**CS F213**

**Object Oriented Programming**

“Analysis of the code for the project”

Project Number: 19

Project Title: Token Bucket Algorithm

Group Number: 147

Team Members:

Saksham Mahajan (2019B4A70627P)

Ritik Thakur (2019B2A70878P)

* **Design Patterns**

Based on Creational Design Pattern:

We have not used creational design pattern in our project since we have hard-coded the instance creation using the new keyword.

For example: IntermediateDevice tempDevice = new IntermediateDevice(tempSource);

Since the project provided to us was based on the Token Bucket Algorithm the classes could be hardcoded since in an algorithm a change other than the prescribed format is highly unlikely.

The problem with approach that we have followed is that the current code execution will not allow change in nature of object according to the requirement without the repetition of code and changing the existing code snippets, thus this compromises the flexibility of the program. Thus, we will face difficulties in extending the project e.g., if we require multiple types of intermediate devices and/or token buckets etc. in the future.

Based on basic design patterns:

1. **Strategy Pattern:**

This pattern is not used since there is no explicit inheritance or variation in instances that occurs throughout the program execution. Since the program executes only a single algorithm, thus no need to separate algorithm from the clients that use it, therefore this pattern is not used.

1. **Observer Pattern:**

This pattern is not used since we have no need of constantly subscribing to the service and change the output on notify since our problem requires only one time execution of the Token Bucket Algorithm. Since there is no one to many kind of dependency between objects this pattern is not used.

1. **Decorator Pattern:**

The program simply takes user input and processes it through a series of classes in specific order to give the output, thus there is no need to add additional responsibilities to any object or instance in between. Thus the Decorator pattern is not helpful.

* **Principles**

1. **Encapsulate what varies**

In our code all the class definitions and signatures remain same throughout the execution of the program and do not have any variation associated with them. Both the attributes and their types are known beforehand thus there is no variation that occurs. Even inheritance has not been used as such in the project other than for extending JFrame and Thread classes to create GUI and threads respectively. So there is no variation related to classes or objects as such and thus no runtime changes occur. As a result, this principle is not used.

Although in classes like IntermediateDevice class we have not separated attributes that remain fixed throughout the execution like capacity, memory, and rate from those like list, number of tokens etc. that vary dynamically. This can lead to problems like data leaks.

1. **Favor composition over inheritance**

In our code we have favored composition since our classes are based on composition and the function that it executes in the program. For example the ExternalDevice class is used only to create instances for sources while the CreateGUI class is used to create JFrame and execute the main logic of the program after receiving input through GUI, while the MainClass serves as the starting point of the program and creates an instance of the CreateGUI class. Similarly the IntermediateDevice class contains the logic related to the Token Bucket and the run method for the threads and the OutputScreen creates the output Java frame. While inheritance has not been used explicitly in our project other than standard one to create JFrame and Thread.

1. **Program to interfaces, not an implementation**

The point here is to exploit polymorphism by programming to a supertype without having to actually use java interface. But the program that we worked on had the same type of objects without any change or variation attached to them. The change that occurs in the ExternalDevice is just the randomization of the number of packets on each instance creation. Since this is a very minimal change in the larger picture of the program there is no need to program it to an interface and then create its reference to create changeable object. In the rest of the classes no change occurs in the object type throughout the program execution because the classes just interact with each other in a specific order to take attribute values and then process them internally to finally give the desired output.

In future if the project is to be extended and we need to add multiple types of Token Buckets or Intermediate device then the current execution will lead to repeated code blocks and confusion.

1. **Strive for loosely coupling between objects that interact**

This principle is violated since the classes used in the program are tightly coupled throughout the execution of the program. For example, the IntermediateDevice class has a method named transmit(ExternalDevice device) which uses the instance device of the type ExternalDevice, thus if we change the ExternalDevice class the functionality of IntermediateDevice class also changes. Similarly we have ArrayList sources in CreateGUI class which stores instances of the ExternalDevice class. Thus all the classes except the MainClass are interwoven in tight coupling.

This can lead to problems if multiple people are working on the same project and class signatures are updated quite frequently, but since we were only two members involved in the project hence this factor was not a major one.

1. **Classes should be open for extension but closed for modification**

Some classes that we have used like ExternalDevice class follow this principle. But the other classes like IntermediateDevice class and CreateGUI class can be modified easily as they contain default variables and in fact are modified throughout the execution of the code. Even if we have to change the capacity, memory or the rate of replenishment of tokens all these changes have to be made in the ExternalDevice class and thus it is not closed for modifications as we need to modify the existing code in order to take them into account.

In future if the project is to be extended and we need to add multiple types of Token Buckets or Intermediate devices then the current execution will lead to repeated code blocks and tiresome process of changing the already tested classes. Thus we would have to test the program thoroughly.

1. **Depend on abstraction, do not depend on concrete classes**

As it has been mentioned throughout this analysis that the classes created in this project are quite concrete and the concept of abstraction has not been used since the project is based on an algorithm so there is not much scope of creating variations in the object designs. Especially since we just take input values and show output values based on processing of the input that occurs in classes internally in a specified order.

This can create problems if we need to introduce any type of change or variation in the instances or extend the domain of the algorithm.