VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Information Technology & Engineering

M-Tech Software Engineering

Fall Semester (2022-2023)

Design Patterns

SLOT - C1

PIZZA - ORDERING SYSTEM USING BUILDER DESIGN PATTERN

REVIEW - III

SUBMMITED BY:-

19MIS0102 - S.DEEPAN
19MIS0068 - M.SHANMUGA MAYAN
19MIS0086 - R.ARUN KUMAR

Under the guidance of

Prof. SENTHIL KUMARAN.U

1. Abstract:

Design patterns are typical solutions to common problems in software design. Each pattern is like a blueprint that you can customize to solve a particular design problem in your code. Patterns are a tool kit of solutions to common problems in software design. Design patterns differ by their complexity, level of detail and scale of applicability. In addition, they can be categorized by their intent and divided into three groups namely creational, structural, and behavioral pattern

2. Introduction:

For the following problem we have chosen builder design pattern because it constructs thecomplex objects step by step manner using simple objects

The Builder design pattern is one of the creational design patterns that describe how to solverecurring design problems in object-oriented software.

The Builder design pattern solves problems like:

- How can a class (the same construction process) create different representations of acomplex object?
- How can a class that includes creating a complex object be simplified?

Creating and assembling the parts of a complex object directly within a class is inflexible. It commits the class to creating a particular representation of the complex object and makes it impossible to change the representation later independently from (without having to change) the class.

The Builder design pattern describes how to solve such problems:

- Encapsulate creating and assembling the parts of a complex object in a separate Builderobject.
- A class delegates object creation to a builder object instead of creating the objectsdirectly.

A class (the same construction process) can delegate to different Builder objects to createdifferent representations of a complex object.

3. Intent:

Separate the construction of a complex object from its representation so that the sameconstruction process can create different representations.

4. Motivation:

In our scenario we are building an interface for a pizza-hut in which it has different types of pizzas and soft drinks namely veg pizza and non-veg pizza in them it has categories like small cheese pizza, medium cheese pizza and large cheese pizza instead of cheese it has all three variants for onion toppings. It has also three variants for non-veg pizza.

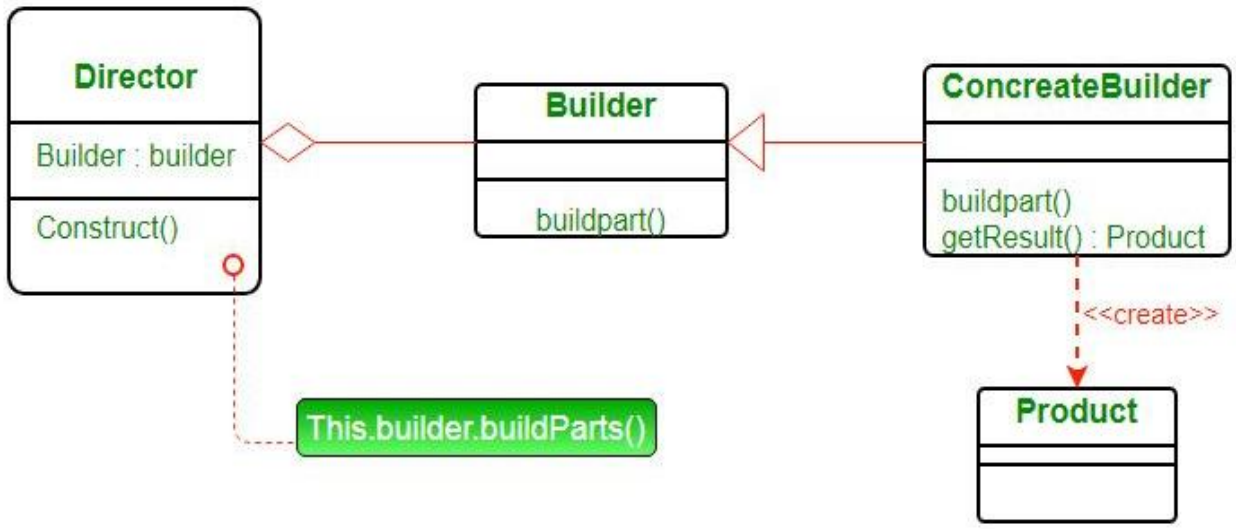
After pizza selection customer can opt soft drink if he/she wanted and 2 drinks are available i,e coke and Pepsi. In drinks also it offers three variants small, medium, and large.

Here we need to build an interface such that customer can order any type of pizza he/she wants and can opt for soft drinks of any type.

We can use BUILDER design pattern for the given problem because the construction of objects is similar and builder For the given problem, first we need to construct different classes for different types of pizzas available and soft drinks available, then we are constructing an order builder class which is responsible for building objects of the ordered items, at last we create builder demo class which uses order builder class to calculate the total cost of the order and shows the items we ordered

Builder design pattern offers step by step construction of them.

5. Structure:



6. Applicability:

Use the Builder pattern when

- The algorithm for creating a complex object should be independent of 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.

7. Participants:

- Builder (Orderbuilder):
  - specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder (BuilderDemo):
  - 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 (OrderedItem):
  - constructs an object using the Builder interface.

- Product (different variants of pizza, soft drinks):
  - represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

8. Collaborations:

- The client creates the Director object and configures it wit the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.

The client retrieves the product from the builder.

9. Consequences:

1. It lets you vary a product's internal representation.

The Builder object provides the director with an abstract interface for constructing the product. The interface lets the builder hide the representation and internal structure of the product. It also hides how the product gets assembled. Because the product is constructed through an abstract interface, all you have to do to change the product's internal representation is define a new kind of builder.

2. It isolates code for construction and representation.

The Builder pattern improves modularity by encapsulating the way a complex object is constructed and represented. Clients needn't know anything about the classes that define the product's internal structure; such classes don't appear in Builder's interface. Each ConcreteBuilder contains all the code to create and assemble a particular kind of product. The code is written once; then different Directors can reuse it to build Product variants from the same set of parts.

3. It gives you finer control over the construction process.

Unlike creational patterns that construct products in one shot, the Builder pattern constructs the product step by step under the director's control. Only when the product is finished does the director retrieve it from the builder. Hence the Builder interface reflects the process of

constructing the product more than other creational patterns. This gives you finer control over the construction process and consequently the internal structure of the resulting product.

