Software Architecture Theory

E03. 아키텍처 설계 사례연구

Elevator Controller System (ECS)

Exercise

2014

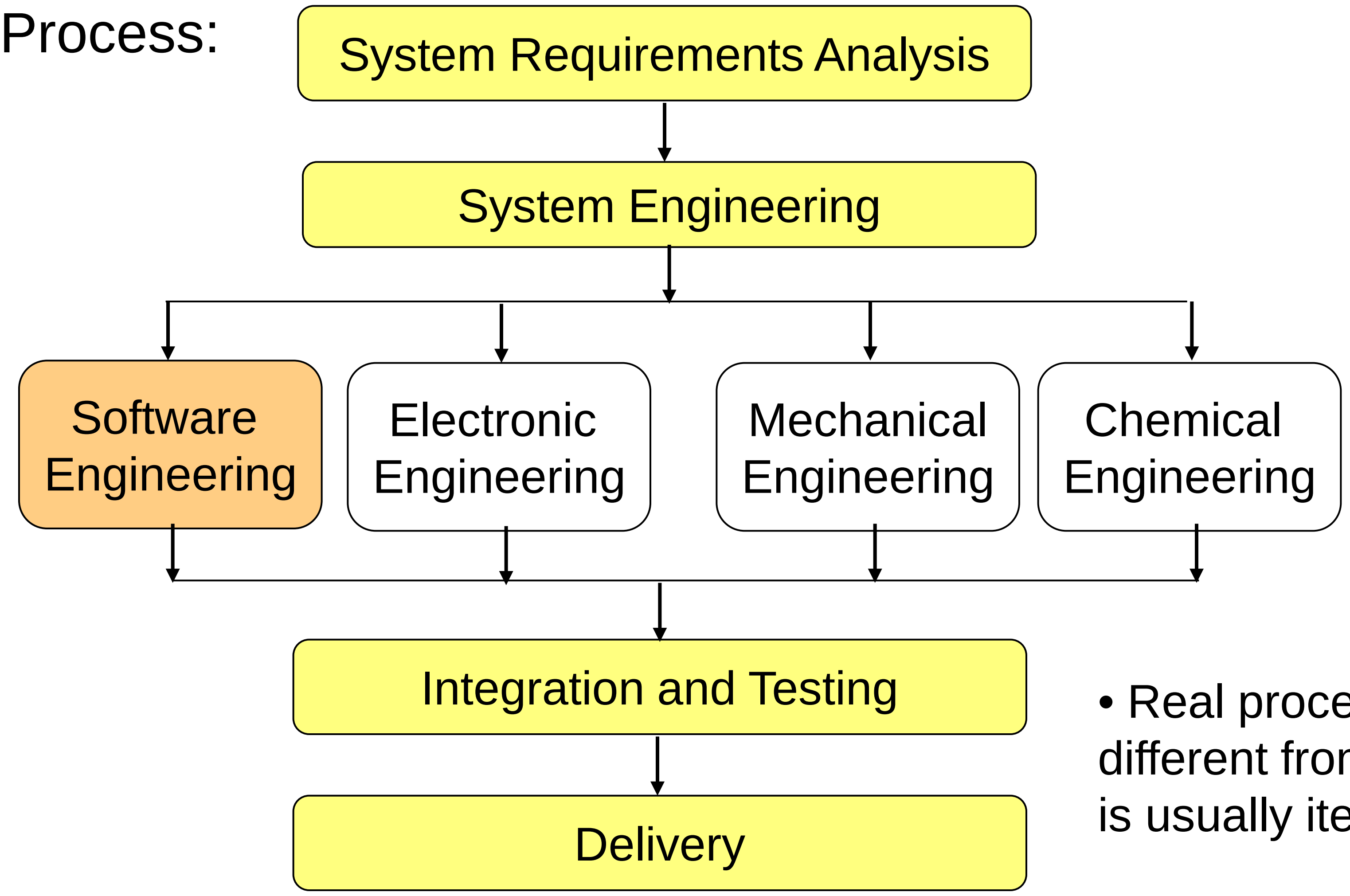
Sungwon Kang

Table of Contents

Acknowledgement. The following contents are based on the Elevator Control System case study in [Gomaa 00].

1. System Engineering

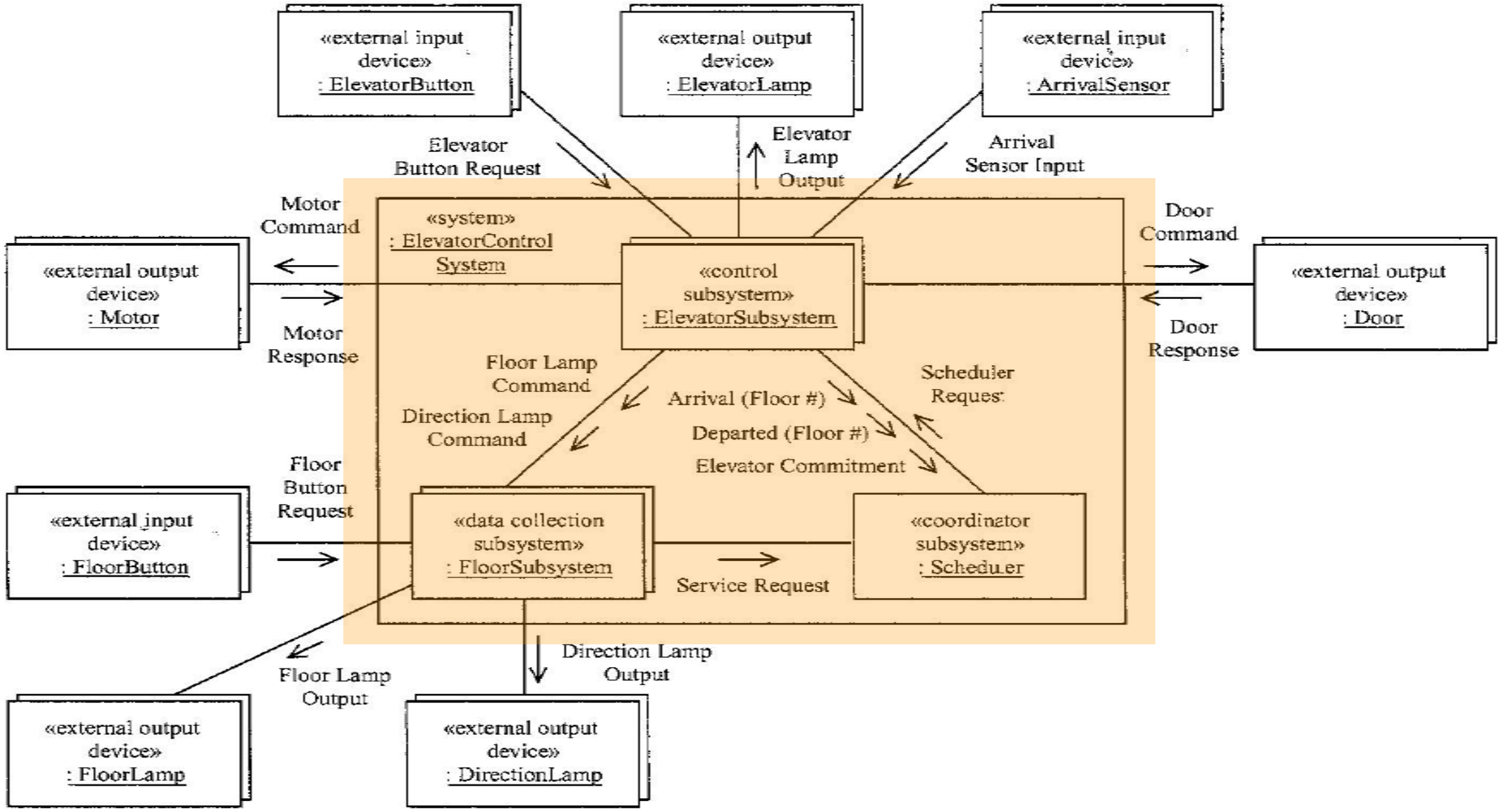
Process:



