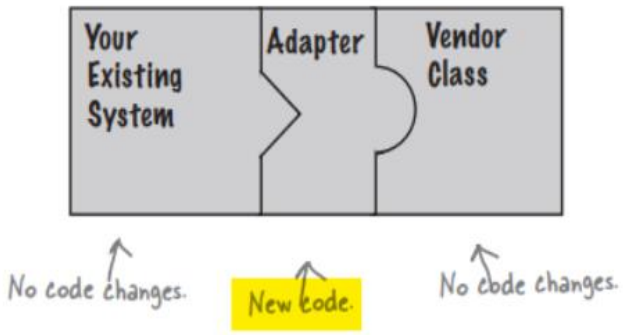
GTU Department of Computer Engineering
CSE443 Object Oriented Analysis and Design
Fall 2021 - Homework 3 Report

Akif KARTAL
171044098

Question 1

Use Adapter Design pattern. Why?



The Adapter Pattern converts the interface of a class into another interface the clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

As you can see adapter design pattern is appropriate for this problem and it is simple compared to decorator and proxy patterns which they can be used for this problem.

Solution

1.1 Implement simple BestDSEver Class

```
4   public class BestDSEver {
5       private ArrayList<Object> arrayList;
6
7       public BestDSEver() {
8           arrayList = new ArrayList<>();
9       }
10
11      public void insert(Object o) {
12          arrayList.add(o);
13      }
14
15      public void remove(Object o) {
16          arrayList.remove(o);
17      }
18
19      public Object get(int index) {
20          return arrayList.get(index);
21      }
22  }
```

1.2 Test BestDSEver Class

```
3       public static void main(String[] args) {
4           BestDSEver ever = new BestDSEver();
5           Integer a = 5;
6           Integer b = 10;
7           ever.insert(a);
8           ever.insert(b);
9           ever.remove(b);
10          System.out.println("Result: " + ever.get(0));
11      }
12  }
```

Output:

```
Result: 5

Process finished with exit code 0
```

1.3 Implement Adapter Design Pattern

1.3.1 Understanding synchronization problem

Since each thread is trying to use same data structure, we have critical sections. We will solve this problem by applying following solution.

```
Thread t1          Thread t2
lock(m)            lock(m)
push(v)            v = pop()      // critical section
unlock(m)          unlock(m)
```

*You can think push and pop methods as insert and remove methods.

1.3.2 Creating Adapter Class

I will apply the **Class Adapter** solution for this problem.

```java
public class BestDSEverAdapter extends BestDSEver {
    private ReentrantLock mutex;

    public BestDSEverAdapter() {
        super(); // create arraylist
        mutex = new ReentrantLock();
    }

    @Override
    public void insert(Object o) {
        mutex.lock(); //lock(m)
        try {
            super.insert(o);
        } finally {
            mutex.unlock(); //unlock(m)
        }
    }

    @Override
    public void remove(Object o) {
        mutex.lock(); //lock(m)
        try {
            super.remove(o);
        } finally {
            mutex.unlock(); //unlock(m)
        }
    }

    @Override
    public Object get(int index) {
        mutex.lock(); //lock(m)
        try {
            return super.get(index);
        } finally {
            mutex.unlock(); //unlock(m)
        }
    }
}
```

*Here, we have a mutex for threads and we are using it in critical regions. Also we don't need a composition here because we are already using super class methods. Here we are making **unlock in finaly block** because if we **get an exception in try block,** still we should be able to unlock otherwise we will leave without unlocking.

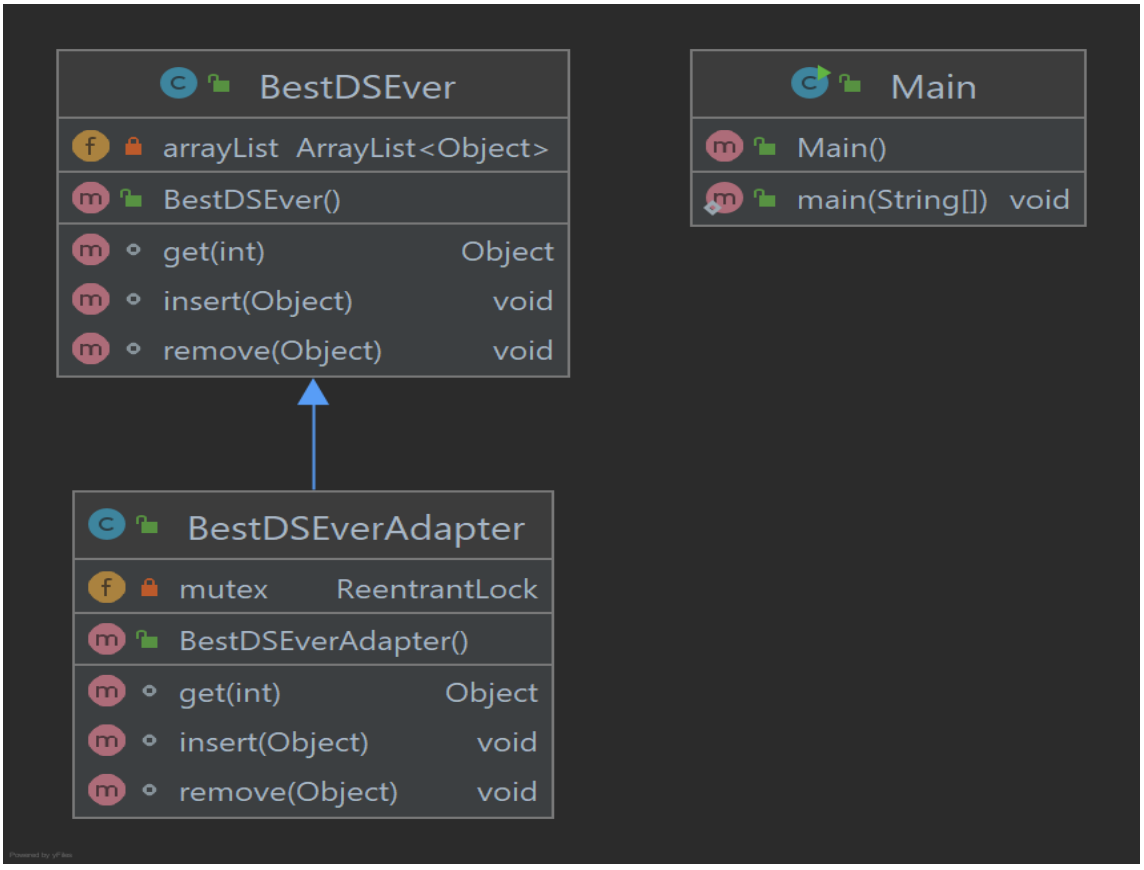
1.4 Testing with multiple Threads

```
public class Main {

    public static void main(String[] args) {
        BestDSEver buffer = new BestDSEverAdapter();
        Integer[] arr = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
        buffer.insert(arr[0]);
        buffer.insert(arr[1]);
        buffer.insert(arr[2]);
        buffer.insert(arr[3]);
        Thread thread0 = new Thread() {
            public void run() {
                System.out.println("Thread0 -> get(0): " + buffer.get(0));
                System.out.println("Thread0 -> insert(Integer(4))");
                buffer.insert(arr[4]);
                System.out.println("Thread0 -> remove(Integer(4))");
                buffer.remove(arr[4]);

            }
        };
        Thread thread1 = new Thread() {
            public void run() {
                System.out.println("Thread1 -> insert(Integer(8))");
                buffer.insert(arr[8]);
                System.out.println("Thread1 -> remove(Integer(8))");
                buffer.remove(arr[8]);
                System.out.println("Thread1 -> get(1): " + buffer.get(1));
            }
        };
        Thread thread2 = new Thread() {
            public void run() {
                System.out.println("Thread2 -> remove(Integer(0))");
                buffer.remove(arr[0]);
                System.out.println("Thread2 -> get(0): " + buffer.get(0));
                System.out.println("Thread2 -> insert(Integer(5))");
                buffer.insert(arr[5]);
            }
        };
        thread0.start();
        thread1.start();
        thread2.start();
        System.out.println("Main Thread -> get(1): " + buffer.get(1));
        System.out.println("Main Thread -> insert(Integer(7))");
        buffer.insert(arr[7]);
        try {
            /*Make sure all threads have finished.*/
            thread0.join();
            thread1.join();
            thread2.join();
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.out.println("Good Bye...");
    }
}
```

