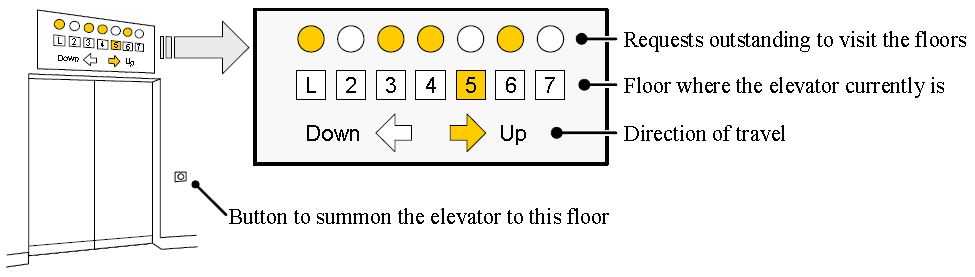
## Practical: Elevator Control

Consider developing a software system to control an elevator in a building. Assume that there will be a **button at each floor** to summon the elevator, and a set of **buttons inside** the **elevator car**—one button per floor to direct the elevator to the corresponding floor. Pressing a button will be detected as a pulse. When pressed, the button is illuminated. At each floor, there will be a **floor sensor** that is “on” when the elevator car is within 10 cm of the rest position at the **floor**.



The information panels will have two lamps representing each floor (see the figure above). A **square lamp** indicates that the car is at the corresponding floor, and a **round lamp** indicates that there is a request outstanding for the elevator to visit the corresponding floor. In addition, there will be two **arrow-shaped lamps** to indicate the current direction of travel. For example, in the figure above, the panel indicates that the elevator car is currently on the fifth floor, going up, and there are outstanding requests to visit the lobby, third, fourth, and sixth floor.

After the elevator visits a requested floor, the corresponding lamp on all information panels should be turned off. Also, the button that summoned the elevator to the floor should be turned off.

Let us assume that the outstanding requests are served so that the elevator will first visit all the requested floors in the direction to which it went first after the idle state. After this, it will serve the requests in the opposite direction, if any. When the elevator has no requests, it remains at its current floor with its doors closed.

**Problem1:** Suppose that you already have designed UML interaction and class diagrams. Your system will execute in a single thread, and your design includes the following classes:

ElevatorMain: This class runs an infinite loop. During each iteration it checks the physical buttons whether any has been pressed and reads the statuses of all the floor sensors. If a button has been pressed or the elevator car arrived/departed a floor, it calls the appropriate classes to do their work, and then starts a new iteration.

CarControl: This class controls the movement of the elevator car. This class has the attribute requests that lists the outstanding requests for the elevator to visit the corresponding floors. It also has three operations:

* addRequest(floorNum : int) adds a request to visit the floor floorNum;
* stopAt(floorNum : int) requests the object to stop the car at the floor floorNum. This operation calls DoorControl.operateDoors() to open the doors, let the passengers in or out, and close the doors.
* When operateDoors() returns, the CarControl object takes this as a signal that it is safe to start moving the car from the current floor (in case there are no pending requests, the car remains at the current floor).

InformationPanel: This class controls the display of information on the elevator information panel. It also has the attribute requests and these operations:

* arrivedAt(floorNum : int) informs the software object that the car has arrived at the floor floorNum.
* departed() which informs the object that the car has departed from the current floor.

OutsideButton: This class represents the buttons located outside the elevator on each floor that serve to summon the elevator. The associated physical button should be illuminated when pressed and turned off after the car visits the floor.  
This class has the attribute illuminated that indicates whether the button was pressed. It also has two operations:

* illuminate() requests the object to illuminate the associated physical button (because it was pressed);
* turnOff()requests the object to turn off the associated physical button (because the elevator car has arrived at this floor).

InsideButton: This class represents the buttons located inside the elevator car that serve to direct the elevator to the corresponding floor. The associated physical button should be illuminated when pressed and turned off after the car visits the floor. It has the same attributes and operations as the class OutsideButton.