- Real process is different from this and is usually iterative

System Engineering

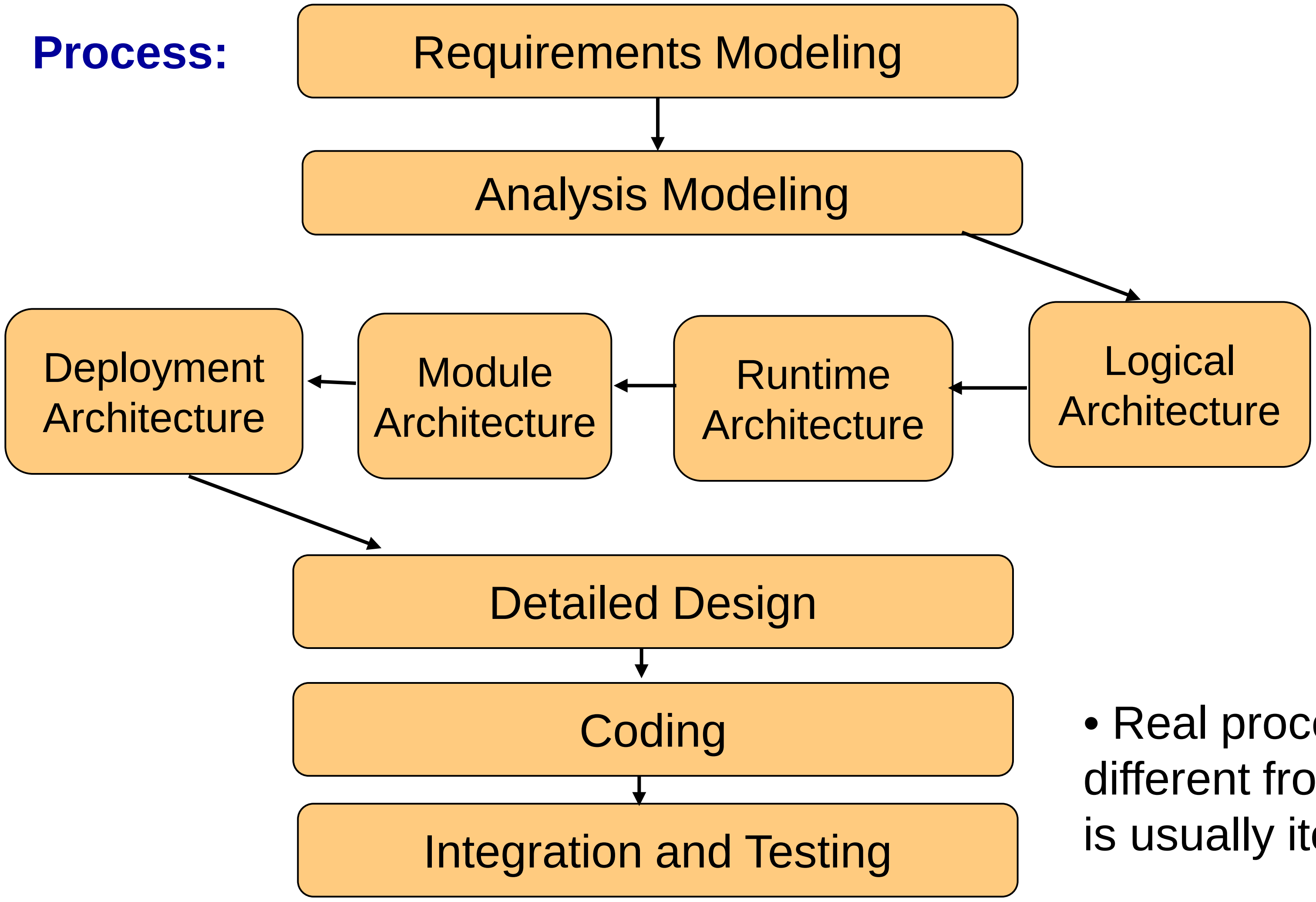
- **System Architecture Design (Elevator System Example)**



«external input device» : ElevatorButton
«external output device» : ElevatorLamp
«external input device» : ArrivalSensor
Elevator Button Request
Elevator Lamp Output
Arrival Sensor Input
Motor Command
«system» : ElevatorControl System
Door Command
«external output device» : Motor
«control subsystem» : ElevatorSubsystem
«external output device» : Door
Motor Response
Door Response
Floor Lamp Command
Direction Lamp Command
Arrival (Floor #)
Departed (Floor #)
Elevator Commitment
Scheduler Request
Floor Button Request
«external input device» : FloorButton
«data collection subsystem» : FloorSubsystem
«coordinator subsystem» : Scheduler
Service Request
Floor Lamp Output
Direction Lamp Output
«external output device» : FloorLamp
«external output device» : DirectionLamp