1.5 Output

```
Main Thread -> get(1): 1
Main Thread -> insert(Integer(7))
Thread1 -> insert(Integer(8))
Thread1 -> remove(Integer(8))
Thread2 -> remove(Integer(0))
Thread1 -> get(1): 1
Thread0 -> get(0): 0
Thread0 -> insert(Integer(4))
Thread0 -> remove(Integer(4))
Thread2 -> get(0): 1
Thread2 -> insert(Integer(5))
Good Bye...

Process finished with exit code 0
```

1.6 Class Diagram



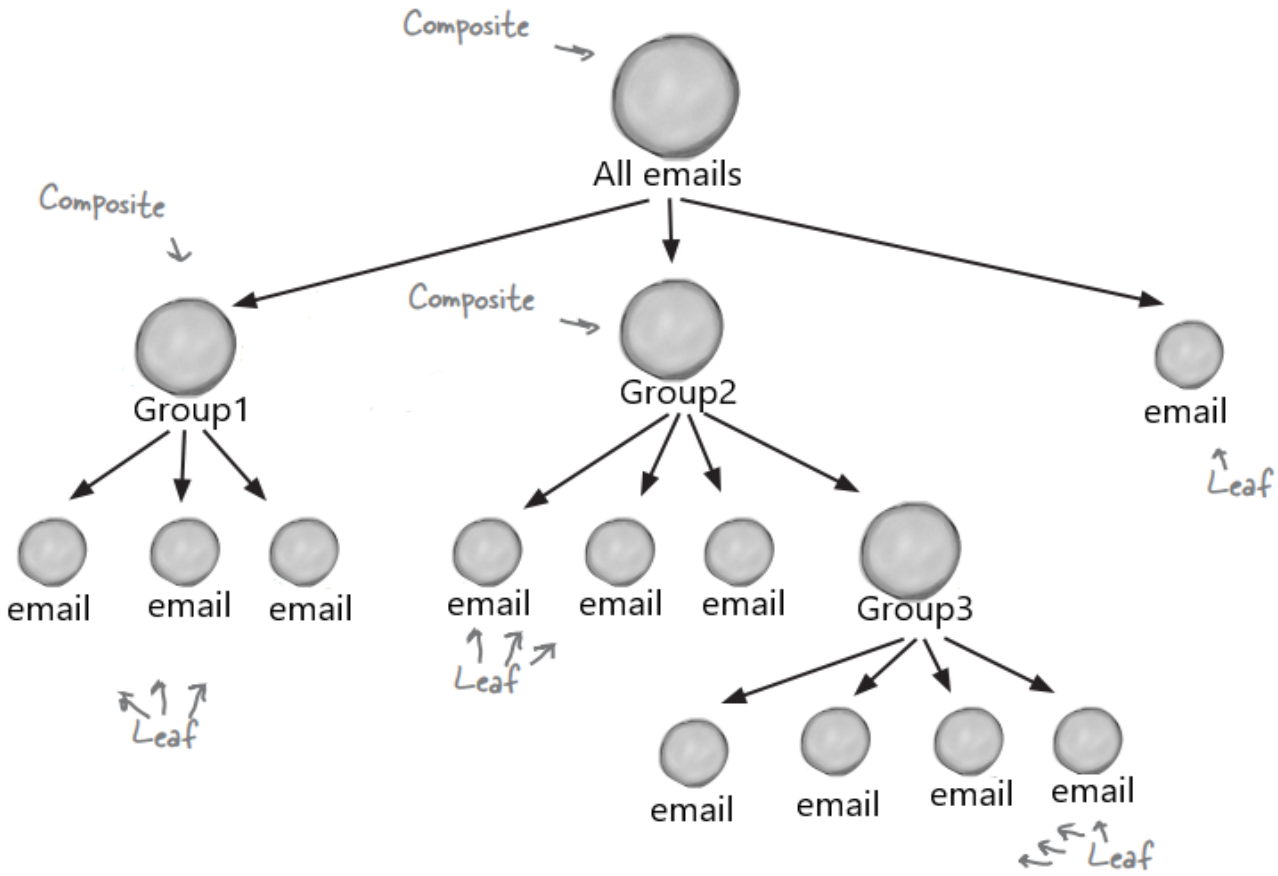
*Here, we are using inheritance(class adapter) because we don't have an interface for BestDSEver class. Also we don't have Adaptee class which means BestDSEverAdapter class is already an Adaptee therefore, we don't use composition in this solution.