12. Implementation:

- In the present scenario we have implemented the following code in java
- First, we have created an interface Item that represents the Pizza and Cold drink.
- Then created an abstract class Pizza and ColdDrink that will implement to the interface Item.
- Now, created concrete sub-classes SmallCheezePizza, MediumCheezePizza, LargeCheezePizza, ExtraLargeCheezePizza that will extend to the abstract class VegPizza. Similarly, we have created for NonVegPizza
- Now, created two abstract classes Pepsi and Coke that will extend abstract class ColdDrink. And also, concrete clasess for them namely small, medium and large that extends the respective classes
- Now we have created an OrderBuilder class that will be responsible to create the objects of OrderedItems class.
- At last, we have created BuilderDemo class which uses orderbuilder class to show the cost os the items we have ordered and show the items we have ordered

13. Implementation Issues:-

1. **Assembly and construction interface.** Builders construct their products in step-by-step fashion. Therefore the Builder class interface must be general enough to allow the construction of products for all kinds of concrete builders**.**

2. **Why no abstract class for products?** In the common case, the products produced by the concrete builders differ so greatly in their representation that there is little to gain from giving different products a common parent class. In the RTF example, the ASCIIText and the TextWidget objects are unlikely to have a common interface, nor do they need one. Because the client usually configures the director with the proper concrete builder, the client is in a position to know which concrete subclass of Builder is in use and can handle its products accordingly.

3. **Empty methods as default in Builder**. In C++, the build methods are intentionally not declared pure virtual member functions. They're defined as empty methods instead, letting clients override only the operations they're interested in.

14. Sample code:

Step 1: Create an interface Item that represents the Pizza and Cold drink.

Item.java

```
public  interface  Item

{

   public String name();

    public String size();

    public float price();

}// End of the interface Item.
```

Step 2: Create an abstract class Pizza that will implement to the interface Item.

Pizza.java

```
public abstract class Pizza implements

        Item{@Override

        public abstract float price();

}
```

Step 3: Create an abstract class Cold-Drink that will implement to the interface Item.

ColdDrink.java

```
public abstract class ColdDrink implements

 Item{@Override

 public abstract float price();
```

Step 4: Create an abstract class Veg-Pizza that will extend to the abstract class Pizza.

VegPizza.java

```
public abstract class VegPizza extends

    Pizza{@Override
```

```
    public abstract float price();
    @Override
    public abstract  String name();
    @Override
    public abstract  String size();
}// End of the abstract class VegPizza.
```

Step 5: Create an abstract class NonVegPizza that will extend to the abstract class Pizza.

NonVegPizza.java

```
public abstract class NonVegPizza extends
    Pizza{@Override
    public abstract float price();
    @Override
    public abstract String name();
    @Override
    public abstract String size();
}// End of the abstract class NonVegPizza.
```

Step 6: Now, create concrete sub-classes SmallCheezePizza, MediumCheezePizza, LargeCheezePizza, ExtraLargeCheezePizza that will extend to the abstract class VegPizza.

SmallCheezePizza.java

```
public class SmallCheezePizza extends
    VegPizza{@Override
    public float price()
        {return 170.0f;
    }
```

```
    @Override
    public String name()
        { return "Cheeze
        Pizza";
    }
    @Override
    public String size()
       {return "Small
       size";
    }
}// End of the SmallCheezePizza class.
```

MediumCheezePizza.java

```
public class MediumCheezePizza extends
    VegPizza{@Override
    public float price()
       {return  220.f;
    }
    @Override
    public String name()
        { return "Cheeze
        Pizza";
    }
    @Override
    public String size()
```

```
    { return "Medium
     Size";
 }
}// End of the MediumCheezePizza class.
```

LargeCheezePizza.java

```
public class LargeCheezePizza extends
    VegPizza{@Override
    public float price()
        {return 260.0f;
    }
    @Override
    public String name()
        { return "Cheeze
        Pizza";
    }
    @Override
    public String size() {
        return "Large Size";
    }
}// End of the LargeCheezePizza class.
```

ExtraLargeCheezePizza.java

```
public class ExtraLargeCheezePizza extends
    VegPizza{@Override
    public float price()
```

```
    {return 300.f;
    }
    @Override
    public String name()
        { return  "Cheeze
        Pizza";
    }
    @Override
    public String size() {
        return "Extra-Large Size";
    }
}// End of the ExtraLargeCheezePizza class.
```

Step 7: Now, similarly create concrete sub-classes SmallOnionPizza, MediumOnionPizza, LargeOnionPizza, ExtraLargeOnionPizza that will extend to the abstract class VegPizza.

SmallOnionPizza.java

```
public class SmallOnionPizza extends VegPizza {
    @Override
    public float price()
        {return 120.0f;
    }
    @Override
    public String name()
        { return  "Onion
```

```
    Pizza";
    }
    @Override
    public String size()
      { return  "Small
      Size";
    }
}// End of the SmallOnionPizza class.
```

MediumOnionPizza.java

```
public class MediumOnionPizza extends VegPizza
    {@Override
    public float price()
        {return 150.0f;
    }
   @Override
    public String name()
        { return  "Onion
        Pizza";
    }
   @Override
    public String size()
      { return  "Medium
      Size";
    }
}// End of the MediumOnionPizza class.
```

LargeOnionPizza.java

```
public class LargeOnionPizza extends
    VegPizza{@Override
    public float price()
        {return 180.0f;
    }
    @Override
    public String name()
         { return "Onion
         Pizza";
    }
    @Override
    public String size()
       { return  "Large
       size";
    }
}// End of the LargeOnionPizza class.
```

ExtraLargeOnionPizza.java

```
public class ExtraLargeOnionPizza extends VegPizza
    {@Override
    public float price()
        {return 200.0f;
    }
    @Override
```

```
    public String name()
        { return  "Onion
        Pizza";
    }
    @Override
    public String size() {
      return  "Extra-Large Size";
    }
}// End of the ExtraLargeOnionPizza class
```

Step 8: Now, similarly create concrete sub-classes SmallMasalaPizza, MediumMasalaPizza, LargeMasalaPizza, ExtraLargeMasalaPizza that will extend to the abstract class VegPizza.

