COMP 2404 -- Assignment #3

Due: Tuesday, March 26, 2019 at 12:00 pm (noon)

Goal

You will modify your student auditing program from a previous assignment or from the base code to implement a simplified version of the Observer design pattern. You will implement two concrete monitor classes that observe changes to student GPA and the number of courses that students fail or from which they withdraw.

Learning Outcomes

With this assignment, you will:

- apply the OO concepts of inheritance and polymorphism
- work with virtual and pure virtual functions in C++
- implement a simplified version of the Observer design pattern

Instructions

1. Draw a UML diagram:

We are going to create a small inheritance hierarchy of monitor objects. There will be an abstract `Monitor` class, and two concrete sub-classes: the `GPAMonitor` class will track the students whose GPA falls below a minimum threshold, and the `FWMonitor` class will track the students who fail or withdraw from more than a preset number of courses.

We will implement the Observer design pattern with `Monitor` objects as the observers and the `Control` object (and indirectly its students) as the subject. Each type of concrete monitor will store of a collection of logs: the `GPAMonitor` object will document which students have a GPA below the preset minimum, and the `FWMonitor` object will document which students have failed or withdrawn from a number of courses higher than another preset value. Each log will be formatted as a string containing the student id and either the GPA or number of failed and withdrawn (FW) courses. Every time a student object is created and added to our student audit system, the monitor objects (the observers) will be notified of a change in that student by the `Control` object (the subject). The monitor objects will examine the student to see if it fits their criteria (either a low GPA or a high number of FW courses). If the student does fit the criteria, the monitor object will create a corresponding log and add it to its log collection. All the logs from both monitor objects will be printed to the screen at the end of the program.

To implement this, we will create a pure virtual `update()` function in the `Monitor` class, and the appropriate behaviour will be implemented in the concrete sub-classes. Update your UML diagram from a previous assignment (or create a new one) to represent the new classes in the program and their associations. The UML diagram that you submit should reflect the design of the entire program for this assignment.

2. Modify the List class

You will need to use a collection class for each `Student` object to store `Course` objects or object pointers. If you did not implement the `List` class in a previous assignment, you can adapt the `Array` class from the tutorials.

To prepare your program to work with the new `Monitor` classes, you will make the following changes to your `List` class (or `Array`, if you don't have a `List` class):

- implement the `computeGPA()` function that returns the average of all course grades (between 0 and 12) for the student, excluding the withdrawals; you will use the function prototype:
  `float computeGPA();`
- implement the `computeNumFW()` function that returns the number of courses that the student has failed or from which the student has withdrawn; you will use the function prototype:
  `int computeNumFW();`

3. Modify the Student and Course classes

To work with the new `Monitor` classes, you will make the following changes to the `Student` and `Course` classes.

You will add the following two member functions to the `Student` class:

- a `computeGPA()` function that returns the GPA for this student, using a function implemented in a previous step
- a `computeNumFW()` function that returns the number of FW courses for this student, using a function implemented in a previous step

You will also modify the `Student` print function to print the student's GPA.

You may need to add getter functions in the `Student` class for the student id and in the `Course` class for the course grade.

4. Implement the Monitor classes

You will create a new `Monitor` abstract class that serves as the base class for the observers in the Observer design pattern. This class will contain the following:

- a data member that stores a collection of logs; you can use a STL `vector` of strings for this collection; we used the `vector` class in our in-class examples on polymorphism
- a pure virtual `update()` function that has the following prototype: `void update(Student*);`
- a `printLogs()` member function that prints the collection of logs to the screen

You will create a new `GPAMonitor` concrete class that derives from the `Monitor` class. This class will contain the following:

- a data member that represents the minimum threshold for GPAs to be flagged; when the `GPAMonitor` object detects a student with a GPA below this minimum threshold, it will create a new log
- a constructor
- an implementation of the `update(Student* stu)` function that checks if the given student's GPA is below the minimum threshold; if it is, then the function will create a new log documenting the student id and the corresponding GPA, and it will add the new log to its collection

You will create a new `FWMonitor` concrete class that derives from the `Monitor` class. This class will contain the following:

- a data member that represents the threshold for flagging a student's number of FW courses; when the `FWMonitor` object detects a student with a number of FW courses greater than this threshold, it will create a new log
- a constructor
- an implementation of the `update(Student* stu)` function that checks if the given student's number of FW courses is greater than the threshold; if it is, then the function will create a new log documenting the student id and the corresponding number of FW courses, and it will add the new log to its collection

5. Modify the Control class

You will modify the `Control` to serve as the subject in the Observer pattern, and work with the new `Monitor` classes as follows:

