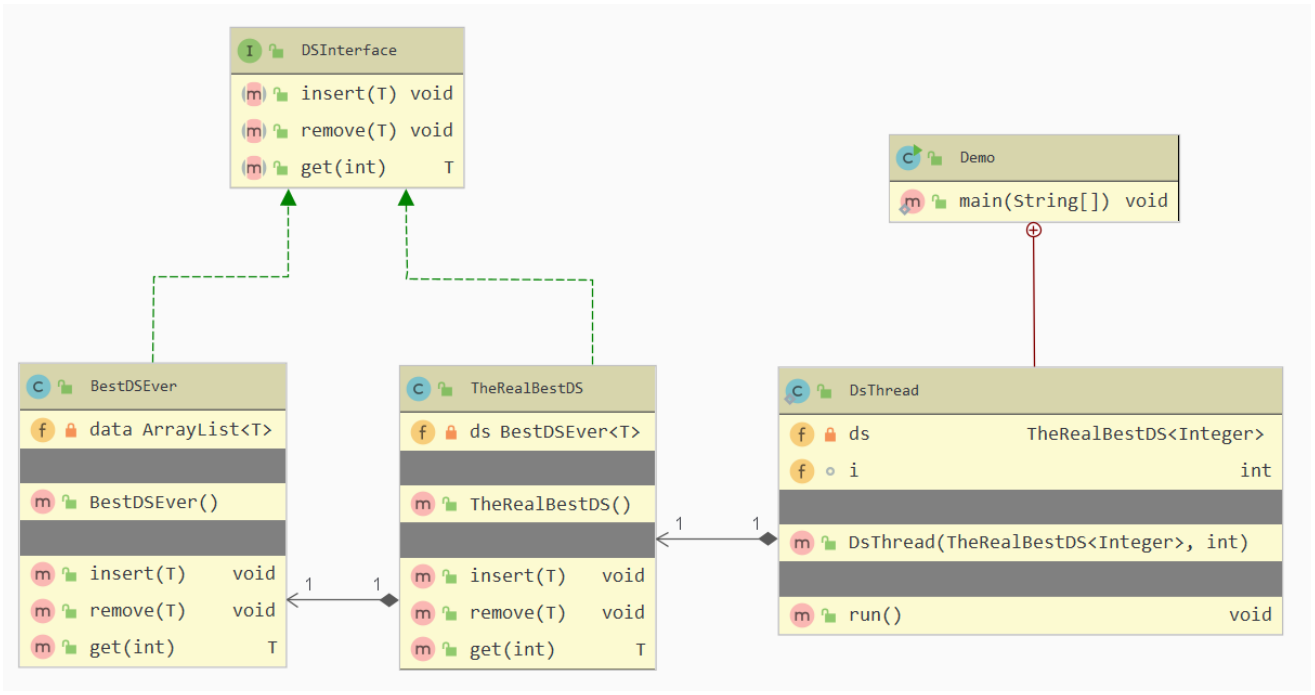
CSE 443 HW 3 Report

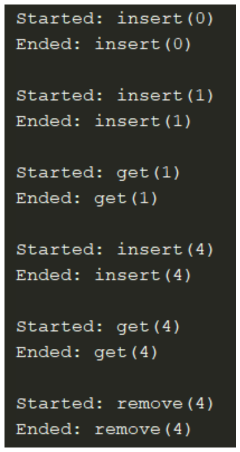
Kaan Can Bozdoğan

161044070

Part 1

Implemented proxy design pattern in order to abstract the BestDSEver class from outside usage. For every method there is a method in the proxy class and those methods all are synchronized. So no more than one of them can not work at the same time. Thus negating those methods from accessing and modifying the data structure at the same time.



