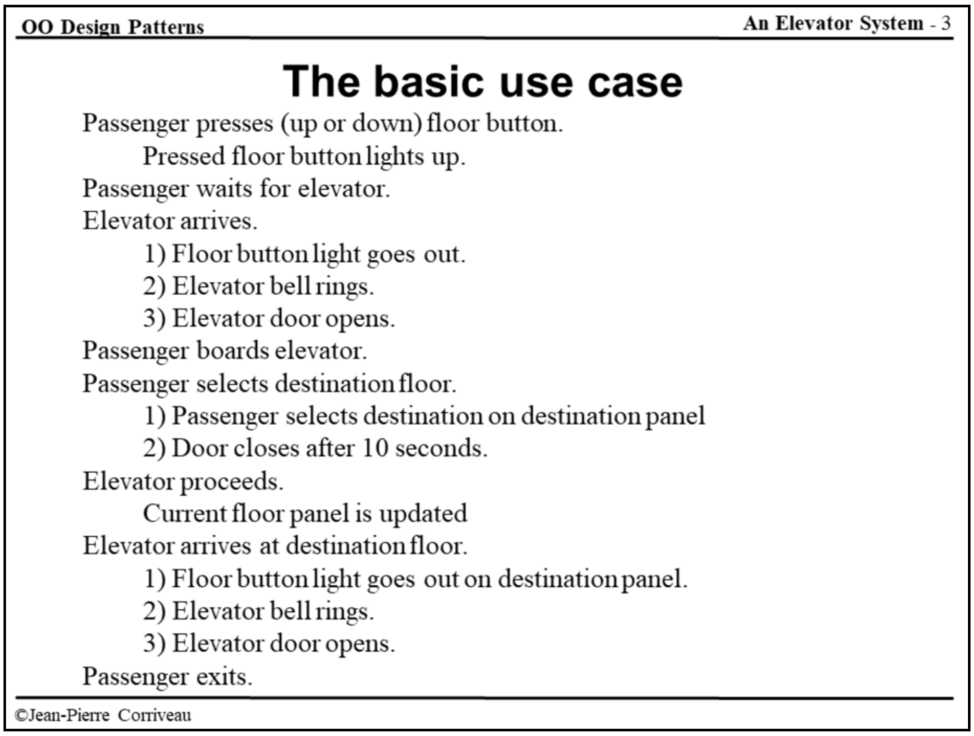
An Elevator System: Analysis and Design

An Elevator System

A building is serviced by a group of M elevators . On each of the N floors is a pair of buttons marked “up” and “down”. When a button is pressed it illuminates, and remains illuminated, until an elevator arrives to transport the waiting passengers. When the elevator arrives, it rings a bell, opens its doors for a fixed time (10 seconds) allowing people to exit or board, rings the bell again, closes its doors and proceeds to another floor. Once on board passengers select a destination floor using a panel of buttons; there is one button for every floor. The elevator has a display which shows passengers the current location (floor) of the elevator. There is also a pair of buttons on the elevator control panel marked “open door” and “close door”. These buttons can be used by a passenger to override the default timing of the doors. The door will remain open beyond its default period if the “open door” button is held depressed; the doors can be closed prematurely by pressing the “door close” button. Each elevator has a sensor which informs it when it arrives at a floor. (The elevator control system should ensure that the group of elevators services the requests expeditiously.)



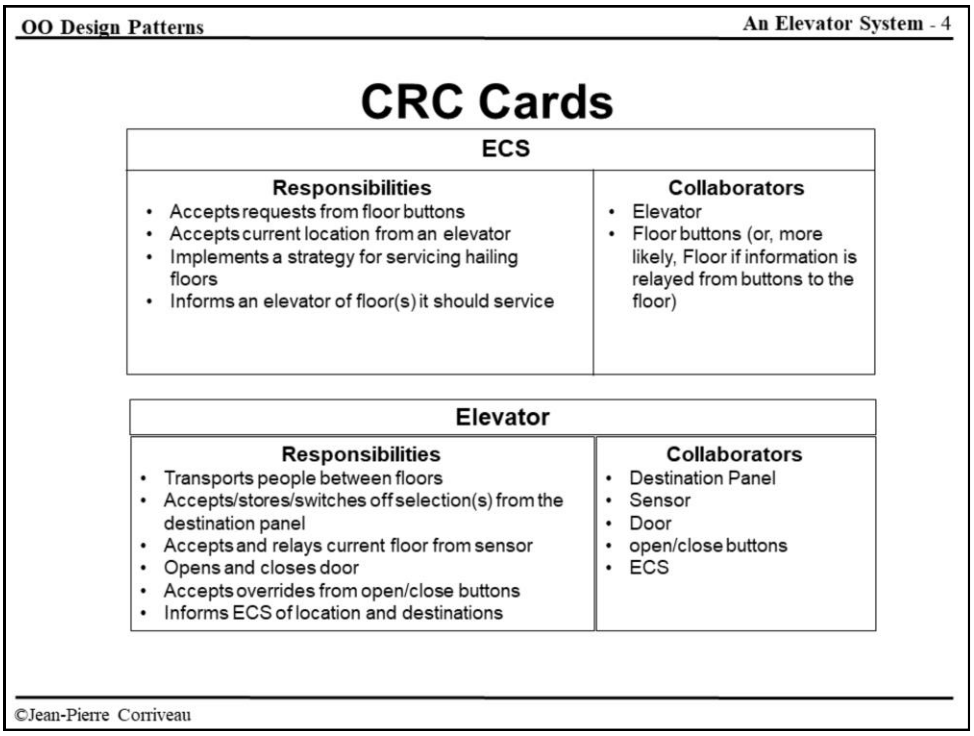
The basic use case

Passenger presses (up or down) floor button.
    Pressed floor button lights up.
Passenger waits for elevator.
Elevator arrives.
    1) Floor button light goes out.
    2) Elevator bell rings.
    3) Elevator door opens.
Passenger boards elevator.
Passenger selects destination floor.
    1) Passenger selects destination on destination panel
    2) Door closes after 10 seconds.
Elevator proceeds.
    Current floor panel is updated
Elevator arrives at destination floor.
    1) Floor button light goes out on destination panel.
    2) Elevator bell rings.
    3) Elevator door opens.
Passenger exits.

Recall that a basic use case describes end-to-end normal behavior. In this problem there is only 1 basic use case for normal behavior.

Let us assume that from analysis of requirements and the problem domain we have generated a use cases model and have also identified the following responsibilities for the elevator system:

- **each floor has button(s) to hail an elevator,**
- **each floor button can be on or off,**
- **pressing a floor button has the effect of creating a (possibly redundant) floor request,**
- **the system manages floor requests and instructs elevators of which floor to service,**
- **an elevator has a floor destination panel, a door, a bell, etc. and must control them,**
- **a bell rings, a floor destination button is on or off, etc.**
- **an elevator must accept requests to service a floor,**
- **an elevator must create floor requests from the selections on its floor destination panel,**
- **an elevator must update its floor destination panel AND its current floor display as it moves from one floor to the next,**
- **an elevator must control both the default and the overridable behavior of its door.**



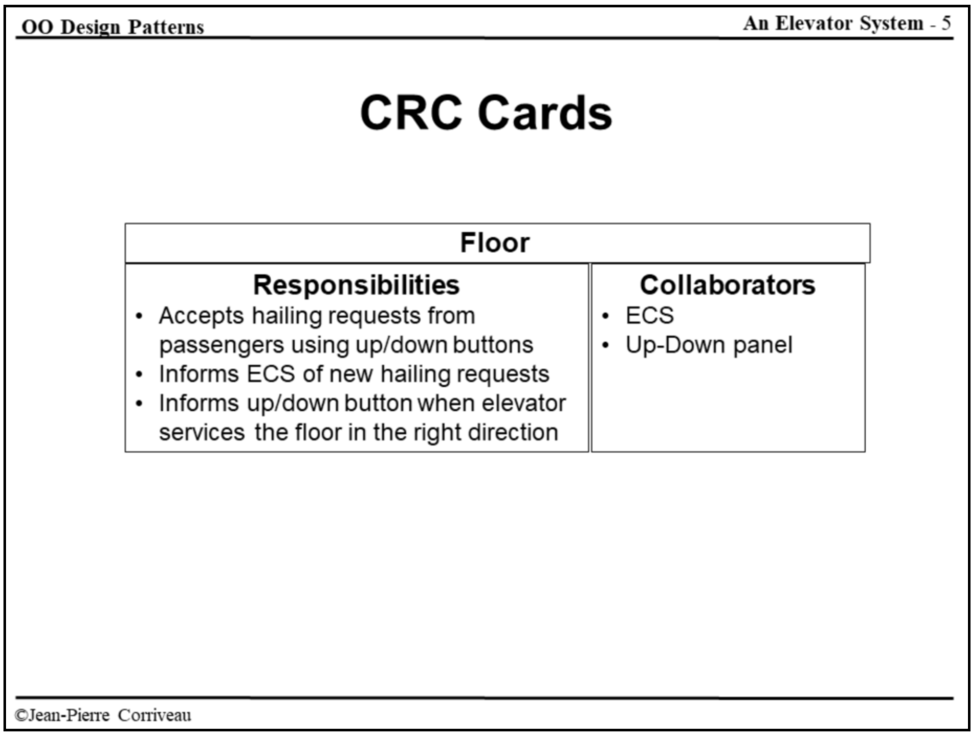
CRC Cards

ECS	
Responsibilities	**Collaborators**
• Accepts requests from floor buttons • Accepts current location from an elevator • Implements a strategy for servicing hailing floors • Informs an elevator of floor(s) it should service	• Elevator • Floor buttons (or, more likely, Floor if information is relayed from buttons to the floor)

Elevator	
Responsibilities	**Collaborators**
• Transports people between floors • Accepts/stores/switches off selection(s) from the destination panel • Accepts and relays current floor from sensor • Opens and closes door • Accepts overrides from open/close buttons • Informs ECS of location and destinations	• Destination Panel • Sensor • Door • open/close buttons • ECS

We have chosen to package the responsibilities into three key classes: ECS, Elevator, and Floor.

