COMP 2404 -- Assignment #3

Due: Tuesday, March 24, 2020 at 12:00 pm (noon)

Goal

For this assignment, you will write a C++ program to implement a simplified version of the Observer design pattern. You will implement two concrete "event logger" classes that observe changes to a library, specifically the events of checking in and checking out of books.

Learning Outcomes

With this assignment, you will:

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

Instructions

1. Draw a UML class diagram:

Event logging is a common practice in a software application, where a portion of the code keeps track of certain events that occur during the execution of the application. The goal of logging events is so that an administrative user can later view when and where these events occurred in the running of the application, and see how the application was used.

We are going to create a small inheritance hierarchy of event logger classes, which will be responsible for logging (tracking) the events in which our program is interested. In this case, these events will be the checking in and the checking out of books in a library. There will be an abstract `Logger` class, and two concrete sub-classes: the `CheckInLogger` class will track the checking in of books, and the `CheckOutLogger` will keep track of the checking out of books. Our design will use the Observer pattern, which is explained in the course material, section 3.3.

We will implement the Observer design pattern with `Logger` objects as the observers and the `Library` object as the subject. Each logger class will store a collection of books: the `CheckInLogger` will track which books have been checked into the library, and the `CheckOutLogger` will track which books have been checked out. Every time the end user chooses to either check out or check in a book, all the logger objects (the observers) will be notified of a change in the status of that book by the `Library` object (the subject). The logger objects will examine the book to see if it fits their criteria (either the book has been checked in or checked out). If the book *does* fit the criteria, the logger object will add the book to its collection. The book collections from both logger 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 `Logger` class, and the appropriate behaviour will be implemented in the concrete sub-classes.

You will begin by drawing a UML class diagram, using a drawing package of your choice, and your diagram will follow the conventions established in the course material covered in section 2.3. A *partial* diagram of the program has been provided for you in Figure 1 as a starting point. Your diagram will represent all the classes in your program, including the control, view, and entity classes in the program, as well as the hierarchy of logger classes. Your diagram will show all attributes, all operations (including the parameters and their role such as in, out, inout), and all associations between classes, including directionality and multiplicity, where applicable. As always, do not show collection classes, as they are *implied* by multiplicity, and do not show getters, setters, constructors, or destructors.

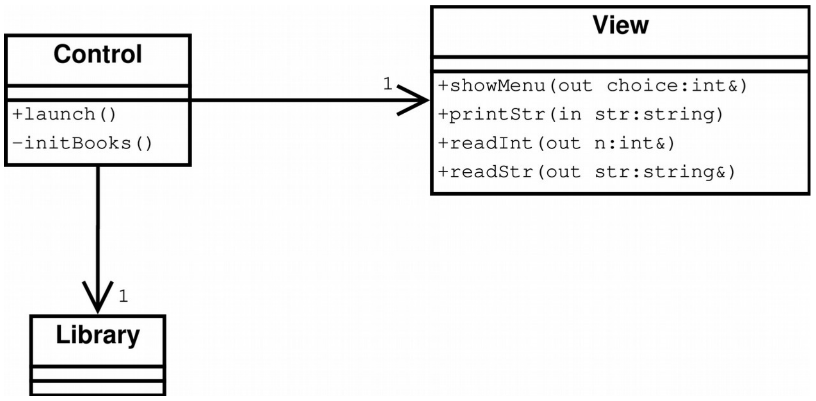

Figure 1 - Partial UML class diagram

2. Modify the `Book` and `List` classes

You will begin by modifying two of the classes that we worked on during the lectures. Download the coding examples posted in cuLearn, in section 3.1, programs #6 and #7. From these two examples, you will keep the `Book` class from program #6, and the `List` class from program #7. You can delete all other files, except for one Makefile that you will modify to use for this assignment.

