creational Pattern

- Intent:
  - Separates the construction of a complex object from its representation so that the same construction process can create different representations.

Applicability

- The Builder pattern assembles a number of objects in various ways depending on the data.

- Use the Builder pattern when

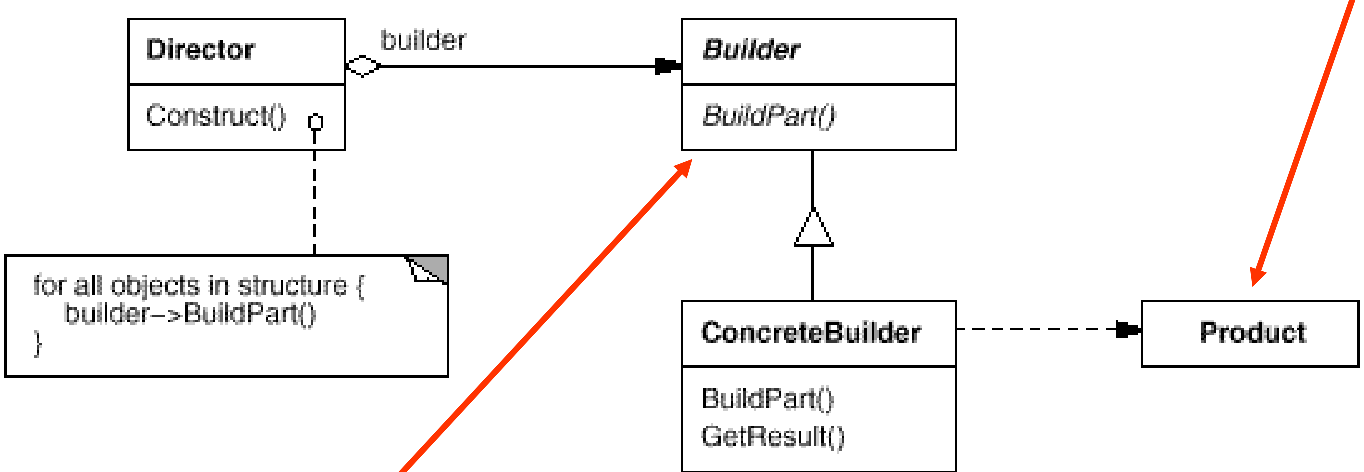
  - the algorithm for creating a complex object should be independent on the parts that make up the object and how they're assembled.

  - the construction process must allow different representations for the object that's constructed.

Structure (UML Model)

construct an object using the Builder interface

the complex object under construction.



specifies an abstract interface for creating parts of a Product object

constructs and assembles parts of the product by implementing the Builder interface

