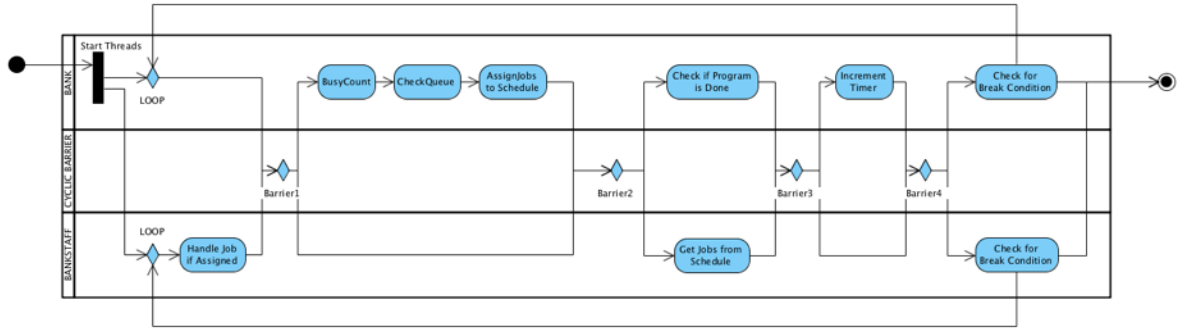
DESIGN

A. CONCURRENCY CONTROL

In our project, we heavily rely on the use of java concurrency utilities, namely cyclic barriers, priority blocking queues, and reentrant read-write locks, as this was clearly not ruled out in the instructions. While we didn't explicitly employ semaphores on their own, some of the utilities mentioned beforehand incorporate them, as well as other concurrency control mechanisms. In the following, we will explain how we used these tools to handle concurrency:

1) **CyclicBarrier (java.util.concurrent.CyclicBarrier)**
   A cyclic barrier is „A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. CyclicBarriers are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called *cyclic* because it can be re-used after the waiting threads are released." (Oracle Docs).
   This barrier allows us to synchronize the looped execution of several BankStaff threads with each other and with the general Bank thread, which contains the global timer variable. Since we know the number of threads beforehand (M+1), we have a *fixed* sized party of threads. The barrier basically forces this party of threads into a linear order of execution, since they have to wait for each other at certain points during their execution. Within the CyclicBarrier source code itself, ReentrantLocks are used are used to handle concurrent access.
   Since our program contains several loops, it is particularly useful, that the barrier is *cyclic*, which is a contrast to, for example, a CountDownLatch. The following graphic gives a great overview how and where we used barriers to handle our threads.



2) **PriorityBlockingQueue (java.util.concurrent.PriorityBlockingQueue)**
   For job-handling, we rely on the use of the PriorityBlockingQueue class. All BanksStaff threads (Consumers) take jobs from these queues, while the Bank class (Producer) puts jobs into them. A PriorityBlockingQueue is „An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations [...]" (Oracle Docs).
   At this point, the priority implementation is of no further interest to us, as it is only used later on in the bonus section, where we will go into more detail. For synchronization matters, it is import to understand the BlockingQueue interface. According to Oracle, the „BlockingQueue implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. [...] Note that a BlockingQueue can safely be used with multiple producers and multiple consumers." (Oracle Docs). This means, that the BlockingQueue basically works like the solution to the BoundedBufferProblem presented during lecture.
3) **ReentrantReadWriteLock (java.util.concurrent.locks.ReentrantReadWriteLock)**
   For synchronizing access to companies' bank accounts, we employ ReentrantReadWriteLocks. Every company instance is assigned with its own lock during construction. Threads that want to access a company's account, must first acquire either the respective ReadLock or WriteLock, dependent on their task. These locks work exactly as the solution to the ReadersAndWriters problem presented in class, and their functionality should be well-known, so no further discussion is needed.

B. OBJECTS

Our project consists of a total of eight .java classes. Below, we will explain each one's purpose and its main functions (getter, setter, etc. are not considered main functions).

1) Main

Contains the well-known main(String[] args) function, starts the program by running the GUI Frame and the methods in BankFrame.

2) BankFrame

This class initializes the Graphical User Interface (GUI) using a JFrame to create the window and JPanels to control the window dynamically. The class also adds two buttons in the GUI that help the user control the program, and a text area where messages on the running of the program are printed. It uses the following methods:

1) public BankFrame()
   Creates the application, calls initialize method.
2) private void initialize()
   Creates JFrame, JPanels, JButtons, JTextArea along with their properties and stylistic design (e.g. fonts and colors used). These are the building blocks of the GUI interface.
3) public void run(int chosenFileNumber)
   This is the function invoked once the user selects a file number from the dropdown selection if the Start Business Day button is pressed. The selected file number is passed in as chosenFileNumber. From here FileScanner class is called, and the rest of the program starts running.
4) public static void printStream(boolean printStream)
   The function activates the printStream that prints all the console messages to the JTextArea in the GUI Frame. However, if the user is in manual mode and chooses to run the program in the console then the else statement is executed and the printStream isn't activated.
5) private static void updateTextArea(final String text)
   Updates the JTextArea with messages from the console, if printStream is invoked.
6) private static void redirectSystemStreams()
   Uses methods that forces program to print to frame instead of to the console.