Question 2 – Composite and Iterator

Solution

2.1 Understanding the problem

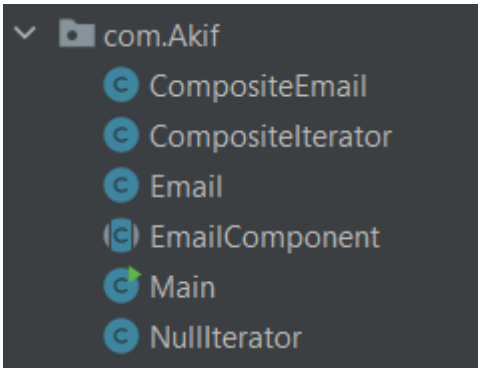
The problem is like the following picture;



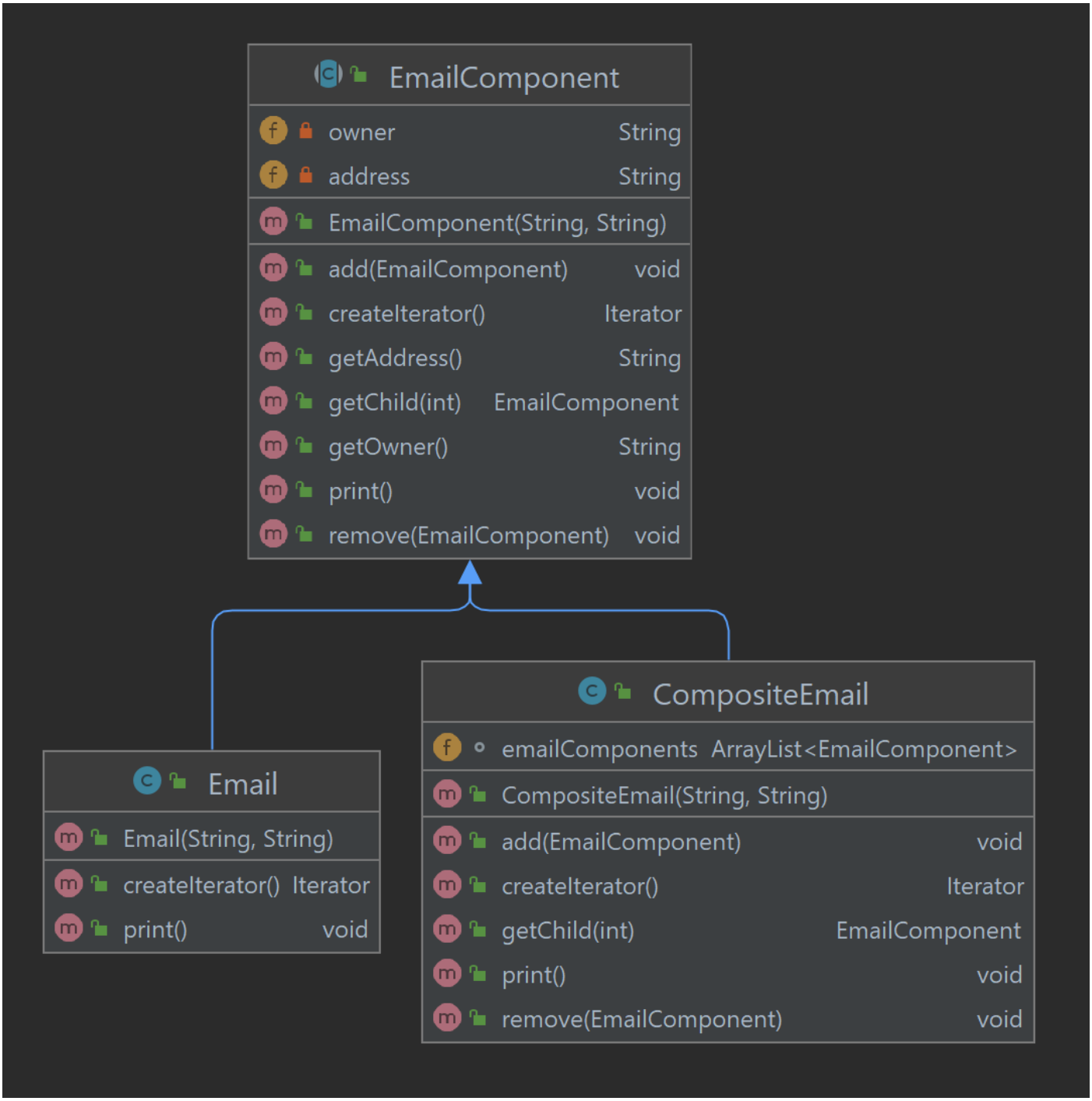
Here, emails contains both address and name of its owner, groups contains an arbitrary number of personal or group addresses also **groups are composite, emails are leaf**.