- add a data member to store a collection of `Monitor` object pointers; you can use a STL `vector` for this
- implement a new member function to notify the monitor objects when a new student is created; the function prototype will be: `void notify(Student* newStu);` this function will loop over the collection of monitors and call the `update()` function on each monitor object, using the new student as parameter; this will ensure that, if the new student meets the criteria of any of the monitor objects, new logs will be created to flag the student

  NOTE: DO NOT simply store two monitor objects! The Observer pattern requires a collection that can store any number of observers.
- modify the `Control` constructor so that:
  - it dynamically creates a new `GPAMonitor` object with a minimum bound of 3.0, and it adds this new object to the collection of monitors
  - it dynamically creates a new `FWMonitor` object with a threshold of 2, and it adds this new object to the collection of monitors
- modify the `Control` destructor so that it loops over every monitor object in its collection and calls the `printLogs()` function on it; the destructor should also deallocate the memory for the monitor objects
- modify the `launch()` function so that it calls the `notify()` function when a new student is added to storage

6. Test the program

- You will provide the `in.txt` file with sufficient datafill for a minimum of 15 different students, each with at least 5 different courses. The nature of the data and its ordering in the file must **thoroughly** test your program. If parts of your program cannot be adequately tested because of inadequate datafill, you could lose 100% of the marks for some marking components.
- Check that the student and course information is correct when the storage is printed out.
- Check that the monitor logs are correct when they are printed out at the end of the program.
- Make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used. Use `valgrind` to check for memory leaks.

Constraints

- your program must follow correct encapsulation principles, including the separation of control, UI, and entity object functionality
- do not use any classes, containers, or algorithms from the C++ standard template library (STL), except in the two instances explicitly permitted for this assignment
- do not use any global variables or any global functions other than `main()`
- do not use structs; use classes instead
- objects must always be passed by reference, not by value
- your classes must be thoroughly documented in every class definition
- all basic error checking must be performed
- existing functions must be reused everywhere possible

Submission

You will submit in *cuLearn*, before the due date and time, the following:

- a UML class diagram (as a PDF file), drawn by you with a drawing package of your choice, that corresponds to the entire program design
- one **`tar`** or **`zip`** file that includes:
  - all source, header, and data files, including the code provided
  - a Makefile
  - a readme file that includes:
    - a preamble (program and revision authors, purpose, list of source/header/data files)
    - compilation. launching, and operating instructions

NOTE: Do **not** include object files, executables, swap files, or duplicate files in your submission.

Grading (out of 100)

Marking components:

- 20 marks: correct UML diagram
  - 8 marks: correct changes to `Control` and `Student`
  - 10 marks: correct attributes and operations in `Monitor` classes
  - 2 marks: correct associations between `Monitor` classes

  NOTE: there will be zero marks for the UML if it doesn't show **all** the classes in the program
- 15 marks: correct modifications to the `List` class
  - 9 marks: correct implementation of `computeGPA()` function
  - 6 marks: correct implementation of `computeNumFW()` function
- 10 marks: correct modifications to the `Student` class
  - 4 marks: correct implementation of `computeGPA()` function
  - 4 marks: correct implementation of `computeNumFW()` function
  - 2 marks: correct change to the print function
- 30 marks: correct implementation of the `Monitor` classes
  - 6 marks: correct definition of `Monitor` class
  - 4 marks: correct implementation of `Monitor` print logs function
  - 2 marks: correct implementation of `GPAMonitor` constructor
  - 8 marks: correct implementation of `GPAMonitor` update function
  - 2 marks: correct implementation of `FWMonitor` constructor
  - 8 marks: correct implementation of `FWMonitor` update function
- 25 marks: correct modifications to the `Control` class
  - 2 marks: correct changes to class definition
  - 6 marks: correct changes to constructor
  - 7 marks: correct changes to destructor
  - 4 marks: correct changes to `launch()` function
  - 6 marks: correct implementation of `notify()` function

Execution requirements:

- all marking components must be called, and they must execute successfully to receive marks
- all data handled must be printed to the screen for marking components to receive marks

Deductions:

- Packaging errors:
  - 10 marks for missing Makefile
  - 5 marks for a missing readme
  - 10 marks for consistent failure to correctly separate code into source and header files
  - 10 marks for bad style or missing documentation
- Major programming and design errors:
  - 50% of a marking component that uses global variables, global functions, or structs
  - 50% of a marking component that consistently fails to use correct design principles
  - 50% of a marking component that uses prohibited library classes or functions
  - 100% of a marking component that is *replaced* by prohibited library classes or functions
  - 50% of a marking component where unauthorized changes have been made to the base code
  - up to 10 marks for memory leaks, depending on severity
- Execution errors:
  - 100% of a marking component that cannot be tested because it doesn't compile or execute in VM
  - 100% of a marking component that cannot be tested because the feature is not used in the code
  - 100% of a marking component that cannot be tested because data cannot be printed to the screen
  - 100% of a marking component that cannot be tested because insufficient datafill is provided