2. Software Engineering

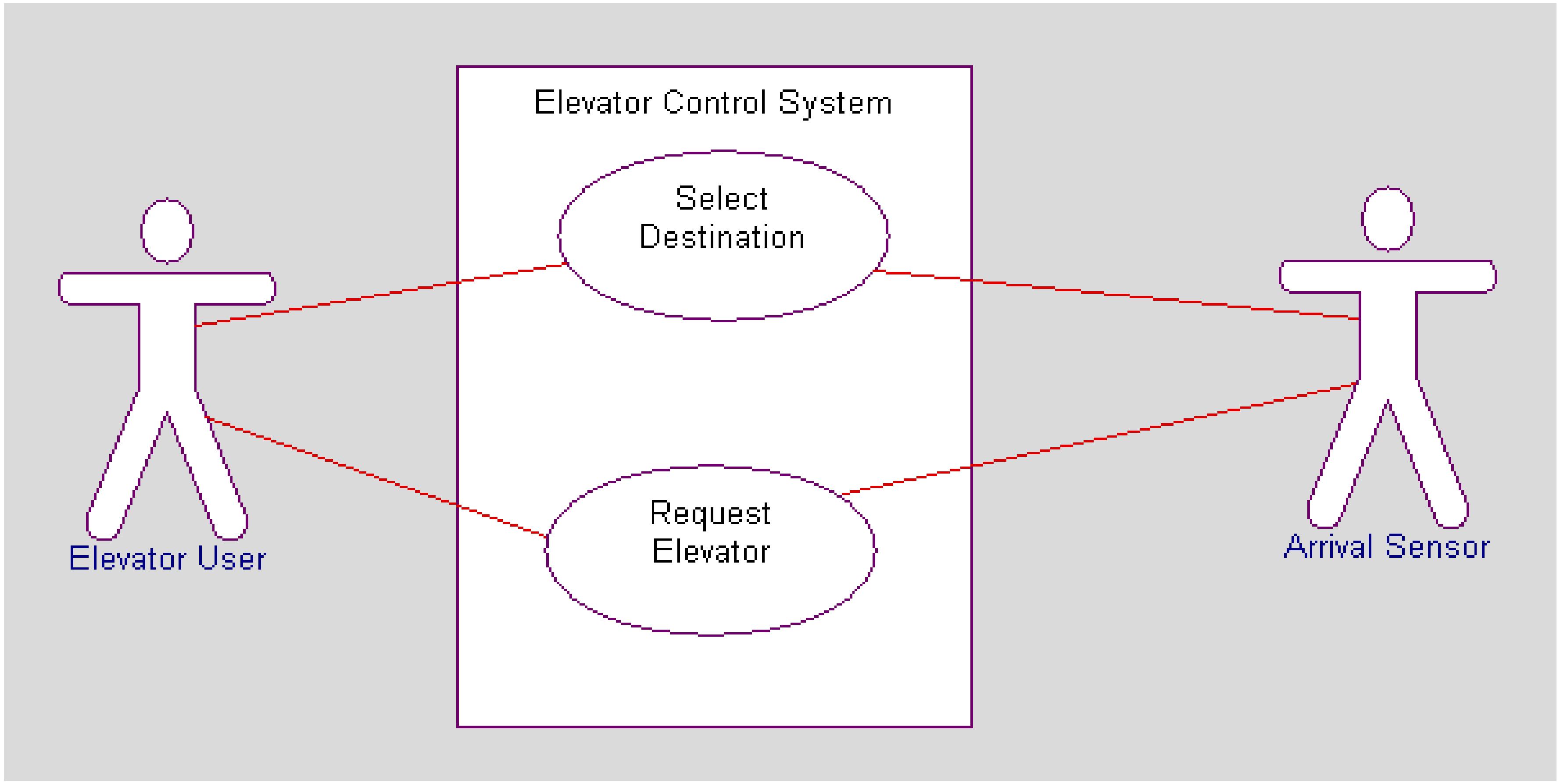
Process:



- Real process is different from this and is usually iterative

Software Engineering

- Requirements modeling



Software Engineering

Request elevator use case

Actors: Elevator User, Arrival Sensor

Precondition: User is at a floor and wants an elevator

Postcondition: Elevator has arrived at the floor in response to the user request.

Description:

1. User presses an up floor button. The floor button sensor sends the user request to the system, identifying the floor number.
2. The system selects an elevator to visit this floor. The new request is added to the list of floors to visit. If the elevator is stationary, the system determines in which direction the elevator should move in order to service the next request.
3. As the elevator moves between floors, the arrival sensor detects the elevator is approaching a floor and notifies the system. The system checks whether the elevator should stop at this floor. If so, the system stops the motor, and opens the door.
4. If there are other outstanding requests, the elevator visits these floors on the way to the floor requested by the user. Eventually, the elevator arrives at the floor in response to the user request.

Alternatives:

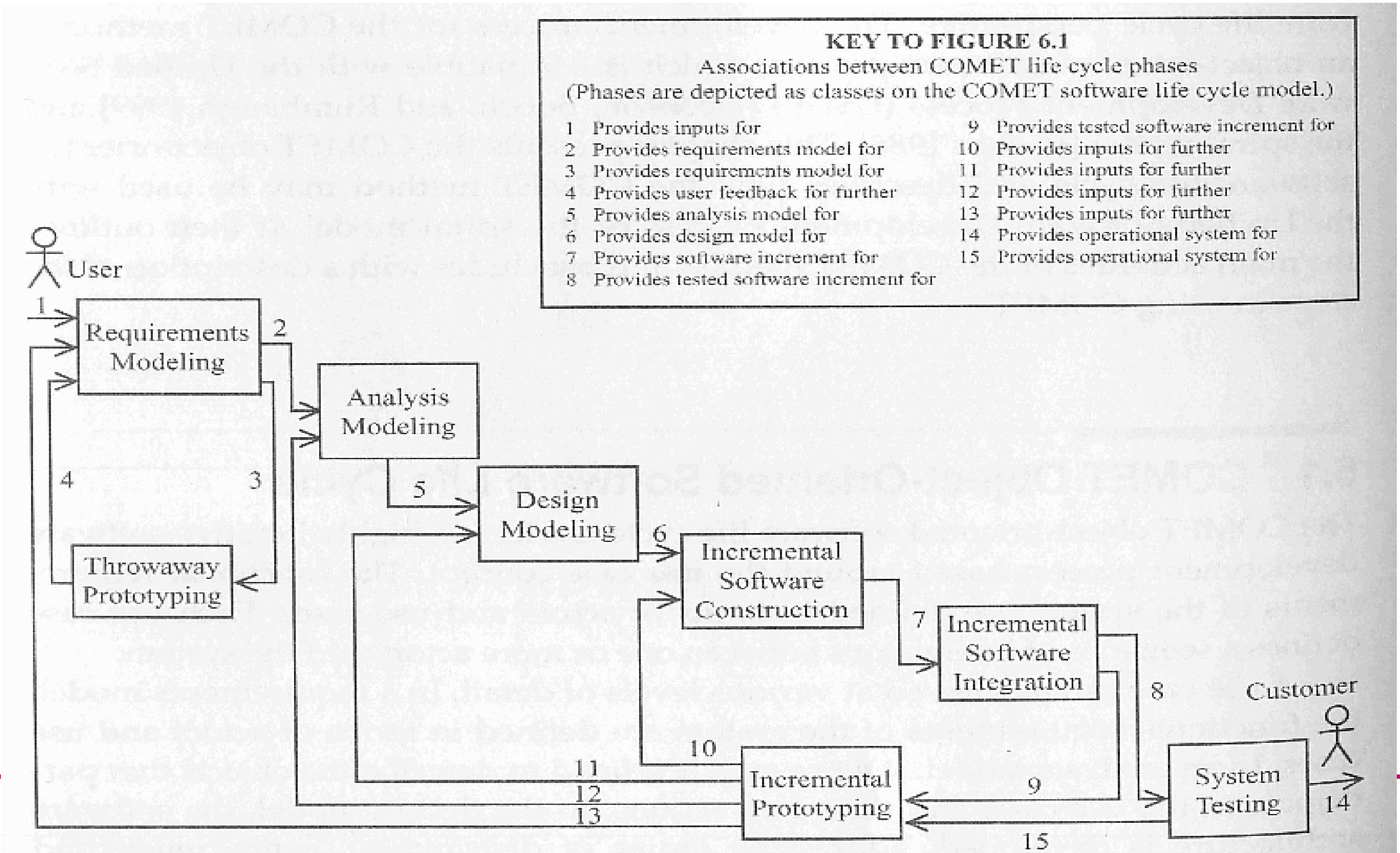
1. User presses a down floor button, the system response is the same as the main sequence.
2. If the elevator is at a floor, and there is no new floor to move to, the elevator stays at the same floor, with the door open.

3. Design Method

- COMET (Concurrent Object Modeling and Architectural Design Method)
- Invented by Prof. Hassan Gomaa (George Mason University)
- For development of concurrent (distributed) applications
- For development of realtime applications
- Iterative
- Object-oriented

COMET Method Overview

- **COMET development process**



KEY TO FIGURE 6.1

Associations between COMET life cycle phases

(Phases are depicted as classes on the COMET software life cycle model.)

1 Provides inputs for
2 Provides requirements model for
3 Provides requirements model for
4 Provides user feedback for further
5 Provides analysis model for
6 Provides design model for
7 Provides software increment for
8 Provides tested software increment for
9 Provides tested software increment for
10 Provides inputs for further
11 Provides inputs for further
12 Provides inputs for further
13 Provides inputs for further
14 Provides operational system for
15 Provides operational system for

COMET Method Overview

1. Requirement modeling
   - Use case modeling
2. Analysis modeling
   - Static modeling
   - Object structuring
   - Finite State Machine modeling
   - Dynamic modeling
3. Design modeling
   - Consolidate the object collaboration model
   - Make decisions about subsystem structure and interfaces
   - Make decisions about how to structure the distributed application into distributed subsystems
   - Make decisions about he characteristic of objects, in particular, whether they are active or passive.
   - Make decision about the characteristics of messages
   - Make decision about class interfaces
   - Develop the detailed software design, addressing detailed issues concerning task synchronization and communication, and the internal design of concurrent tasks.

COMET Method (1/2)

1. Requirements Modeling (Develop use case model)

1.1 Develop use case diagrams to depict actors, use cases and the relationships between use cases

1.2 Document each use case with a use case description

2. Analysis Modeling

2.1 Develop static model

2.1.1 Develop static model of problem domain (real world) by using class diagrams

2.1.2 Develop a system context model by using a class diagram

2.2 Structure the system into classes and objects

2.3 Develop dynamic model (for each use case)

2.3.1 Determine the objects participating in the use case

2.3.2 Develop an object interaction diagram (collaboration or sequence). Analyze the sequence of interactions between the objects. Analyze the information passed between the objects.

2.3.3 Develop a statechart for each state-dependent object

2.3.4 Develop message sequence descriptions for each interaction diagram

Problem: Domain analysis and architecture design not clearly separated

COMET Method (2/2)

3. Design Modeling

Architec-
tural
Design

- **3.1 Synthesize analysis model artifacts to produce initial software architecture**
  - 3.1.1 Synthesize statecharts
  - 3.1.2 Synthesize a collaboration model
  - 3.1.3 Synthesize a static model
- **3.2 Design overall software architecture**
- **3.3 Design distributed component-base software architecture**
- **3.4 Design the concurrent task architecture for each subsystems**
  - 3.4.1 Structure subsystems into concurrent tasks
  - 3.4.2 Define the tasks and their interfaces
  - 3.4.3 Develop concurrent collaboration diagrams for each subsystem
  - 3.4.4 Document the design of each task in a task behavior specification
- **3.5 Analyze the performance of the design**

Detailed
Design

- **3.6 Design the classes in each subsystem**
- **3.7 Develop the detailed software design for each subsystem**
  - 3.7.1 Design the internals of composite tasks with nested passive objects
  - 3.7.2 Design the details of task synchronization mechanisms for objects
  - 3.7.3 Design the connector classes that encapsulate inter task communication
  - 3.7.4 Design and document each task's internal event sequencing logic
- **3.8 Analyze the performance of the real-time design in greater detail for each subsystem. If necessary, repeat 3.4-3.7.**

