Pete's Pizza Parlor

By: Shayak Khatua

Course #: CSCI 3353

Course Title: Object-Oriented Design
Section: 01

Instructor: Professor Naomi Bolotin

Table of Contents

Project Description

Pete's Pizza Parlor is a fun simulation of an ordering and dining experience at a pizzeria. The program greets the user and begins walking them through the process of ordering a custom-made pizza. The user can choose different pizza styles (New York, Extra Thin, and Deep Dish), crust types (normal or gluten-free), and cheese types (Mozzarella, Provolone, and Parmesan). The user can then select unlimited toppings (Mushrooms, Onions, Jalapeño Peppers, Pepperoni, Sausage) for additional charges. The user then has the choice to eat the pizza in the restaurant or eat it later. They also have the choice to pay for the pizza or run away. Depending on their decisions, the outcome of the simulation changes.

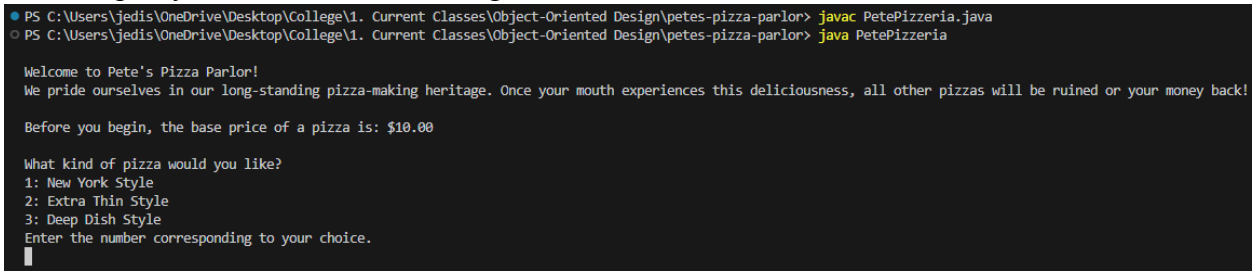
Please see Justification of Design Patterns to learn about the program's design.

How to Run

To run the program, perform the following steps:

1. Unzip the **Petes-Pizza-Parlor** folder.
2. Open your terminal.
3. Navigate to the **Petes-Pizza-Parlor** folder.
4. Run the command **javac PetePizzeria.java** (this is the driver program).
5. Run the command **java PetePizzeria**.
6. Follow the on-screen prompts until the completion of the program.

After Step 5, you should see something like this:



Justification of Design Patterns

Creational: Builder

The builder design pattern was chosen to create the initial base pizza. As stated before, this involves selecting the style of pizza, type of crust, and type of cheese. While these are all basic steps in a pizza recipe, there is already a great amount of variation in the possible base pizzas before toppings. So, a design pattern is needed that allows for some degree of shared properties as well as some degree of variability. Furthermore, when creating a pizza, you have to follow a strict order. The style of pizza influences the thickness of the crust, so it must be chosen first. The cheese has to go on top and at the end of the process (for the base pizza).

Therefore, the builder design pattern satisfies these requirements as it allows for the creation of any kind of pizza, but in incremental steps. It separates the construction of a pizza with the intermediate builder object from its representation, allowing for customizability at run-time.

Structural: Decorator

The decorator design pattern was chosen to add toppings to the base pizza. In this program, it was intended for customers to be able to add as many toppings as they want. They can even add the same topping again if they desire more. Furthermore, while there are only five toppings in this program, it is reasonable to design a program that allows for more toppings to be added to the menu in the future, especially since pizza is a highly customizable food.