(1) You will make the following changes to the `Book` class:
  - (a) add a getter member function for the book id
  - (b) add a getter member function for the author
  - (c) add a boolean data member called `checkedIn` to indicate whether or not the book is currently checked into the library
  - (d) modify the constructor so that it initializes the `checkedIn` flag to true
  - (e) add a getter member function for the `checkedIn` data member; to prevent later confusion, you should call this function `isCheckedIn()`
  - (f) add a `void checkIn()` member function that sets the `checkedIn` flag to true
  - (g) add a `void checkOut()` member function that sets the `checkedIn` flag to false
  - (h) modify the `print()` function so that it prints out all the data members, including the book id and whether or not the book is currently checked in

(2) You will make the following changes to the `List` class:
  - (a) change the `List` and `Node` classes so that they store `Book` pointers as data
  - (b) change the `add()` member function so that:
    - books are added in ascending order by author
    - the list does **not** allow the same book to be added twice; you can use pointer comparisons to verify this
  - (c) change the `del()` member function so that books are deleted by book id; you will have to change the parameter type to do this
  - (d) add a `Book* find(int id)` member function that returns the `Book` pointer currently in the linked list where the book id corresponds to the given `id` parameter; this member function will return null if the book id is not found
  - (e) add a `void cleanup()` member function that deallocates the memory for the data **only**; this function is necessary because we will have books stored in multiple lists, and we have to ensure that each book is deallocated only **once** at the end of the program; the `List` destructor should not deallocate the data, only the nodes

3. Implement the `Logger` classes

You will create a new hierarchy of `Logger` classes that track the checking in and checking out of books in the library. These classes also serve as the observers in our Observer design pattern. A skeleton of the `Logger` classes (without inheritance) has been provided for you in the `a3-posted.tar` file in *cuLearn*. You may keep all the `Logger` class definitions in the same header file, and all the function implementations in the same source file.

You will implement the `Logger` classes as follows:

(1) You will create a new `Logger` abstract class that serves as the base class for the other logger classes. The `Logger` class will contain the following:
   - (a) a string data member for the name of the logger object, and a default constructor that initializes this data member from a parameter
   - (b) a data member that stores the log information as a collection of books; this collection must be an object of the `List` class, as modified above
   - (c) a pure virtual `void update(Book*)` member function that takes as parameter a book whose `checkedIn` status has just been changed, either from true to false, or false to true
   - (d) a `void printLogs()` member function that prints to the screen the logger object's name and its logs (its collection of books)

(2) You will create a new `CheckInLogger` concrete class that derives from the `Logger` class. This class will contain the following:
   - (a) a constructor that initializes the logger name to "Checked-in", using *base class initializer syntax*
     - ▪ see the coding examples in section 3.2, program #1, for an example of this
   - (b) an implementation of the `void update(Book* b)` member function that does the following:
     - ▪ if the given book `b` has been checked in, it must be added to the `CheckInLogger` object's collection of books
     - ▪ if the given book `b` has been checked out, it must be removed from the `CheckInLogger` object's collection of books
     - ▪ you must reuse existing functions in both of these operations

(3) You will create a new `CheckOutLogger` concrete class that derives from the `Logger` class. This class will contain the following:
   - (a) a constructor that initializes the logger name to "Checked-out", using base class initializer syntax
   - (b) an implementation of the `void update(Book* b)` member function that does the following:
     - ▪ if the given book `b` has been checked out, it must be added to the `CheckOutLogger` object's collection of books
     - ▪ if the given book `b` has been checked in, it must be removed from the `CheckOutLogger` object's collection of books

4. Implement the `Library` class

You will create a new `Library` class that manages a collection of books. This class also serves as the subject in our Observer design pattern.

The `Library` class will contain the following:

(1) a data member that stores a master collection of books; you will use the modified `List` class for this

(2) a data member that stores the observers as a collection of `Logger` object pointers; you must use a C++ standard template library (STL) `vector` for this

(3) a destructor that:
   - (a) loops over the observers (every logger object in the collection) and calls its `printLogs()` function, and then deallocates the logger object
   - (b) deallocates all the books in the master collection; you will use an existing `List` member function for this

(4) a `void subscribe(Logger* logger)` member function that adds the given `logger` object to the collection of observers

