Project Summary

Andrew Engel

Rasmussen College

**Needs Analysis**

**Requirements**

The course project application will be designed to simulate the reception of a bank. It will simulate customer transactions, bank teller allocation, and the passing of time. The application will also be required to print a report of the interactions with customers. The following will describe what is required of each of these simulated components.

The customer simulation will allow customers to enter the bank at random times. When a customer enters, they are immediately assigned to a teller. Each customer may request to open an account, close an account, withdraw money or deposit money. Whichever request is made, a random amount of time will be associated with how long the transaction will take. Once a transaction is complete, the customer will no longer be assigned to a teller.

The bank teller simulator is intended to provide customers with timely service. There may be a minimum of one teller working at a time to a maximum of three, which is dependent on the number of customers in the bank. One teller will be working until there are more than four customers, which is when the second teller comes to help. Once there are more than eight customers in the bank is when the third teller comes to help. As the number of customers decreases, the tellers are removed in opposite order of their addition. In other words, they operate on a last in, first out basis.

The simulation of time passing will handle a few areas of time. First, it will keep track of the current time. The bank’s hours of operation are from 10:00 AM until 1:00 PM, which is when the simulation will start and end. The time will increment based on when customers enter the bank and how long each transaction takes.

Lastly, the simulation will end with a report of all customer transactions that took place during the day. The report will display each customer’s first name, last name, account number, time of arrival, type of transaction, and amount of money involved. This information will be sorted by customer last name, amount of money involved, and time of arrival. At the end of the report, the average waiting time per customers and average number of customers waiting will be displayed.

**Assumptions**

The application is assumed to be designed as a console application and require no user input. Once the application runs, it performs all the logic and print the results. The application will be created using the C++ programming language and utilize object oriented design. There will be classes for customers, time, and reports.

It is assumed that all customers and their attributes will be created/defined prior to entering the bank. Customers will be held in a stack, and a random number generator will determine when a customer is removed from the stack and enters the bank. Customers can only make one transaction and cannot re-enter the bank after they’ve left. Customers transaction requests will pre-determined, and the time in which they take will be randomized between 1 and 20 minutes.

As mentioned in the requirements, tellers will be assigned depending on the number of customers in the bank. It is assumed that any number of customers beyond than the first eight will all be assigned to the third teller.

Lastly, the reporting will take place by keeping track of log objects. This means that each time a customer enters the bank, a log object is created, which keeps track of all events associated with that customer. Each log object will be kept within a vector. The contents of these log objects will then be calculated and printed for the end of day report.

**Scope**

This section describes what is in scope and what is out of scope for the project. In scope deals with the simulation of customers, allocation of bank tellers, the passing of time, and reporting. All areas within the requirements and assumptions are considered in scope of this project.

There are a few components that are outside of scope of this application. One area considered out of scope is a graphically user interface. The logic and design of this project is for a console based application only. In addition, user input is out of scope. All customer attributes are pre-defined within the program and any variation of results are randomized.

This also means that users and permissions are out of scope. The application will not require a user to enter a username and password to access it. The application will only need to be compiled and run to see the results of the simulation. It is considered to be a read-only application. Lastly, it is out of scope of the application to use the client’s hard drive or a database to store data. All data will be temporarily stored and allocated within memory.

**Design Documentation**

The following is the design of the Bank Simulation Application’s three classes and the main.cpp file. The classes I will be creating are Customer, Clock and ReportLog. The main.cpp file will have access to each of these classes and help facilitate the simulation. A UML diagram can be viewed at page 7 of this document.

**Customer**

The Customer class will have the most data and methods associated with it. Customer objects will have data members that store customers’ names, arrival times, transaction types, transaction minutes and money. Get and Set methods will be available for many of these data members, as well as a method to decrement the transaction time for each minute they are being serviced. The Customer class will not be coupled with any other class besides main.cpp.

