Assignment – 2

Question 1: Find an application of Singleton design pattern in your application and draw its corresponding class diagram and a collaboration diagram and implement it in Java. Explain in a text document why singleton should be applied in the scenario and how it improves your design (30 points).

1. Singleton Class:
    a. private class constructor
    b. static variables and static method

Admin class: attributes

i. adminUsername
ii. adminPassword

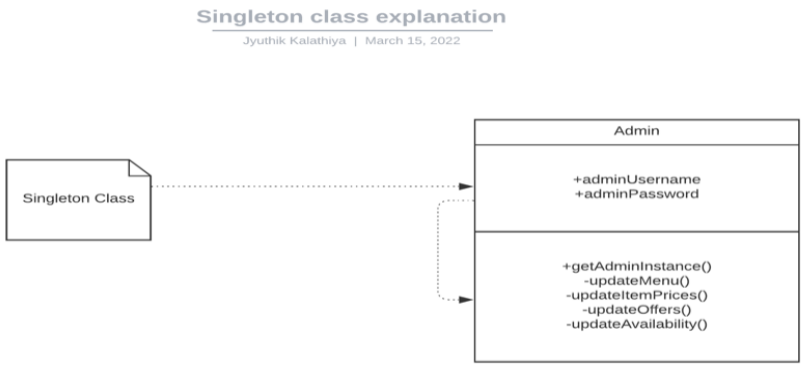
Methods:

- getAdminInstance()
- validateAdmin()
- updateMenu()
- updateItemPrices()
- updateOffers()
- updateAvailibilty()

How Admin class becomes more secure by using singleton design pattern?

- Singleton class makes a class to generate a single instance object throughout the system. Similarly, Administrator uses admin class for login onto server side. When multiple users try to access admin at one time, it should prohibit since already one admin is logged in. But using singleton design pattern we remove this vulnerability. Making more secure and reliable server-side administration. The credentials get verified from database's admin details.

Here is the class diagram:



Java Code Implementation:

```
import java.util.*;

class Admin {
   static Admin admin;
   String adminUsername;
   String adminPassword;

   private Admin(String adminUsername, String adminPassword) {
         adminUsername = this.adminUsername;
         adminPassword = this.adminPassword;
   }

   public static Admin getAdminInstance(String adminUsername, String
adminPassword) {

      if(admin==null){
         return new Admin(adminUsername,adminPassword);
      }

      return admin;
   }
```

```
   public static boolean validateAdmin(String adminUsername, String
adminPassword){
      if(adminUsername == admin.adminUsername && adminPassword ==
admin.adminPassword) {
         System.out.println("Welcome Admin");
         return true;
      }

      return false;
   }
   public void updateMenu() {
      //returns the existing menu and asks for update if yes then new items details
are added.
   }

   public void updateItemPrices() {
      //updates the list of prices associated with each item.
   }

   public void updateOffers() {
      //add or remove any offers for customers.
   }

   public void updateAvailabilty() {
      //updates timings or days off for maintenance of application.
   }
}
```

```
import java.util.*;

class demo {
    public static void main(String[] args) {
        Admin ad1 = Admin.getAdminInstance("xyz","Boolean");
        System.out.println(ad1);
        Admin ad2 = Admin.getAdminInstance("xyz","Boolean");
        System.out.println(ad2);
    }
}
```

Output:

```
[Running] cd "/Users/juythikkalathiya/Desktop/Coding/" && javac demo.java && java demo
Admin@251a69d7
Admin@7344699f
```

Question 2: Find an application of Singleton design pattern in your application and draw its corresponding class diagram and a collaboration diagram and Implement it in Java. Explain in a text document why singleton should be applied in the scenario and how it improves your design (30 points).

1) Builder Pattern:

Here, as we know pizza is prepared with different types such as Chicken Pizza, Flat Bread, Pepporoni Pizza with extra cheese, etc. with a set of specifications.

