CSE 443
Object Oriented Analysis and Design

Homework - 3

Muharrem Ozan Yeşiller
171044033

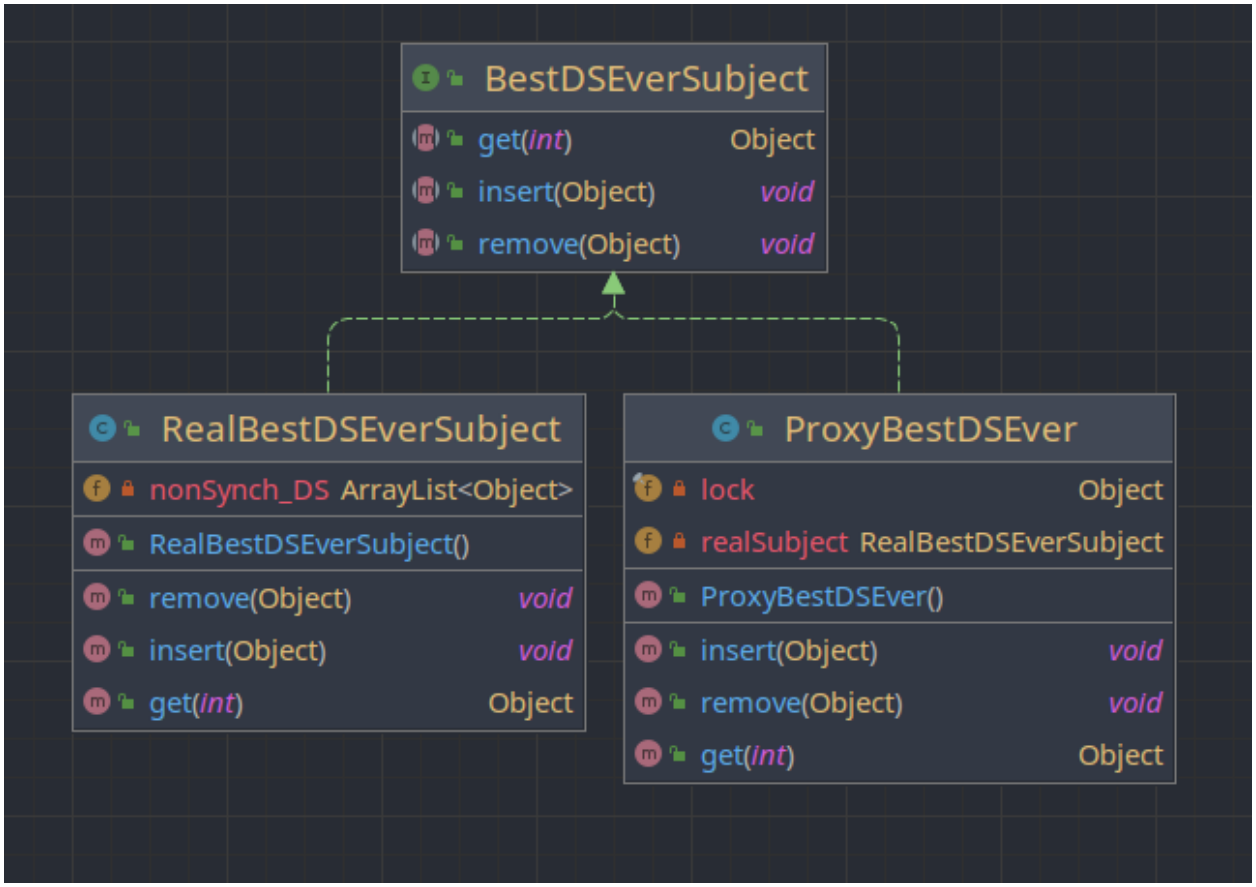
Design Patterns and Class Diagrams

Part 1, Proxy Design Pattern

A proxy class allows us to represent another class. In this way, the Proxy class can perform all the operations or operations it wants without interfering with the main class. In my design, there is a Subject interface, and there is a Real Subject class that implements this interface. BestDSEver, on the other hand, keeps an arraylist in it and uses the arraylist methods when implementing this interface. This class is the BestDSEver class. While implementing this design pattern, I implemented the ProxyBestDSEver class by implementing the Subject interface. In it, it encapsulates the BestDSEver functions without interfering with the BestDSEver class. In addition, thread safety is provided with a mutex object. There is one final object in the class. This object acts as a mutex in the interventions of different threads for an object produced from this class.