SmallMasalaPizza.java

```
public class SmallMasalaPizza extends
    VegPizza{@Override
    public float price()
        {return 100.0f;
    }
    @Override
    public String name()
        { return  "Masala
        Pizza";
    }
    @Override
    public String size()
```

```
    { return  "Samll
    Size";
  }
}// End of the SmallMasalaPizza class
```

MediumMasalaPizza.java

```
public class MediumMasalaPizza extends VegPizza
  {@Override
  public float price()
    {return 120.0f;
  }
  @Override
  public String name()
    { return  "Masala
    Pizza";
  }
  @Override
  public String size()
    { return  "Medium
    Size";
  }
File: LargeMasalaPizza.java
public class LargeMasalaPizza extends
  VegPizza{@Override
```

```
    public float price()
        {return 150.0f;
    }
     @Override
    public String name()
        { return  "Masala
        Pizza";
    }
    @Override
    public String size()
       { return  "Large
       Size";
    }
} //End of the LargeMasalaPizza class
```

ExtraLargeMasalaPizza.java

```
public class ExtraLargeMasalaPizza extends VegPizza
    {@Override
    public float price()
        {return 180.0f;
    }


    @Override
    public String name()
        { return  "Masala
```

```
    Pizza";
    }
    @Override
    public String size() {
      return  "Extra-Large Size";
    }
}// End of the ExtraLargeMasalaPizza class
```

Step 9: Now, create concrete sub-classes SmallNonVegPizza, MediumNonVegPizza, LargeNonVegPizza, ExtraLargeNonVegPizza that will extend to the abstract class NonVegPizza.

SmallNonVegPizza.java

```
public class SmallNonVegPizza extends NonVegPizza
    {@Override
    public float price()
        {return 180.0f;
    }
    @Override
    public String name()
      { return "Non-Veg
      Pizza";
    }
    @Override
    public String size()
        { return "Samll
        Size";
```

```
    }
}// End of the SmallNonVegPizza class
```

MediumNonVegPizza.java

```
public class MediumNonVegPizza extends
    NonVegPizza{@Override
    public float price()
        {return 200.0f;
    }
    @Override
    public String name()
       { return "Non-Veg
      Pizza";
    }
    @Override
    public String size()
        { return "Medium
        Size";
    }
```

LargeNonVegPizza.java

```
public class LargeNonVegPizza extends
    NonVegPizza{@Override
    public float price()
        {return 220.0f;
    }
```

```
    @Override
    public String name()
       { return "Non-Veg
      Pizza";
    }
    @Override
    public String size()
        { return "Large
        Size";
    }

}// End of the LargeNonVegPizza class
```

ExtraLargeNonVegPizza.java

```
public class ExtraLargeNonVegPizza extends NonVegPizza
    {@Override
    public float price()
        {return 250.0f;
    }
    @Override
    public String name()
       { return "Non-Veg
      Pizza";
    }
```

```
    @Override
    public String size() {
        return "Extra-Large Size";
}
}
 // End of the ExtraLargeNonVegPizza class
```

Step 10: Now, create two abstract classes Pepsi and Coke that will extend abstract class ColdDrink.

Pepsi.java

```
public abstract class Pepsi extends ColdDrink
    {@Override
    public abstract  String name();
    @Override
    public abstract  String size();
    @Override
    public abstract  float price();
}// End of the Pepsi class
```

Coke.java

```
public abstract class Coke  extends ColdDrink
    {@Override
    public abstract  String name();
    @Override
    public abstract  String size();
    @Override
```

```
    public abstract  float price();

}// End of the Coke class
</textaea></div>

<p>Step 11:<b>Now, create concrete sub-classes SmallPepsi, MediumPepsi, LargePepsi that will extend to the abstract class Pepsi.</b></p>

<div id="filename">File: SmallPepsi.java</div>

<div class="codeblock"><textarea  name="code" class="java">
public class SmallPepsi  extends Pepsi{
    @Override
    public String name()
       { return "300 ml
       Pepsi";
    }
    @Override
    public float price()
        {return 25.0f;
    }
    @Override
    public String size()
        { return "Small
        Size";
    }
}// End of the SmallPepsi class
```

MediumPepsi.java

```
public class MediumPepsi extends Pepsi {


    @Override
    public String name() {
       return "500 ml Pepsi";
    }
    @Override
    public String size()
        { return "Medium
        Size";
    }
    @Override
    public float price()
        {return 35.0f;
    }
}// End of the MediumPepsi class
```

LargePepsi.java

```
public class LargePepsi extends
    Pepsi{@Override
    public String name()
       { return "750 ml
       Pepsi";
    }
```

```
    @Override
    public String size()
        { return "Large
        Size";
    }
    @Override
    public float price() {
        return 50.0f;
    }
}// End of the LargePepsi class
```

Step 12: Now, create concrete sub-classes SmallCoke, MediumCoke, LargeCoke that will extend to the abstract class Coke.

SmallCoke.java

```
public class SmallCoke extends
    Coke{@Override
    public String name()
         { return "300 ml
         Coke";
    }
    @Override
    public String size()
        { return "Small
        Size";
    }
```

```
    @Override
    public float price()
        {return  25.0f;
    }
 }// End of the SmallCoke class
```

MediumCoke.java

```
public class MediumCoke extends
    Coke{@Override
    public String name()
         { return "500 ml
         Coke";
    }
    @Override
    public String size()
        { return "Medium
        Size";
    }
    @Override
    public float price()
        {return  35.0f;
   }
}// End of the MediumCoke class
```

LargeCoke.java

```
public class LargeCoke extends Coke

 {@Override

public String name()

    { return "750 ml

    Coke";

}

@Override

public String size()

    { return "Large

   Size";

}

@Override
public float price()

    {return  50.0f;

}

}// End of the LargeCoke class

</textrea></div>

<p>Step 13:<b>Create an OrderedItems class that are having Item objects defined above.</b></p>

<div id="filename">File: OrderedItems.java</div>

<div class="codeblock"><textarea  name="code" class="java">

import java.util.ArrayList;

import java.util.List;

public class OrderedItems {
```

```
    List<Item> items=new ArrayList<Item>();
    public void addItems(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.println("Item is:" +item.name());
            System.out.println("Size is:" +item.size());
            System.out.println("Price is: " +item.price());


        }
    }
}// End of the OrderedItems class
```

Step 14: Create an OrderBuilder class that will be responsible to create the objects of OrderedItems class.