- Builder Design Pattern is an object creation pattern.
- Here the use case is that the pizza will have different types, toppings, crust etc.
- So, we have used four Enum classes such as Toppings, Crust, Size, Cheese type, etc.
- Builder is basically used in case if we want to limit the object creation with certain properties/features.
- For example, here we have shown 4 attributes that are mandatory to set before the object is created or if we want to freeze the object creation until certain attributes are not set yet.
- Basically, we have used is instead of constructor.
- The builder will expose the attributes that the generated object should have but it also hides how to set them.

Builder design pattern is used when we must create complex objects with many mandatory or optional parameters. So, in this case we have many parameters such as Toppings, Crust, Size, Cheese Type, etc.
Also, it is used for building something which afterwards will exist and should not be altered.

Java Code –

```
package Pizza;
import java.util.*;
public class OrderPizza {
   public static void main(String[] args) {
      PizzaBuilder myPizza = new PizzaBuilder();
      //PizzaBuilder myPizza = new PizzaBuilder();
      myPizza.setCheese(PizzaCheeseType.AMERICAN);
      myPizza.setCrust(PizzaCrustType.STUFFED);
      myPizza.setSize(PizzaSize.LARGE);
      myPizza.setTopping(PizzaToppings.CHICKEN);
      myPizza.setTopping(PizzaToppings.MUSHROOM);
      Pizza pizza = myPizza.buildPizza();
      System.out.println("Pizza Specifications : ");
```

```
        System.out.println("Pizza Crust : "+pizza.getCrust());
        System.out.println("Pizza Size : "+pizza.getSize());
        System.out.println("Pizza Cheese : "+pizza.getCheese());
        System.out.println("Pizza Toppings : "+pizza.getTopping());
        System.out.println("Total Price for the Pizza : $"+pizza.calculateTotalCost());
    }
}

class PizzaBuilder {
    private PizzaCrustType crust;
    private PizzaCheeseType cheese;
    private PizzaSize size;
    private HashSet<PizzaToppings> toppings = new HashSet<PizzaToppings>();
    public PizzaBuilder(){

    }
    public void setCheese(PizzaCheeseType cheese) {
        this.cheese = cheese;
    }

    public void setSize(PizzaSize size) {
        this.size = size;
    }

    public void setTopping(PizzaToppings topping) {
        this.toppings.add(topping);
        //this.topping = topping;
    }
    public void setCrust(PizzaCrustType crust) {
        this.crust = crust;
    }

    public Pizza buildPizza(){
        return new Pizza(this.crust, this.cheese, this.toppings, this.size);
    }


}

class Pizza {

    private float totalPrice = 0;
    private PizzaSize pizzaSize;
    //private PizzaToppings topping;
```

```
   private PizzaCrustType crust;
   private PizzaCheeseType cheese;
   private HashSet<PizzaToppings> toppings = new HashSet<PizzaToppings>();

   public Pizza(PizzaCrustType crust, PizzaCheeseType cheese, HashSet<PizzaToppings>
toppings, PizzaSize size){
      this.crust = crust;
      this.cheese = cheese;
      this.toppings = toppings;
      this.pizzaSize = size;
   }
   public PizzaSize getSize() {
      return pizzaSize;
   }

   public float getToppingsPrice(HashSet<PizzaToppings> toppings) {
      Iterator<PizzaToppings> topping = toppings.iterator();
      float totalToppingPrice = 0;
      while (topping.hasNext()){
         totalToppingPrice = totalToppingPrice + topping.next().getCost();
      }
      return totalToppingPrice;

   }

   public HashSet<PizzaToppings> getTopping() {
      return this.toppings;
   }

   public PizzaCrustType getCrust() {
      return crust;
   }

   public PizzaCheeseType getCheese() {
      return cheese;
   }
   public double calculateTotalCost() {
      //pizza.setTopping(topping);
      totalPrice = this.cheese.getCost() + this.crust.getCost() +
this.getToppingsPrice(this.toppings) + this.pizzaSize.getCost();
      return totalPrice;
   }

}
```

```
enum PizzaCheeseType {
   AMERICAN {
      public float getCost() {
         return 5;
      }
   }, ITALIAN {
      public float getCost() {
         return 5;
      }
   };

   public abstract float getCost();

}

enum PizzaSize {
   REGULAR {
      public float getCost() {
         return 8;
      }
   },
   MEDIUM {
      public float getCost() {
         return 10;
      }
   }, LARGE {
      public float getCost() {
         return 15;
      }
   };

   public abstract float getCost();

}

enum PizzaToppings {
   ONIONS {
      public float getCost(){
         return 1.5f;
      }
   }, TOMATO{
      public float getCost(){
         return 2;
      }
```

```
    }, BELLPEPPERS{
        public float getCost(){
            return 1.5f;
        }
    },
    PEPPERONI {
        public float getCost(){
            return 2;
        }
    }, CHICKEN{
        public float getCost(){
            return 3;
        }
    }, MUSHROOM{
        public float getCost(){
            return 2;
        }
    };

    public abstract float getCost();

}
enum PizzaCrustType {
    THIN  {
        public float getCost(){
            return 3;
        }
    } , STUFFED{
        public float getCost(){
            return 7;
        }
    };

    public abstract float getCost();

}
```

