COMP 2404 -- Assignment #2

Due: Tuesday, November 6, 2018 at 12:00 pm (noon)

Goal

You will modify your event management 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. 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 `Calendar` 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, create a new dynamically allocated `Event` object and add it to the calendar using existing functions
  - use the `View` object to print the content of the calendar 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 all the information from the user about one event
- a member function for printing out the calendar; this function will take the calendar as a parameter, and it will use *delegation*, as seen in Tutorial #3, to ask the `Calendar` class to print to the screen

Except for printing the calendar 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. **Implement the List class**

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

- a data member for the head of the list
- a constructor for initialization
- a destructor for cleaning up dynamically allocated memory
- a member function with the prototype `void add(Event*)` that adds a new event to the list
  - the new event will be added in its correct position in the list, in ascending order by date, using the `Date` class's `lessThan()` function
  - you will need to implement a nested `Node` class, as we saw in the course material
- a `print()` function for printing the events to the screen

Change the `Calendar` class to use a new `List` object instead of the event array. There should be zero impact on existing classes because of this change, except for minor changes to the `Calendar` class.

5. **FOR BONUS MARKS ONLY: Implement the formatting functions**

As explained in Tutorial #5, we need to implement some formatting functions in order to follow the correct principles of encapsulation and information hiding. We cannot have entity or collection classes printing data to the screen, and we cannot have the `View` class knowing the details of how a collection is stored or traversed.

You will modify the code as follows:

- rename the `print()` functions in the `Time`, `Date`, `Event`, `List`, and `Calendar` classes to the following:

  ```
  void format(string& outStr)
  ```
- write the `format()` functions so that they format the object data in the same way that it was printed to the screen, but instead of outputting, they place the formatted data into the `outStr` output parameter
  - you can use the `stringstream` class to help with the correct formatting (see Tutorial #5)
  - the `format()` functions will not print anything to the screen
  - the `outStr` parameter must be declared in the calling function
- change the code in the `Control` class to format the calendar data (through delegation) before asking the `View` to print it, using a new member function in the `View` class that simply prints a given string to the screen
- the calendar's `format()` function will ask the event list to format itself itself
- the list's `format()` function will loop through the events and format all the event data into one big string, using the `Event` class's formatting function
- each event will ask its `Date` object to format itself, which will in turn ask its `Time` object to do so

6. Test the program

- You will modify the `in.txt` file so that it provides sufficient datafill for a minimum of 15 events. The ordering of event dates and times in the file must be such that the program is thoroughly tested.
- Check that the event information is correct when the calendar is 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)
- 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

Grading (out of 100)

Marking components:

- 35 marks: correct UML diagram
  - 25 marks: correct classes, attributes, and operations
  - 10 marks: correct associations between classes
- 20 marks: correct implementation of `Control` class
  - 4 marks: correct definition of new data members
  - 16 marks: correct implementation of `launch()` function

- 10 marks: correct implementation of `View` class
  - 4 marks: correct implementation of main menu function
  - 4 marks: correct implementation of read event information function
  - 2 marks: correct implementation of print calendar function
- 30 marks: correct implementation of `List` class
  - 5 marks: correct class definition
  - 3 marks: correct implementation of constructor
  - 7 marks: correct implementation of destructor
  - 10 marks: correct implementation of `add()` function
  - 5 marks: correct implementation of `print()` function
- 5 marks: correct changes to `Calendar` class for use of `List` class
- **Bonus 10 marks**: correct implementation of `format()` functions

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