Video: **https://drive.google.com/file/d/1VDN0WfH7F4QaXdPUrUP94eIZRPyN0s0v/view?usp=sharing**

Drive Link:

https://drive.google.com/drive/folders/1M4zv71\_pVepbUPHDSQcSacvDxD7nSS-8?usp=sharing

Document Evaluation

Class Description:

There are 3 classes namely **Bucket, AddTokenThread** and **SendPacketThread**. The Bucket class acts like the device on which the other two class will work on to implement the algorithm.

**Bucket** class has 2 class variables namely *maxsize* and *tokens*. It also has 2 methods *AddToken* and *SendPacket* . The *AddToken* method adds 1 to the *tokens* variable present in the class. *SendPacket* is the method that deducts *tokens* variable. If the number of packets(n) > number of tokens(t) then all the tokens are consumed and the method returns (n-t) and t=0.Whereas if(t>n) then t=t-n and the method returns 0.

**AddTokenThread** is the child class of Thread class. It takes an object of Bucket class as input. The *run* method of this class adds tokens to the bucket at a constant rate using *AddToken* method (of Bucket class) and *sleep* method (of Thread class).

**SendPacketThread** is also child class of Thread class. It also takes an object of Bucket class as input. In its *run* method, the Thread is put to sleep initially for a random amount of time(using Math.random() function) so that bucket can fill up a bit. Then using the Math.random() method, the number of packets is generated using the local variable *j* according to the given formula. A local variable *i* is also declared which takes the return value of b.SendPacket(j)(b is object of Bucket class which was taken as input by the constructor. if(*i*==0) then nothing happens sendPacket just completes its job and then cycle starts again. But if (*i*!=0) then the thread is again put to sleep so that the bucket can fill up for the packets to be cleared before the infinite loop can begin again).

Known limitations/bugs:

The limitations/bugs we could identify are:

**1)**In the SendPacketThread class, when the thread is sent to sleep for the second time because the number of tokens are less than the number of packets. By multiplying time b/w two addition of tokens and no of token required. But this seems to be true only if the integer returned by the SendPacket<10. If it is >10 then we have to wait for time obtained by multiplying time between two addition of tokens and no of token required.

**2)**During the time the program is running sometimes a bug appears which causes some problem in the SendPacketThread while the thread is sent to sleep for the second time because the number of tokens are less than the number of packets. When the sleep time finishes mathematically the number of tokens in bucket should be equal to the number of packets held but sometimes(Yes sometimes only) it ends up one token lesser or one token greater than the desired amount. We guessed that it is some problem with multithreading(nondeterminism which is causing this)

OOP Strategies Implementation

**Favour Composition over inheritance:**

It is one of the strategy which has been used while coding for the project. As can be seen in the code the two classes have used an object of the Bucket class rather than inheriting it. Doing this certainly helped in reducing the amount of complication that might have arisen if for instance AddTokenThread had inherited Bucket class with the same class doing everything in the code. In this case while considering the code the object of the bucket class has been simply called whenever one of its methods was required thereby simplifying the code with Bucket class having methods which other class which have the objects of the buckets class can use as and when required.

**Strive for Loose Coupling:**

It is also one of the strategies used in the project. Classes in the code are loosely coupled since not one of them has any parent child relationship within classes. The loosely the 2 classes(AddTokenThread and AddPacketThread) are interconnected to Bucket class is because of object of bucket is a class variable in the 2 classes but other than that they would largely be unaffected if some modifications are done in future thus reducing effort required in maintaining the code.

**Depend on abstraction don’t depend on concrete classes:**

Its not been implemented in this code. Concrete classes are only used in this code. If the principle was followed interfaces would have been used more as a result code reusability would have increased. Since the code is rather short it did’nt cause any inefficiency but had the code been long it would have been very redundant disobeying the DRY principle.

**Classes should be open for extension closed for modification:**

Its not been implemented in the code. Since there was no parent-class child-class in the code it did’nt impact the efficiency of code but at the same time there is no denying that in case of long codes this rule must be strictly followed otherwise the slightest of changes in the classes leads to plenty of tedious corrections needed to be done