For more on CRC cards refer to
http://agilemodeling.com/artifacts/crcModel.htm



CRC Cards

Floor	
Responsibilities	**Collaborators**
• Accepts hailing requests from passengers using up/down buttons • Informs ECS of new hailing requests • Informs up/down button when elevator services the floor in the right direction	• ECS • Up-Down panel

Other CRC cards which are a bit more trivial:

Up/Down floor buttons

- switches on when pressed and informs Floor

- accepts order to switch off from Floor

Open/close buttons

- inform Elevator that they are pressed

Bell

- accepts order to ring from door

Door

- accepts order to open or close from Elevator

- tells Bell to ring

Sensor

- detects arriving at a floor

- informs Elevator and Elevator Display of upcoming floor

Destination panel button

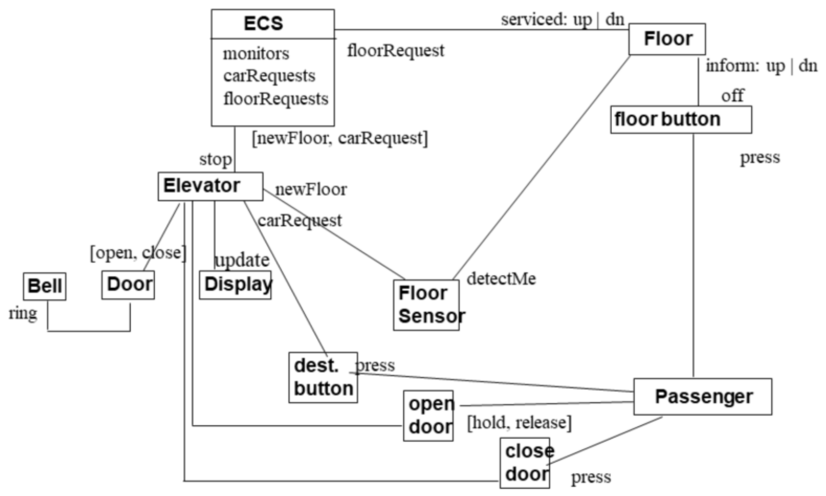
- switches on when pressed and informs Elevator

- accepts order to switch off from Elevator

Display

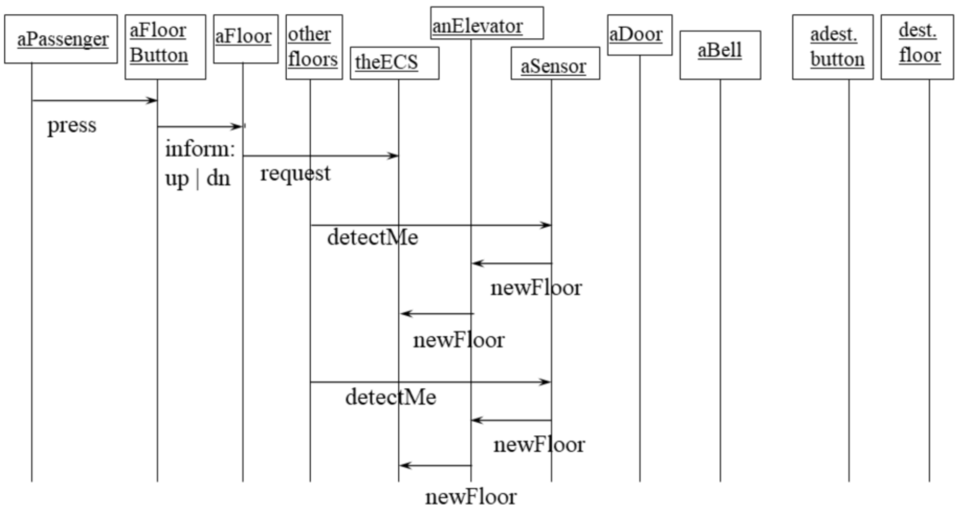
- accepts update message from Sensor

Elevator System: OOA structural model



The above diagram illustrates the associations between elements of the elevator system. The signals or events that these associations imply are also shown.

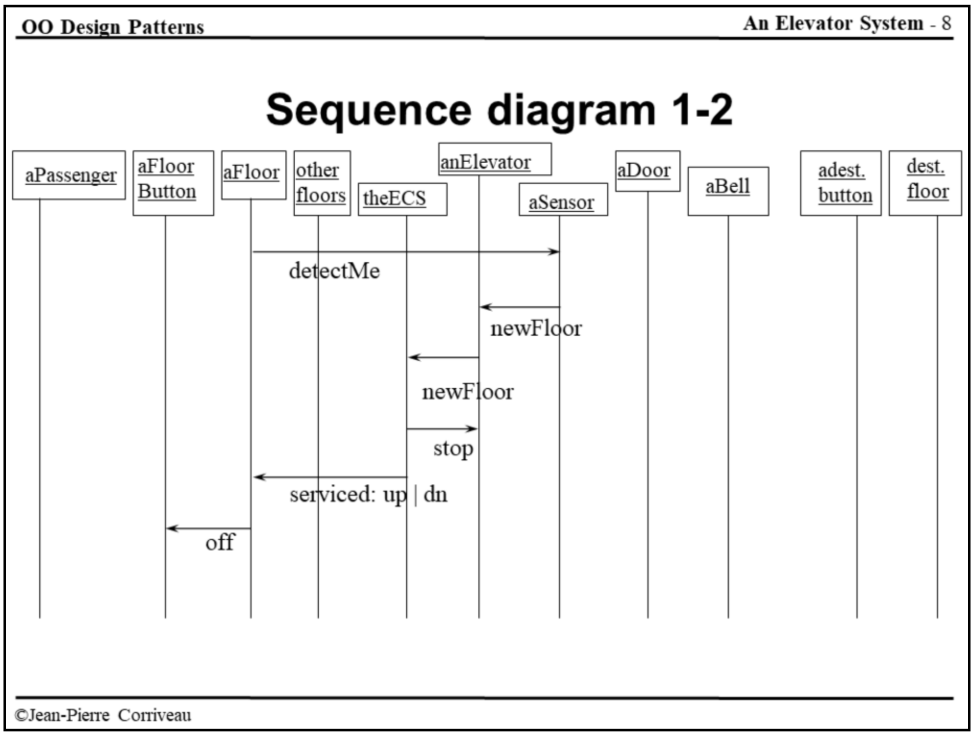
Sequence diagram 1-1



The sequence diagram (also called an event trace, or a message sequence chart, or a timing diagram..) is useful for capturing interactions, once and only once objects and responsibilities have been established: analysis from interaction diagrams typically leads to OOD considerations.

The above portion simply addresses the request being communicated to the ECS and an elevator traversing floors that need not service. We could include some of the other floors being serviced as well...