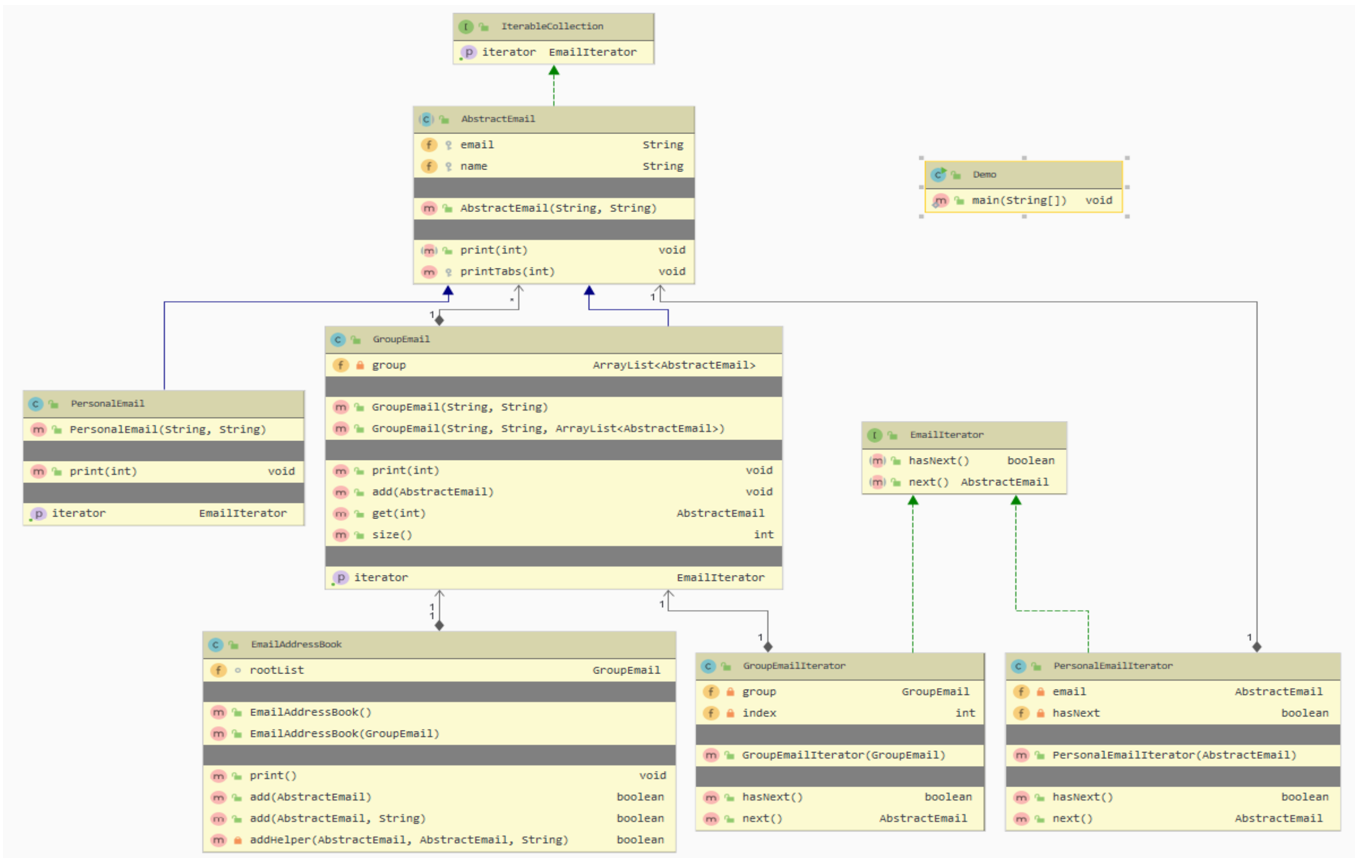


```
Started: insert(0)
Ended: insert(0)

Started: insert(1)
Ended: insert(1)

Started: get(1)
Ended: get(1)

Started: insert(4)
Ended: insert(4)

Started: get(4)
Ended: get(4)

Started: remove(4)
Ended: remove(4)
```

In the main method I run multiple threads which calls all three methods to show that it is thread safe. At the same time I added prints to the proxy class methods to demonstrate which thread is calling which function. And how they work.

As seen from a few method calls, a method starts and finishes without any interference from any other method call.