OrdereBuilder.java

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
public class OrderBuilder {
public OrderedItems preparePizza() throws IOException{
OrderedItems itemsOrder=new OrderedItems();
BufferedReader br =new BufferedReader(new InputStreamReader(System.in));
System.out.println(" Enter the choice of Pizza ");
System.out.println("============================");
System.out.println(" 1. Veg-Pizza ");
System.out.println(" 2. Non-Veg Pizza ");
System.out.println(" 3. Exit ");
System.out.println("============================");
int pizzaandcolddrinkchoice=Integer.parseInt(br.readLine());
switch(pizzaandcolddrinkchoice)
{
   case 1:{
   System.out.println("You ordered Veg Pizza");
   System.out.println("\n\n");
   System.out.println(" Enter the types of Veg-Pizza ");
   System.out.println(" ");
   System.out.println(" 1.Cheeze Pizza ");
   System.out.println(" 2.Onion Pizza ");
   System.out.println(" 3.Masala Pizza ");
   System.out.println(" 4.Exit ");
   System.out.println(" ");
   int vegpizzachoice=Integer.parseInt(br.readLine());
   switch(vegpizzachoice)
   {
      case 1:
      {
         System.out.println("You ordered Cheeze Pizza");
         System.out.println("Enter the cheeze pizza size");
         System.out.println(" ");
         System.out.println(" 1. Small Cheeze Pizza ");
         System.out.println(" 2. Medium Cheeze Pizza ");
         System.out.println(" 3. Large Cheeze Pizza ");
```

```
        System.out.println(" 4. Extra-Large Cheeze Pizza ");
        System.out.println(" ");
        int cheezepizzasize=Integer.parseInt(br.readLine());
        switch(cheezepizzasize)
        {
           case 1:
           itemsOrder.addItems(new SmallCheezePizza());
           break;
           case 2:
           itemsOrder.addItems(new MediumCheezePizza());
           break;
           case 3:
           itemsOrder.addItems(new LargeCheezePizza());
           break;
           case 4:
           itemsOrder.addItems(new ExtraLargeCheezePizza());
           break;
        }
      }
      case 2:
        {
             System.out.println("You ordered Onion Pizza");
             System.out.println("Enter the Onion pizza size");
             System.out.println(" ");
             System.out.println(" 1. Small Onion Pizza ");
             System.out.println(" 2. Medium Onion Pizza ");
             System.out.println(" 3. Large Onion Pizza ");
             System.out.println(" 4. Extra-Large Onion Pizza ");
             System.out.println(" ");
             int onionpizzasize=Integer.parseInt(br.readLine());
             switch(onionpizzasize)
             {
case 1:
itemsOrder.addItems(new SmallOnionPizza());
```

```
break;
case 2:
itemsOrder.addItems(new MediumOnionPizza());
break;
case 3:
itemsOrder.addItems(new LargeOnionPizza());
break;
case 4:
itemsOrder.addItems(new ExtraLargeOnionPizza());
break;
}
}
break;
case 3:
{
System.out.println("You ordered Masala Pizza");
System.out.println("Enter the Masala pizza size");
System.out.println(" ");
System.out.println(" 1. Small Masala Pizza ");
System.out.println(" 2. Medium Masala Pizza ");
System.out.println(" 3. Large Masala Pizza ");
System.out.println(" 4. Extra-Large Masala Pizza ");
System.out.println(" ");
int masalapizzasize=Integer.parseInt(br.readLine());
switch(masalapizzasize)
{
case 1:
itemsOrder.addItems(new SmallMasalaPizza());
break;
case 2:
itemsOrder.addItems(new MediumMasalaPizza());
break;
case 3:
itemsOrder.addItems(new LargeMasalaPizza());
```

```
break;
case 4:
itemsOrder.addItems(new ExtraLargeMasalaPizza());
break;
}
}
break;
}
}
break;// Veg- pizza choice completed.
case 2:
{
System.out.println("You ordered Non-Veg Pizza");
System.out.println("\n\n");
System.out.println("Enter the Non-Veg pizza size");
System.out.println(" ");
System.out.println(" 1. Small Non-Veg Pizza ");
System.out.println(" 2. Medium Non-Veg Pizza ");
System.out.println(" 3. Large Non-Veg Pizza ");
System.out.println(" 4. Extra-Large Non-Veg Pizza ");
System.out.println(" ");
int nonvegpizzasize=Integer.parseInt(br.readLine());
switch(nonvegpizzasize)
{
case 1:
itemsOrder.addItems(new SmallNonVegPizza());
break;
case 2:
itemsOrder.addItems(new MediumNonVegPizza());
break;
case 3:
itemsOrder.addItems(new LargeNonVegPizza());
break;
case 4:
```

```
itemsOrder.addItems(new ExtraLargeNonVegPizza());
break;
}
}
break;
default:
{
break;
}
}//end of main Switch
//continued?..
System.out.println(" Enter the choice of ColdDrink ");
System.out.println("============================");
System.out.println(" 1. Pepsi ");
System.out.println(" 2. Coke ");
System.out.println(" 3. Exit ");
System.out.println("============================");
int coldDrink=Integer.parseInt(br.readLine());
switch (coldDrink)
{
case 1:
{
System.out.println("You ordered Pepsi ");
System.out.println("Enter the Pepsi Size ");
System.out.println(" ----------------------");
System.out.println(" 1. Small Pepsi ");
System.out.println(" 2. Medium Pepsi ");
System.out.println(" 3. Large Pepsi ");
System.out.println(" ----------------------");
int pepsisize=Integer.parseInt(br.readLine());
switch(pepsisize)
{
case 1:
itemsOrder.addItems(new SmallPepsi());
```

```
break;
case 2:
itemsOrder.addItems(new MediumPepsi());
break;
case 3:
itemsOrder.addItems(new LargePepsi());
break;
}
}
break;
case 2:
{
System.out.println("You ordered Coke");
System.out.println("Enter the Coke Size");
System.out.println(" ----------------------");
System.out.println(" 1. Small Coke ");
System.out.println(" 2. Medium Coke ");
System.out.println(" 3. Large Coke ");
System.out.println(" 4. Extra-Large Coke ");
System.out.println(" ----------------------");
int cokesize=Integer.parseInt(br.readLine());
switch(cokesize)
{
case 1:
itemsOrder.addItems(new SmallCoke());
break;
case 2:
itemsOrder.addItems(new MediumCoke());
break;
case 3:
itemsOrder.addItems(new LargeCoke());
break;
}
}
```