**Clock**

The Clock class will be responsible for keeping track of time and ensuring the bank is operating within business hours. The data members will consist of the current hour, current minute, the bank’s opening time and the bank’s closing time. A get method will be available for to obtain the current time. Current time will only be changed by the incrementMinute() method, which in turn will call an incrementHour() method. Lastly, the Clock class includes a function called withinBusinessHours() which returns a Boolean value.

**ReportLog**

The ReportLog class will be responsible for retaining data about events of the bank. A ReportLog object will have data members holding the customer’s last name, amount of money used in the transaction, the transaction type, and time of arrival. Get and set functions will be available for each of these members. There will also be four static members that will act as accumulators (total wait times, total new customers, number of customers waiting per minute, and total number of minutes) so that average waiting time and average number of customers waiting can be calculated.

**Main**

The main.cpp file will have a few global members available for its use. These global members will be three queues named teller1, teller2, and tell3, and will be available for use depending on the number of customers in the bank. Lastly, a vector of ReportLog objects will be available for each time an event needs to be logged.

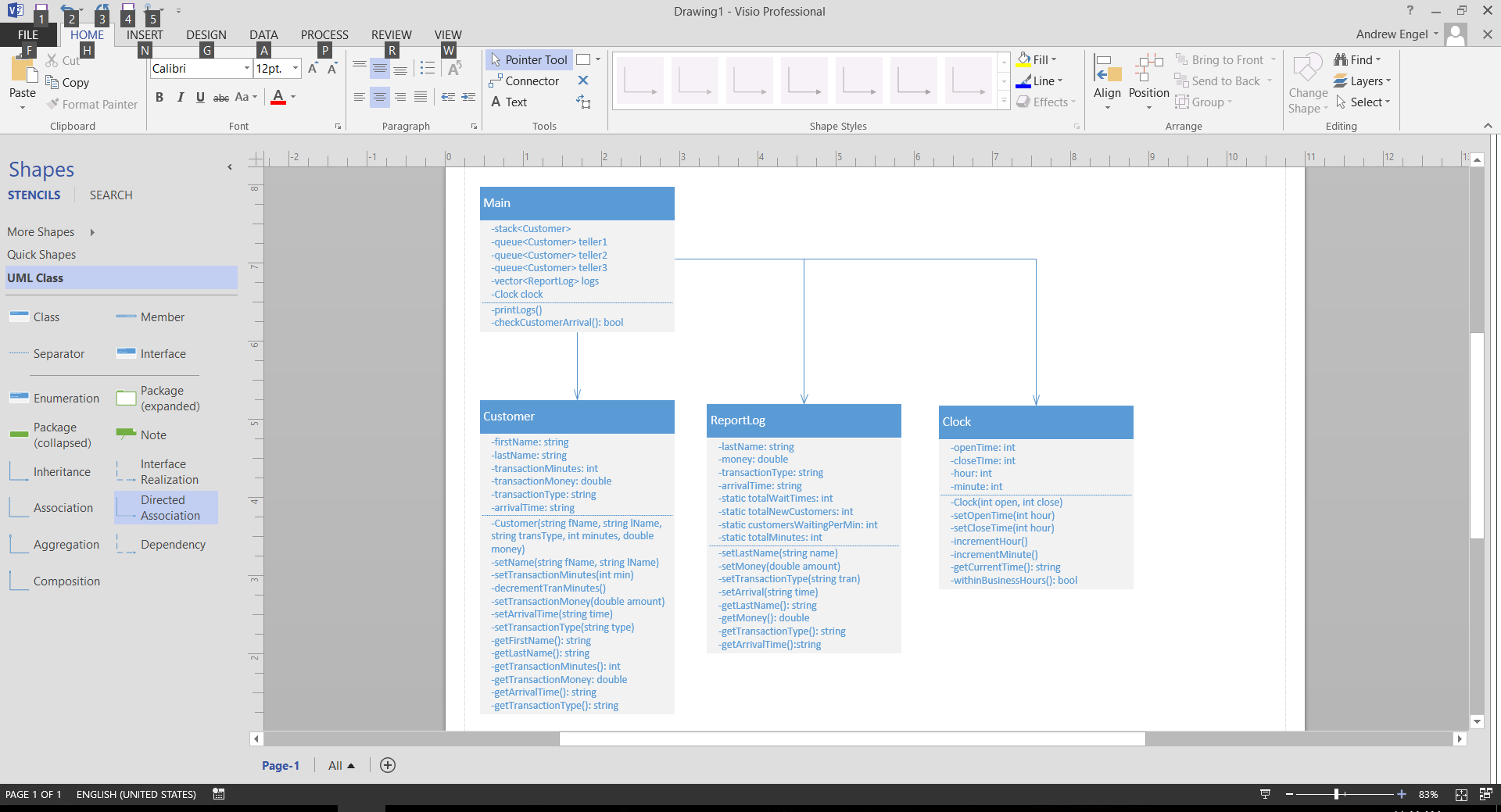
The flow of the program will begin in the main function of the main.cpp file. It will begin by creating a Clock object and setting its start and end time (i.e. business hours). Then a while loop will begin and continue running while there are customers objects in the customer stack and the time is within business hours. If these conditions are true, one minute is incremented to the clock, using its incrementMinute() function. Then, an if statement checks to see if a customer has arrived using the main.cpp’s function checkCustomerArrival(). This function will have a random number that will generate between 1-10. If the number is a 1 or 2, it will return true (giving it a 20% chance a customer will arrive in that minute – though, that percentage will need to be tested). If the checkCustomerArrival() function returns true, a ReportLog object is created and appended to the vector, documenting the top customer from the stack’s details and adding to the ReportLog accumulators.

