CMPT 225 Assignment 2 - Bank Simulation

Due: Friday, June 24 by 15:30 - Submission via CourSys

To be done in pair

How to form a pair

We must create a "group" of 2 people on CourSys. One partner of the pair creates the "group" on CourSys and the other partner of the pair must respond to CourSys "group" notification. Then, only one partner of the pair needs to submit the assignment and when the assignment is marked, both partners will receive a mark. If the second partner missed acknowledging the CourSys "group" notification, only the submitting partner will receive a mark for the assignment.

Bank Simulation

Problem Statement

In this Assignment 2, we are to design and implement an event-driven bank simulation application. The problem statement we are solving in this assignment is described in Programming Problem 6 of Chapter 13 of our textbook and in its Section 13.4.

Before we dive into this assignment, let's first read Programming Problem 6 and Section 13.4 of Chapter 13.

Design

Let's start this assignment with the design of our solution. How many classes shall we need? To answer this question, have a look at the section "Implementation - Classes" below. What would be the responsibility of each of our classes? What attributes and operations would our classes have? What relationship would exist between them? Let's capture our design in a UML class diagram and let's make sure we include the public interfaces of our data collection classes. Let's save our UML class diagram in a pdf file and name this file **UMLClassDiagram.pdf**.

Implementation - Classes

In order to solve this problem, we will need the following classes.

1. A simulation application "class", i.e., the file with the main( ) function and perhaps other functions. Use the algorithm outlined in Section 13.4 of our textbook to guide the implementation of our simulation application. Name this file **BankSimApp.cpp**.

2. An Event class as described in Section 13.4. It must be designed and implemented as an ADT and have getters and setters for the attributes "type", "time" and "length". This class ***must not*** print anything on the computer monitor screen. Name this class Event (**Event.h** and **Event.cpp**).

3. A data collection Queue ADT class. This class ***must not*** print anything on the computer monitor screen. Name this class Queue (**Queue.h** and **Queue.cpp**). Use an array-based implementation for this Queue class and make sure our implementation abides to its Public Interface described below (expressed in C++):

```
Class invariants: FIFO or LILO

   // Description: Returns "true" is this queue is empty, oth
   // Time Efficiency: O(1)
   bool isEmpty() const;

   // Description: Adds newElement to the "back" of this queu
   //              returns "true" if successful, otherwise "f
   // Time Efficiency: O(1)
   bool enqueue(const Event& newElement);

   // Description: Removes the element at the "front" of this
   //              returns "true" if successful, otherwise "f
   // Precondition: This queue is not empty.
   // Time Efficiency: O(1)
   bool dequeue();

   // Description: Retrieves (but does not remove) the elemen
   //              "front" of this queue and returns it.
   // Precondition: This queue is not empty.
   // Postcondition: This queue is unchanged.
   // Exceptions: Throws EmptyDataCollectionException if this
   // Time Efficiency: O(1)
   Event peek() const throw(EmptyDataCollectionException);
```

When designing our Queue ADT, we need to figure out how an enqueue, a dequeue and a peek operation will work, i.e., where is the "front" and the "back" of our Queue. Note that the "front" and "back" of our Queue is not necessarily the "front" and "back"

of the Queue's underlying data structure. In this assignment, we do not have to expand (resize) our array when it becomes full. Let's set our array capacity to 100.

4. A data collection Priority Queue ADT class. This class ***must not*** print anything on the computer monitor screen. Name this class PQueue (**PQueue.h** and **PQueue.cpp**). A priority queue keeps its elements in a particular "priority" sort order. Its peek operation peeks at the element with the "highest" priority and its dequeue operation removes the element with the "highest" priority. Note that *"highest" priority* does not always mean largest value.

   In order to figure out what the sort order is for elements of Event class type, consider the following examples from our textbook:

   - Figure 13-8 of our textbook shows that an Event object of type "A" and time "20" has a "higher" priority than an Event object of type "A" and time "22" and therefore would be the next to be dequeued from the priority queue *eventListPQueue* even though "20" < "22".
   - Figure 13-8 also shows that an Event object of type "A" and time "30" has a "higher" priority than an Event object of type "D" and time "30" and therefore would be the next to be dequeued from the priority queue *eventListPQueue* even though both Event objects have the same time.