```
break;
default:
{
break;
}
}//End of the Cold-Drink switch
return itemsOrder;
}
}
```

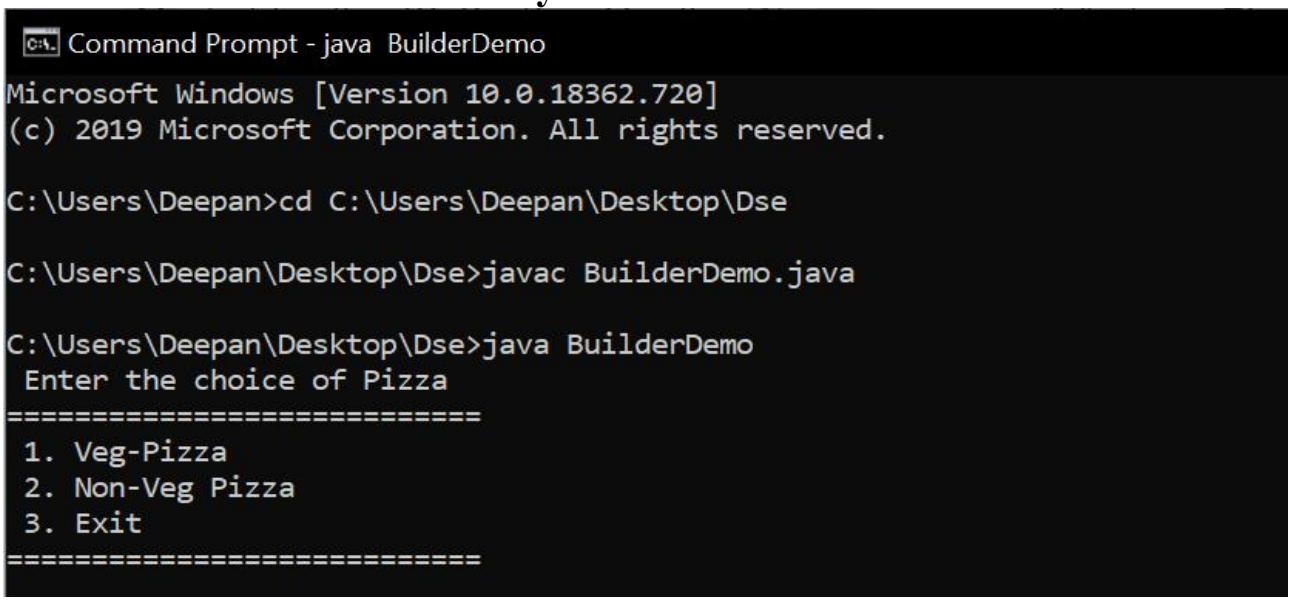
Step 15: Create a BuilderDemo class that will use the OrderBuilder class.

BuilderDemo.java

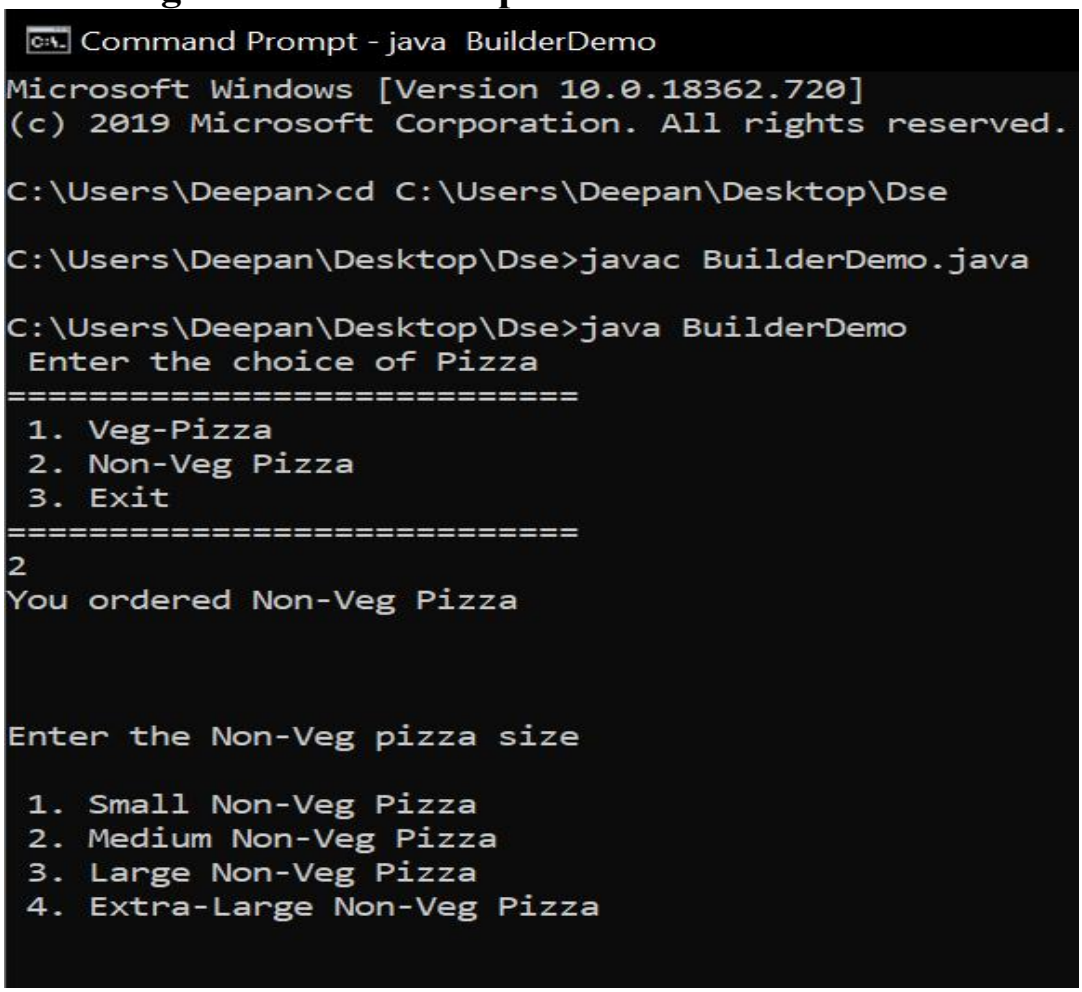
```
import java.io.IOException;
public class BuilderDemo {
public static void main(String[] args) throws IOException {
// TODO code application logic here
OrderBuilder builder=new OrderBuilder();
OrderedItems orderedItems=builder.preparePizza();
orderedItems.showItems();
System.out.println("\n");
System.out.println("Total Cost : "+ orderedItems.getCost());
}
}// End of the BuilderDemo class
```

15. Output:

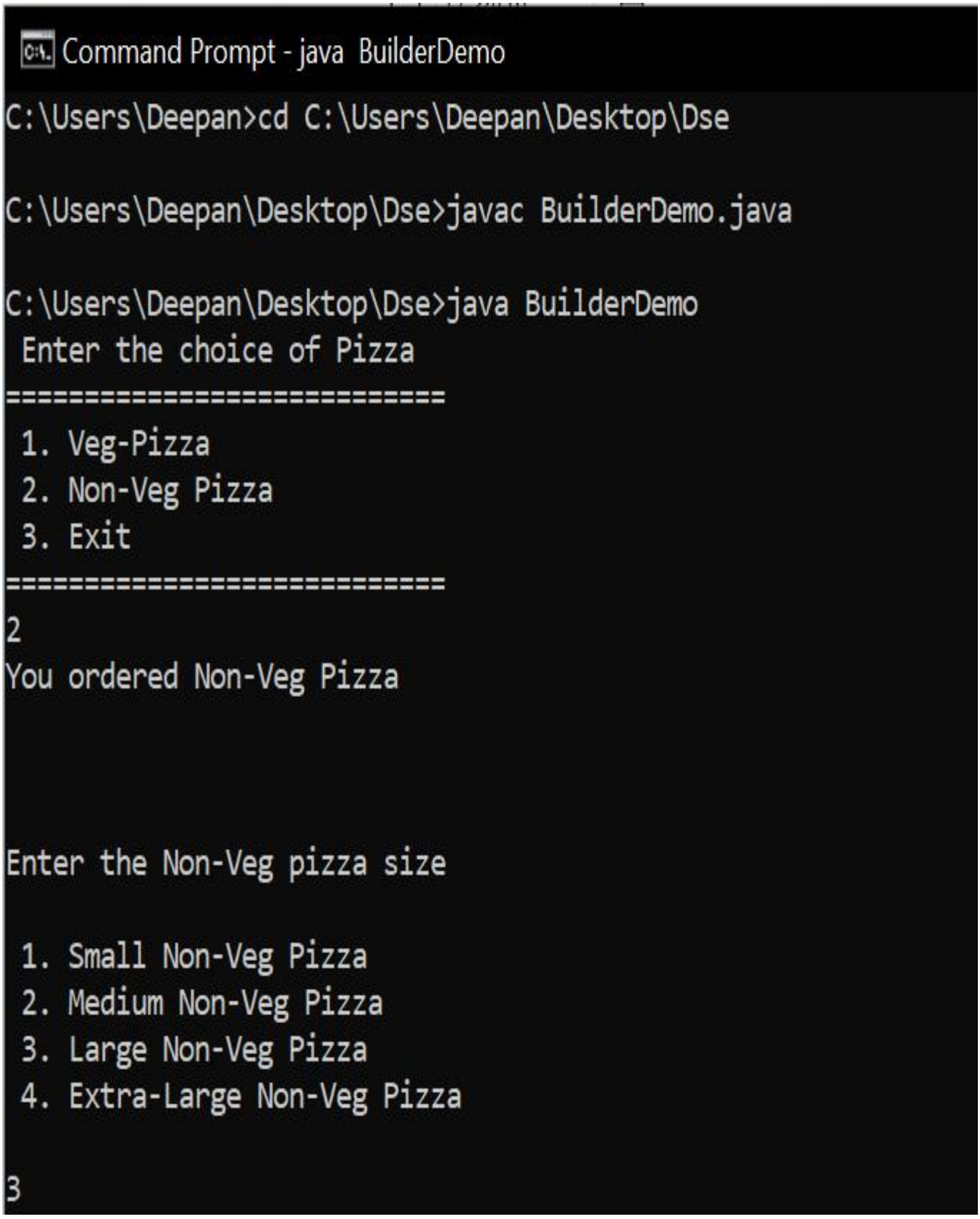
The main interface of the system



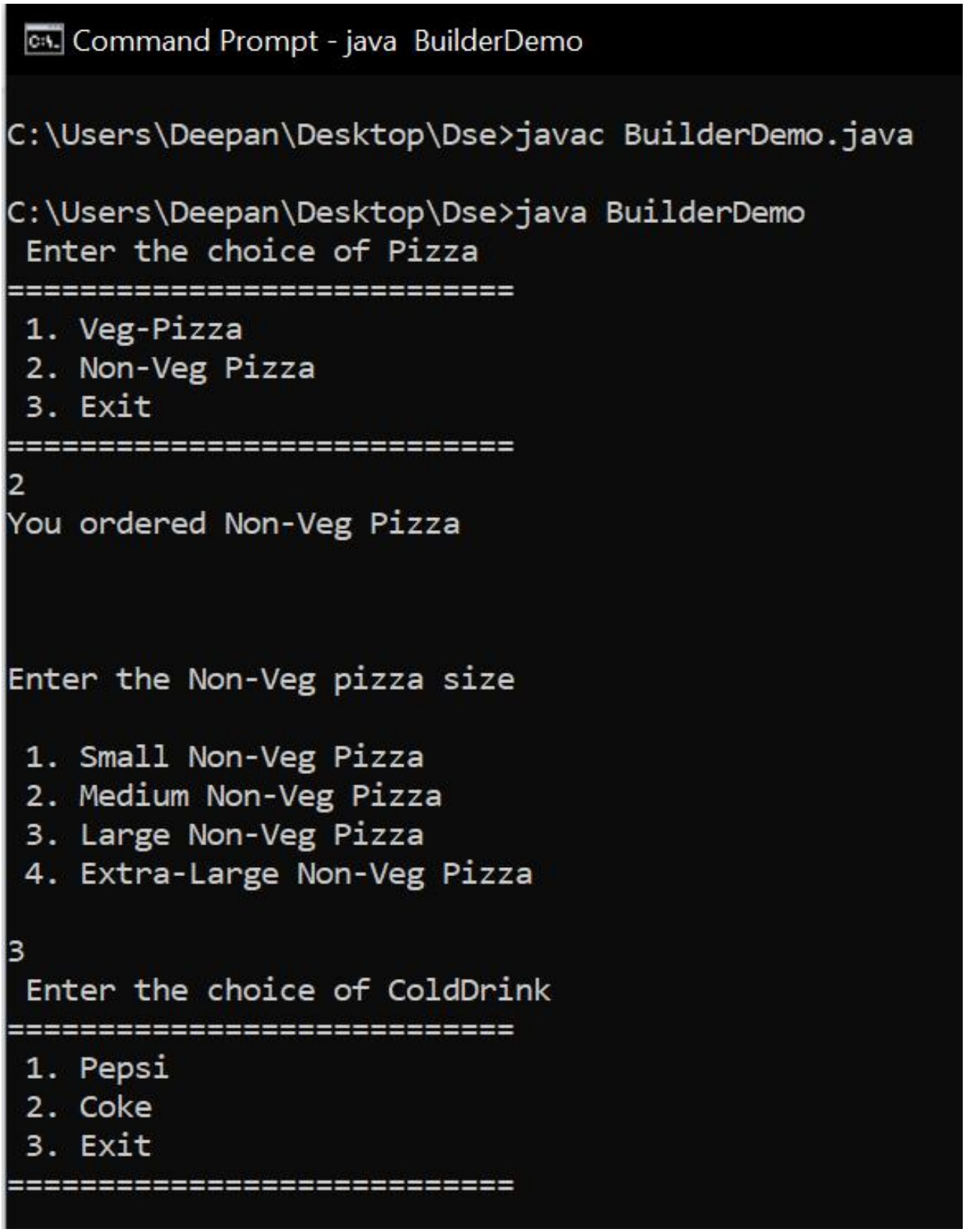
Entering the choice of the pizza user needs



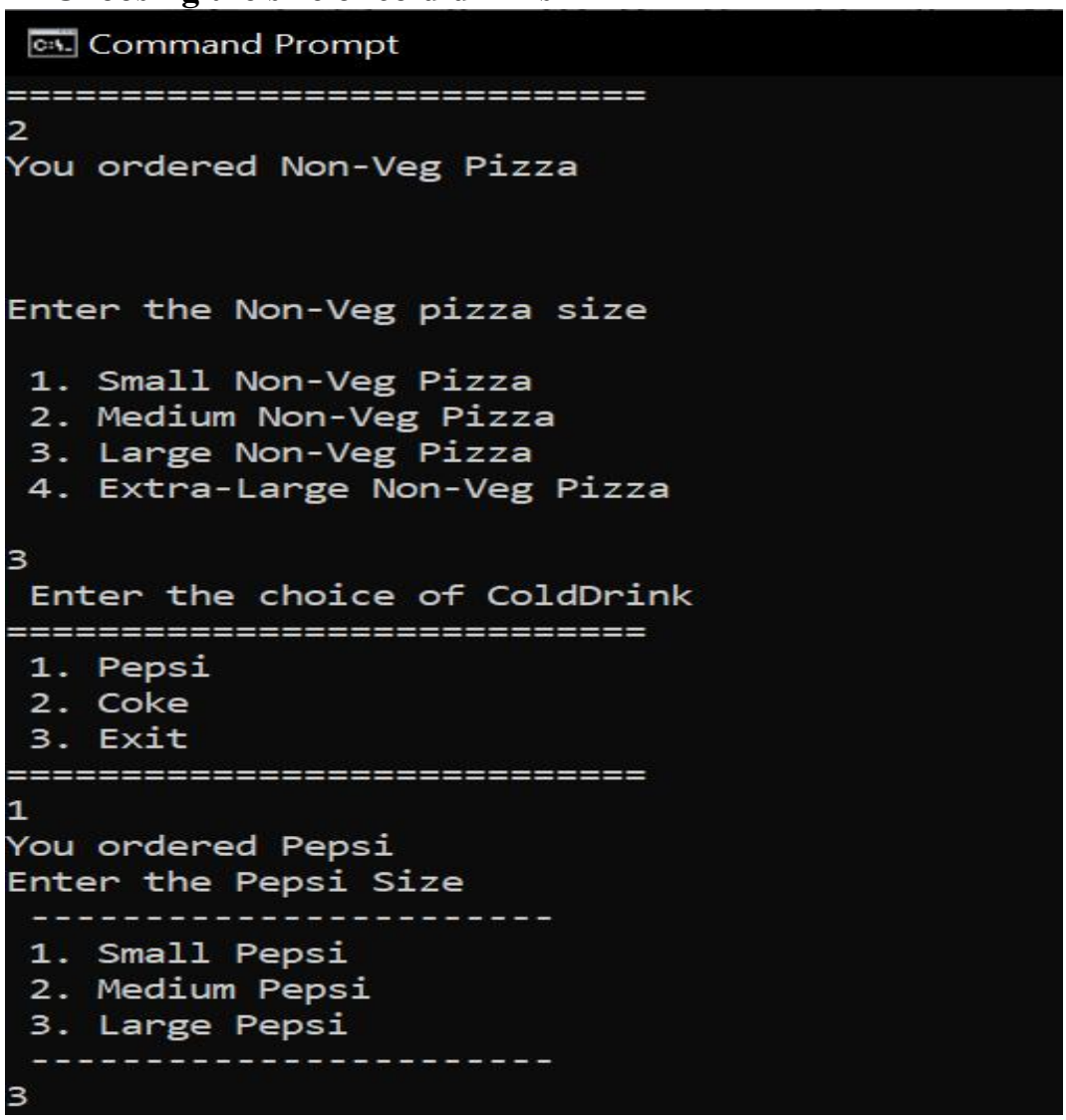
Selecting the size of the pizza



