Raising Your Skill – Elevator Model

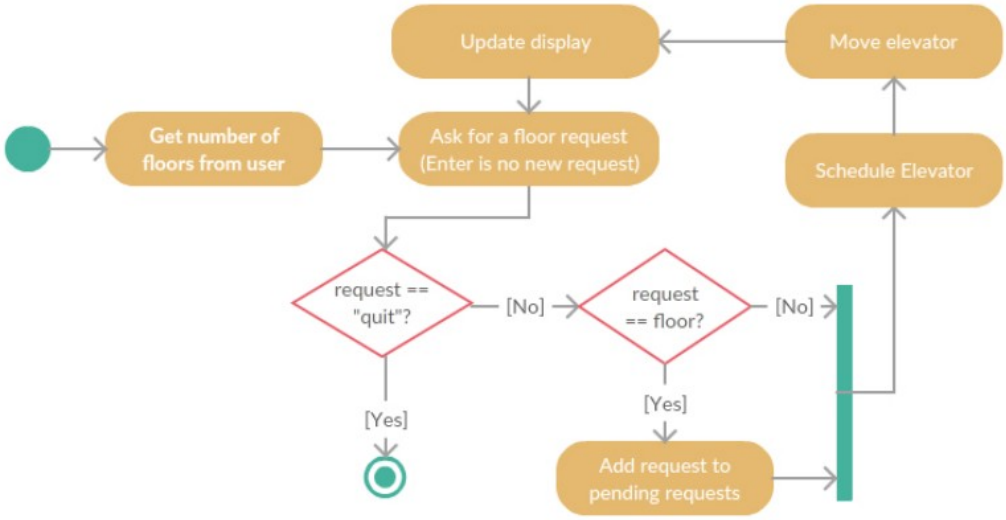
CSE 1325 – Fall 2016 – Homework #3
Due Thursday, September 15 at 8:00 am

Much of the technology that we take for granted is actually the result of deep research and iteratively improved software algorithms. In this homework, we will implement a *very* basic individual elevator model as a C++ class, and then (for bonus) iteratively improve the system, including inventing your own basic elevator scheduling algorithm.[1]

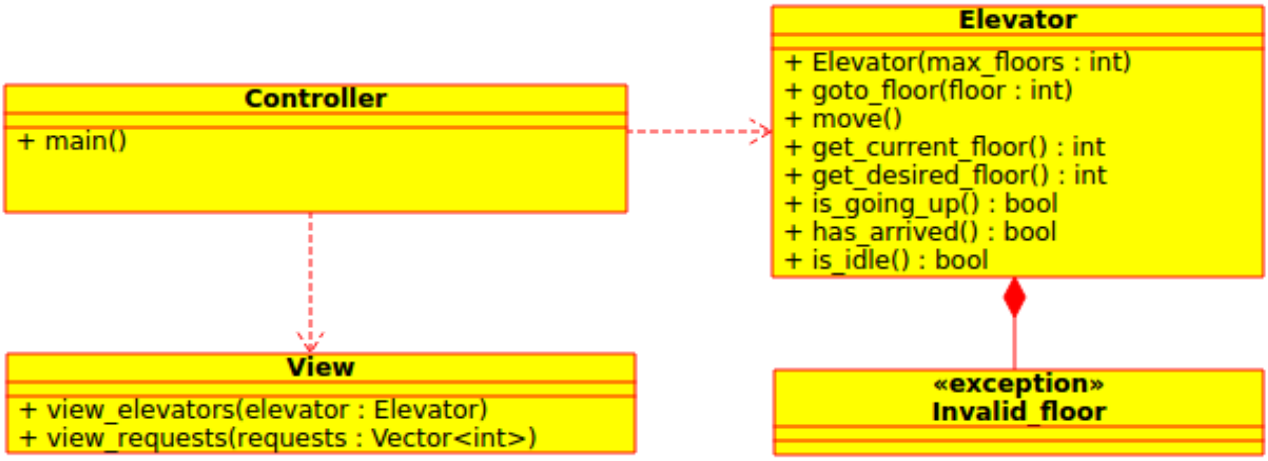
Requirements

The elevator model will follow the Model-View-Controller (MVC) pattern (to be covered shortly if not already), but since this is a 1-week homework exam, the files that comprise the Controller and View (and some of the Model) will be provided to you. For Full Credit, you will only need to write the Elevator model, elevator.cpp, which will represent a single elevator.

To the right is a basic UML activity diagram of the program logic.



Below is a simple UML class diagram of the Full Credit version of the elevator simulation. Note that Controller isn't actually a class for this level, although it will be for the Advanced Bonus (ahem).



1 This understanding of the inherent complexity of good scheduling algorithms may have the side effect of reducing your complaints when waiting for elevators. Or perhaps not.

The name of the class you must create is Elevator, and it does not inherit from any other classes (don't worry, we cover inheritance soon).

A constructor should be provided for this class that accepts the maximum number of floors that the elevator should cover, from floor 0 (the basement) to the top floor (the parameter to this constructor). It is permissible but not required to also provide a default constructor that assumes 9 floors + basement. If the controller directs an Elevator instance to move outside of that range, it should throw an Elevator::Invalid_floor exception (i.e., your Elevator class must both define this exception and throw it if directed to move to a floor that is outside its range).