4. Elevator Controller System: ECS

(Partial solution for a non-distributed version)

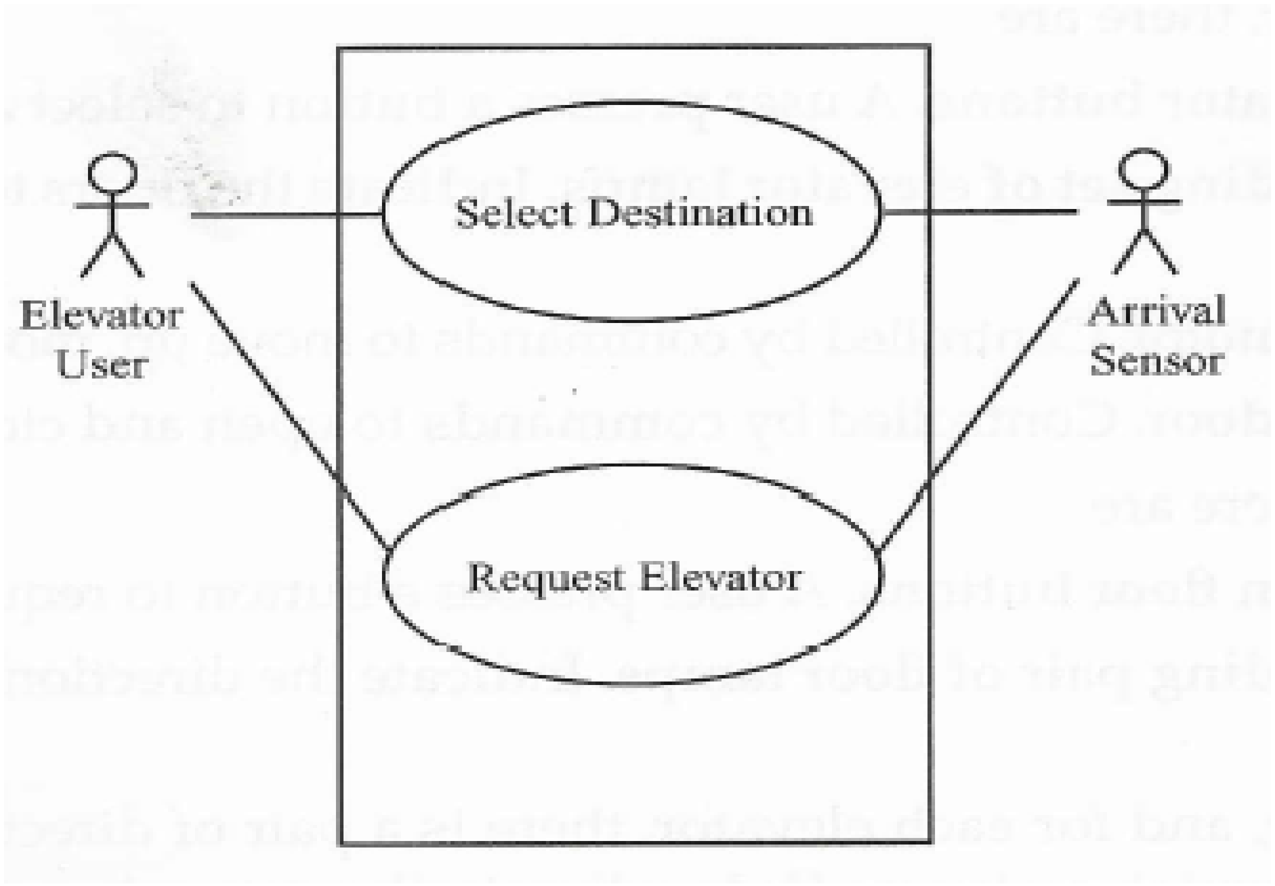
Problem Description (ECS)

- For each elevator, there are
  - A set of elevator buttons
  - A corresponding set of elevator lamps
  - An elevator motor
  - An elevator motor
  - An elevator door
- For each floor, there are
  - Up and down floor buttons
  - A corresponding pair of floor lamps
- At each floor and each elevator, there is a pair of direction lamps to indicate whether an elevator is heading up or down. For top and bottom floors, there is only one floor button, one floor lamp, and (for each elevator) one direction lamp.
- There is an arrival sensor at each floor in each elevator shaft to detect the arrival of an elevator at the floor.
- The hardware characteristics of the I/O devices are that
  - The elevator buttons, floor buttons and arrival sensor are asynchronous
  - The other I/O devices are all passive
  - The elevator and floor lamps are switched on by the hardware, but must be switched off by the software. The direction lamps are switched on and off by the software.

Phase1. Requirements Modeling

(1.1) Use Case Diagram

- ECS actors and use cases



(1.2) Use Case Description

Select Destination use case

Actors: Elevator User(primary), Arrival Sensor
Precondition: User is in the elevator
Postcondition: Elevator has arrived at the destination floor selected by the user.
Description:
1. User presses an up elevator button. The elevator button sensor sends the elevator button request to the system, identifying the destination floor the user wishes to visit.
2. The new request is added to the list of floors to visit. If the elevator is stationary, the system determines in which direction the system should move in order to service the next request. The system commands the elevator door to close. When the door has closed, the system commands the motor to start moving the elevator, either up or down.
3. As the elevator moves between floors, the arrival sensor detects that the elevator is approaching a floor and notifies the system. The system checks whether he elevator should stop at this floor. If so, the system commands the motor to stop. When the elevator has stopped, the system commands the elevator door to open..
4. If there are other outstanding requests, the elevator visits these floors on the way to the floor requested by the user. Eventually, the elevator arrives at the destination selected by to the user.

Alternatives:
1. User presses a down elevator button to move down. System response is the same as for the main sequence.
2. If the elevator is at a floor, and there is no new floor to move to, the elevator stays at the same floor, with the door open.