2.2 Class Diagram

2.2.1 Classes in my solution



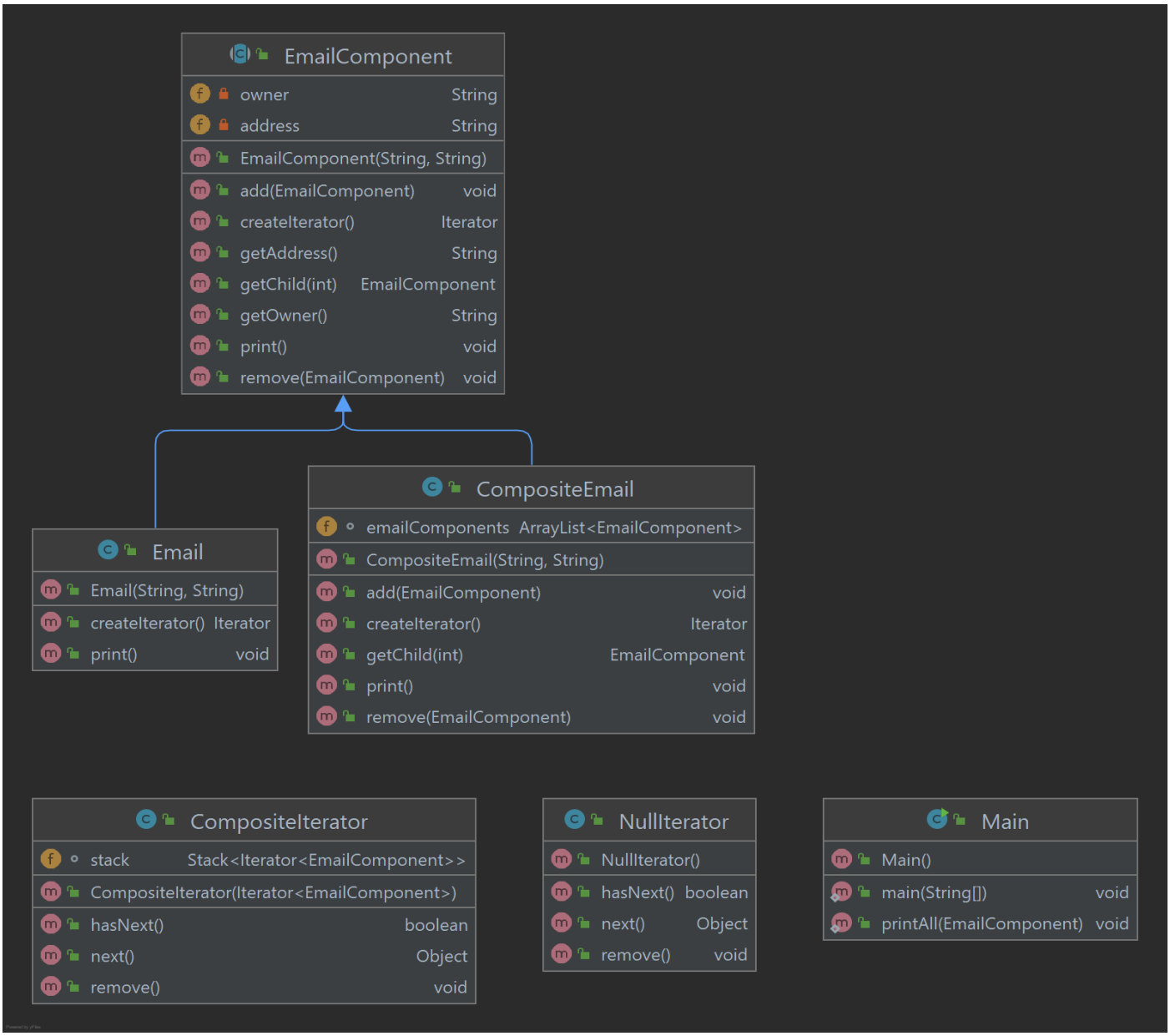
2.2.2 Composite Pattern Class Diagram



Here, EmailComponent class is abstract class, Email is leaf class and CompositeEmail is a composite class with add, remove and get methods.

2.2.3 Full Class Diagram

Here, we will see full diagram note that I used iterator design pattern to traverse all tree easily as expected in homework pdf file.



Here, Composite iterator implements **java.util iterator** interface to use in composite email class. NullIterator class is used in Email leaf class. Main class is for test purpose.

2.3 Test Results

Check Main.java class and run to see results. Some part of result is following;

```
Owner: GTU Ceng All
--------------------------------------------------
akif.kartal2017@gtu.edu.tr Akif Kartal
djuro2017@gtu.edu.tr Djuro RADUSINOVIC
mustafa.tokgoz2017@gtu.edu.tr Mustafa TOKGÖZ
m.karakaya2018@gtu.edu.tr Muhammed Emin KARAKAYA
m.kurtcebe2018@gtu.edu.tr Mehdi KURTCEBE
sinan.sari2016@gtu.edu.tr Sinan SARI
hboubati@gtu.edu.tr Alp Eser
--------------------------------------------------
```

Question 3 – Concurrency patterns

Solution

3.1 Understanding the synchronization barrier problem

In order to solve this problem, we will apply following solution in 2 different ways with java;

Example: **synchronization barrier** with N threads.

Condition variable `c`, mutex `m`, `arrived = 0`

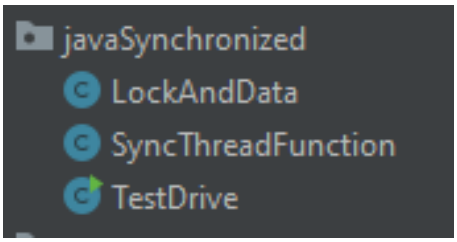
```
lock(m)
++arrived
if(arrived < N)    // if this thread is not last
   cwait(c,m)      // then wait for others
else
   broadcast(c)    // i'm last, awaken the other N-1
unlock(m)
```

Some notes on my solution

- In order **to create thread** in java I will use **Runnable interface** for both solutions.
- Note that, since all threads works with **different portion of the matrix** which means different buffer, we don't have any synchronization problem other than synchronization barrier.
- In my solution, all threads will use **same thread function** with different coordinates.
- I will explain my solutions with simple example. You can check the source code after reading.