Notice ‘newFloor’ really means ‘number of next upcoming floor’

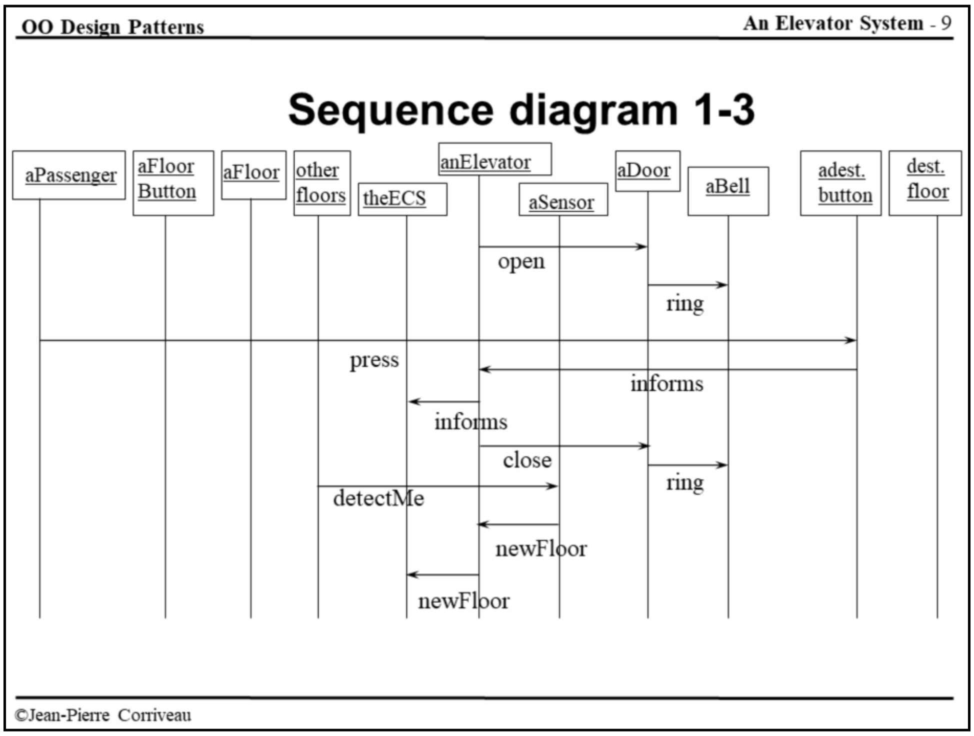


Sequence diagram 1-2

This portion of the Sequence diagram shows an elevator arriving at a floor that needs to be serviced. The ECS informs the elevator to stop at that floor. In fact, the ECS could have made the decision that this particular elevator should service that specific floor as early as the time of the floor request. But we abstract from the details which will be explored in OOD. For now, the ECS encapsulates the allocation strategy and merely tells (at one point in time or another) an elevator to stop at a floor.

The ECS informs the floor that is waiting for service that it is now being serviced in a particular direction.

In turn, the floor tells the relevant floor button to switch off.

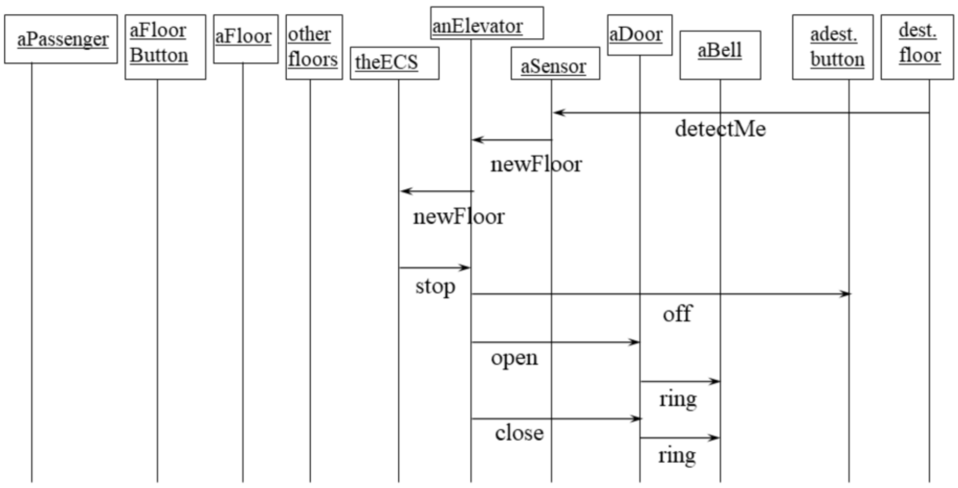


Sequence diagram 1-3

This portion of the sequence diagram shows what happens when the elevator arrives at a floor to service. For abstraction we are assuming an elevator will systematically inform the ECS of a requested destination.

The bottom of the sequence diagram again shows the elevator moving through floors that need not be serviced. Again an extension to this sequence diagram could show servicing some other floors.

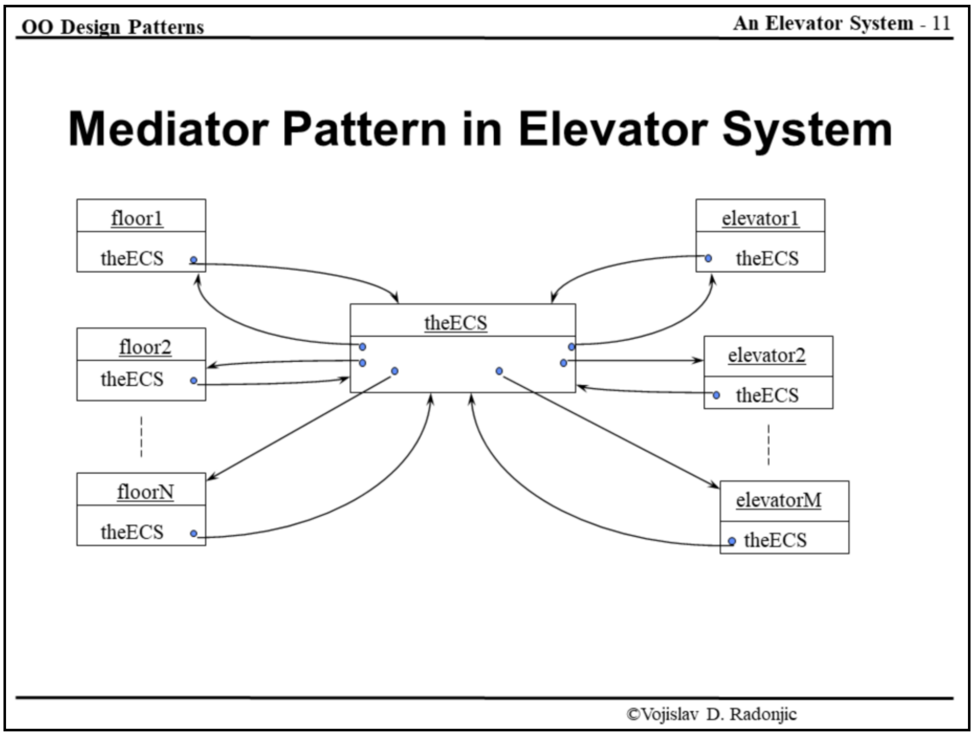
Sequence diagram 1-4



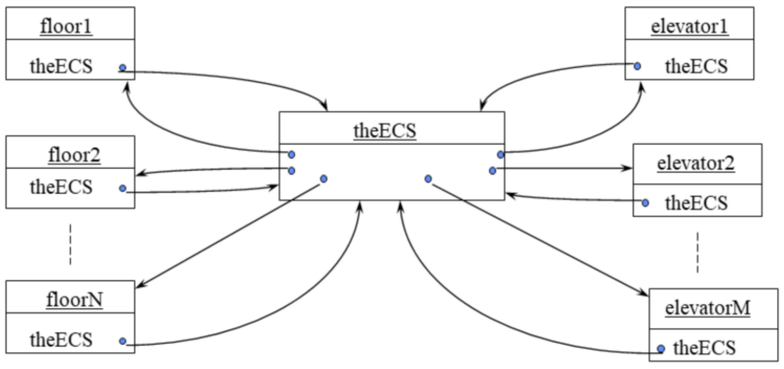
Finally, we get to the destination floor and open the door for passengers to get off.

This portion of the diagram would be more complex if we had to consider the fact that the destination floor could in fact also be a floor waiting for service. This could be simply be treated as an extension to the basic case, as previously suggested.

In fact, each extension should be developed in a portion of an sequence diagram and be documented as to where it is to be inserted in the sequence diagram(s) of the basic use cases.