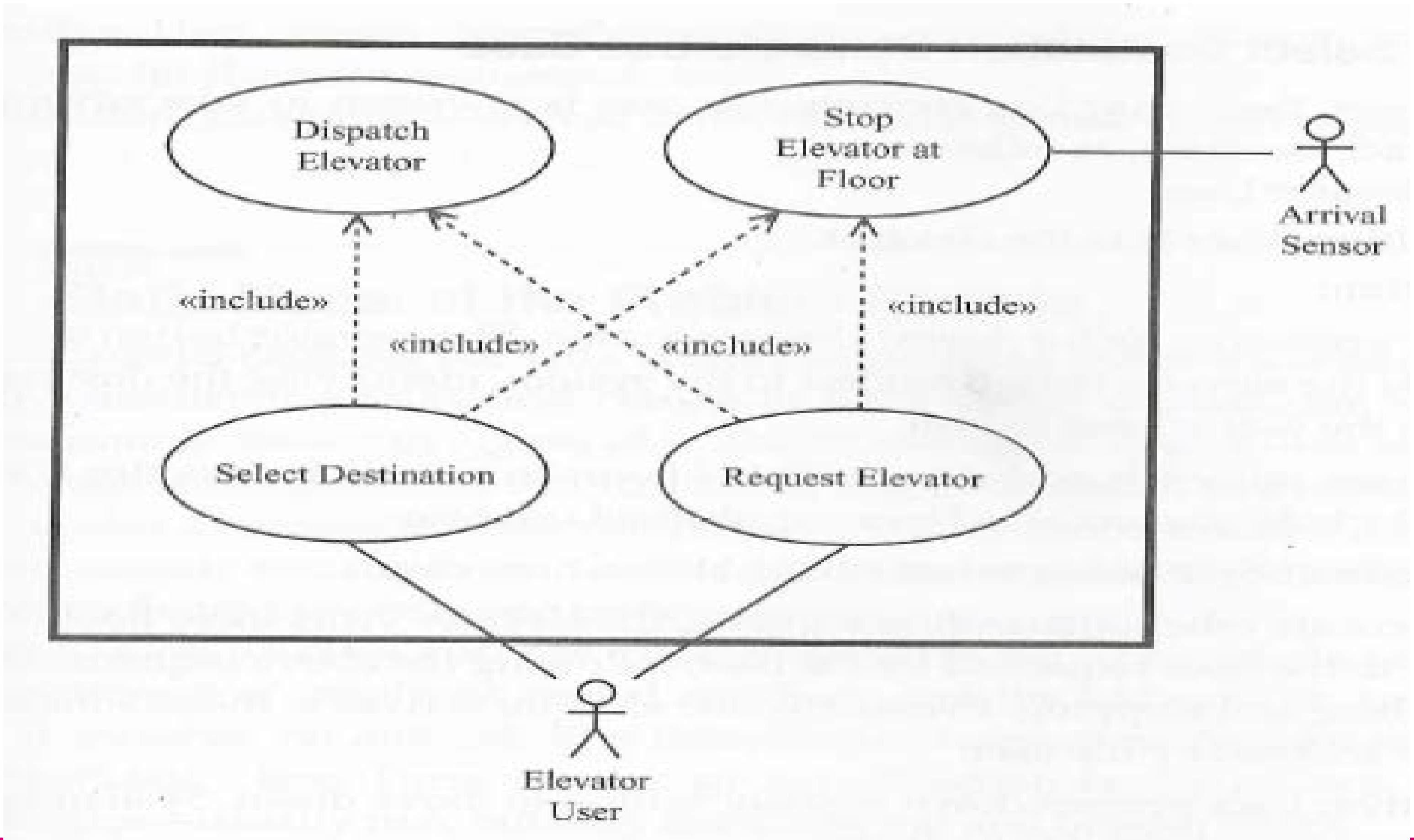
Request elevator use case

Actors: Elevator User, Arrival Sensor
Precondition: User is at a floor and wants an elevator
Postcondition: Elevator has arrived at the floor in response to the user request.
Description:
1. User presses an up floor button. The floor button sensor sends the user request to the system, identifying the floor number.
2. The system selects an elevator to visit this floor. The new request is added to the list of floors to visit. If the elevator is stationary, the system determines in which direction the elevator should move in order to service the next request.
3. As the elevator moves between floors, the arrival sensor detects the elevator is approaching a floor and notifies the system. The system checks whether he elevator should stop at this floor. If so, the system stops the motor, and opens the door.
4. If there are other outstanding requests, the elevator visits these floors on the way to the floor requested by the user. Eventually, the elevator arrives at the floor in response to the user request.

Alternatives:
1. User presses a down floor button, the system response is the same as for the main sequence.
2. If the elevator is at a floor, and there is no new floor to move to, the elevator stays at the same floor, with the door open.

(1.1') Use Case Diagram

- Analysis of the two use cases indicates two common sequences that can be factored into abstract use cases.



(1.2’) Use Case Description (1/4)

UC1) Stop Elevator at Floor abstract use case

Actors: Arrival Sensor

Precondition: Elevator is moving.

Postcondition: Elevator has stopped at floor, with door open.

Description:

As the elevator moves between floors, the arrival sensor detects that the elevator is approaching a floor and notifies the system.

The system checks whether the elevator should stop at this floor.

If so, the system commands the motor to stop.

When the elevator has stopped, the system commands the elevator door to open.

Alternatives:

The elevator is not required to stop at this floor and so continues past the floor.

(1.2') Use Case Description (2/4)

UC 2) Dispatch Elevator abstract use case

Precondition: Elevator is at a floor with the door open.

Postcondition: Elevator is moving in the commanded direction.

Description:

The system determines in which direction the elevator should move in order to service the next request.

The system commands the elevator door to close.

When the door has closed, the system commands the motor to start moving the elevator, either up or down.

Alternatives:

If the elevator is at a floor and there is no new floor to move to, the elevator stays at the current floor, with the door open.

(1.2’) Use Case Description (3/4)

UC 3) Select Destination concrete use case
Actors: Elevator User
Precondition: User is at a floor
Postcondition: Elevator has arrived at the destination floor selected by the user.
Description:
1. User presses an up (or down) elevator button. The elevator button sensor sends the elevator button request to the system, identifying the destination floor the user wishes to visit.
2. The new request is added to the list of floors to visit. If the elevator is stationary, include Dispatch Elevator abstract use case.
3. Include Stop Elevator at Floor abstract use case.
4. If there are other outstanding requests, the elevator visits these floors on the way to the floor requested by the user. Eventually, the elevator arrives at the floor in response to the user request.