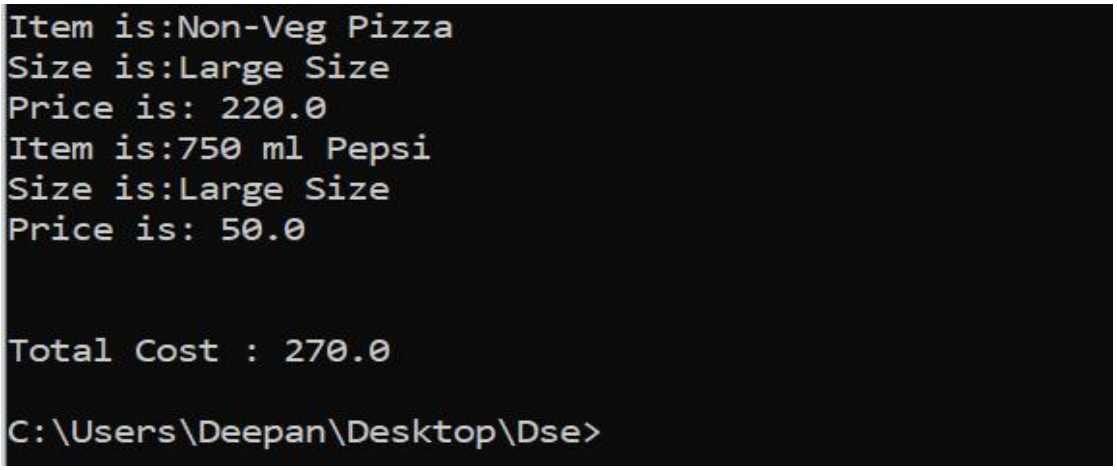
Selecting the type of cold drinks



Choosing the size of cold drinks



It prints the items we have chosen with the total cost:



16.References and links followed:

1. Dürschmid, Tobias. "Design pattern builder: a concept for refinable reusable design pattern libraries." Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity. 2016.
2. Dürschmid, Tobias. "Design Pattern Builder."
3. refactoring.guru/design-patterns/builder
4. howtodoinjava.com/design-patterns/creational/builder-pattern-in-java/
5. en.wikipedia.org/wiki/Builder_pattern

Design Patterns

Title : Pizza ordering system using builder pattern

Team Details

19MIS0068 - Shanmuga Mayan
19MIS0086 - Arun Kumar
19MIS0102 - Deepan.S

Abstract

- Design patterns are typical solutions to common problems in software design.
- Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- Patterns are a toolkit of solutions to common problems in software design.