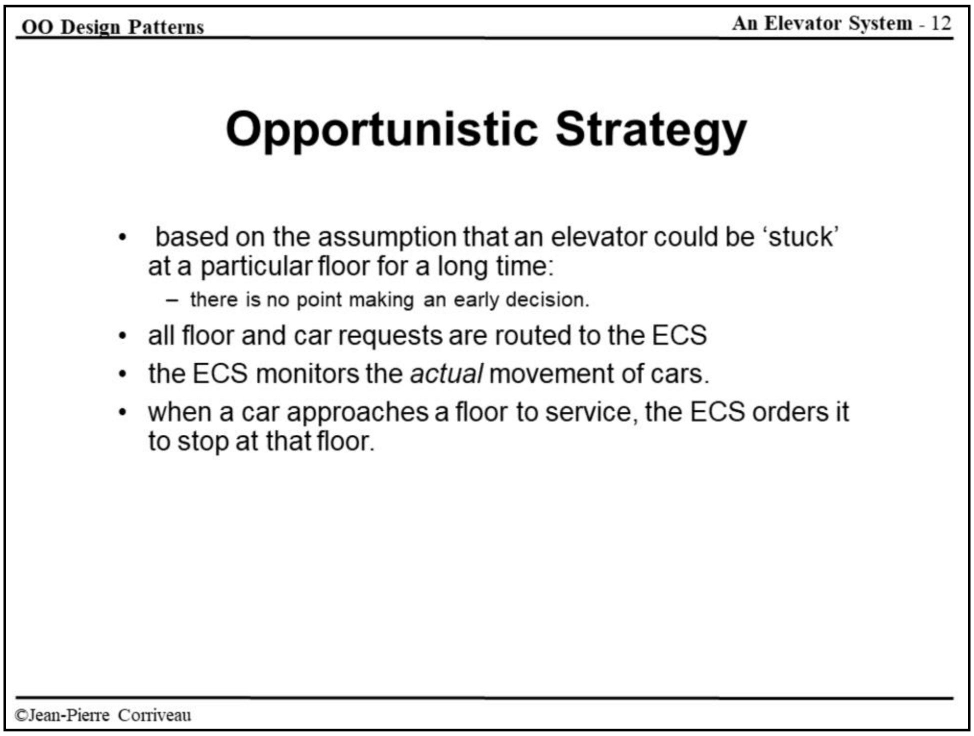


Mediator Pattern in Elevator System



©Vojislav D. Radonjic

The ECS encapsulates and localizes system level behavior. In doing so it decouples floors from elevators, and elevators from each other.



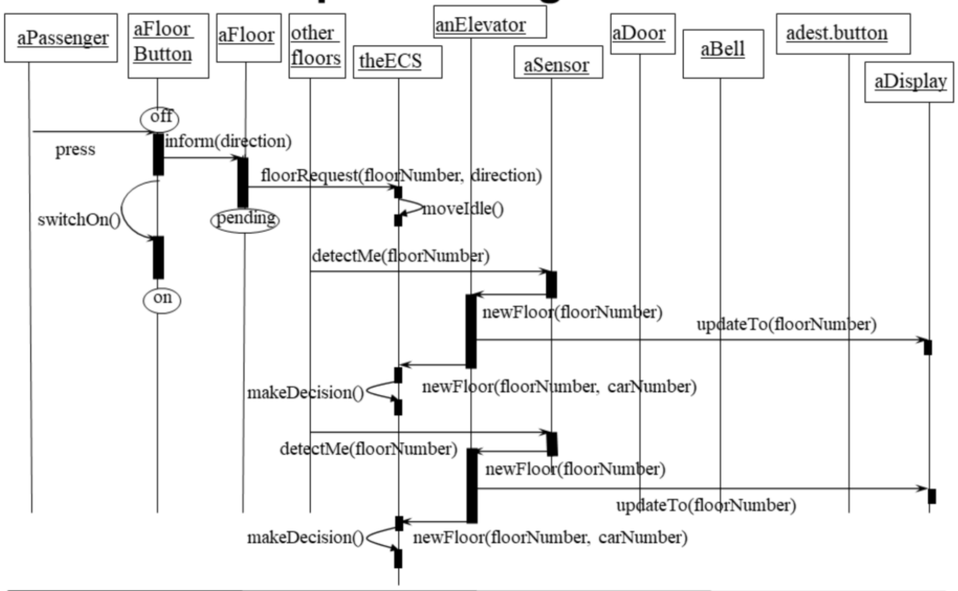
Opportunistic Strategy

- based on the assumption that an elevator could be 'stuck' at a particular floor for a long time:
  - there is no point making an early decision.
- all floor and car requests are routed to the ECS
- the ECS monitors the *actual* movement of cars.
- when a car approaches a floor to service, the ECS orders it to stop at that floor.

Because we do not know how many people will get on/off at a particular floor, any planning can be very hypothetical. Instead, we could just adopt the strategy that when a car approaches a floor, the ECS has enough time, once informed of this, to send a message to that car instructing it to stop at that floor.

OO Design Patterns — An Elevator System - 13

Sequence Diagram 1-1



©Vojislav D. Radonjic

The idea of OOD is working out the details:

Floor button has a method to react to pressing from user. The behavior of this method is explicit in the sequence diagram.

A floor has the responsibility of knowing which directions are waiting for services. Can use for example, two booleans (to be drawn on the OOD structural model): waitingForUp, waitingForDown.

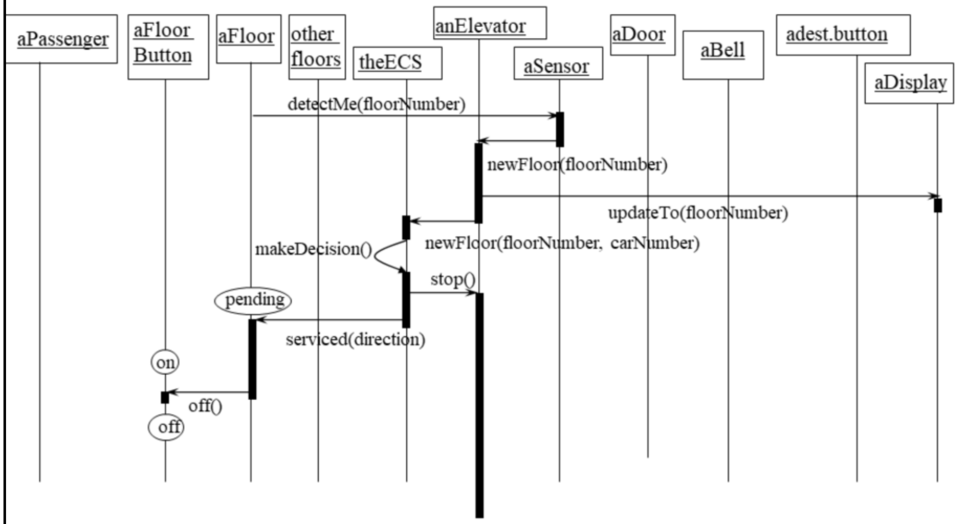
Superfluous presses in either directions are ignored by Floor, after consulting its 2 booleans.

Simultaneous presses are treated as two separate requests. The order should not matter.

Upon receiving a floor request, the ECS updates its pendingFloorRequest dictionary. We can still abstract from exact data structures early in OOD.

It is less useful to worry about the state of the ECS: thinking of it as reacting to messages is enough, we don’t really need an FSM and state variables for it yet.

Sequence Diagram 1-2



To understand the behavior of the ECS, the key idea is to realize that it **may** take a decision only upon receiving a notice of an elevator coming up to a floor (i.e. reception of message newFloor). It did not in the previous sequence diagram. In this sequence diagram, the ECS decides that the elevator that has just notified it, should service the floor this elevator is approaching. The makeDecision? method is responsible for i) determining if the notifying elevator should stop at the next floor, ii) informing the elevator if it needs to stop at the floor it is approaching and, in the case of a pending floor request, iii) informing the floor that it is serviced in a particular direction.

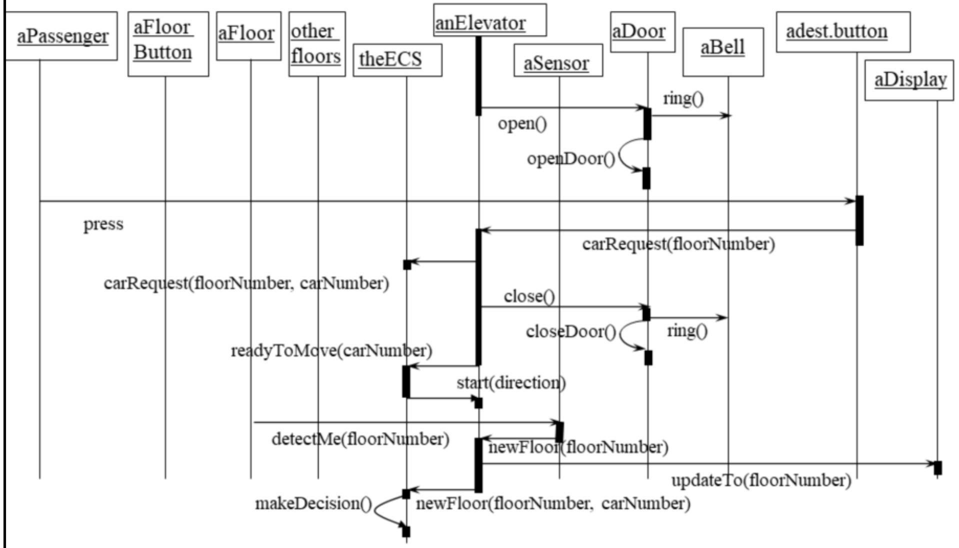
More details would explicit i) the decision process and ii) the use of the attributes of the ECS (e.g., pendingFloorRequests, directions, etc.).

The description of what is going inside an object can be done with an FSM or, often more simply, with pseudo-code. Both of these tools are refinable.