Alternatives:
User presses a down floor button, the system response is the same as the main sequence.

(1.2’) Use Case Description (4/4)

UC4) Request elevator concrete use case
Actors: Elevator User
Precondition: User is at a floor and wants an elevator
Postcondition: Elevator has arrived at the floor in response to the user request.
Description:
1. User presses an up floor button. The floor button sensor sends the user request to the system, identifying the floor number.
2. The system selects an elevator to visit this floor. The new request is added to the list of floors to visit. If the elevator is stationary, then include Dispatch Elevator abstract use case.
3. Include Stop Elevator at Floor abstract use case.
4. If there are other outstanding requests, the elevator visits these floors on the way to the floor requested by the user. Eventually, the elevator arrives at the floor in response to the user request.

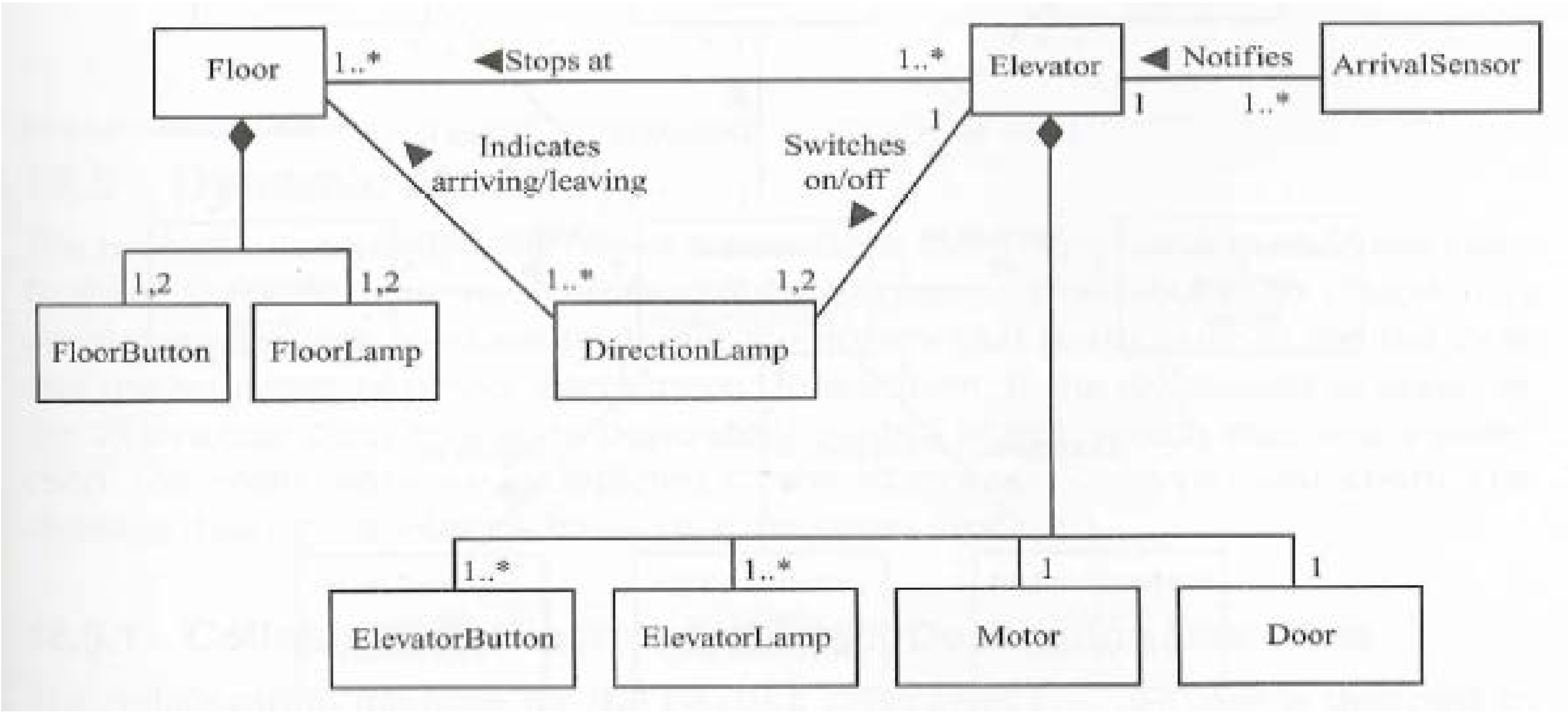
Alternatives:
User presses a down floor button, the system response is the same as the main sequence.