Output :

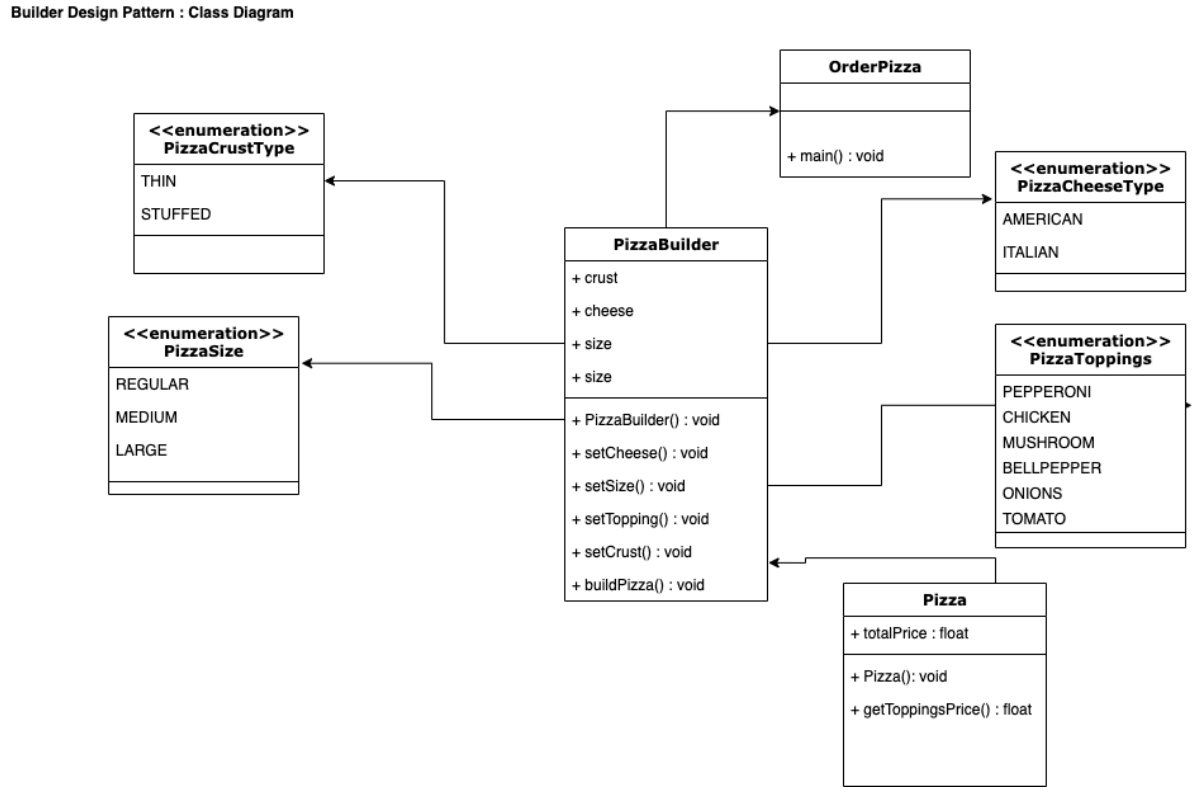
```
25      private PizzaSize size;
26      private HashSet<PizzaToppings> toppings = new HashSet<PizzaToppings>();
27      public PizzaBuilder(){
28
29      }
30      public void setCheese(PizzaCheeseType cheese) {
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

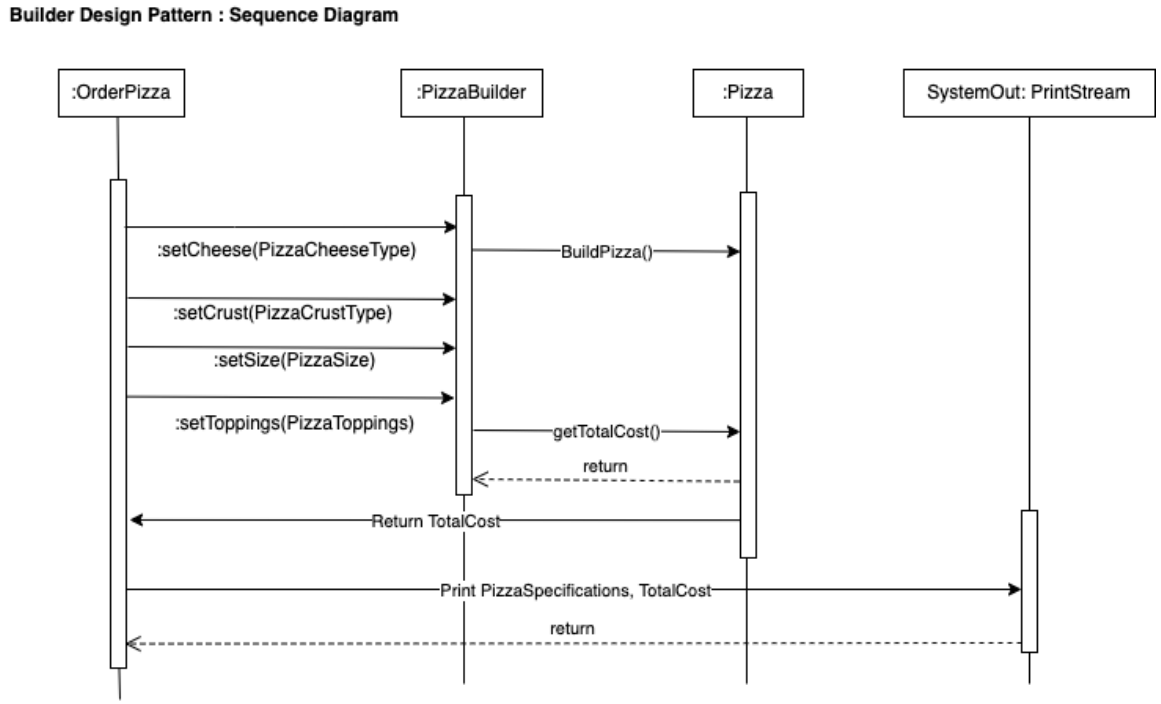
```
Total Price for the Pizza : $345.0
(base) deeshanirmal@Deeshas-Air Pizza % javac OrderPizza.java
(base) deeshanirmal@Deeshas-Air Pizza % java OrderPizza.java
Pizza Specifications :
Pizza Crust : STUFFED
Pizza Size : LARGE
Pizza Cheese : AMERICAN
Pizza Toppings : [MUSHROOM, CHICKEN]
Total Price for the Pizza : $31.0
(base) deeshanirmal@Deeshas-Air Pizza % java PizzaStore.java
Error: Could not find or load main class PizzaStore.java
Caused by: java.lang.ClassNotFoundException: PizzaStore.java
(base) deeshanirmal@Deeshas-Air Pizza % javac PizzaStore.java
error: file not found: PizzaStore.java
Usage: javac <options> <source files>
```

OUTLINE
TIMELINE
JAVA PROJECTS
SPRING BOOT DASHBOARD

Class Diagram:



Sequence Diagram:



2) Factory Pattern

- Factory pattern is one of the widely used patterns for creating an object in Java. It helps in figuring out which item class instance to make.
- In Factory design, we make object without uncovering the creation rationale to the client and allude to recently made object utilizing a typical connection point.