   Use a link-based implementation for our PQueue. We must use the provided Node class (Node.h, Node.cpp). Let's make sure our implementation of the PQueue class abides to its Public Interface described below (expressed in C++):

```
Class Invariant: The elements stored in this Priority Queue a

   // Description: Returns "true" is this Priority Queue is e
   // Time Efficiency: O(1)
   bool isEmpty() const;

   // Description: Inserts newElement in sort order.
   //              It returns "true" if successful, otherwise
   // Precondition: This Priority Queue is sorted.
   // Postcondition: Once newElement is inserted, this Priori
   bool enqueue(const Event& newElement);

   // Description: Removes the element with the "highest" pri
   //              It returns "true" if successful, otherwise
   // Precondition: This Priority Queue is not empty.
   bool dequeue();

   // Description: Retrieves (but does not remove) the elemen
   // Precondition: This Priority Queue is not empty.
   // Postcondition: This Priority Queue is unchanged.
```

```
    // Exceptions: Throws EmptyDataCollectionException if this
    Event peek() const throw(EmptyDataCollectionException);
```

5. An exception class called EmptyDataCollectionException. This class has been provided for us to use: "EmptyDataCollectionException.h and "EmptyDataCollectionException.cpp

   Please, refer to Interlude 3 of our textbook to learn how to use exceptions. Feel free to refer to other resources found on the Web regarding exceptions such as this web site.

Implementation - Public Interface

As in Assignment 1, copy and paste the above public interfaces into respective *.h and *.cpp files then design and implement the Queue and the PQueue classes such that they abide to their respective public interface.

As in Assignment 1, we cannot modify these public interfaces because our Assignment 2 will be marked using test scripts that are implemented based on these public interfaces and class descriptions given above.

We can add more methods to our public interfaces, as long as they are be private methods. If we make use of a "printing" method for testing purposes, let's make sure we comment calls to this method before we submit the code for our Assignment 2.

Implementation - Documentation

Let's make sure we include in our classes the documentation discussed in class:

- Comment header block containing: filename, class description, class invariant (if any), author, date of creation.
- A description, a precondition (if any) and a postcondition (if any) for each of the class method (public and private).
- All attributes (and methods) must be descriptively named. If needed, comments must be added to the attributes.
- Let's make sure our code satisfies the "good programming style" described on the GPS web page of our course web site.

Testing

Input File

Here is an input file to be used to test our simulation simulation.in

Contrary to what is stated in Propgramming Problem 6 of our textbook, our BankSimApp is not to open and read input files, but is to read input from the command line. This way, we can redirect input as follows:

```
uname@hostname: ~$ ./bsApp < simulation.in
```

where *bsApp* is the executable form of our BankSimApp.

Output Format

As indicated in Programming Problem 6, our BankSimApp must produce a very specifically formatted result. Let's make sure our BankSimApp prints on the computer monitor screen its result in exactly the same format as the results displayed in Programming Problem 6 (same layout, same words with same lower/upper cases, etc...).

Marking Scheme

Assignment 2 is worth 5% and marks are allocated as follows:

- Correct UML class diagram - 1%
  - Are the UML notation and syntax used in the diagram proper and correct?
  - Have the public interface of the data collection classes been included?
  - Does the UML class diagram represent what has been implemented?
- Correctness of solution and classes - 3%
- Design and implementation of data collection classes and their data structures - 0.5%
- Coding style - 0.5%

Submission

Assignment #2 is due Friday June 24 at 15:30.

We can use IDE's to develop our application, ***but before submitting our files, let's make sure our code compiles and executes in the environment we use in the lab: Ubuntu Linux command line and g++.***

We must submit the following files via CourSys:

1. UMLClassDiagram.pdf
2. BankSimApp.cpp
3. Event.h and Event.cpp
4. Queue.h and Queue.cpp

5. PQueue.h and PQueue.cpp