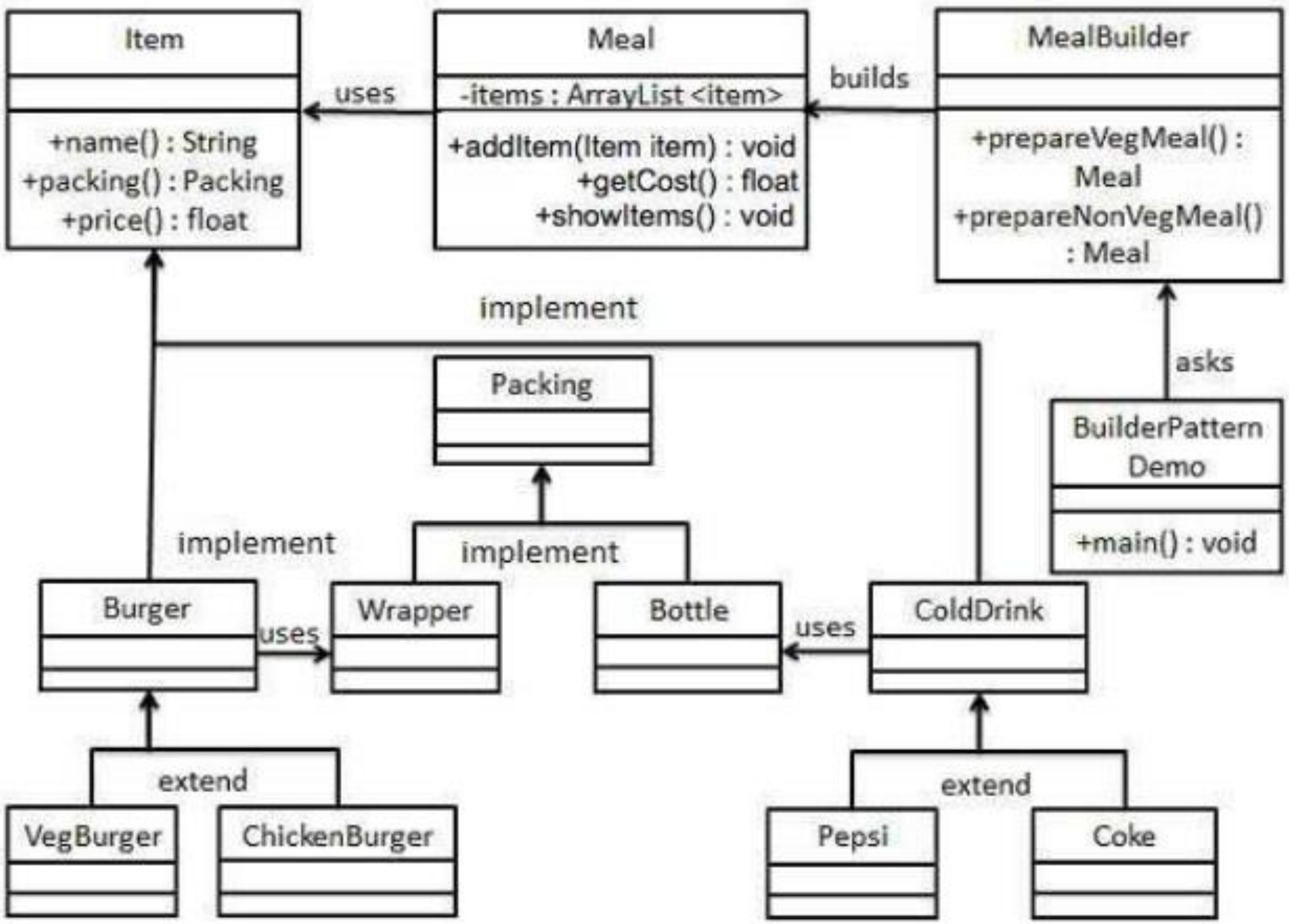
Participants

- **Builder:** specifies an abstract interface for creating parts of a Product object.
- **ConcreteBuilder:**
  - constructs and assembles parts of the product by implementing the Builder interface.
  - Defines and keeps track of the representation it creates.
  - Provides an interface for retrieving the product.
- **Director:** constructs an object using the Builder interface.
- **Product:** represents the complex object under construction.

Consequences

- Abstracts the construction implementation details of a class type. It lets you vary the internal representation of the product that it builds.
- Encapsulates the way in which objects are constructed improving the modularity of a system.
- Finer control over the creation process, by letting a builder class have multiple methods that are called in a sequence to create an object.
- Each specific Builder is independent of any others.

Example



Item
+name() : String
+packing() : Packing
+price() : float
uses
Meal
-items : ArrayList <item>
+addItem(Item item) : void
+getCost() : float
+showItems() : void
builds
MealBuilder
+prepareVegMeal() : Meal
+prepareNonVegMeal() : Meal
implement
asks
Packing
BuilderPattern Demo
+main() : void
implement
implement
Burger
uses
Wrapper
Bottle
uses
ColdDrink
extend
VegBurger
ChickenBurger
extend
Pepsi
Coke

Item.java

```
public interface Item {
   public String name();
   public Packing packing();
   public float price();
}
```

Packing.java

```
public interface Packing {
   public String pack();
}
```

Wrapper.java

```
public class Wrapper implements Packing {

   @Override
   public String pack() {
      return "Wrapper";
   }
}
```

Bottle.java

```
public class Bottle implements Packing {

   @Override
   public String pack() {
      return "Bottle";
   }
}
```

```
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();
}
```

VegBurger.java

```java
public class VegBurger extends Burger {

   @Override
   public float price() {
      return 25.0f;
   }

   @Override
   public String name() {
      return "Veg Burger";
   }
}
```

ChickenBurger.java

```java
public class ChickenBurger extends Burger {

   @Override
   public float price() {
      return 50.5f;
   }

   @Override
   public String name() {
      return "Chicken Burger";
   }
}
```

Coke.java

```java
public class Coke extends ColdDrink {

   @Override
   public float price() {
      return 30.0f;
   }

   @Override
   public String name() {
      return "Coke";
   }
}
```

Pepsi.java

```java
public class Pepsi extends ColdDrink {

   @Override
   public float price() {
      return 35.0f;
   }

   @Override
   public String name() {
      return "Pepsi";
   }
}
```

Meal.java

```
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());
      }
   }
}
```

MealBuilder.java

```
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;
   }
}
```

Step 7

BuiderPatternDemo uses MealBuider to demonstrate builder p

BuilderPatternDemo.java

```
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());
   }
}
```