- For this specific project "Online Pizza Delivery", the use case showcases the final stages of pizza making which includes the processes like preparation, bake, pick, route, deliver, and finally order.
- Therefore, we have created three classes:
  1) Pizza

     a) bake ()
     b) pick ()
     c) route ()
     d) deliver ()

  2) Menu ()
     a) VegPizza ()
     b) NonvegPizza ()
     c) Sandwich ()

  3) Order
     Subclass: Menu ()
     Margherita ()
     FarmHouse ()
     PeppyPaneer ()
     MexicanGreenWave ()
     DeluxeVeggie ()
     PepperBarbequeChicken ()
     ChickenSausage ()
     ChickenGoldenDelight ()
     ChickenDominator ()
     ChickenFiesta ()
     BuffaloChicken ()
     ChickenHabanero ()
     MediterrareanVeggie ()
     PhillyCheeseSteak ()
     ChickenBaconRanch ()

- Production line technique design configuration plot: conceptual the launch capacity of pizza project into unique strategies and execute.
- Menu () class instigates further order process from three categories namely Veg, Nonveg and Sandwich respectively.

- The fifteen examples above are subclasses of the abstract pizza class. Based on user input, these classes are called from the menu classes.
- Here we have shown credits that are required to set before the item is made or on the other hand to freeze the article creation until specific ascribes are not yet set.
- Finally, when all the steps are completed, and code compiles and runs successfully than the last stage is executed.
- At last Oder will be placed.

Step summary

Step 1: create a Pizza abstract class