3) FileScanner

Is used to read input from the .txt configuration files. Every time when we iterate over the tokens in the .txt-files, a new Scanner object is created to reset the iterator. This is more economical than to reset the scanner (which is also why Oracle didn't even incorporate this function to begin with).

1) public FileScanner(int chosenFileNumber)
   Takes in the chosen file number from user in GUI frame as a parameter. If the chosen file is 1,2 or 3 then it creates the file name automatically and calls setup() and createEconomy(), effectively starting the running of the program. If chosen file is other, then the program enters manual mode, and the keyboard scanner opens, letting the user manually inputting a file manually. Once a valid file is selected (i.e. it exists in the resources folder), the user then has the option to continue in the console or print messages to frame.
2) private void setup()
   Reads the configuration data (M, T_d, T_w, T_b, T_in, T_out) at the beginning of each .txt-file and saves it in a global array. Since all the provided config-samples had this form, it is assumed that every configuration file contains these details in exactly this order.
3) private void createEconomy()
   This is used to search the config-file for details on companies. Creates an instance of Company for every company specified in the configuration and stores it in a global array.
4) public PriorityBlockingQueue<Job> assignJobs(int timer)
   This is very frequently invoked by the Bank class. Searches the config-file for jobs where the assigned time equals the timer-parameter, and creates Job instances if there are any. All created instances are stored in a PriorityBlockingQueue which is then returned to the caller.
5) public boolean isDone(int timer)
   Is also invoked by the Bank class. Checks if all the entries in the config-file have been read. and returns a boolean variable depending on the result.

4) Bank

Is basically the most important class in the program. Contains and handles the global timer variable as well as the CyclicBarriers, both of which are used to synchronize the threads.

1) public Bank(FileScanner reader)
   Is used only once. Sets up all the variables in the class. The passed on reader instance is especially important, since the Bank class needs to have access to the config-file

2) public void doBusiness()
   If Bank is the main class of the program, then this is its main method. Creates all other threads, reads input from the config-file (acts as a producer), assigns jobs, handles the timer, and determines when the program is done. For better understanding of this method, refer to the graphic presented in the CyclicBarrier subsection of the concurrency control section.
3) public void awaitBarrier(int barrierNo)
   This is used to give everyone access to the CyclicBarriers. We created an extra method for this to make the source code more readable, since every await() instruction needs an extra try-multicatch surrounding it.

5) BankStaff

Implements the Runnable interface so it can run in parallel to the Bank thread. There are several instances of this, dependent on M.

1) public BankStaff(int i, Bank bank)
   Sets up an instance of BankStaff. i represents the tellerID, bank is used so this instance has a dynamic reference to Bank.
2) public void run()
   Most important method in this class. This is basically the corresponding code to Bank's doBusiness() method. Simulates how tellers handle jobs and customers. Same as in Bank doBusiness(), refer to the graphic provided earlier to get a better understanding of this method

6) Job

There is one instance of Job created for every job in the config-file. Every instance contains a lot of information which is set when the constructor is called. Implements the Comparable interface, which, in combination with PriorityBlockingQueue, is used for one of the bonus points.

1) public Job(...)
   Creates an instance of Job. Needs a lot of information and dynamic references in order to function properly.
2) public void setAdmitted(int admitted, BankStaff teller)
   When a teller starts to process a Job, admitted is set as the current timer and a dynamic reference to said teller is created.
3) public void execute(int timer)
   This is used to pass a task BankStaff to Employee for execution

7) Employee

Instances of Employee are created within FileScanner every time a customer comes to the Bank.

1) public Employee(int ID, Company employer)
   The constructor is called from within the FileScanner class and assigns every employee his own ID. Furthermore, a dynamic link to an instance of company is created.
2) public void doTask(...)
   This is called from within Job and signals that the employee can now execute his task. The function is overloaded with a lot of parameters that contain information about the Job, as well as a dynamic link to the teller that supervises the action. Calls the action which is defined in transactionType.
3) private void withdraw(...)
   Is called from doTask and executes a withdraw command. For this, a WriteLock from the associated company is acquired, then the method forces the thread to idle for a while to simulate the transaction time. Once the transaction is done, the lock is released.
4) private void deposit(...)
   Is called from doTask and executes a deposit command. For this, a WriteLock from the associated company is acquired, then the method forces the thread to idle for a while to simulate the transaction time. Once the transaction is done, the lock is released.
5) private void checkBalance(...)
   Is called from doTask and executes a checkBalance command. For this, a ReadLock from the associated company is acquired, then the method forces the thread to idle for a while to simulate the transaction time. Once the transaction is done, the lock is released.