(5) a `void notify(Book* b)` member function that loops over every logger object in the collection of observers, and calls its `update()` function with the given book parameter `b`

(6) a `void add(Book* b)` member function that adds the given book `b` to the collection of books, and notifies all logger objects of the change

(7) add a `Book* find(int id)` member function that searches the book collection for a book with the given identifier `id`, and returns either that book pointer or null if the book was not found

(8) a `void checkOut(Book* b)` member function that checks book `b` out of the library and notifies all the logger objects of the change

(9) a `void checkIn(Book* b)` member function that checks book `b` into the library and notifies all the logger objects of the change

(10) a `void print()` member function that prints to the screen all the books in the library

NOTE: The implementation of the above member functions **must** make use of existing functions everywhere possible.

5. Implement the `Control` class

You will create a new `Control` class that implements the control flow for the entire program. A skeleton of the `Control` class has been provided for you in the `a3-posted.tar` file in *cuLearn*, including a partial `launch()` member function, and the complete `initBooks()` member function. The `Control` class must also contain the following:

(1) a data member for the `Library` object that represents the library to be managed

(2) a data member for the `View` object that will be responsible for most user I/O
  (a) the `View` class is provided for you in the `a3-posted.tar` file

(3) a constructor that:
  (a) dynamically creates a new `CheckInLogger` object and subscribes it to the `Library` object
  (b) dynamically creates a new `CheckOutLogger` object and subscribes it to the library

(4) a `launch()` member function that does the following:
  (a) read a menu selection and, if required, a book id from the user
  (b) find the book with that book id in the library; you must perform basic error checking
  (c) check the book out of the library, if required by the user
  (d) check the book into the library, if required by the user
  (e) print out the content of the library once the user chooses to exit

6. Test the program

(1) run your program several times, each time checking in and checking out some books

(2) check that the book information is correct when it is printed at the end of the program

(3) check that the logs for both logger objects are correct when they are printed at the end of the program

(4) make sure that all dynamically allocated memory is explicitly deallocated when it is no longer used; use `valgrind` to check for memory leaks

Constraints

1. your program must follow correct encapsulation principles, including the separation of control, UI, entity, and collection object functionality
2. do not use any classes, containers, or algorithms from the C++ standard template library (STL), except where explicitly permitted in the instructions
3. do not use any global variables or any global functions other than `main()`
4. do not use structs; use classes instead
5. objects must always be passed by reference, never by value
6. functions must return data using parameters, not using return values, except for getter functions
7. existing functions must be reused everywhere possible
8. all basic error checking must be performed
9. all dynamically allocated memory must be explicitly deallocated
10. your classes must be thoroughly documented in every class definition, as discussed in the course material, section 1.3

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 and header 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 and launching instructions

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

Grading (out of 100)

Marking components:

- 25 marks: correct UML diagram
- 5 marks: correct changes to `Book` and `List` classes
- 20 marks: correct implementation of `Logger` classes
- 35 marks: correct implementation of `Library` class
- 15 marks: correct implementation of `Control` 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:

1. Packaging errors:
   (1) 10 marks for missing Makefile
   (2) 5 marks for a missing README
   (3) 10 marks for consistent failure to correctly separate code into source and header files
   (4) up to 10 marks for bad style or missing documentation
2. Major programming and design errors
   (1) 50% of a marking component that uses global variables, or structs
   (2) 50% of a marking component that consistently fails to use correct design principles
   (3) 50% of a marking component that uses prohibited library classes or functions
   (4) 100% of a marking component that is *replaced* by prohibited library classes or functions
   (5) 50% of a marking component where unauthorized changes have been made to the provided code
   (6) up to 100% of a marking component where Constraints listed are not followed
   (7) up to 10 marks for memory leaks
   (8) up to 10 marks for bad style
3. Execution errors
   (1) 100% of a marking component that cannot be tested because it doesn't compile or execute in VM
   (2) 100% of a marking component that cannot be tested because the feature is not used in the code
   (3) 100% of a marking component that cannot be tested because data cannot be printed to the screen
   (4) 100% of a marking component that cannot be tested because insufficient datafill is provided