```
public abstract class Pizza {
   public String name;
   public abstract void prepare();
   public void bake() {
      System.out.println(name + " is being prepared");
   }
   public void pick() {
      System.out.println("Your order has been picked up.");
   }
   public void route() {
      System.out.println("Your order is en route");
   }
   public void deliver() {
      System.out.println("Your order is being delivered");
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Analysis: this class is used to represent the whole process of making pizza: bake(), pick(), route()and deliver(). Assuming that the materials required in each pizza preparation stage are different, the preparation stage is defined as an abstract method, and the other three stages are the same.

Step 2: create pizza classes

```
public class FARMHOUSE extends Pizza {
   @Override
   public void prepare() {
      setName("FARM HOUSE Pizza");
      System.out.println("FARM HOUSE Pizza preparing raw materials");
   }
}
```

Analysis: this class is used to create FARM HOUSE category pizza.

```
public class PEPPYPANEER extends Pizza {
   @Override
   public void prepare() {
      setName("PEPPY PANEER Pizza");
      System.out.println("PEPPY PANEER Pizza preparing raw materials");
   }
}
```

Analysis: This class is used to create Peppy Paneer category pizza

```
public class MARGHERITA extends Pizza {
   @Override
   public void prepare() {
      setName("MARGHERITA Pizza");
      System.out.println("MARGHERITA Pizza");
   }
}
```

Analysis: This class is used to create Margherita category pizza.

```
public class DELUXEVEGGIE extends Pizza {
   @Override
   public void prepare() {
      setName("DELUXE VEGGIE Pizza");
      System.out.println("DELUXE VEGGIE Pizza preparing raw materials");
   }
}
```

Analysis: This class is used to create Delux Veggie category pizza.

```
public class MEXICANGREENWAVE extends Pizza {
   @Override
   public void prepare() {
      setName("MEXICAN GREEN WAVE Pizza");
      System.out.println("MEXICAN GREEN WAVE flavoured pizza is preparing");
   }
}
```

Analysis: This class is used to create Mexican Green Wave category pizza.

```
public class CHICKENGOLDENDELIGHT extends Pizza{
   @Override
   public void prepare() {
      setName("CHICKEN GOLDEN DELIGHT Pizza");
      System.out.println("CHICKEN GOLDEN DELIGHT Pizza preparing raw materials ");
   }
}
```

Analysis: This class is used to create Chicken Golden Delight category pizza.

```
public class CHICKENFIESTA extends Pizza{
   @Override
   public void prepare() {
      setName("CHICKEN FIESTA Pizza");
      System.out.println("CHICKEN FIESTA Pizza preparing raw materials ");
   }
}
```

