BIM492

DESIGN PATTERNS

PROJECT 2

Şevval Türkyılmaz	**49927389416**	**sevvaltrkylmzz**
Cansu Gürel	**16157401836**	**CansuG**

1. STATEMENT OF WORK: Write your problem definition in detail using 100-200 words.

We have a very well known dessert shop, which is known for it amazing cakes and cookies. The chefs are very experienced and creates different kinds of them. And of course people tend to come to this fancy store more often.

The problem we are trying to solve in this project is about an dessert shop that has so many orders that they can't quite catch up. Creating the desserts are not a very easy task when we have lots of orders and getting the orders from clients is alone also very time-consuming process. In this project, we are trying to help the desert shop manage the orders and take the orders more efficiently. The more hectic and well-known the business is, the more workload is possible. To overcome this workload and keep everything simple and efficient, we come up with this project.

2. DESIGN PATTERNS: Select at least two design patterns that are useful in solving the problem you defined. Explain why and how you employ such design patterns in your Project.

- WHY WE USED OUR DESIGN PATTERNS:

We choose Command and Factory design patterns.

The main focus of the command pattern which comes under behavioral design patterns is to inculcate a higher degree of loose coupling between involved classes. Classes that are interconnected, making the least use of each other defines loose coupling.

The reason we chose the command design pattern is that when requests need to be sent without consciously knowing what you are asking for or who the receiver is. In this pattern, the invoking class is decoupled from the class that actually performs an action. The invoker class only has the callable method execute, which runs the necessary command, when the client requests it.

Factory pattern is one of the most used design patterns in Java. This type of design pattern comes under creational pattern, as this pattern provides one of the best ways to create an object.

In the Factory pattern, we create an object without exposing the creation logic to the client and refer to the newly created object using a common interface. We wanted to separate the object creation from our code. When we use the new() operator in code, what happening is "coupling" the code to a very specific version of an object. The code (client) and the object are tied together. This is inflexible. If we,

in the future, need to use a newer, more capable version, such as a subclass with more functionality, we have to update all the new() statements in the codebase, which is time-consuming and may require a lengthy re-compile. With factory design pattern, we can simply update the factory to return the new version of our code. Not changing the entire code, but only modifying the factory itself. This allows loose-coupling.

- HOW WE IMPLEMENTED OUR DESIGN PATTERNS:

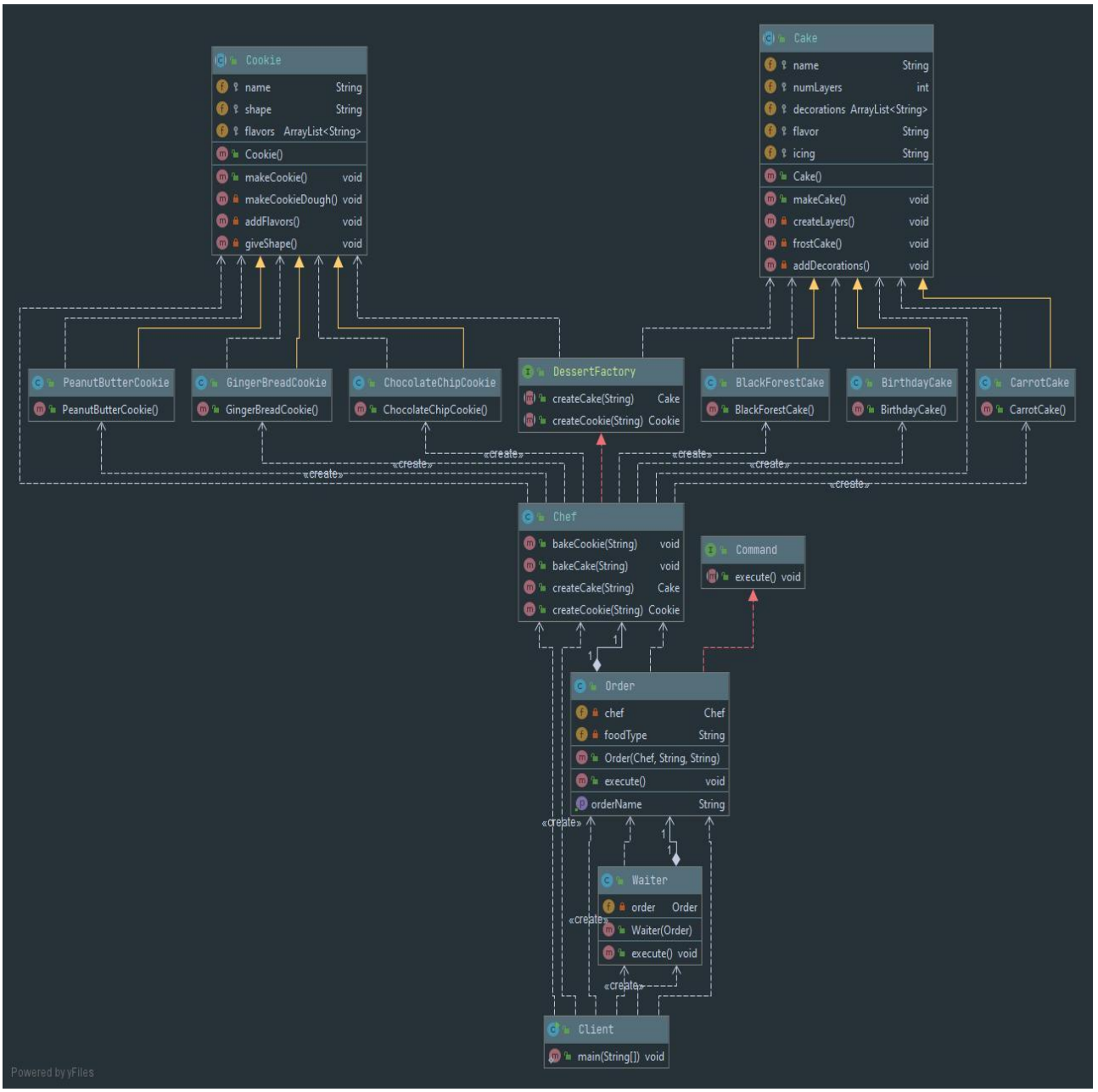
We have an interface for ‘Command’, a class for ‘Order’ that implements Command interface, a class ‘Waiter’ which is invoker and a class ‘Chef’ which is receiver.