Then, this customer will be copied to a teller’s queue and popped off the stack. If the total number of customers in the bank is less than 4, the customer is added to teller1’s queue. Otherwise, if the total number of customers in the bank is less than 8, the customer is added to teller2’s queue. If neither of these conditions are met, the customer is added to teller3’s queue.

Once the above mentioned logic is executed, or if the checkCustomerArrival() function returned false, another if statement will evaluate whether there are any customers currently in the bank. If there are customers in the bank, each customer at the front of each queue will have their transactionMinutes properties decremented by one. Then, if any of those customers’ transactionMinutes are equal to zero, they are removed from the queue.

Finally, the clock object’s withinBusinessHours() is called to evaluate whether the while loop continues. Once the while loop ends, the vector of ReportLog objects is sorted by last name and printed to the console. In addition, accumulators are calculated together to print average waiting time for customers and the average number of customers waiting.

**UML Diagram**



**Test Plan**

**Generating Random Numbers**

Random numbers will be generated for use in checking whether a customer has arrived for each minute the bank is open. The responsibility for checking whether a customer has arrived will be delegated to a function that will generate a random number between 1 and 10. If the number is a one or two, it will return true; else, it will return false. This is to simulate a 20% chance that a customer will arrive each minute of the working day.

To test both the randomization and the probability of a customer arriving (to ensure approximately 20% arrival rate), I will add a couple components to the function. First, I will add a cout statement to print the number that is being generated each time the method is called. In addition, I will include two accumulators: one that will increment each time the method is called, and one to increment whenever a one or two is generated. Finally, on the last call to the function, I will have the program divide the number of times a one or two is generated by the total number of calls to this method. This number will be printed to the console, telling me the customer arrival rate. Based on the results, I will know if I need to adjust my probability to increase/decrease the arrival rate.

**Assigning Customers to the Correct Teller**

Each time the method that checks for a customer arrival returns true, a customer will be placed into a teller’s queue. The number of customers currently in the bank will determine which teller the most recent customer is assigned to. To ensure customers are being assigned to the correct teller, I will set up a test case telling me which teller was assigned and how many customers are in the bank each time a new customer arrives.