3.2 Using Java's synchronized

3.2.1 Classes in my solution



3.2.2 Shared Data and Lock Object Class

```
7   /***
8    * This class is used for both shared data and
9    * as a lock object for synchronized.
10   */
11  public class LockAndData {
12      private ComplexNumber[][] matrixA;
13      private ComplexNumber[][] matrixB;
14      private ComplexNumber[][] matrixSum;
15      private AtomicInteger arrived;
16
17      public LockAndData(AtomicInteger arrived) {
18          this.arrived = arrived;
19      }
20
```

*Atomic integer is much better between threads in java.

3.2.3 Common Thread Function between Threads

```
7   public class SyncThreadFunction implements Runnable{
8       private final LockAndData lockData;
9       private Coordinates coordinates;
10
11      public SyncThreadFunction(LockAndData lockData, Coordinates coordinates) {
12          this.lockData = lockData;
13          this.coordinates = coordinates;
14      }
15
16      @Override
17      public void run() {
18          System.out.println("Task1 -> XStart: " + coordinates.getxLow() + " YStart: "+
19          synchronized (lockData){
20              try{
21                  lockData.getArrived().getAndIncrement(); // ++arrived
22                  if(lockData.getArrived().get() < 4){
23                      lockData.wait(); // cwait(c,m)
24                  }
25                  else{
26                      lockData.notifyAll(); // broadcast(c)
27                  }
28              } catch (InterruptedException e) {
29                  e.printStackTrace();
30              }
31          }
32          System.out.println("Task2 -> XStart: " + coordinates.getxLow() + " YStart: "+
33      }
34  }
```

***Here, synchorized keyword acts like a mutex and it locks and unlock code in its scope. We are using **lockData object is** like a mutex(with the help of object class) since **it is common and shared between** all threads.

3.2.4 Creating Threads and Testing

```
14  public class TestDrive {
15      public static void main(String[] args) {
16          // create thread shared data number of arrived
17          AtomicInteger arrivedCount = new AtomicInteger(0);
18
19          //set common data and lock object
20          LockAndData data = new LockAndData(arrivedCount);
21
22          //create threads and inject shared data and its responsible coordinates in matrix
23          Thread thread0 = new Thread(new SyncThreadFunction(data, new Coordinates(0, 4096, 0
24          Thread thread1 = new Thread(new SyncThreadFunction(data, new Coordinates(0, 4096, 4
25          Thread thread2 = new Thread(new SyncThreadFunction(data, new Coordinates(4096, 8192
26          Thread thread3 = new Thread(new SyncThreadFunction(data, new Coordinates(4096, 8192
27
28          //start threads
29          thread0.start();
30          thread1.start();
31          thread2.start();
32          thread3.start();
33
34          try {
35              /*Make sure all threads have finished.*/
36              thread0.join();
37              thread1.join();
38              thread2.join();
39              thread3.join();
40          } catch (Exception e) {
41              e.printStackTrace();
42          }
43          System.out.println("All threads are finished. Good Bye...");
44
45      }
46  }
```

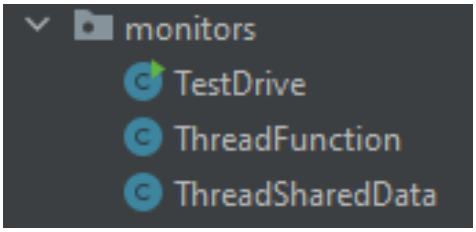
3.3.5 Output

```
Task1 -> XStart: 4096 YStart: 0
Task1 -> XStart: 4096 YStart: 4096
Task1 -> XStart: 0 YStart: 0
Task1 -> XStart: 0 YStart: 4096
Task2 -> XStart: 0 YStart: 4096
Task2 -> XStart: 4096 YStart: 0
Task2 -> XStart: 0 YStart: 0
Task2 -> XStart: 4096 YStart: 4096
All threads are finished. Good Bye...

Process finished with exit code 0
```

*As you can see all task2s **didn't start** all task1s are finished.

3.3 Using mutex(es) and monitor(s)

3.3.1 Classes in my solution



3.3.2 Shared Data between Threads

```
5   import java.util.concurrent.atomic.AtomicInteger;
6   import java.util.concurrent.locks.Condition;
7   import java.util.concurrent.locks.ReentrantLock;
8
9   public class ThreadSharedData {
10      //private ComplexNumber[][] matrixA;
11      //private ComplexNumber[][] matrixB;
12      //private ComplexNumber[][] matrixSum;
13      private AtomicInteger arrived;
14      private ReentrantLock mutex;
15      private Condition cond;
16
17      public ThreadSharedData(AtomicInteger arrived, ReentrantLock mutex, Condition cond) {
18          this.arrived = arrived;
19          this.mutex = mutex;
20          this.cond = cond;
21      }
```

*Atomic integer is much better between threads in java.

3.3.3 Common Thread Function between Threads

```
6   public class ThreadFunction implements Runnable{
7       private ThreadSharedData data;
8       private Coordinates coordinates;
9
10      public ThreadFunction(ThreadSharedData data ,Coordinates coordinates) {
11          this.data = data;
12          this.coordinates = coordinates;
13      }
14
15      @Override
16      public void run() {
17          System.out.println("Task1 -> XStart: " + coordinates.getxLow() + " YStart: "+ coordinates.getyLow());
18          data.getMutex().lock(); // lock(m)
19          try{
20              data.getArrived().getAndIncrement(); // ++arrived
21              if(data.getArrived().get() < 4){
22                  data.getCond().await(); // cwait(c,m)
23              }
24              else{
25                  data.getCond().signalAll(); // broadcast(c)
26              }
27          } catch (InterruptedException e) {
28              e.printStackTrace();
29          } finally {
30              data.getMutex().unlock(); // unlock(m)
31          }
32          System.out.println("Task2 -> XStart: " + coordinates.getxLow() + " YStart: "+ coordinates.getyLow());
33
34      }
35  }
```