We make a concrete command, which implements the Command interface, asking the receiver to complete an action, and send the command to the invoker. The invoker is the person that knows when to give this command. The chef is the only one who knows what to do when given the specific command/order. So, when the execute method of the invoker is run, it, in turn, causes the command objects’ execute method to run on the receiver, thus completing necessary actions.

The Client makes an Order and sets the Receiver as the Chef. The Order is sent to the Waiter, who will know when to execute the Order i.e. when to give the chef the order to cook. When the invoker is executed, the Orders’ execute method is run on the receiver i.e. the chef is given the command to either bake cookie or cake.

We created ‘DesertFactory’ interface that has two methods which are createCookie and createCookie. And our Factory class, ‘Chef’, implements our interface. In our chef class we create objects, with this way we separate object creation from the rest of our code. In Chef class we have got several concrete classes BlackForestCake, CarrotCakeCake, BirthdayCake, ChocolateChipCookie, PeanutButterCookie, GingerBreadCookie being instantiated, and the decision of which class to instantiate is made at run-time depending on the user’s choice.

3. UML: Draw a detailed class diagram of your proposed solution using some UML Diagram Drawing Tool.



4. RESEARCH:

- SOURCE CODE:

  The code is from a pizza application. Here is the code snippet that used the Singleton design pattern:

```
public enum PizzaDeliverySystemConfiguration {
   INSTANCE;
   PizzaDeliverySystemConfiguration() {
      // Initialization configuration which involves
      // overriding defaults like delivery strategy
   }

   private PizzaDeliveryStrategy deliveryStrategy =
PizzaDeliveryStrategy.NORMAL;

   public static PizzaDeliverySystemConfiguration getInstance() {
      return INSTANCE;
   }

   public PizzaDeliveryStrategy getDeliveryStrategy() {
      return deliveryStrategy;
   }
}
```

- EXPLANATION:

Enum was introduced in Java 5 and provides a thread-safe implementation. The objects returned by Enum are Singleton in nature and therefore can be effectively used for implementing the Singleton design pattern in the multi-threaded environment. Normally, implementing a class using the Singleton pattern is quite non-trivial. Enums provide a quick and easy way of implementing singletons.

In addition, since the enum class implements the *Serializable* interface under the hood, the class is guaranteed to be a singleton by the JVM. This is unlike the conventional implementation, where we have to ensure that no new instances are created during deserialization.