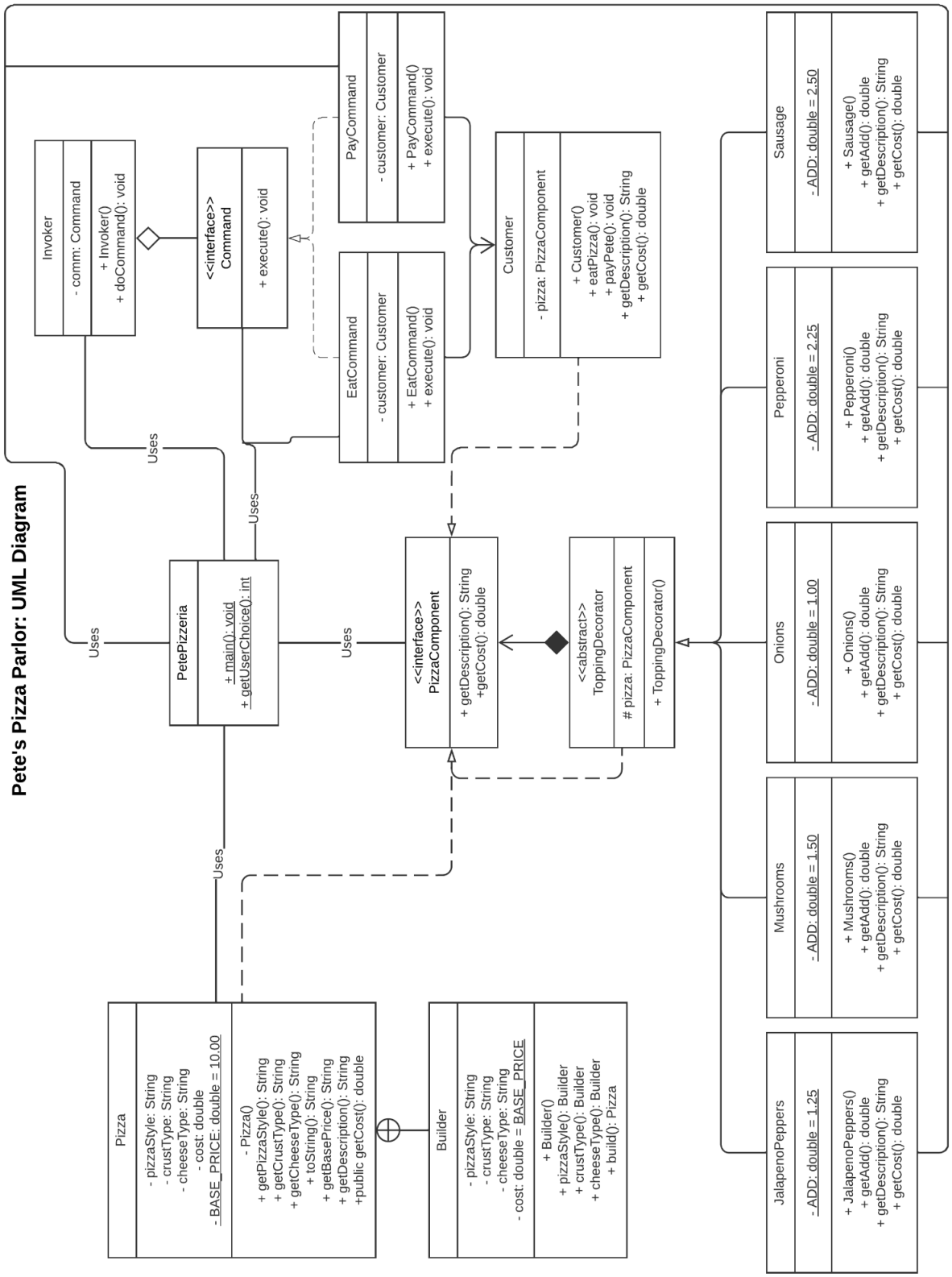
Therefore, the decorator design pattern satisfies these requirements as it allows for unlimited combinations of toppings with very little code. It is not at all necessary to make a new class for every possible combination of toppings, which would be extremely inefficient. Instead, relatively few classes can be used to create “infinite” combinations. Thus, the decorator design pattern is a very streamlined and efficient choice for pizza toppings.

Behavioral: Command

The command design pattern was chosen to allow the user to eat and pay for their pizza. Eating and paying for pizzas are both actions, but these actions do not have that much to do with pizza itself. A pizza cannot eat or pay for itself. If Pete’s Pizza Parlor were to expand its menu, then eat and pay would apply to more than just pizza. Thus, the object is not so much the focus as the action itself.

Therefore, the command design pattern satisfies these requirements as it allows for the focus to be on actions that use objects as instance variables (receivers). The eat and pay commands receive pizza in this project, but could receive other food items in the future. Thus, command offers greater adaptability and longevity to the code base.

UML Diagram


Pete's Pizza Parlor: UML Diagram
Invoker
- comm: Command
+ Invoker()
+ doCommand(): void
<<interface>>
Command
+ execute(): void
PayCommand
- customer: Customer
+ PayCommand()
+ execute(): void
EatCommand
- customer: Customer
+ EatCommand()
+ execute(): void
Customer
- pizza: PizzaComponent
+ Customer()
+ eatPizza(): void
+ payPete(): void
+ getDescription(): String
+ getCost(): double
Uses
PetePizzeria
+ main(): void
+ getUserChoice(): int
<<interface>>
PizzaComponent
+ getDescription(): String
+getCost(): double
<<abstract>>
ToppingDecorator
pizza: PizzaComponent
+ ToppingDecorator()
Pizza
- pizzaStyle: String
- crustType: String
- cheeseType: String
- cost: double
- BASE_PRICE: double = 10.00
- Pizza()
+ getPizzaStyle(): String
+ getCrustType(): String
+ getCheeseType(): String
+ toString(): String
+ getBasePrice(): String
+ getDescription(): String
+public getCost(): double
Builder
- pizzaStyle: String
- crustType: String
- cheeseType: String
- cost: double = BASE_PRICE
+ Builder()
+ pizzaStyle(): Builder
+ crustType(): Builder
+ cheeseType(): Builder
+ build(): Pizza
Sausage
- ADD: double = 2.50
+ Sausage()
+ getAdd(): double
+ getDescription(): String
+ getCost(): double
Pepperoni
- ADD: double = 2.25
+ Pepperoni()
+ getAdd(): double
+ getDescription(): String
+ getCost(): double
Onions
- ADD: double = 1.00
+ Onions()
+ getAdd(): double
+ getDescription(): String
+ getCost(): double
Mushrooms
- ADD: double = 1.50
+ Mushrooms()
+ getAdd(): double
+ getDescription(): String
+ getCost(): double
JalapenoPeppers
- ADD: double = 1.25
+ JalapenoPeppers()
+ getAdd(): double
+ getDescription(): String
+ getCost(): double