3.3.4 Creating Threads and Testing

```
12  public class TestDrive {
13      public static void main(String[] args) throws IOException, InterruptedException {
14          // create thread shared data
15          ReentrantLock mutex = new ReentrantLock();
16          Condition cond = mutex.newCondition();
17          AtomicInteger arrivedCount = new AtomicInteger(0);
18
19          //set common data
20          ThreadSharedData data = new ThreadSharedData(arrivedCount, mutex, cond);
21
22          //create threads and inject shared data and its responsible coordinates in matrix
23          Thread thread0 = new Thread(new ThreadFunction(data, new Coordinates(0, 4096, 0, 4
24          Thread thread1 = new Thread(new ThreadFunction(data, new Coordinates(0, 4096, 4096
25          Thread thread2 = new Thread(new ThreadFunction(data, new Coordinates(4096, 8192, 0
26          Thread thread3 = new Thread(new ThreadFunction(data, new Coordinates(4096, 8192, 4
27
28          //start threads
29          thread0.start();
30          thread1.start();
31          thread2.start();
32          thread3.start();
33
34          try {
35              /*Make sure all threads have finished.*/
36              thread0.join();
37              thread1.join();
38              thread2.join();
39              thread3.join();
40          } catch (Exception e) {
41              e.printStackTrace();
42          }
43          System.out.println("All threads are finished. Good Bye...");
44      }
45  }
```

3.3.5 Output

```
Task1 -> XStart: 4096 YStart: 4096
Task1 -> XStart: 0 YStart: 4096
Task1 -> XStart: 0 YStart: 0
Task1 -> XStart: 4096 YStart: 0
Task2 -> XStart: 4096 YStart: 0
Task2 -> XStart: 4096 YStart: 4096
Task2 -> XStart: 0 YStart: 4096
Task2 -> XStart: 0 YStart: 0
All threads are finished. Good Bye...

Process finished with exit code 0
```

*As you can see all task2s **didn't start** all task1s are finished.

3.4 Calculating A+B and Discrete Fourier Transform

3.4.1 A+B Implementation

```
for (int i = coordinates.getxLow(); i < coordinates.getxUp() ; i++) {
    for (int j = coordinates.getyLow(); j <coordinates.getyUp() ; j++) {
        lockData.setSumByIndex(i,j, Helper.addNumbers(lockData.getAByIndex(i,j),lockData.getBByIndex(i,j)));
    }
}
```

3.4.2 Discrete Fourier Transform Formula and Implementation

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn}$$
$$= \sum_{n=0}^{N-1} x_n \cdot \left[\cos\left(\frac{2\pi}{N}kn\right) - i \cdot \sin\left(\frac{2\pi}{N}kn\right)\right], \quad \text{(Eq.1)}$$

Implementation

```
37          int n = (coordinates.getxUp() - coordinates.getxLow());
38          int k = 0;
39          for (int i = coordinates.getxLow(); i < coordinates.getxUp() ; i++) {
40              double sumReal = 0;
41              double sumImag = 0;
42              for (int j = coordinates.getyLow(); j <coordinates.getyUp() ; j++) {
43                  double angle = (2 * Math.PI * k * coordinates.getPortion()) / n;
44                  sumReal +=  lockData.getSumByIndex(i,j).getReal() * Math.cos(angle) +
45                          lockData.getSumByIndex(i,j).getImg() * Math.sin(angle);
46                  sumImag += -1*lockData.getSumByIndex(i,j).getReal() * Math.sin(angle) +
47                          lockData.getSumByIndex(i,j).getImg() * Math.cos(angle);
48              }
49              lockData.setResByIndex(coordinates.getPortion(),k,new ComplexNumber((int)sumReal,
50                      (int)sumImag));
51              k++;
52          }
```

*Here, as you can see thread calculates dft for only its responsible portion of the matrix.

3.5 Testing full implementation as console application

In order to test we need to create 8192x8192 size of matrix and calculate dft on this matrix. But if I choose this size, I am getting **java.lang.OutOfMemoryError**. Therefore, **I tested my application with 4096x4096 size of matrix.**

Java synchronized result

```
Threads are starting...
Time Taken in java synchronized: 1632 ms
All threads are finished. Good Bye...

Process finished with exit code 0
```

Java monitor result

```
Threads are starting...
Time Taken in java monitor: 1601 ms
All threads are finished. Good Bye...

Process finished with exit code 0
```

Using a single thread result

```
Calculating is starting...
Time Taken in single thread: 3340 ms
Single thread is finished. Good Bye...

Process finished with exit code 0
```

*As you can see in multi thread version we have gained about 1700 ms.

Note: As you have seen, main thread is used only for test purpose such as creating threads etc.