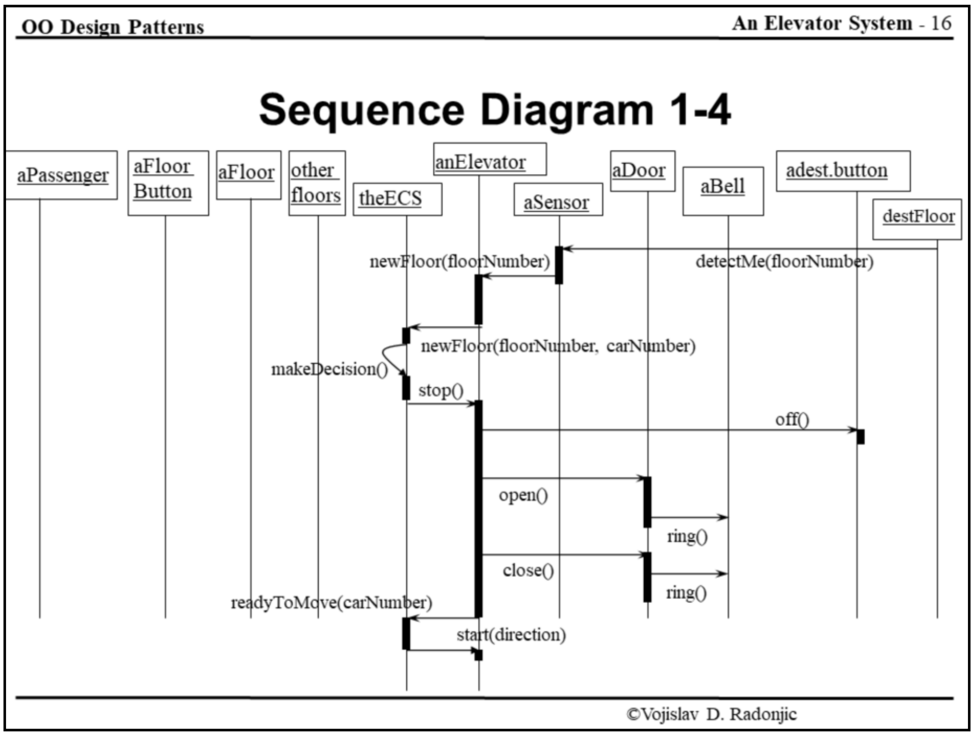
Notice how the messages got parameters. For example: each floor must pass its number to sensors so that ultimately each car informs the ECS of its number and next floor (assuming the sensor picks up the floor beacon BEFORE the elevator arrives at the floor).

Also notice how easy it is to get inconsistencies (e.g., in names of messages and parameters, in the order of parameters, etc.) between the different pages of an sequence diagram, and between different sequence diagrams... **Homogenization is the process of making everything consistent.**

Sequence Diagram 1-3



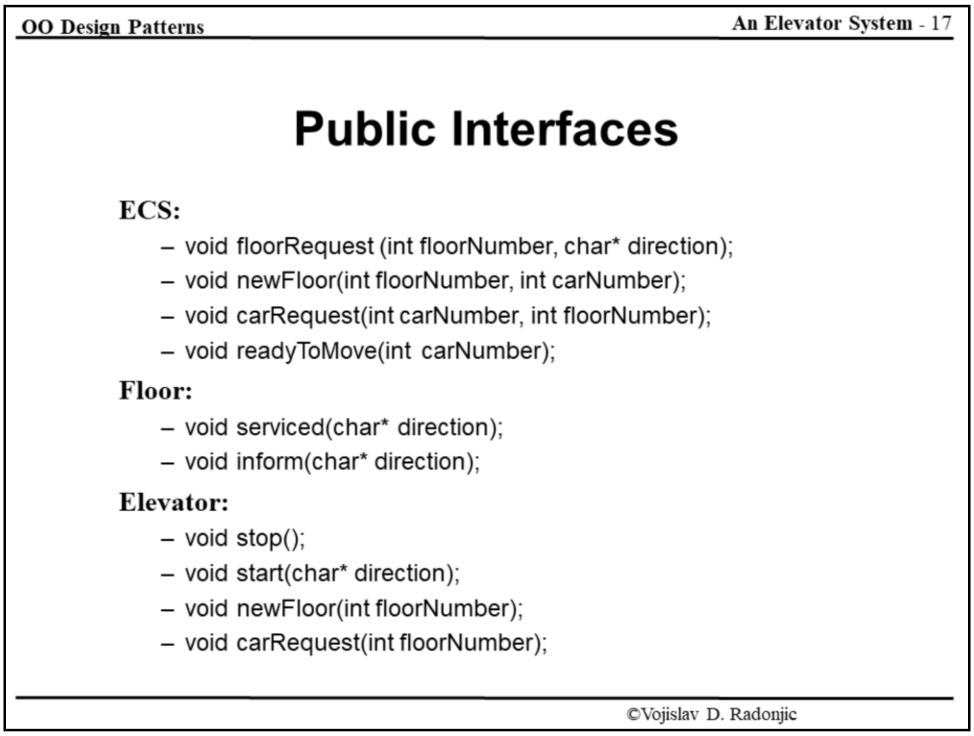
The sequence diagram makes explicit 1) how the door first rings the bell and then opens or closes itself, and 2) how the elevator informs theECS it is ready to move and how theECS then decides in what direction to move the elevator.



At this point we have a sequence diagram for the basic use case.

Unless objects have been added, joined or removed, updating the structural model merely consists of updating the names of the messages, if necessary.

At this point the elevator problem is too small to illustrate the importance of subsystems. However, does this set of sequence diagrams suggest any particular separation of objects in subsystems? What about all the mechanical parts?



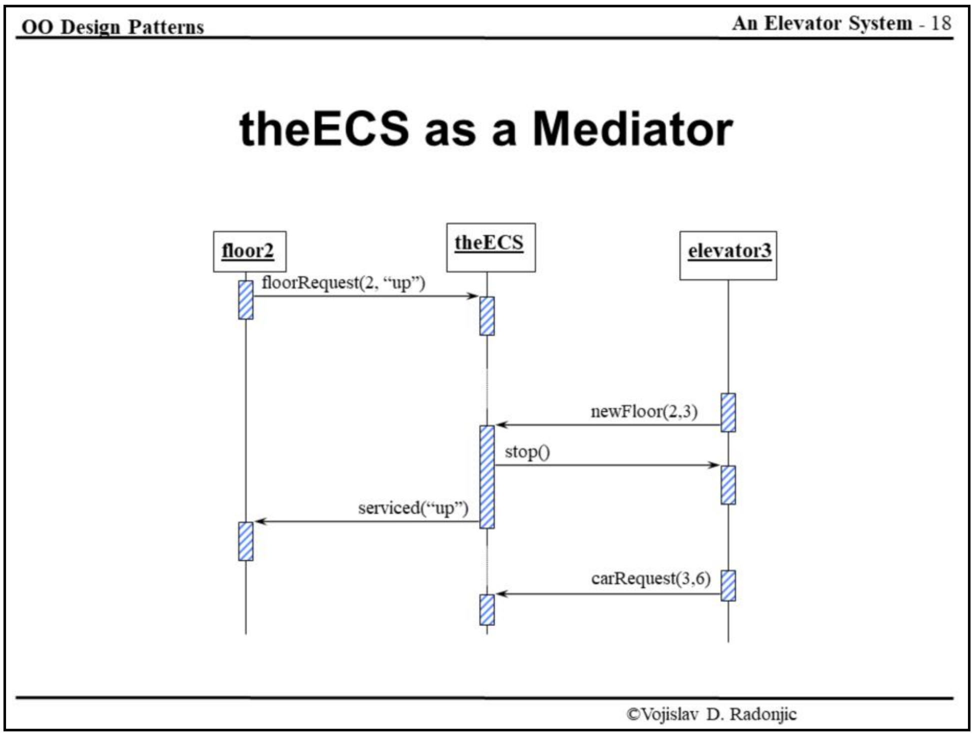
Public Interfaces

ECS:
- void floorRequest (int floorNumber, char* direction);
- void newFloor(int floorNumber, int carNumber);
- void carRequest(int carNumber, int floorNumber);
- void readyToMove(int carNumber);

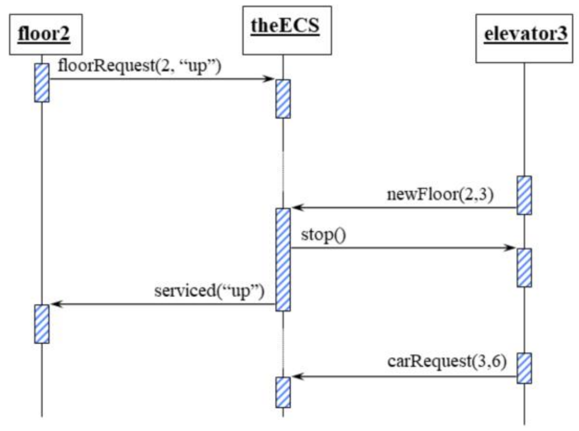
Floor:
- void serviced(char* direction);
- void inform(char* direction);

