COMP 2404 -- Assignment #2

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

Goal

You will be modify your student auditing program from either Assignment #1 or from the base code to separate the architecture into control, view, and entity objects.

Learning Outcomes

With this assignment, you will:

- practice the correct design of an object-oriented program
- use UML to document the design

Instructions

1. Draw a UML diagram:

Using a drawing package of your choice, draw a UML class diagram for your code from Assignment #1 or for the base code. Remember, functions are not classes, so `main()` is not a class! Now add a `Control` class that will be responsible for all control flow, and a `View` class to deal with all user I/O. Think about what functions will be required in each class. You must keep adding to your diagram as you modify the code in each of the steps below. Your UML diagram should reflect the design of the entire program for this assignment.

2. Implement the Control class

You will create a new `Control` class that implements the control flow from the `main()` function. The `Control` class will contain:

- a data member for the `Storage` object that used to be declared in `main()`
- a data member for a new `View` object that will be responsible for user I/O
- a `launch()` member function that implements the program control flow and does the following:
  - use the `View` object to display the main menu and read the user's selection, until the user chooses to exit
  - if required by the user:
    - use the `View` object to read in all the student and course information,
    - create a new dynamically allocated `Student` object, containing the corresponding dynamically allocated `Course` objects
    - add the new student to storage using existing functions
  - use the `View` object to print the content of the storage to the screen at the end of the program

The `Control` class will perform all user I/O using the `View` class. It will not interact with the user directly.

You will change the `main()` function so that its only responsibility is to declare a `Control` object and call its `launch()` function.

3. Implement the View class

You will create a new `View` class that is responsible for interacting with the user. The `View` class will contain:

- a member function for displaying the main menu and reading the user's selection
- a member function for reading the student id
- a member function for reading all the information from the user about one course
- a member function for printing out the storage; this function will take a `Storage` object as parameter by reference, and it will use *delegation*, as seen in Tutorial #3, to ask the `Storage` class to print to the screen

Except for printing the storage at the end of the program, only the `View` class will interact with the user. You must change the program so that user I/O goes through this class.

After the `Control` and `View` classes are correctly implemented, your code should have no global functions other than `main()`.

4. Modify the Course class

You will modify the `Course` class to add a new member function that compares two courses. The new member function will have the prototype `bool lessThan(Course*)`, and it will compare the given parameter with the `Course` object on which the function is called. The lesser of two courses is the one with the lower course code. If two courses have the same course code, the lesser course is the one with the lesser term.

This function requires the new `Course` class data member called `term` from Assignment #1.

5. Implement the List class

You will create a new `List` class that holds a singly linked list of `Course` pointers. You will implement the linked list as we saw in class, with no dummy nodes. The `List` class will contain the following:

- a data member for the head of the list
- a data member for the tail of the list
- a constructor
- a destructor to clean up the dynamically allocated memory
- a member function with the prototype `void add(Course*)` that adds a new course to the list
  - the new course will be added in its correct position in the list, in ascending (increasing) order by course, using the `Course` class's `lessThan()` function
  - you will need to implement a `Node` class, as we saw in class
- a `print()` member function that prints the courses to the screen
  - after all data is printed, indicate which course is at the head and which course is at the tail

Change the `Student` class to use a new `List` object instead of the course array. It will no longer need to track the number of courses. There should be zero impact on existing classes because of this change.

6. Test the program

- You will modify the `in.txt` file so that it provides sufficient datafill for a minimum of 15 students, each with at least 5 courses. The ordering in the file of each student's courses must be such that the program is thoroughly tested.
- Check that the student and course information is correct when the storage is printed out.
- 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 the 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)
- 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:

- 35 marks: correct UML diagram
  - 27 marks: correct classes, attributes, and operations
  - 8 marks: correct associations between classes
- 20 marks: correct implementation of `Control` class
  - 2 marks: correct definition of new data members
  - 18 marks: correct implementation of `launch()` function
- 10 marks: correct implementation of `View` class
  - 2 marks: correct implementation of main menu function
  - 2 marks: correct implementation of read student id
  - 4 marks: correct implementation of read course information
  - 2 marks: correct implementation of print storage function
- 5 marks: correct change to `Course` class
  - 5 marks: correct implementation of `lessThan()` function

- 25 marks: correct implementation of `List` class
  - 4 marks: correct class definition
  - 2 marks: correct implementation of constructor
  - 5 marks: correct implementation of destructor
  - 10 marks: correct implementation of `add()` function
  - 4 marks: correct implementation of `print()` function
- 5 marks: correct changes to `Student` class for use of `List` class

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