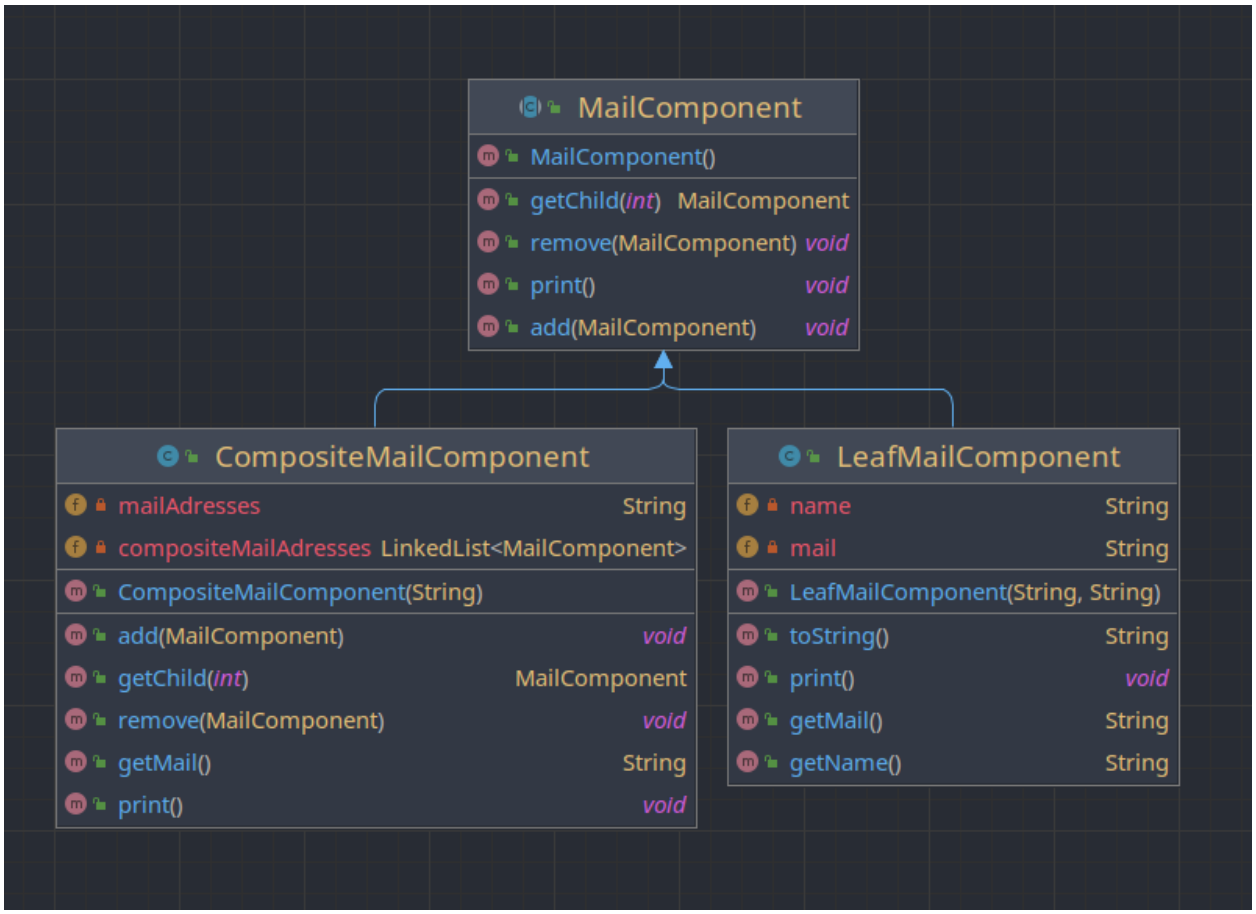
```
private RealBestDSEverSubject realSubject;
private final Object lock = new Object();
    ...
    ...
 @Override
 public void insert(Object o) {
     synchronized (lock) {
         System.out.println("Insert Start");
         this.realSubject.insert(o);
         System.out.println("Insert finish");
     }
 } // Example
```