Elevator:
- void stop();
- void start(char* direction);
- void newFloor(int floorNumber);
- void carRequest(int floorNumber);

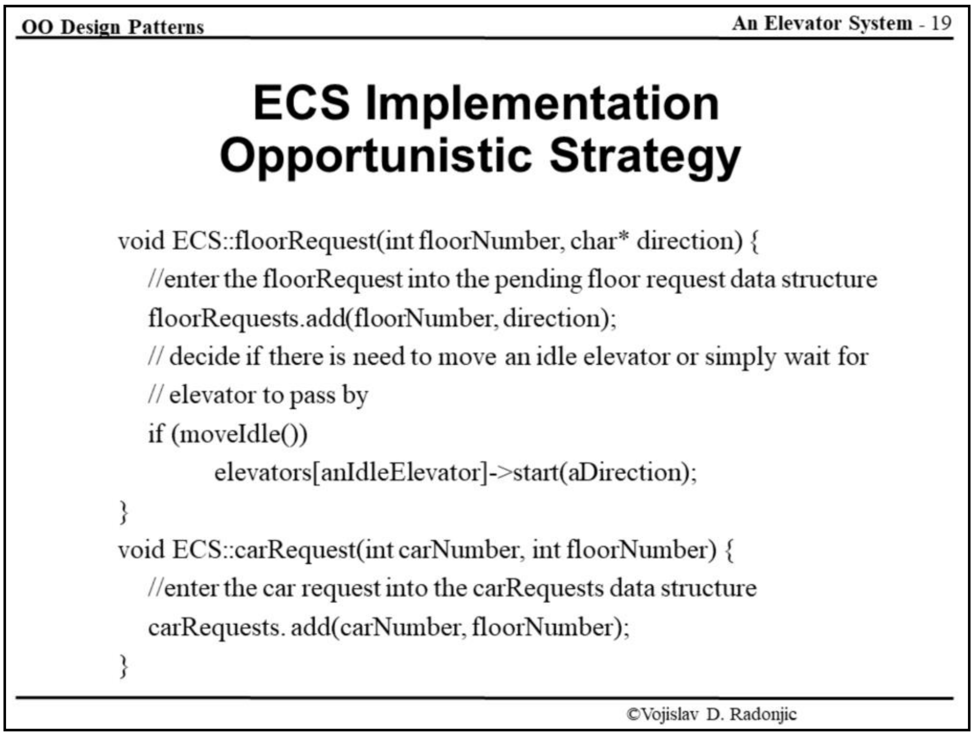
From a collection of sequence diagrams that cover all the use cases we would identify the interfaces of each of the elements in our elevator system. In our case the key elements of interest are theECS, the floors and the elevators because we want to focus on the system-level control aspects.



theECS as a Mediator

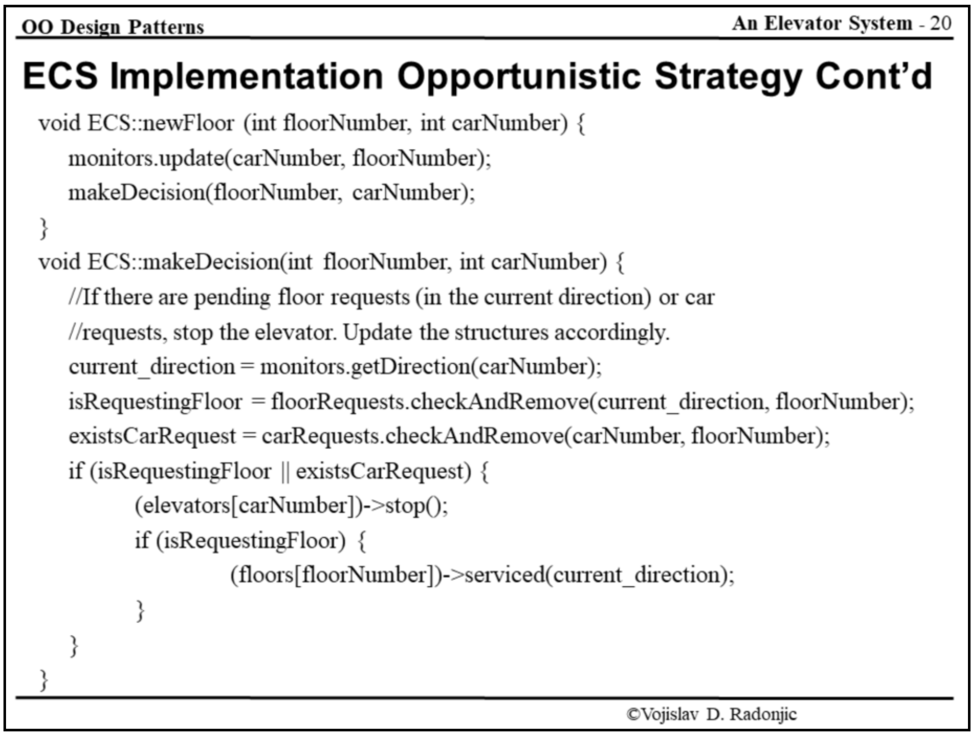


Notice how theECS decouples floors and elevators from each other. Elevators appear to collaborate amongst themselves to service request generated from the floors without knowing anything about other elevators or floors.



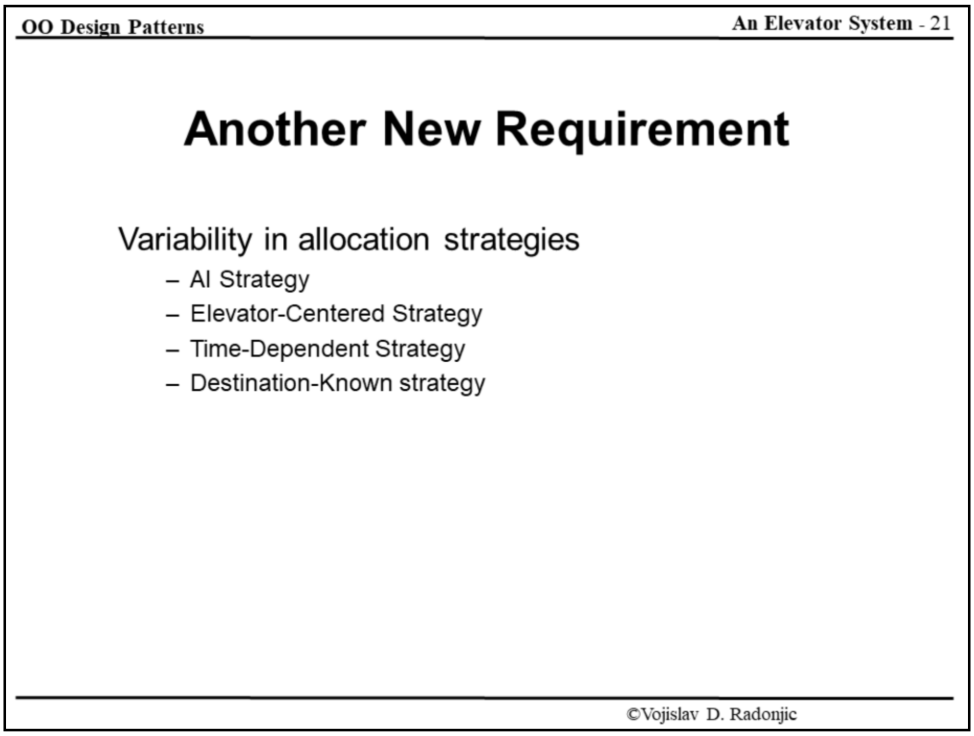
ECS Implementation Opportunistic Strategy

```
void ECS::floorRequest(int floorNumber, char* direction) {
   //enter the floorRequest into the pending floor request data structure
   floorRequests.add(floorNumber, direction);
   // decide if there is need to move an idle elevator or simply wait for
   // elevator to pass by
   if (moveIdle())
         elevators[anIdleElevator]->start(aDirection);
}
void ECS::carRequest(int carNumber, int floorNumber) {
   //enter the car request into the carRequests data structure
   carRequests. add(carNumber, floorNumber);
}
```