8) Company

Instances of Company are created at the beginning of the program within the FileScanner class. Each company has a balance, a distinct number and a ReentrantReadWriteLock which is used to protect access to its account (balance).

1) public Company(int companyNo, int balance)
   The constructor assigns values to the global variables and creates a lock object.

C. PSEUDO CODE

```
public class Main {

        public static void main(String[] args) {
          //RUNS THE GUI FRAME
        }
}
```

```
public class Bank {

        public Bank(...) {
          //CONSTRUCTOR; SETS UP CLASS
        }

        public void doBusiness()  {
                for(EVERY BANKSTAFF) {
                  //START THREAD
                }
                while(PROGRAM NOT DONE) {
                        awaitBarrier(1);
                        for(EVERY BANKSTAFF) {
                          //COUNTS HOW MANY TELLERS ARE BUSY
                        }
                        for(EVERY JOB IN QUEUE) {
                                  //DECREMENT QUEUEING TIME
                                if(JOB ARRIVED IN QUEUE AND TELLER AVAILABLE) {
                                  //REMOVE FROM QUEUE AND ADD TO SCHEDULE
                                }
                        }
                          //READ JOBS FROM CONFIG TO TEMP QUEUE
                        for(EVERY JOB IN TEMP QUEUE) {
                                if(TELLER AVAILABLE) {
                                  //ADD JOB TO SCHEDULE
                                } else {
                                  //ADD JOB TO QUEUE
                                }
                        }
                        awaitBarrier(2);
                        if(EVERYTHING IS DONE) {
                          //SET PROGRAM DONE TRUE
                        }
                        awaitBarrier(3);
                          //INCREMENT GLOBAL TIMER
                        awaitBarrier(4);
                        if(PROGRAM DONE) {
                          //EXIT LOOP
                        }
                }
        }

        public void awaitBarrier(int barrierNo) {
                switch(barrierNo) {
                  //CALLS AWAIT() ON THE SPECIFIED BARRIER
                }
        }
}
```

```
public class BankFrame {

	public BankFrame() {
	  //CONSTRUCTOR; SETS UP CLASS
	}

	private void initialize() {
	  //CREATES BUTTONS, TEXT AREAS, MENUS; PLACES THEM IN FRAME
	}

	public void run(...) {
	  //RUNS THE CHOSEN CONFIG FILE IN BANK
	}

	public static void printStream(...) {
		if (PRINT STREAM TRUE ) {
		  //REDIRECT TO PRINTSTREAM
		} else {
		  //PRINT TO CONSOLE
		}
	}

	 private static void updateTextArea(...) {
	  //UPDATE THE TEXT IN TEXTAREA
	 }

	 private static void redirectSystemStreams() {
	  //REDIRECT TEXT OUTPUT FROM CONSOLE TO SWING INTERFACE
	 }

}
```

```
public class BankStaff implements Runnable {

	public BankStaff(...) {
	  //CONSTRUCTOR; SETS UP CLASS
	}

	@Override
	public void run() {
		while(true) { //MAIN LOOP FOR TELLER
			if(TELLER IS BUSY) {
				if(ASSIGNED TASK COMING FROM QUEUE) {
				  //DECREMENT QUEUE TIMER
				} else {
				  //EXECUTE ASSIGNED TASK
			}
			employer.awaitBarrier(1);
			employer.awaitBarrier(2);
			if(TELLER NOT BUSY){
				  //TRY TO ACQUIRE JOB
				if(JOB ACQUIRED) {
				  //ASSIGN TASK TO TELLER
				}
			}
			employer.awaitBarrier(3);
			employer.awaitBarrier(4);
			if(PROGRAM IS DONE) {
			  //END THREAD
			}
}}}
```

```
public class Job implements Comparable<Job> {

	public Job(...) {
	  //CONSTRUCTOR; SETS UP CLASS
	}

	public void setAdmitted(...) {
	  //ASSIGNS TELLER AND ADMITTED TIME TO JOB
	}

	public void execute(...) {
	  //HANDS TASK TO EMPLOYEE FOR EXECUTION
	}

	@Override
	public int compareTo(...) {
	  //RANK JOBS AGAINST EACH OTHER
	}
}
```

```
public class FileScanner {

	public FileScanner(...) {
		do {
		  //ASKS WHICH CONFIG FILE SHOULD BE RUN
		} while(NO LEGAL SELECTION);
		do {
		  //ASKS IF PROGRAM SHOULD BE RUN IN INTERFACE OR CONSOLE
		} while(NO LEGAL SELECTION MADE);
		  //READS CONFIG AND SETS IT UP
		  //CALLS SETUP
		  //CALLS CREATE ECONOMY
		}

	private void setup() {
		  //CREATE ARRAY
		while(ITERATE THROUGH CONFIG) {
		  //READ M, T_D, T_W, T_B, T_IN, T_OUT FROM CONFIG
		}
	}

	private void createEconomy() {
		while(ITERATE THROUGH CONFIG) {
		  //READ HOW MANY COMPANIES THERE ARE
		}
		  //CREATE ARRAY
		while(ITERATE THROUGH CONFIG) {
		  //CREATE INSTANCES OF COMPANY AND PLACE THEM IN ARRAY
		}
	}

	public PriorityBlockingQueue<Job> assignJobs(...) {
		while(ITERATE THROUGH CONFIG) {
			if(NEXT OBJECT IS JOB) {
				if(OBJECT IS JOB WITH CURRENT TIME) {
				  //CREATE INSTANCE OF EMPLOYEE
				  //DETERMINES TYPE OF JOB
				  //CREATE NEW JOB
				} else if(OBJECT IS JOB WITH FUTURE TIME) {
				  //EXIT LOOP
				}
```

```
                }
            }
             //RETURN ALL FOUND JOBS WITH CURRENT TIME
        }

        public boolean isDone(...) {
            while(ITERATE THROUGH CONFIG) {
             //COUNT NUMBER OF JOBS IN CONFIG
            }
            while(ITERATE THROUGH CONFIG CONFIG) {
             //COUNT HOW MANY JOBS ARE ALREADY DONE
            }
            if(NUMBER OF JOBS EQUALS NUMBER OF DONE JOBS) {
             //ALL JOBS ARE READ FROM CONFIG
            }
             //RETURN RESULT
        }
}
```

```
public class Employee {

        public Employee(...) {
         //CONSTRUCTOR; SETS UP CLASS
        }

        public void doTask(...) {
         //DETERMINES TRANSACTION TYPE AND CALLS APPROPRIATE METHOD
        }

        private void deposit(...) {
            while(NO LOCK) {
                if {
                 //TRIES TO ACQUIRE WRITELOCK FROM COMPANY
                } else {
                 //WAIT FOR ONE TURN
                }
            }
            while(JOB IS NOT DONE) {
             //WAIT FOR ONE TURN
            }
             //DEPOSIT INTO ACCOUNT
             //RELEASE LOCK
        }

        private void withdraw(...) {
            while(NO LOCK) {
                if {
                 //TRIES TO ACQUIRE WRITELOCK FROM COMPANY
                } else {
                 //WAIT FOR ONE TURN
                }
            }
            while(JOB NOT DONE) {
             //WAIT FOR ONE TURN
            }
             //WITHDRAW FROM ACCOUNT
             //RELEASE LOCK
        }

        public void checkBalance(...) {
            while(NO LOCK) {
```

```
                        if {
                          //TRIES TO ACQUIRE READLOCK FROM COMPANY
                        } else {
                          //WAIT FOR ONE TURN
                        }
                }
                while(JOB NOT DONE) {
                  //WAIT FOR ONE TURN
                }
                  //CHECK BALANCE OF ACCOUNT
                  //RELEASE LOCK
        }
}
```

```
public class Company {

        public Company(...) {
          //CONSTRUCTOR; SETS UP CLASS
        }

}
```