Part 2

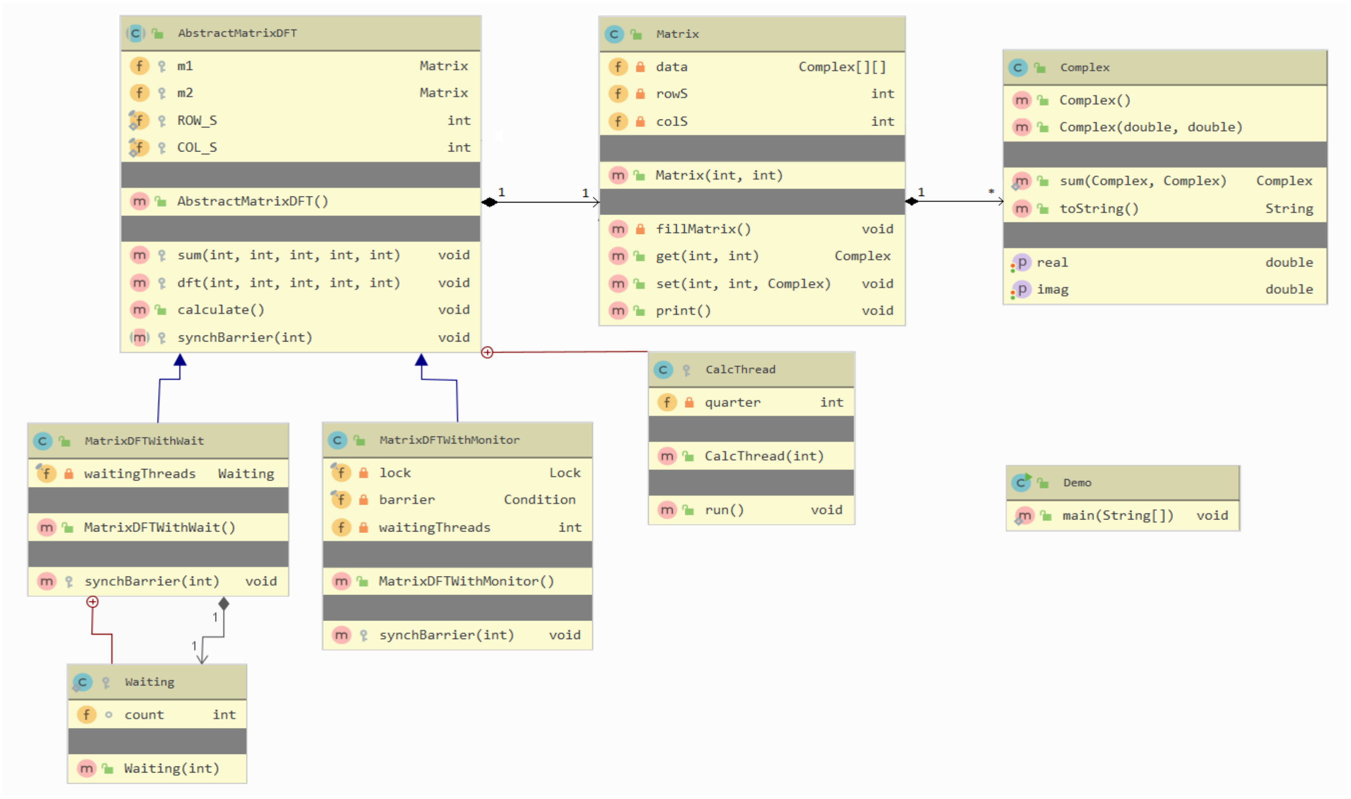


Since every item in the list has email and owner name, I created an abstract email class for this purpose. AbstractEmail class implements IterableCollection at the same time. So every class extending it needs to be able to create an iterator. Personal and group email classes extends AbstractEmail class. Each email class implements its print method. Group email of course have a list of AbstractEmail since its the composite class.

With the iterator of a GroupEmail class we can traverse every child email item in the group. With the iterator of PersonalEmail class what so ever, there is only one class to traverse and it is itself. next() returns only one time and becomes obsolete after it. With this we can just get the iterator of an email class and print it and its children (if it is a group email) without checking for any other thing. It comes handy because emails are stored in a AbstractEmail list, so we dont know if the email is group or personal. Just get the iterator and call next() as long as hasNext() returns true and call print to the returning value next(). With group email printing all of its children, all the email tree is printed without any complication.

EmailAdressBook class has a GroupEmail class as its root and all the tree structure of the email system is stored in it. So in the print method of it, just gets the iterator of root and calls print in every iteration. This task is solved easily with the combination of composite and iterator design pattern.

Part 3



Created an abstract calculator class because other than barrier handling, everything was the same in wait-notify and monitor solutions. Between summation and dft calculations synchBarrier method is called, which provides waiting to the threads until the last one of them overcomes the barrier. Then the dft calculation continues. I am sad to say that I couldn't implement the dft code. So I just put a little sleep there to simulate it. But all the required synchronization is provided.

Wait & Notify Solution:

Initialized a variable which indicates how many threads are waiting to continue for the dft calculation, to 0 at the start of the program. Every thread which finished the summation phase calls synchBarrier method. In it there is synchronized block which waits on variable I initialized at the start. First increments the variable then checks if it is the last thread to finish summation. If not, it will call wait() on the variable. If it is, then it calls notifyAll() which wakes up every thread waiting for the last thread. And they continue their calculations. Synchronized block provides mutual exclusion on checking the variable. Without it there would be complications. Like thread 1 finishing summation and checks for the variable, sees it is not the last one, then thread 2 comes in and checks the variable and sees it is the last one and notifies all, then thread one continues and calls wait. Thanks to synchronized block it is not possible.

Monitor Solution:

In this solution we have a lock, a condition object and a variable that holds the count of waiting threads. In the synchBarrier method thread calls lock to provide mutual exclusion because of the reason I explained previously. Then checks for the waiting count. If it is not the last one, calls await() on condition object. If it is, calls signalAll(), waking up other awaiting threads. The logic is the same. Provide mutual exclusion for the critical phase and check for the condition if it is okay to continue or wait for the others to overcome synchronization barrier.