Class Diagram



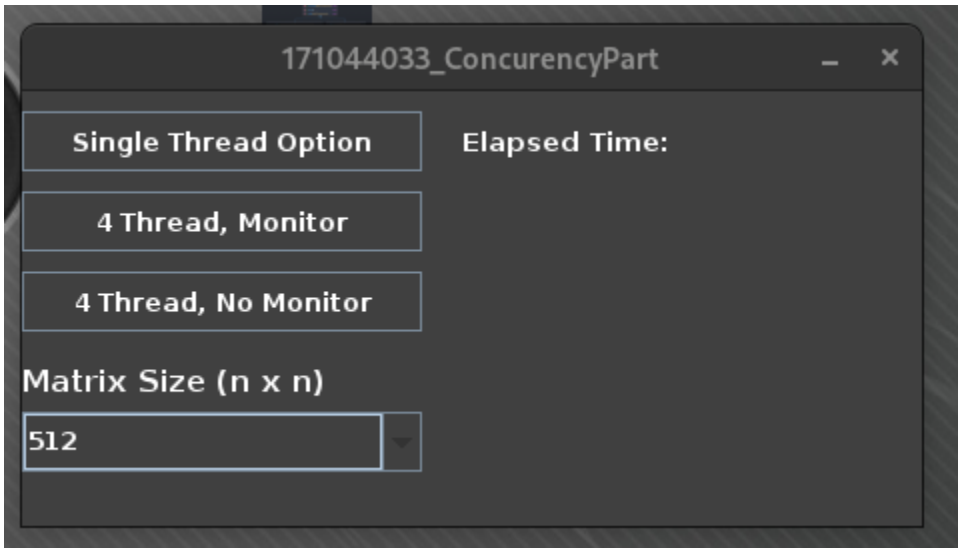
Part 2, Composite and Iterator Design Pattern

A composite design pattern allows a group of individual components and different components to act similarly in a hierarchical structure, that is, a group of objects that differ from each other in themselves, to be used as if they were a single whole object. The task of compound patterns is to combine objects into a tree structure, rearrange and shape the whole relationship of parts throughout the application.



Part 3, Concurrency Design Pattern

In my design in this section, there are inputs on a JPanel. Single thread, Multiple threads with or without mutex buttons. It takes a size parameter as an extra. We were also asked to use an 8192 x 8192 matrix in PDF, but I kept this part flexible because I got an out of memory exception on my computer.



Single Thread Button: With a single thread, 2 square matrices of the given size are generated, they are added up and the DFT operation is applied.