3.5 Responsive GUI

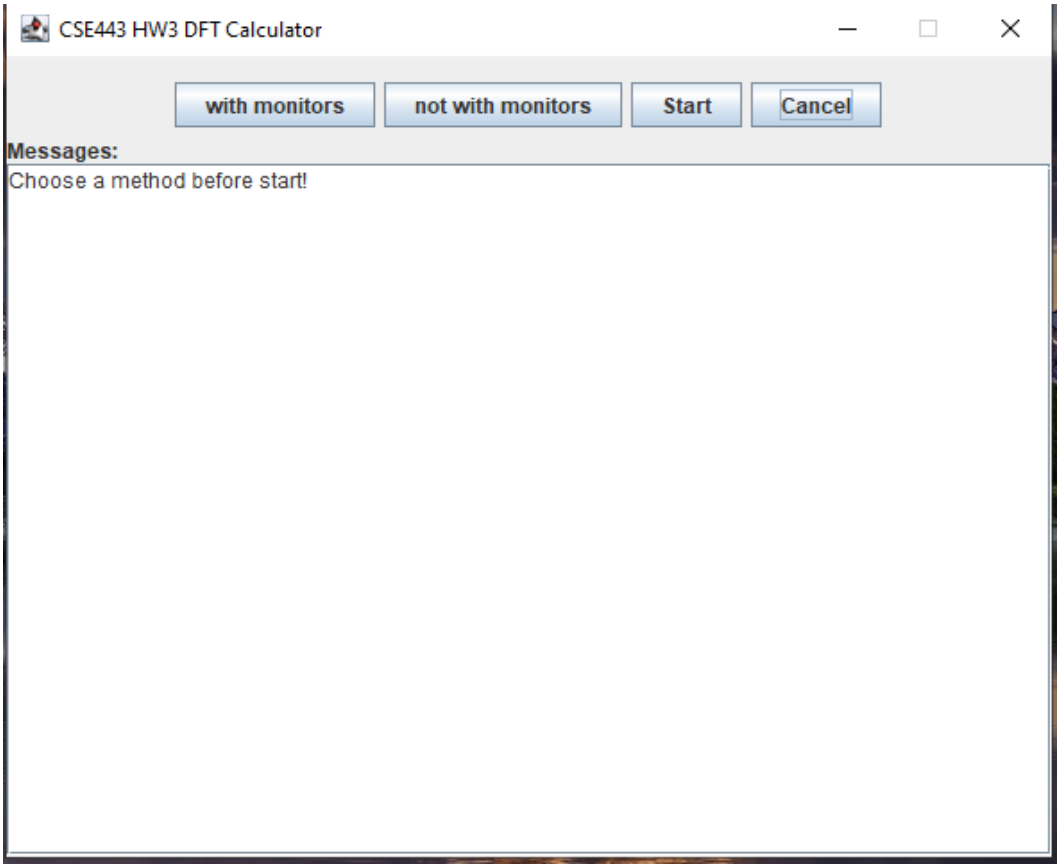
In java Swing GUI, **if we use these implementations directly** our gui **will not be respond** until calculation is done.

Therefore in order to solve this problem, I have used **java SwingWorker<T,V> class** and it worked.

Definition:

SwingWorker is designed for situations where you need to have a long running task run in a background thread and provide updates to the UI either when done, or while processing. *docs.oracle.com*

3.5.1 My Simple GUI



*with monitors = multi thread

* not with monitor = single thread

3.5.2 Making it responsive while calculating

First See start button action listener code

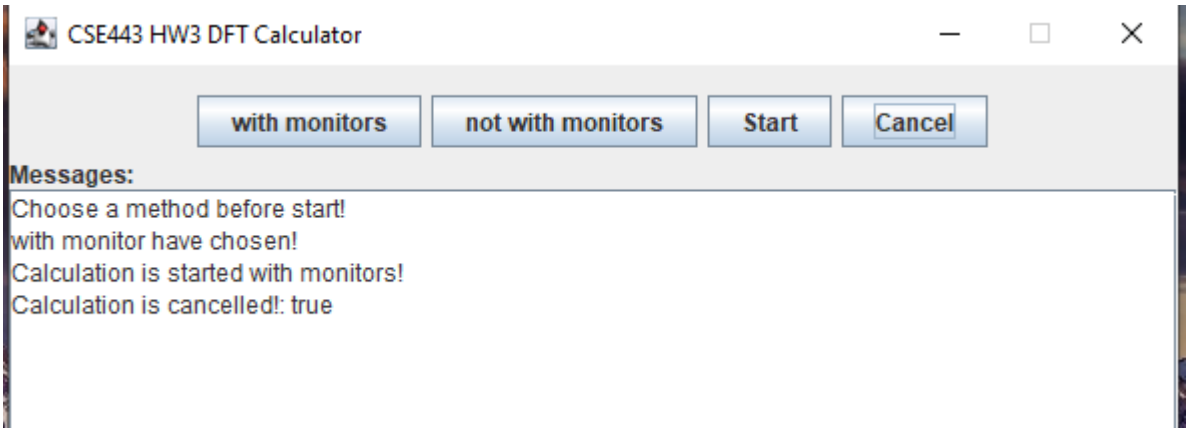
```
btnNewButton_2.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (type == 0) {
            JOptionPane.showMessageDialog(null, "Choose a method!"
                    , "Warning", JOptionPane.INFORMATION_MESSAGE);
        } else {
            calculator = new SwingWorker<Long, Void>() {
                @Override
                public Long doInBackground() {
                    isWorking = true;

                    //logArea.append(String.valueOf(type));
                    if (type == 1) {
                        logArea.append("\nCalculation is started with monitors!");
                        Long start = System.currentTimeMillis();
                        TestDrive.main(null);
                        return System.currentTimeMillis() - start;
                    } else {
                        logArea.append("\nCalculation is started with single thread!");
                        Long start = System.currentTimeMillis();
                        SingleThread.main(null);
                        return System.currentTimeMillis() - start;
                    }
```