To implement this test case, I will create a function that will be called each time the check customer arrival method returns true. This function will add together the sizes of each teller’s queue and assign that number to a variable. In addition, it will check to see which teller the most recent customer was assigned to, and assign a one, two, or three to another variable (representing teller one, two, or three). Then, these two variables will be printed to the console so I can see whether the correct teller was assigned. For example, if there are 4 customers in the bank, and the customer was assigned to teller 2, I know there is an issue with the logic.

**Validating Customers Leaving the Bank**

Next, I will need to validate whether customers are properly being removed from teller queues (i.e. leaving the bank). Each Customer object has a data member, transactionMinutes, assigned to them. Based on who is at the front of each queue, they will have their transaction minutes decremented by one for each minute that passes in the business day until they hit zero. Once their transaction minutes hit zero, they are removed from the queue.

To ensure customers are being removed from queue at the appropriate time, I will check to see that customer transaction minutes are decrementing and that they are being removed once they hit zero. For each minute that passes, the customers at the front of the teller queues will print their transactionMinutes values. In addition, before a customer is removed queue a message indicating which customer is being removed will be printed. This will allow me to view the decrementing of minutes for each customer. After each customer has one minute printed to the console, the next minute in sequence should show the customer being removed.

**Ensure the Application Runs within Business Hours**

The application will simulate a bank business day with working hours from 10:00 AM to 1:00 PM. The Clock class will have a method called withinBusinessHours() that will return a Boolean value, which will signal when the application should end. The test case for this will be fairly simple. I will add an if statement to this method that checks if a true value will be returned. If true, it will print the current time to the console. This will allow me to run the application and see what time the application ended (i.e. bank closed).

**Validating Report Data**

Once the bank is outside operating hours, a report log will be printed to the console, which will be handled by the ReportLog class. The parts I want to create a test case for are the calculations of average number of waiting customers and the average wait time for customers. These averages will be calculated through static accumulator variables that are held within the ReportLog class. These variables will accumulate total wait times, total new customers, number of customers waiting per minute, and total number of minutes.

Again, the test case for this will be relatively straight forward. I will tie a cout statement to each of the calls to increment these accumulators. The result will be a statement printed to the console displaying the current value of each accumulator and the current timestamp. That will allow me to see exactly when each accumulator is incremented so I can see exactly at what point in the application any improper aggregations occur. In addition, it will allow me to see the final values that are used in the calculations of the averages.

**Reflective Summary**

Reflecting upon the experience I had developing code for this project, I feel as though I learned a lot about C++, general coding and my progress as a developer. I feel that C++ is an exciting language that requires a much more thorough approach, as there isn’t as much built-in functionality than many other languages. However, C++ provides so many manual controls, that you can do much more with it than other languages.

One of the things I learned about C++ while developing this project was how to use static variables and methods. I’ve used them in other languages, but C++ is a bit different. My ReportLog class had static members and methods, which I called in the main.cpp file. I learned that you need to declare members and methods as static in the header file, but not in the class file. If declared static in the .cpp file, you will get a compilation error. In addition, in the main.cpp file, I needed to use the class name, scope resolution operator, and the static member/function in sequence in order to call/reference the static member/function.

A big lesson I learned about coding in general was the use of data structures. Before doing this project, I would often rely on arrays or vectors to handle most of my work with collections. This project helped me understand the use cases or other collections, which saved me from overly-complicated code, and probably some overhead. I will always put more thought toward appropriate data structures/collections with each development project moving forward.

Lastly, I learned a bit about myself and my progress toward my goal of becoming a software developer. This project has further reinforced the importance of gathering requirements, thinking about the problem, and designing the solution before writing a single line of code. I used a lot of pseudo code to map out my project, which helped me quickly figure out solutions. The actual coding of this project took me far less time than I thought, and typically I run into the opposite. Overall, there was little rework I needed to do while coding, and I mainly stuck to my written assignments and UML diagram to lead the way. It was a mind-opening experience.