Analysis: This class is used to create Chicken Fiesta category pizza.

```
public class CHICKENDOMINATOR extends Pizza{
   @Override
   public void prepare() {
      setName("CHICKEN DOMINATOR Pizza");
      System.out.println("CHICKEN DOMINATOR Pizza preparing raw materials ");
   }
}
```

Analysis: This class is used to create Chicken Dominator category pizza.

```
public class PEPPERBARBECUECHICKEN extends Pizza{
   @Override
   public void prepare() {
      setName("PEPPER BARBECUE CHICKEN Pizza");
      System.out.println("PEPPER BARBECUE CHICKEN preparing raw materials ");
   }
}
```

Analysis: This class is used to create Pepper Barbeque Chicken category pizza.

```
public class CHICKENSAUSAGE extends Pizza{
   @Override
   public void prepare() {
      setName("Chicken Sausage Pizza");
      System.out.println("Chicken Sausage Pizza preparing raw materials ");
   }
}
```

Analysis: This class is used to create Chicken Sausage category pizza.

```
public class BuffaloChicken extends Pizza{
   @Override
   public void prepare() {
      setName("Buffalo Chicken Sandwich");
      System.out.println("Buffalo Chicken Sandwich preparing raw materials ");
   }
}
```

Analysis: This class is used to create Buffalo Chicken Sandwich.

```
public class PhillyCheeseSteak extends Pizza{
   @Override
   public void prepare() {
      setName("Philly Cheese Steak Sandwich");
      System.out.println("Philly Cheese Steak Sandwich preparing raw materials ");
   }
}
```
Analysis: This class is used to create Philly Cheese Steak Sandwich.

```
public class MediterraneanVeggie extends Pizza{
   @Override
   public void prepare() {
      setName("Mediterranean Veggie Sandwich");
      System.out.println("Mediterranean Veggie Sandwich preparing raw materials ");
   }
}
```
Analysis: This class is used to create Mediterranean Veggie Sandwich category pizza.

```
public class ChickenHabanero extends Pizza{
   @Override
   public void prepare() {
      setName("Chicken Habanero Sandwich");
      System.out.println("Chicken Habanero Sandwich preparing raw materials ");
   }
}
```
Analysis: This class is used to create Chicken Habanero Sandwich category pizza.

```
public class ChickenBaconRanch extends Pizza{
   @Override
   public void prepare() {
      setName("Chicken Bacon Ranch Sandwich");
      System.out.println("Chicken Bacon Ranch Sandwich preparing raw materials ");
   }
}
```
Analysis: This class is used to create Chicken Bacon Ranch Sandwich.

Step 3: create an abstract class for ordering pizza

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

public abstract class OrderPizza {
   abstract Pizza createPizza(String orderType);
   public OrderPizza() {
      do {
```

```
            String orderType = getType();
            Pizza pizza = createPizza(orderType); //Abstract method, which is completed by the
factory subclass
            if (pizza == null){
               System.out.println("Failed to order pizza");
               break;
            }
            pizza.prepare();
            pizza.bake();
            pizza.pick();
            pizza.route();
            pizza.deliver();
            System.out.println("Do you want order again");
            System.out.println("1.Yes");
            System.out.println("2.No");
            String orderAgain = getType();
            if(orderAgain.equals("2"))
            {
               System.out.println("Thank you for your order");
               break;
            }

         } while (true);
      }
      private String getType() {
         try {
            BufferedReader strin = new BufferedReader(new InputStreamReader(System.in));
            System.out.println("Please enter your choice:");
            String str = strin.readLine();
            return str;
         } catch (IOException e) {
            e.printStackTrace();
            return "";
         }
      }
}
```

Analysis: an abstract method createPizza() is defined in this class, which allows each factory subclass to implement itself, and the code logic for ordering pizza is written in the constructor; The getType() function is use to get pizza type from user.