Canceling operation

To cancel current operation, SwingWorker cancel method is used

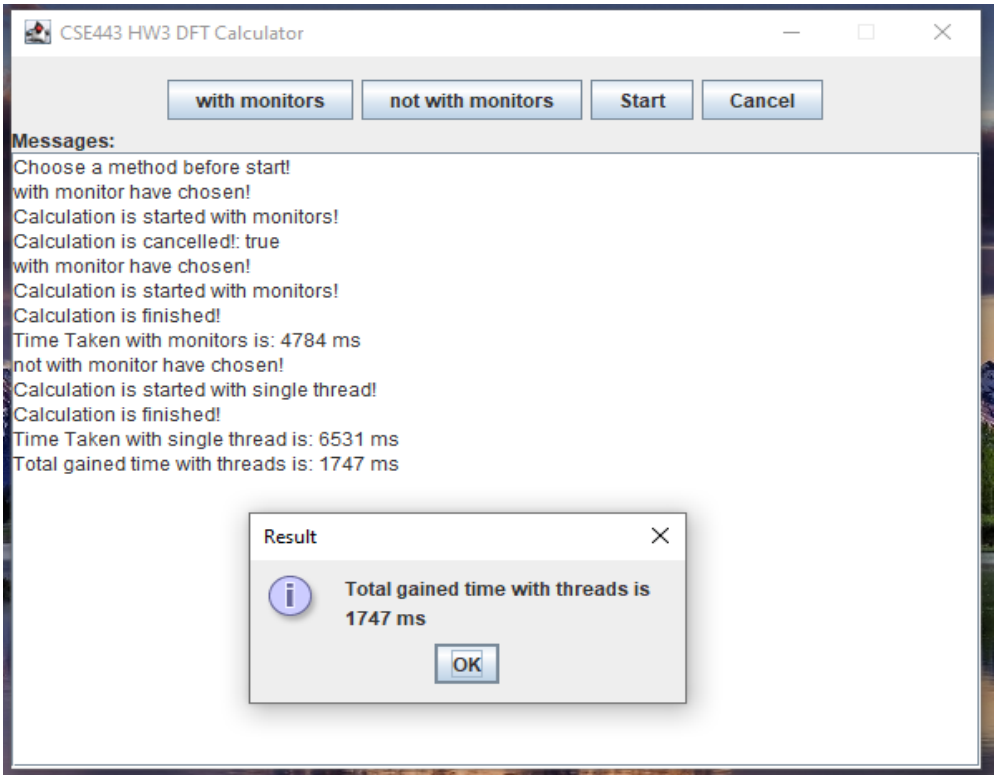
```
if (isWorking) {
    logArea.append("\nCalculation is cancelled!: " + calculator.cancel(false));
} else {
    JOptionPane.showMessageDialog(null, "Nothing is working to cancel!"
            , "Warning", JOptionPane.INFORMATION_MESSAGE);
}
```

Output:



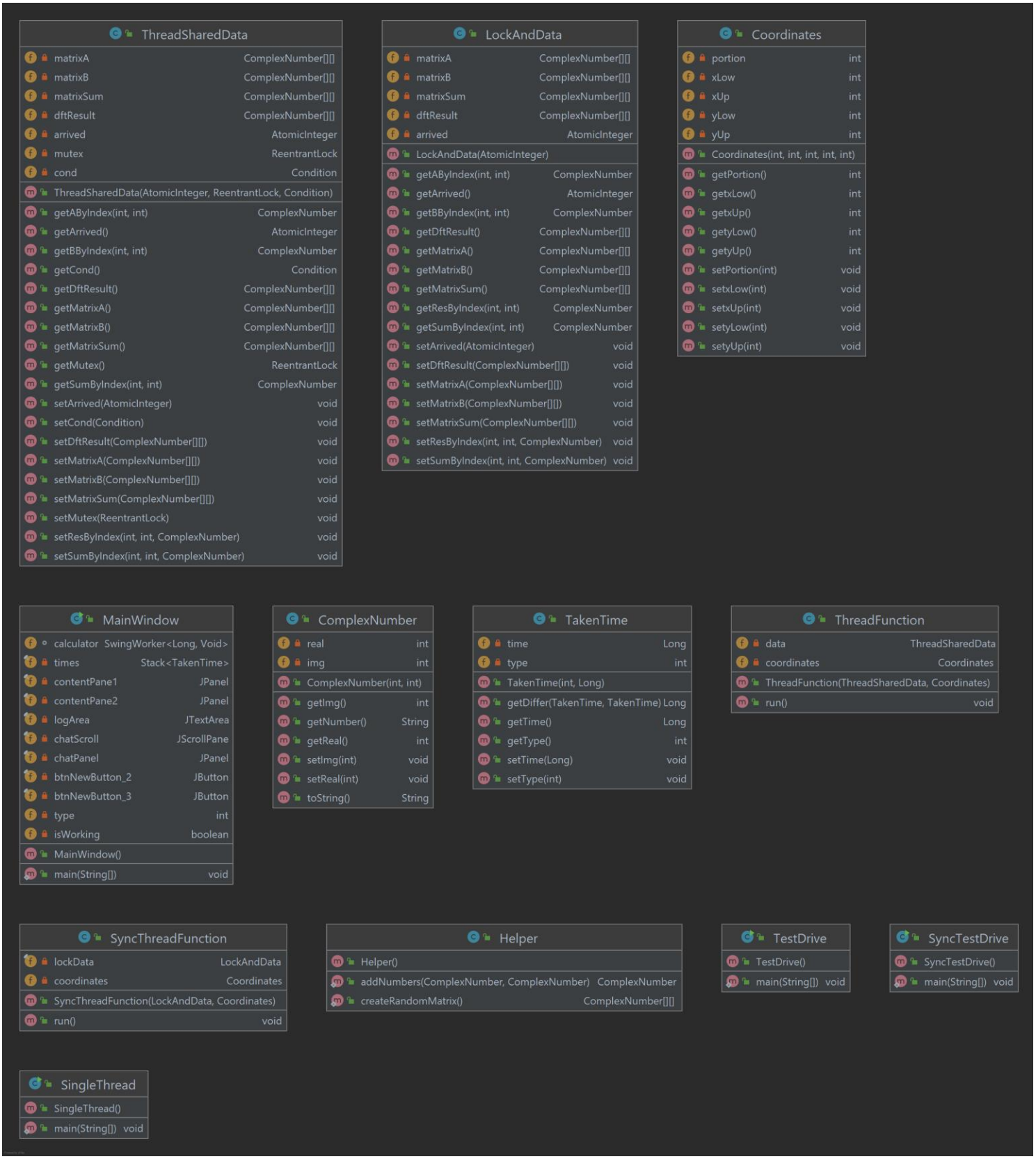
3.5.3 See the total time gained by using multiple threads

Output:



*As you can see, we gained **nearly same amount of time as in console test**. Note that in this test I have used 4096x4096 size of matrix.

3.5 Full Class Diagram



References

- CSE344 System Programming Course slides
- Head First Design Patterns, 2nd Edition.
- https://en.wikipedia.org/wiki/Discrete_Fourier_transform
- https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html
- https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html

Java Version

Project default java version is **17**. Therefore, executables(jar files) may not be working, if java 17 is not installed.