Design Pattern

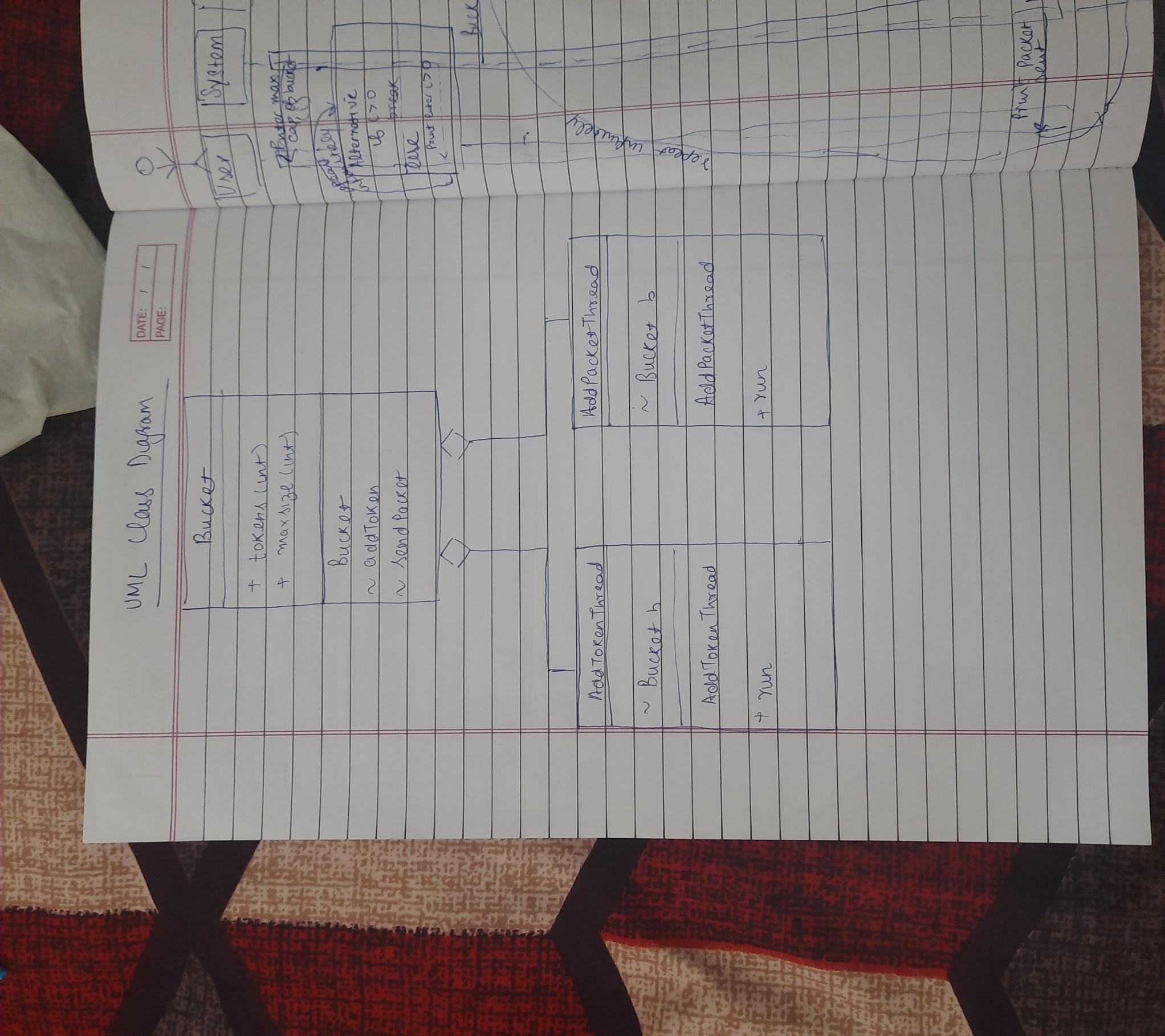
Strategy Design:

This code does’nt implement Strategy design pattern. It can be implemented by creating an interface for Bucket class which will serve as abstraction of the context class (Bucket).This interface will be the strategy

Let the interface name Bi its methods will be AddToken and SendPacket and its variables are int tokens and int maxsize;

**package** com.company;  
**import** java.util.ArrayList;  
**import** java.util.Queue;  
**import** java.util.Scanner;  
  
**interface** BI{  
 **void** addToken();  
 **int** sendPacket(**int** n);  
}

**class** Bucket{  
 **public int tokens**, **maxsize**;  
 Bucket(**int** max){  
 **tokens** = 0;  
 **maxsize** = max;  
 }  
  *public* **void** addToken() {  
 **if** (**tokens** >= **maxsize**)  
 **return**;  
 **else** {  
 **tokens** +=1;  
 System.***out***.println(**"Added a token. Total:"** + **tokens**);  
 }  
 } *public* **int** sendPacket(**int** n){  
 System.***out***.println(**"No. of packet "** + n);  
 **if**(n > **tokens**){System.***out***.println(**tokens**+**"packets spent "**+(n-**tokens**)+**" packets held"**);  
 **int** s= n-**tokens**;  
 **tokens**=0;  
 **return** s;  
 }  
 **else**{  
 **tokens** -= n;  
 System.***out***.println(**"Packet sent and the required tokens consumed"**);  
 **return** 0;  
 }  
 }  
}

UML Diagrams

UML Sequence Diagram