Introduction

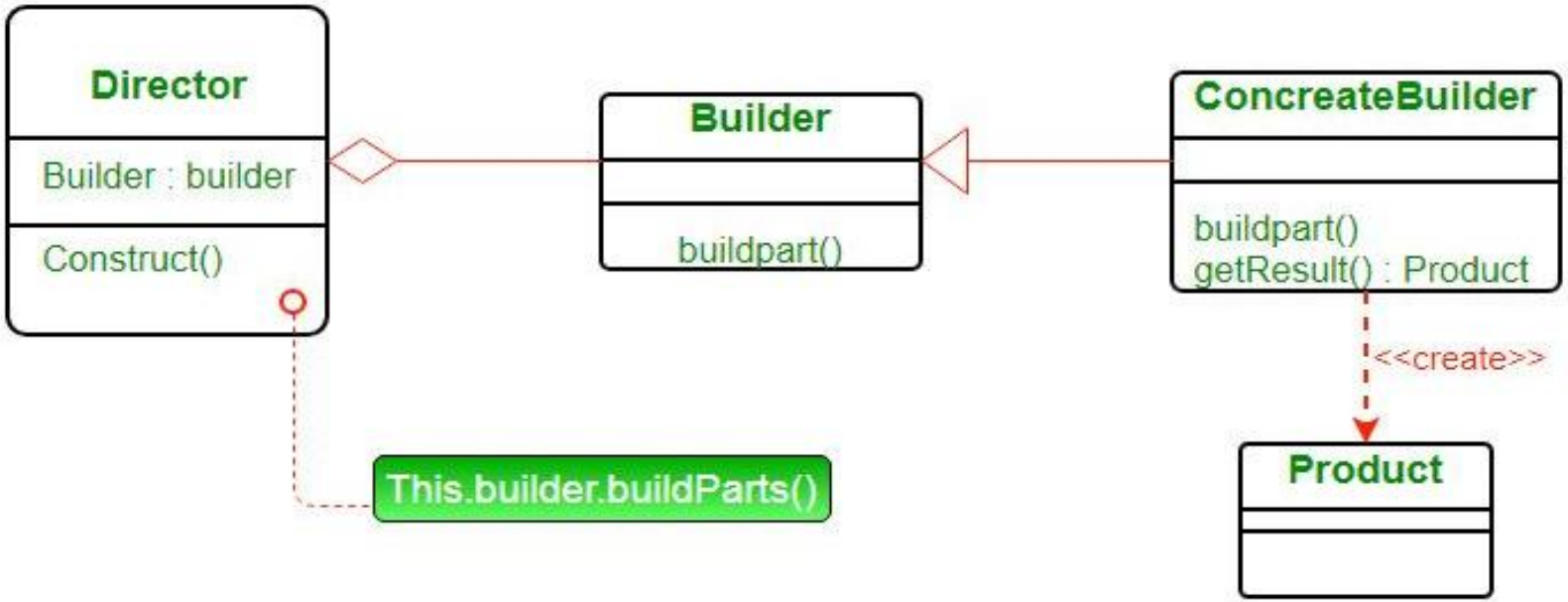
- The Builder design pattern is one of the creational design patterns that describe how to solve recurring design problems in object-oriented software
- The Builder design pattern describes how to solve such problems :

1. Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
2. A class delegates object creation to a builder object instead of creating the objects directly.

Intent

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Structure


Director
Builder : builder
Construct()
Builder
buildpart()
ConcreateBuilder
buildpart()
getResult() : Product
<<create>>
Product
This.builder.buildParts()

Collaborations

- The client creates the Director object and configures it with the desired Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Consequences

- It lets you vary a product's internal representation
- It isolates code for construction and representation
- It gives you finer control over the construction process

Implementation issues

- Assembly and construction interface
- Why no abstract class for products
- Empty methods as default in Builder

System Implementation

- In the present scenario we have implemented the following code in java
- First, we have created an interface Item that represents the Veg and Non Veg Pizza.
- Now, created concrete sub-classes SmallCheezePizza, MediumCheezePizza, LargeCheezePizza, ExtraLargeCheezePizza that will extend to the abstract class VegPizza. Similarly, we have created for NonVegPizza.
- Now we have created an OrderBuilder class that will be responsible to create the objects of OrderedItems class.
- At last, we have created BuilderDemo class which uses orderbuilder class to show the cost os the items we have ordered and show the items we have ordered

THANK YOU ☺