```
public class VegPizza extends OrderPizza {

   @Override
   Pizza createPizza(String orderType) {
      Pizza pizza = null;
      if(orderType.equals("1")) {
```

```
            pizza = new MARGHERITA();
        } else if (orderType.equals("2")) {
            pizza = new FARMHOUSE();
        }
        else if (orderType.equals("3")) {
            pizza = new PEPPYPANEER();
        }
        else if (orderType.equals("4")) {
            pizza = new MEXICANGREENWAVE();
        }
        else if (orderType.equals("5")) {
            pizza = new DELUXEVEGGIE();
        }
        return pizza;
    }
}
```

Analysis: This class inherits the OrderPizza class and serves as a pizza ordering distributor for the store's vegetarian menu.

```
public class NonVegPizza extends OrderPizza {
    @Override
    Pizza createPizza(String orderType) {
        Pizza pizza = null;
        if(orderType.equals("1")) {
            pizza = new PEPPERBARBECUECHICKEN();
        } else if (orderType.equals("2")) {
            pizza = new CHICKENSAUSAGE();
        }
        else if (orderType.equals("3")) {
            pizza = new CHICKENGOLDENDELIGHT();
        }
        else if (orderType.equals("4")) {
            pizza = new CHICKENDOMINATOR();
        }
        else if (orderType.equals("5")) {
            pizza = new CHICKENFIESTA();
        }
        return pizza;
    }
}
```

Analysis: This class inherits the OrderPizza class and serves as a pizza ordering distributor for the store's non vegetarian menu.

```
public class Sandwich extends OrderPizza {

    @Override
```

```
    Pizza createPizza(String orderType) {
        Pizza pizza = null;
        if(orderType.equals("1")) {
            pizza = new BuffaloChicken();
        } else if (orderType.equals("2")) {
            pizza = new ChickenHabanero();
        }
        else if (orderType.equals("3")) {
            pizza = new MediterraneanVeggie();
        }
        else if (orderType.equals("4")) {
            pizza = new PhillyCheeseSteak();
        }
        else if (orderType.equals("5")) {
            pizza = new ChickenBaconRanch();
        }
        return pizza;
    }
}
```

Analysis: This class inherits the OrderPizza class and serves as a pizza ordering distributor for the store's sandwich menu.

Step 4: create a run class

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

public class PizzaStore {
    public static void main(String[] args) throws java.io.IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
         System.out.println("Please choose the category from the menu:");
         System.out.println("1.Veg Pizza");
         System.out.println("2.NonVeg Pizza");
         System.out.println("3.Sandwich");

            int n = Integer.parseInt(br.readLine());
        if (n==1) {
            System.out.println("Please select the pizza");
            System.out.println("1.MARGHERITA");
            System.out.println("2.FARM HOUSE");
            System.out.println("3.PEPPY PANEER");
            System.out.println("4.MEXICAN GREEN WAVE");
            System.out.println("5.DELUXE VEGGIE");
            new VegPizza();
        } else if (n==2){
            System.out.println("Please select the pizza");
            System.out.println("1.PEPPER BARBECUE CHICKEN");
```