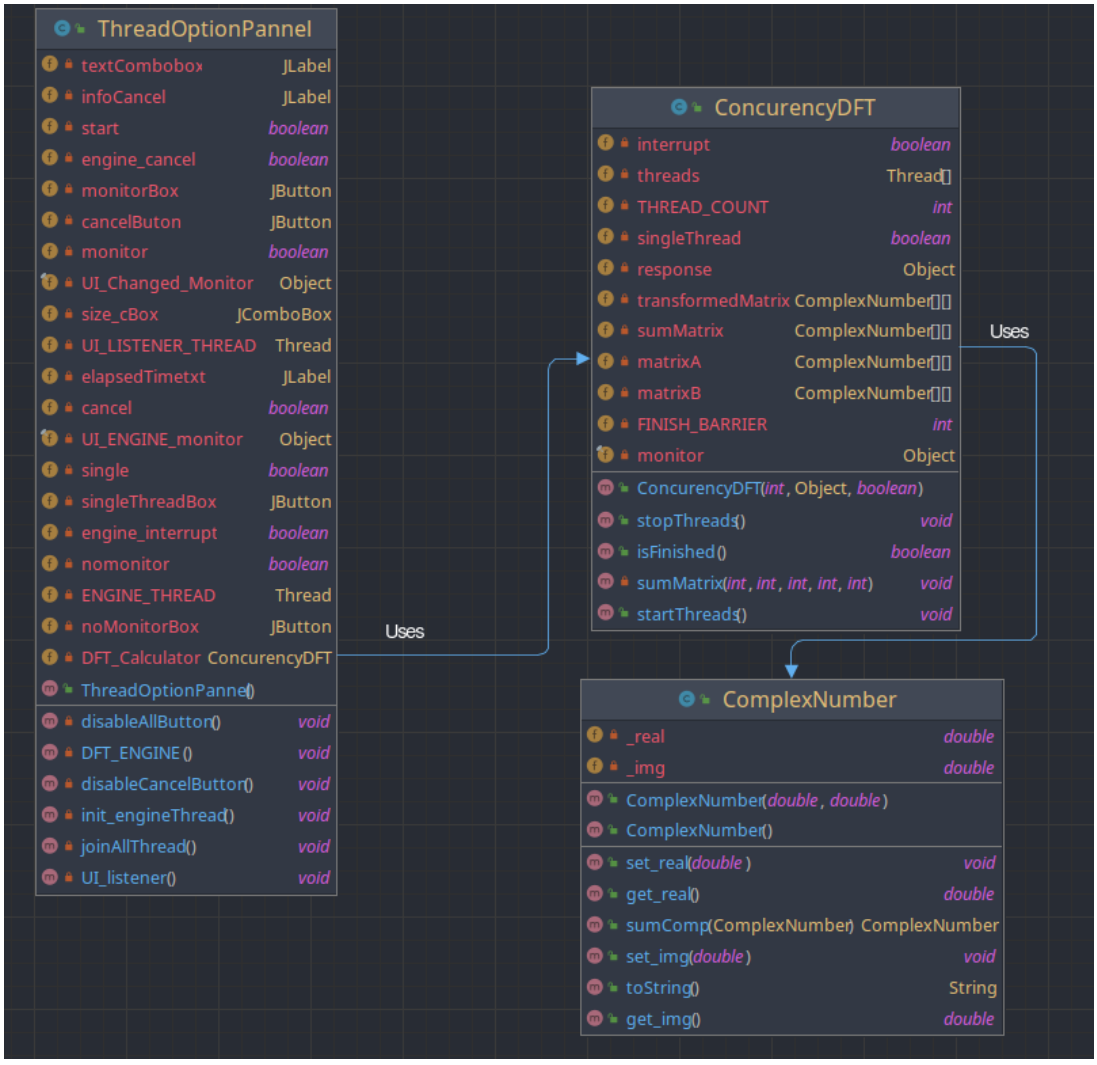
4 Threads, Monitor Button: 4 threads these square matrices are processed simultaneously with 4 separate parts: 4 Threads. It should be noted here that when collecting 2 matrices, the thread that finishes its work waits for all threads to finish their work. This is called the threshold timing barrier.

```
synchronized (monitor) {
            --THREAD_COUNT;
            if (THREAD_COUNT != 0)
                monitor.wait();
            else
                monitor.notifyAll();
}
```

A ConcurencyDFT class has been implemented that is responsible for these operations, and the monitor here is a final object. Because this class starts many threads, the thread that receives the mutex of this monitor enters the block and reduces the counter, if it is not the last thread, it leaves the mutexi on hold. If it is the last thread, it leaves the mutexi and wakes up every pending thread.

In the graphical interface, 2 different threads were used. One of these threads is the thread that is waiting for notifications from listeners from the user interface. The other thread is the one that receives a notification from the first thread and starts the calculation processes.

Class Diagram



The co-scheduling between these two threads is provided as follows dec:

First, the thread that listens to the UI switches to standby. The other thread creates the object that will manage the necessary operations for the DFT account and goes on hold.

This thread waits for notifications from button listeners while waiting.

```
monitorBox.addActionListener(t -> {
            this.start = true;
            this.monitor = true;
            this.single = false;
            this.nomonitor = false;

            synchronized (UI_Changed_Monitor) {
                UI_Changed_Monitor.notify();
            }
        });









private void UI_listener() {
        while (true) {
            synchronized (UI_Changed_Monitor) {
                try {
                    UI_Changed_Monitor.wait();
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
```

If it receives a notification, it updates the required values. The second thread sends a notification and goes into standby.

The second thread receives the notification and runs the start method of the class that makes the DFT account to start the processes.

```
if (monitor || nomonitor) {
                DFT_Calculator = new ConcurencyDFT(size,
UI_ENGINE_monitor, false);
            } else if (single) {
                DFT_Calculator = new ConcurencyDFT(size,
UI_ENGINE_monitor, true);
            }

            DFT_Calculator.startThreads();

            while (true) {

                synchronized (UI_ENGINE_monitor) {
                    if (!engine_interrupt)
                        UI_ENGINE_monitor.wait();
                }


            if (engine_interrupt) {
                    DFT_Calculator.stopThreads();
                    engine_interrupt = false;
                    engine_cancel = true;
                    break;
                }

                if (DFT_Calculator.isFinished()) {
                    engine_interrupt = false;
                    engine_cancel = true;
```

```
                synchronized (UI_Changed_Monitor) {
                    UI_Changed_Monitor.notify();
                }
                break;
            }
        }
```