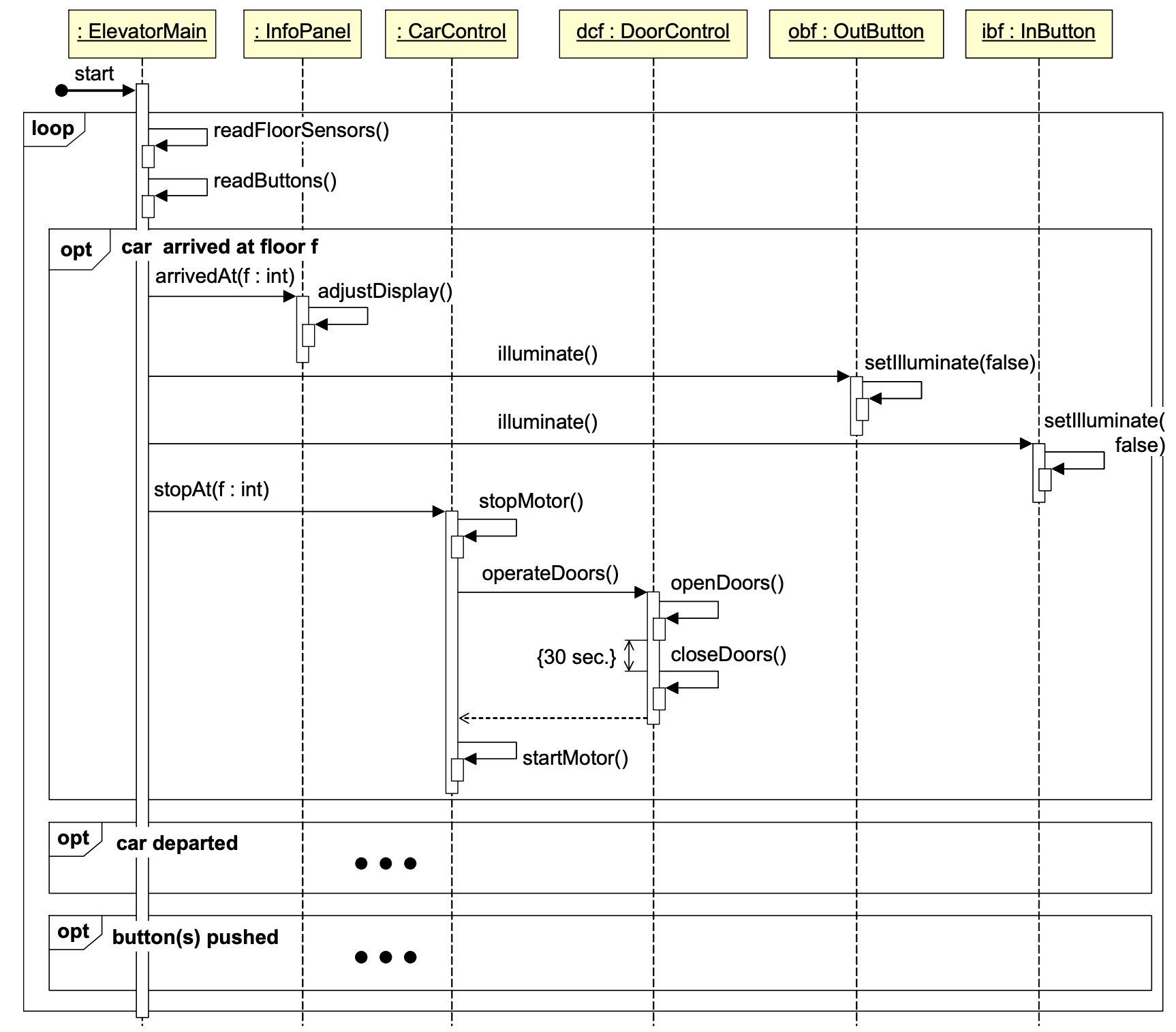
DoorControl: This class controls opening and closing of the elevator doors on each floor. This class has the Boolean attribute doorsOpen that is set true when the associated doors are open and false otherwise. It also has the operation:

* operateDoors() : void tells the software object when to open the doors. This operation sets a timer for a given amount of time to let the passengers in or out; after the timer expires, the operation closes the doors automatically and returns.

Note that some classes may have multiple instances (software objects), because there are multiple corresponding physical objects. For example, there is an information panel, outside button, and doors at each floor. In addition, we do not have a special class to represent a floor sensor that senses when the elevator car is in or near the rest position at the floor. The reason for this choice is that this system is single-threaded and the ElevatorMain object will notify the interested objects about the floor sensor status, so there is no reason to keep this information in a class dedicated solely for this purpose.

Draw the interaction and class diagrams corresponding to the design described above.

**Solution:** Part of the interaction diagram is shown in Figure H-14.



**Figure H-14.**  **Partial interaction diagram for the elevator problem**

The UML diagram shows only the interaction sequence for the case when the elevator car arrives at floor *f*. The other two cases, when the car departs from the current floor and when a physical button is pushed are left to the reader as exercise. Notice that we do use several “opt” choices rather than an “alt” choice, because the events (car-arrived, car-departed, button pushed) are not mutual alternatives. Although car-arrived and car-departed cannot happen at the same time, they should not be represented with an “alt.” Because it may be that neither one of them happened, it is ***not*** appropriate to show them as:

IF (car-arrived) THEN do-actions-when-car-arrived

ELSE do-actions-when-car-departed

**Problem2:** Consider the elevator control problem. Your task is to determine whether the Publisher-Subscriber design pattern can be applied in this design. Explain clearly your answer. If the answer is yes, identify which classes are suitable for the publisher role and which ones are suitable for the subscriber role. Explain your choices, list the events generated by the Publishers, and state explicitly for each Subscriber to which events it is subscribed to.

**Solution:** In summary, the Publisher will generate three types of events:

* arrivedAt(floorNum : int) informs a Subscriber that the elevator car arrived at floor floorNum.
* departed()informs a Subscriber that the car has departed from the current floor.
* pressed(floorNum : int) informs a Subscriber that the physical button associated with floor floorNum was pressed.