ECS Implementation Opportunistic Strategy Cont'd

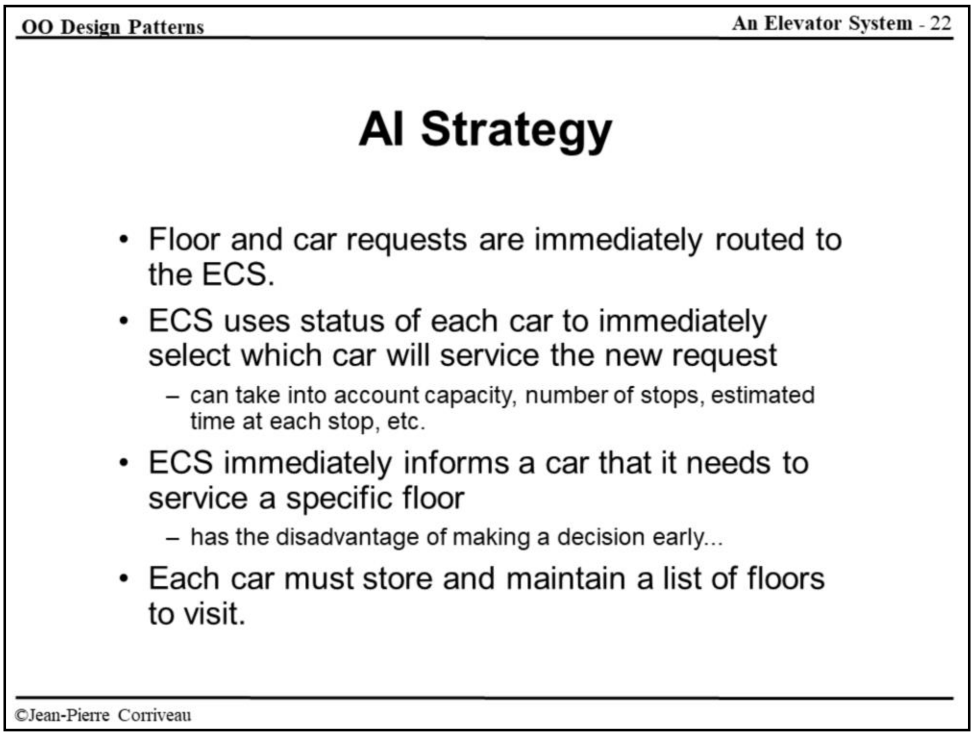
```
void ECS::newFloor (int floorNumber, int carNumber) {
    monitors.update(carNumber, floorNumber);
    makeDecision(floorNumber,  carNumber);
}
void ECS::makeDecision(int  floorNumber, int carNumber) {
    //If there are pending floor requests (in the current direction) or car
    //requests, stop the elevator. Update the structures accordingly.
    current_direction = monitors.getDirection(carNumber);
    isRequestingFloor  = floorRequests.checkAndRemove(current_direction, floorNumber);
    existsCarRequest = carRequests.checkAndRemove(carNumber, floorNumber);
    if (isRequestingFloor  || existsCarRequest) {
            (elevators[carNumber])->stop();
            if (isRequestingFloor)  {
                    (floors[floorNumber])->serviced(current_direction);
            }
    }
}
```



Another New Requirement

Variability in allocation strategies

- AI Strategy
- Elevator-Centered Strategy
- Time-Dependent Strategy
- Destination-Known strategy

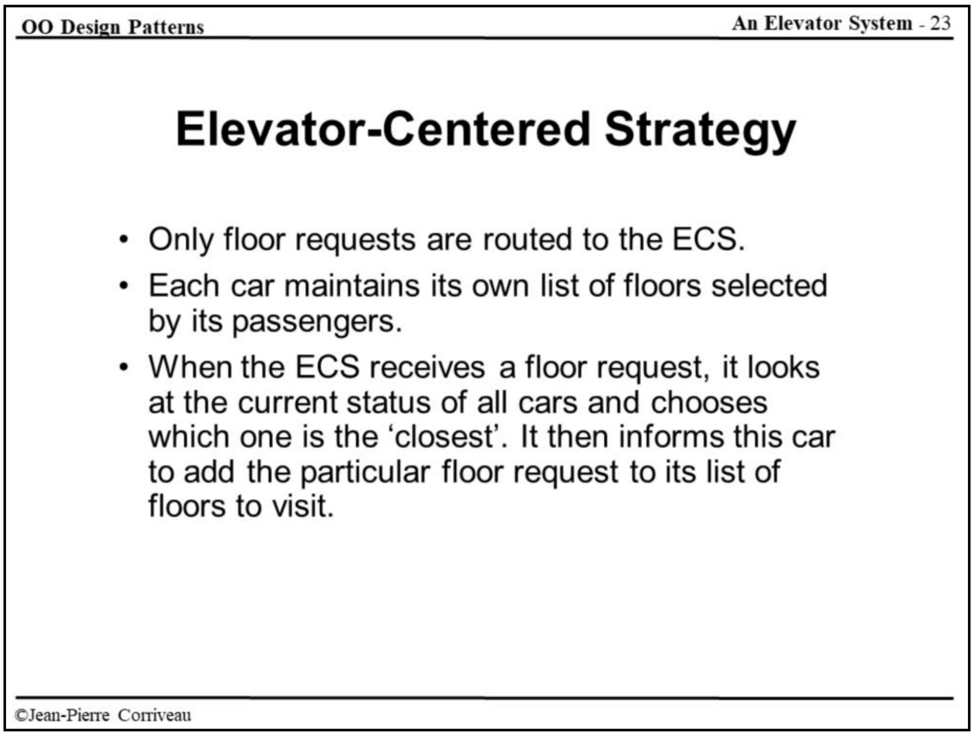


AI Strategy

- Floor and car requests are immediately routed to the ECS.
- ECS uses status of each car to immediately select which car will service the new request
  - can take into account capacity, number of stops, estimated time at each stop, etc.
- ECS immediately informs a car that it needs to service a specific floor
  - has the disadvantage of making a decision early...
- Each car must store and maintain a list of floors to visit.

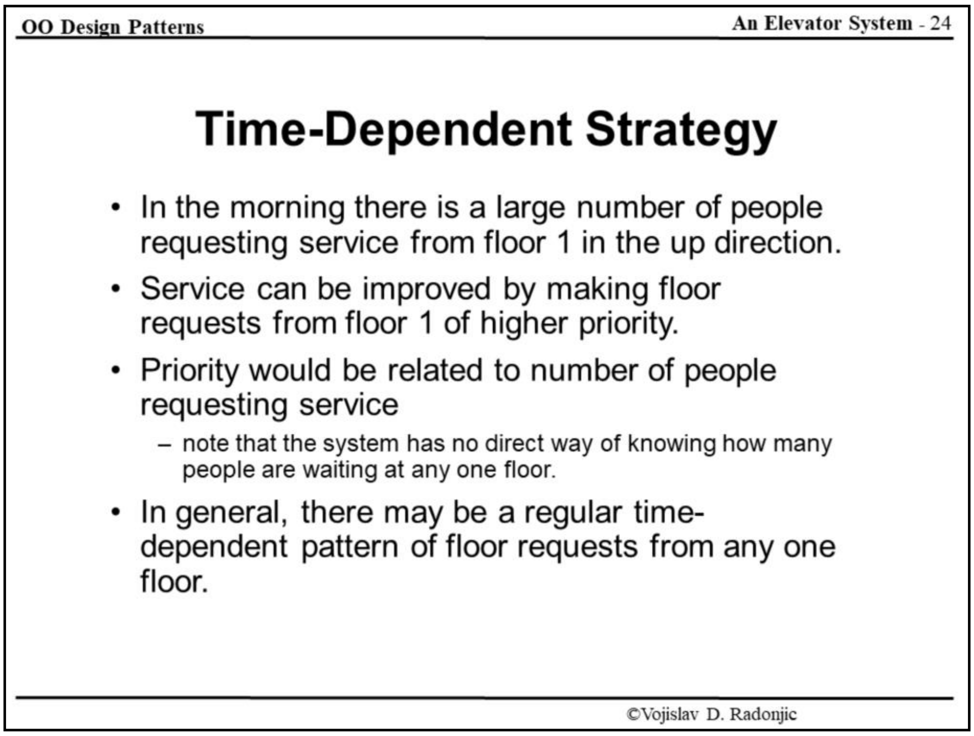
In the AI solution, the complexity of the problem is hidden in the ECS.

Each car maintains a list of floors to visit (as instructed by the ECS). Each car is therefore responsible of removing a floor from this list, once it is visited.