D. BONUS POINTS

1) **You can develop a reasonable method so that all the customers get served in the shortest amount of time;**
   Our program contains two methods which try to assure that every customer gets served in the shortest time possible. These two methods are the ReentrantReadWriteLock and the PriorityBlockingQueue.
   The ReentrantReadWriteLock gives threads, which only wish to check the balance of a company, the ability to access the company's account concurrently, which speeds up the program in general.
   The PriorityBlockingQueue imposes a FCFS rule on employees arriving at the bank. We chose FCFS because we wanted to achieve a minimal average turnaround time, a goal which a real world bank would probably pursue as well. SJF would have been a possibility as well, but since we did not have an easy way to implement aging, starvation was a serious problem when applying this algorithm.
2) **The system comes with a nice interface to illustrate the process;**
   We implemented a GUI interface to make it easier to use our program and improve the user experience. The interface uses a JFrame to create the window and JPanels to control the window dynamically. The class also adds two buttons that help the user control the program, and a text area where messages on the running of the program are printed. The text area implements the JScroll option: this is a useful design feature, allowing the user the ability to scroll within the text area.
3) **You implement creative/innovative way of applying synchronization.**
   We achieve synchronization mainly through the use of CyclicBarriers. We figured that, within the scope of the lecture, this qualifies as an innovative way of achieving synchronization, since we did not learn about this before. We only discovered this utility while exploring the java concurrency utensils, so it was new to us.