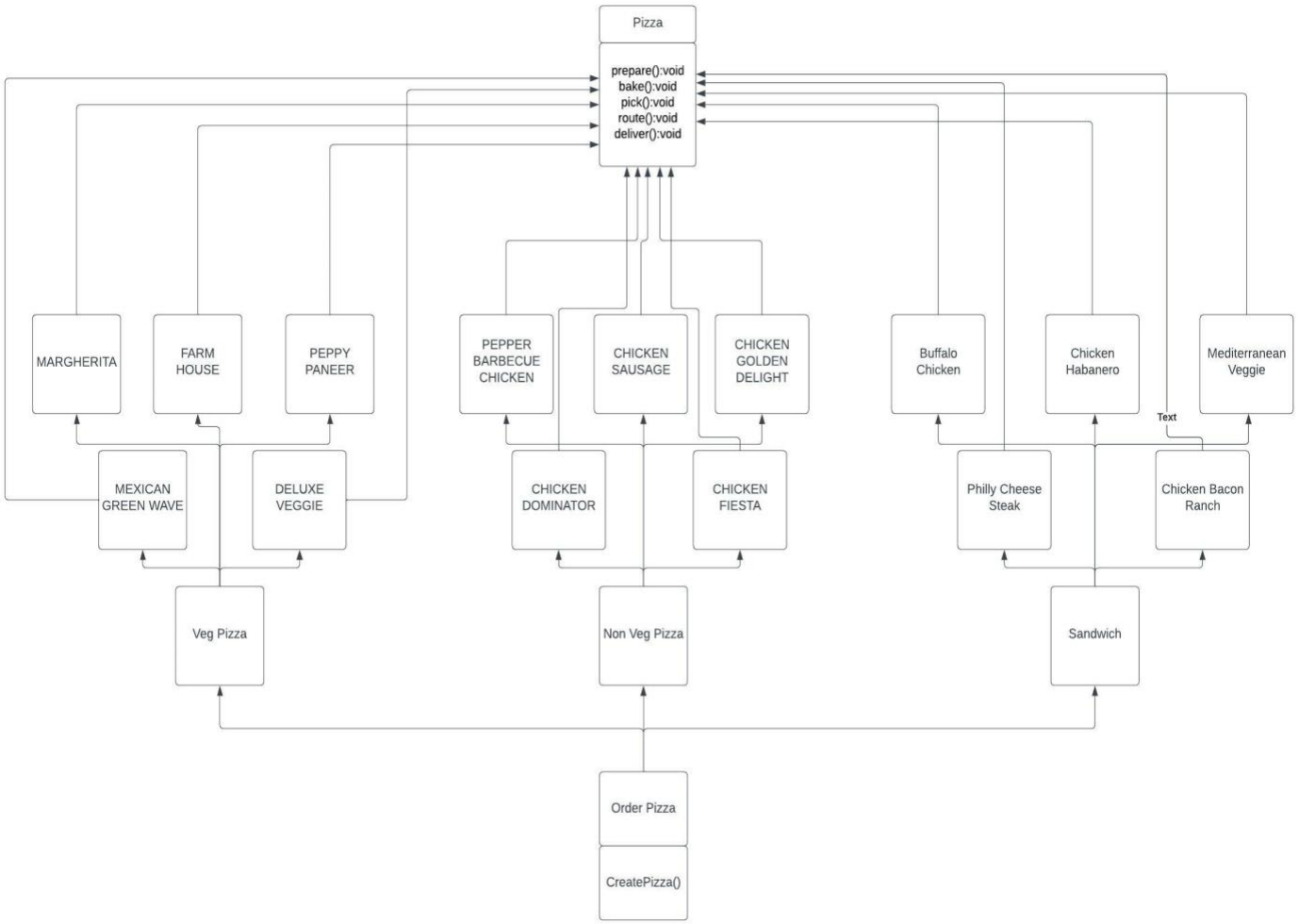
```
            System.out.println("2.CHICKEN SAUSAGE");
            System.out.println("3.CHICKEN GOLDEN DELIGHT");
            System.out.println("4.CHICKEN DOMINATOR");
            System.out.println("5.CHICKEN FIESTA");
            new NonVegPizza();
        }
        else if(n==3){
            System.out.println("Please select the sandwich");
            System.out.println("1.Buffalo Chicken");
            System.out.println("2.Chicken Habanero");
            System.out.println("3.Mediterranean Veggie");
            System.out.println("4.Philly Cheese Steak");
            System.out.println("5.Chicken Bacon Ranch");
            new Sandwich();
        }
    }
}
```

Analysis: This is the main class, and it is divided into three parts: vegetarian, non-vegetarian, and sandwich. From these three choices, the user may select an option.

Output

```
Please choose the category from the menu:
1.Veg Pizza
2.NonVeg Pizza
3.Sandwich
2
Please select the pizza
1.PEPPER BARBECUE CHICKEN
2.CHICKEN SAUSAGE
3.CHICKEN GOLDEN DELIGHT
4.CHICKEN DOMINATOR
5.CHICKEN FIESTA
Please enter your choice:
1
PEPPER BARBECUE CHICKEN preparing raw materials
PEPPER BARBECUE CHICKEN Pizza is being prepared
Your order has been picked up.
Your order is en route
Your order is being delivered
Do you want order again
1.Yes
2.No
Please enter your choice:
2
Thank you for your order
```

Class Diagram



Sequence Diagram