Elevator must provide a public goto_floor(int) method, which directs the Elevator instance to which desired floor it should move when directed. This method returns nothing (i.e., it is “void”).

Elevator must also provide a public move() method, which directs an instance to move **one** floor in the direction necessary to reach the desired floor. Thus, multiple move() calls will usually be necessary for an Elevator instance to reach the desired floor specified by the most recent goto_floor(int) call. If the instance is already at the desired floor, calling move() does not result in changing floors. This method also returns nothing (it is also “void”).

The Elevator class should also include 5 public utility methods:

- int get_current_floor(), which should return the floor at which the elevator is currently positioned (this is a “getter” method).
- int get_desired_floor(), which should return the floor specified by the most recent goto_floor(int) call (also a “getter” method).
- bool is_going_up(), which should return true if the elevator is current ascending (i.e., get_current_floor() will increment on the next call to move()) or false if the elevator is currently descending.
- bool has_arrived(), which is true only if the elevator reached the floor specified by the most recent goto_floor(int) call *on the most recent move() call*. So, has_arrived() will return false while the elevator instance is moving, return true after the move() call in which it reaches the desired floor, and then return false again after the next move() call.
- bool is_idle() will return true only if the elevator instance did not change floors *on the last call* to move().

Note that Elevator is NOT responsible for scheduling requested floors or reporting its movement (indeed, Elevator should never generate text at all!). That is handled by the Controller and its peripheral functions and by View, respectively.

Given

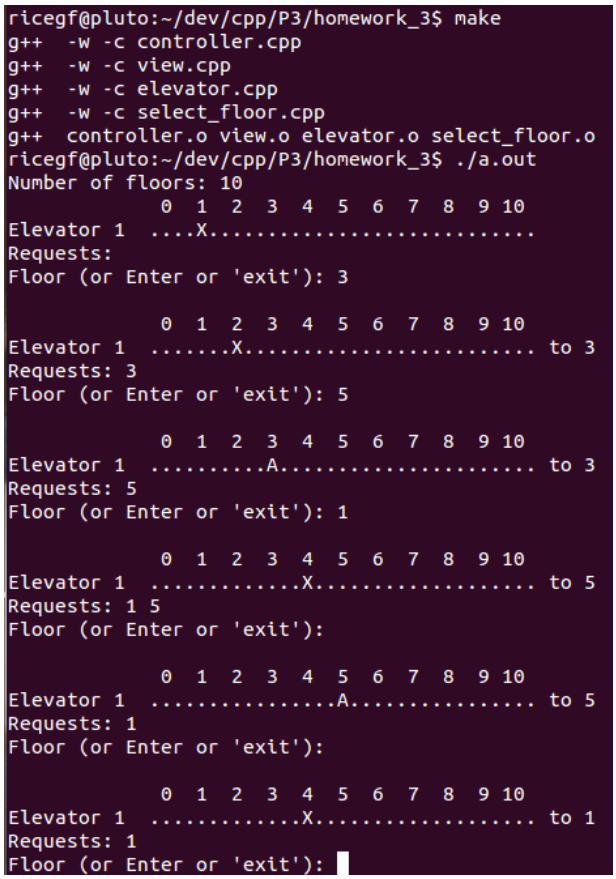
You will be provided a set of files, including an elevator.h (which maps to the class diagram above), and a controller.cpp and a view.cpp class representing the Controller and View. These will #include your Elevator class in the file called elevator.cpp. Compiling controller.cpp will include the other files, and the resulting executable will allow you to interactively test your class (a Makefile is provided to assist you in building – we'll cover Makefiles in more detail shortly).

Deliverables

You will provide a file called elevator.cpp and a screenshot named elevator.png (or .jpg or .gif) of your interactive testing session. The interactive testing session need not be comprehensive, but should include a full screen height's worth of useful testing.

Grading

- **Full Credit** – If you deliver a working elevator.cpp and a screenshot of the interactive test session. See example at right.



- **Bonus** – If you also deliver an automated test program called test_elevator.cpp (which may optionally be based on controller.cpp, but that likely does NOT actually use controller.cpp or view.cpp directly) that ***non-interactively*** verifies that your Elevator class operates correctly, along with a text copy of the test program output in a file named test_elevator.txt. NO input may be accepted by your test program. If all tests pass, your test program may output only one word: “pass”. If any tests fail, then your test program should print “fail” and then, for each failure, print helpful information to identify which test failed and why. **You must use the identical elevator.cpp class that you developed for Full Credit.**

- **Advanced Bonus** – If you modify the provided controller.cpp file to represent an actual Controller *class* that also implements the Singleton pattern to ensure exactly one elevator controller can exist in the system. Update your automated test program to include one or more tests verifying that additional instances of Controller cannot be created. Also create a new file, main.cpp, that implements a new main() function that creates an instance of Controller and then initiates elevator operation using it. **You must use the identical elevator.cpp class that you developed for Full Credit and Bonus.**

- **Extreme Bonus** – If you deliver a multicontroller.cpp program, which may optionally be based on or an extension (via inheritnce) of controller.cpp, that uses multiple instances of your (unmodified) Elevator class to model a building with *multiple* elevators, along with a screenshot of your interactive test session. **Your elevator.cpp class must be identical to that for the Full Credit, Bonus, and Advanced Bonus levels.** You will need to implement a reasonable control algorithm for scheduling multiple elevators. Below is an example of the Extreme Bonus (your View may differ).