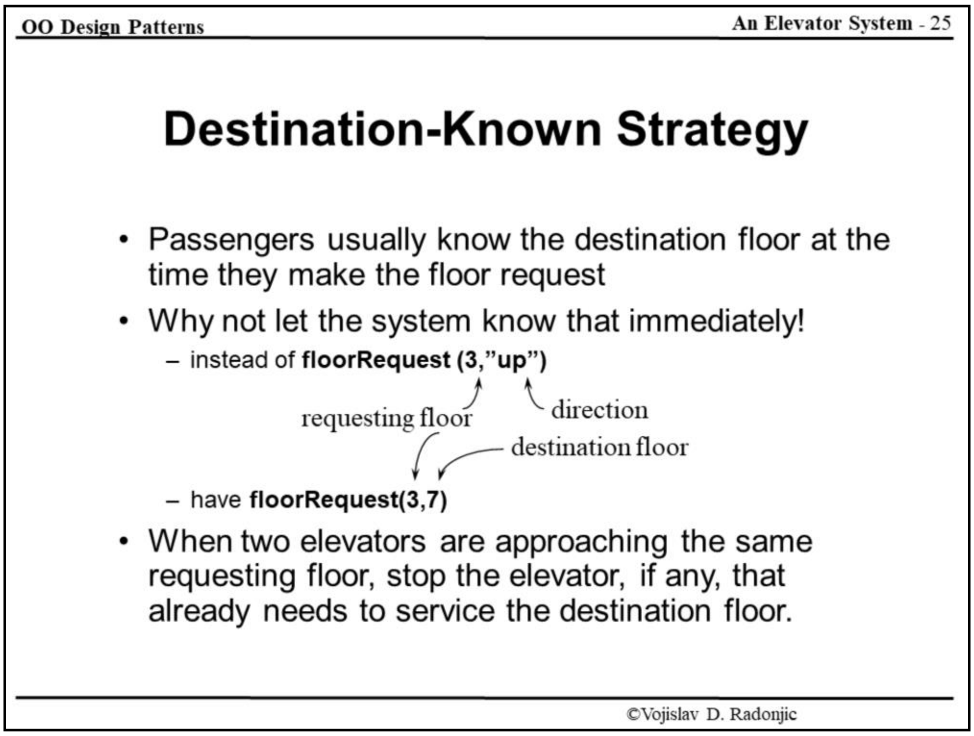
Elevator-Centered Strategy

- Only floor requests are routed to the ECS.
- Each car maintains its own list of floors selected by its passengers.
- When the ECS receives a floor request, it looks at the current status of all cars and chooses which one is the ‘closest’. It then informs this car to add the particular floor request to its list of floors to visit.



Time-Dependent Strategy

- In the morning there is a large number of people requesting service from floor 1 in the up direction.
- Service can be improved by making floor requests from floor 1 of higher priority.
- Priority would be related to number of people requesting service
  - note that the system has no direct way of knowing how many people are waiting at any one floor.
- In general, there may be a regular time-dependent pattern of floor requests from any one floor.



Destination-Known Strategy

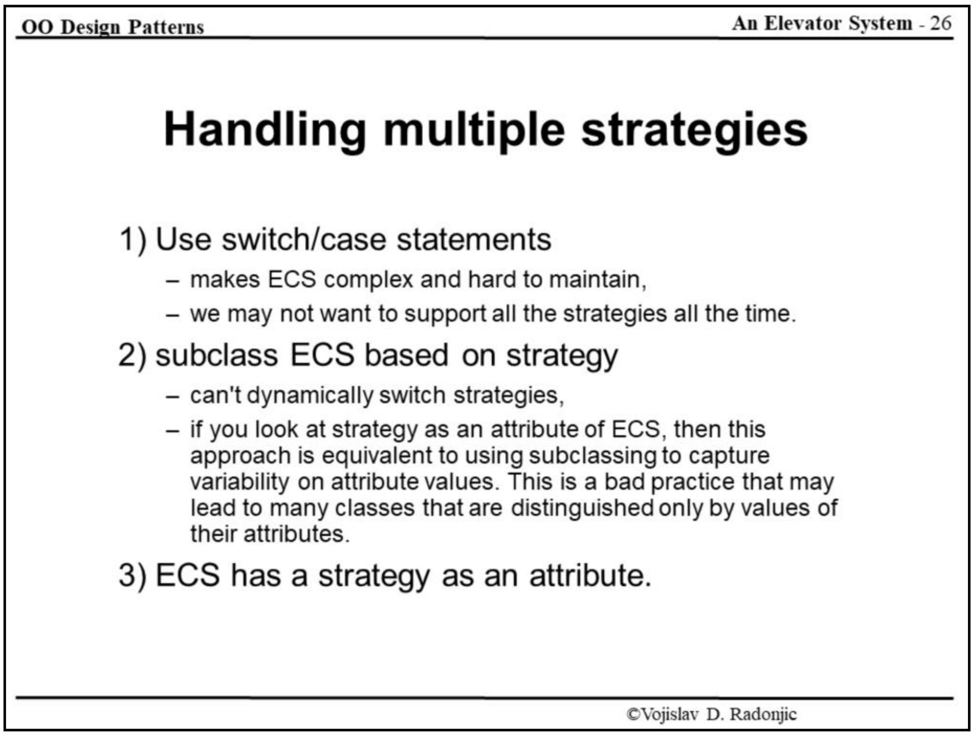
- Passengers usually know the destination floor at the time they make the floor request
- Why not let the system know that immediately!
  - instead of **floorRequest (3,"up")**
  - have **floorRequest(3,7)**
- When two elevators are approaching the same requesting floor, stop the elevator, if any, that already needs to service the destination floor.

Yet another strategy:

Load-balancing Strategy

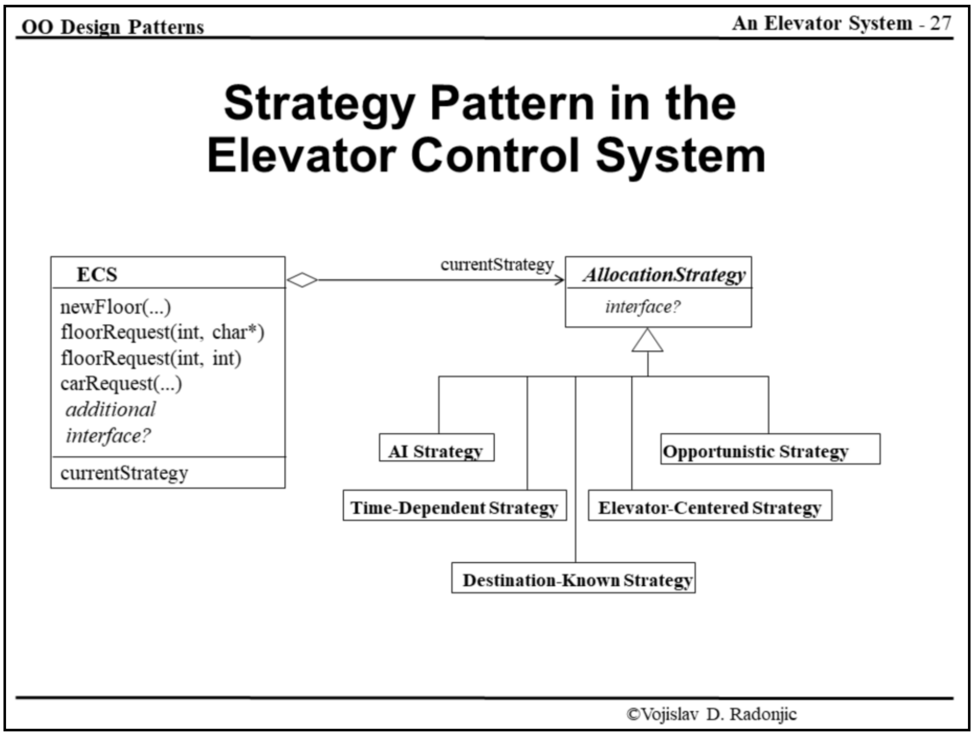
•distribute evenly the requests on the elevators,

•flag and automatically take out of service an elevator that is over the time or distance limit from last service.

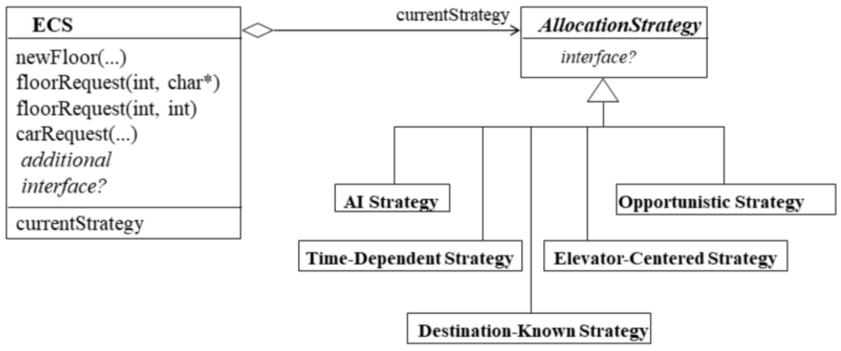


Handling multiple strategies

1) Use switch/case statements
   - makes ECS complex and hard to maintain,
   - we may not want to support all the strategies all the time.
2) subclass ECS based on strategy
   - can't dynamically switch strategies,
   - if you look at strategy as an attribute of ECS, then this approach is equivalent to using subclassing to capture variability on attribute values. This is a bad practice that may lead to many classes that are distinguished only by values of their attributes.
3) ECS has a strategy as an attribute.



Strategy Pattern in the Elevator Control System



The above diagram shows how the Strategy pattern could be used in the elevator system to have the **ECS** provide support for multiple allocation strategies.

Exercise: complete the above picture by defining an interface between the **ECS** and the ***Allocation Strategy***.