Phase 2. Analysis Modeling

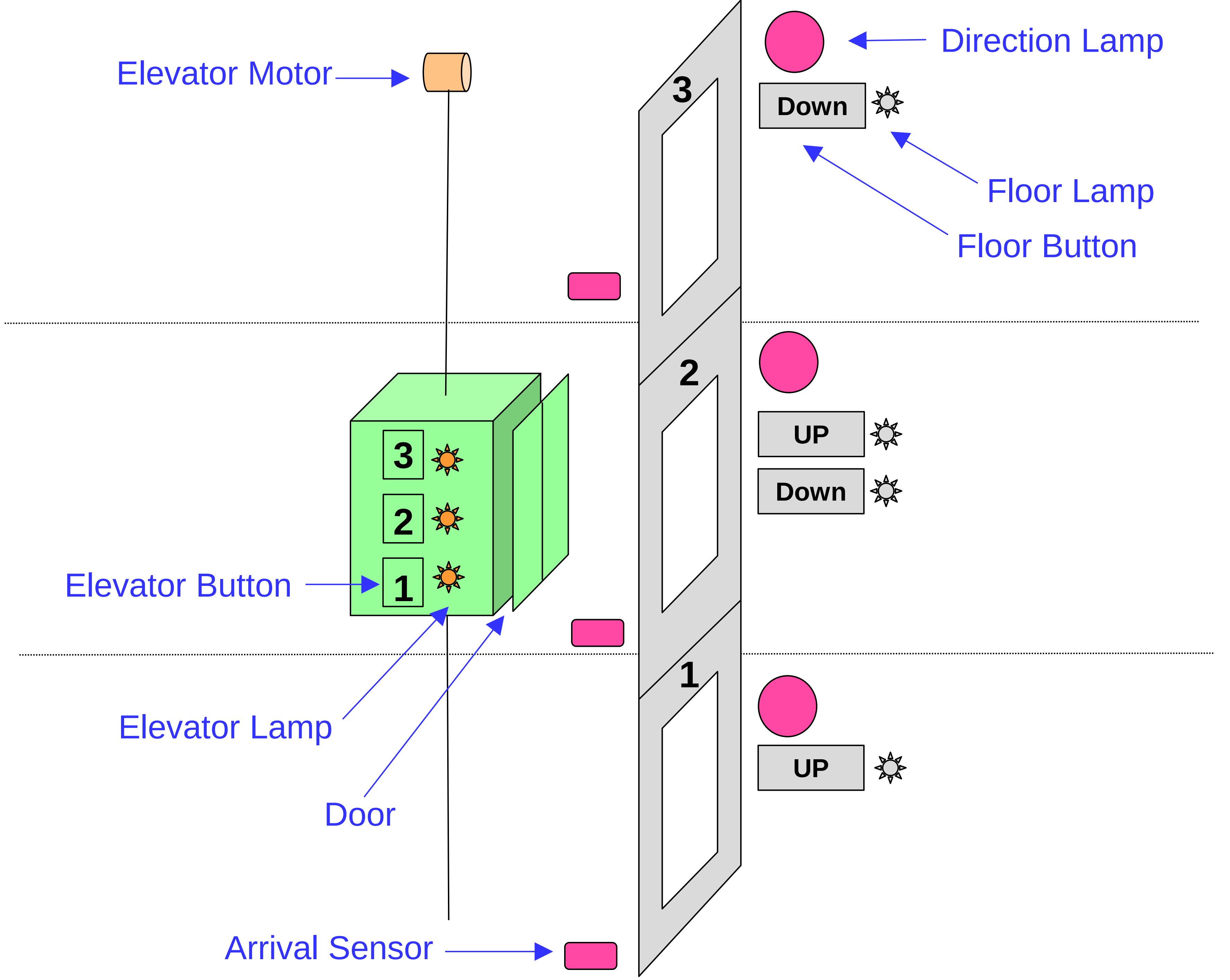
2.1 Develop static model
2.2 Structure the system into classes and objects
2.3 Develop dynamic model (for each use case)

(2.1.1) Static Modeling: Conceptual Static Model

- Static Model captures the static relationships for ECS.
- A model of the Problem Domain

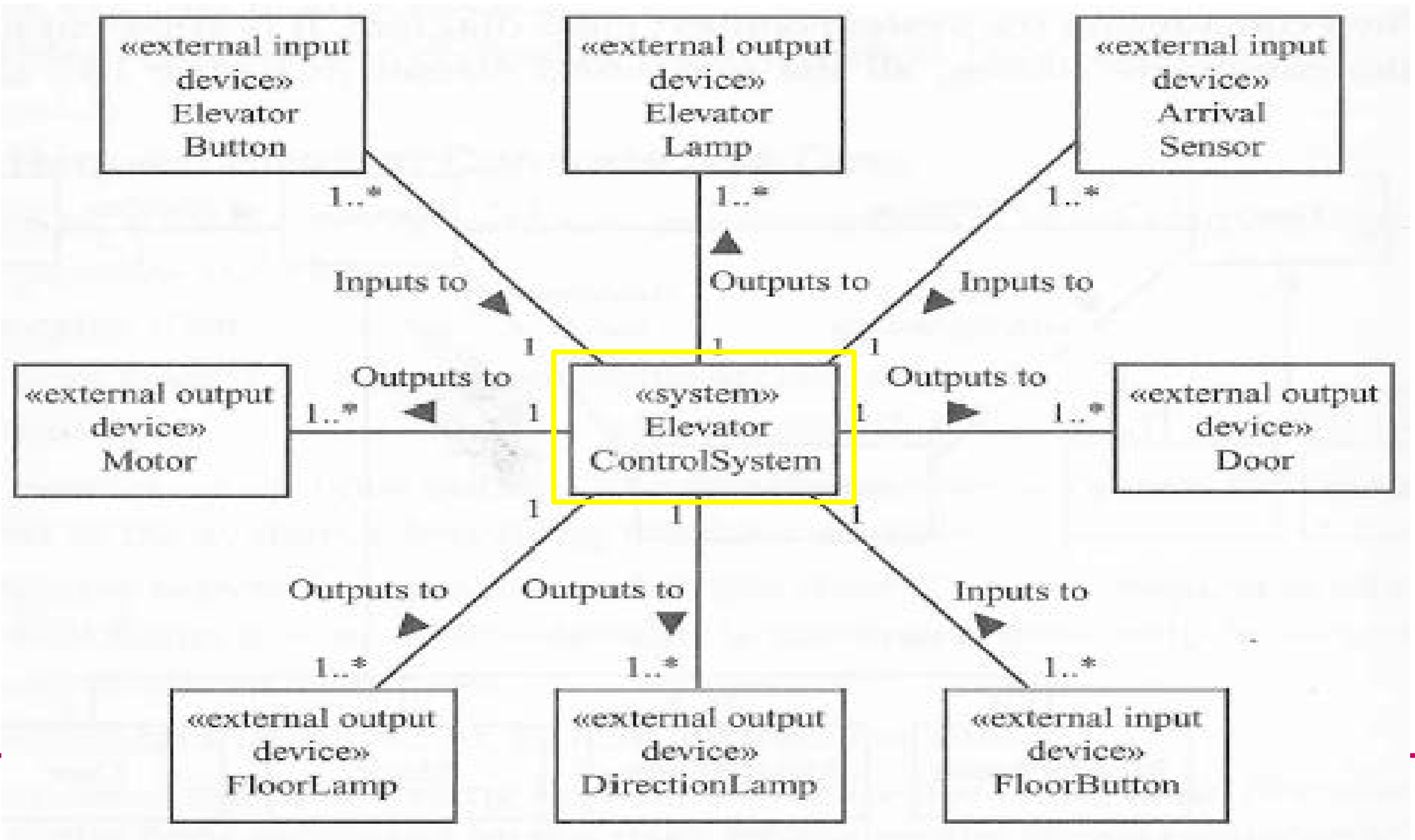


Sungwon Kang


Direction Lamp
Elevator Motor
3
Down
Floor Lamp
Floor Button
2
3
2
1
UP
Down
Elevator Button
1
Elevator Lamp
UP
Door
Arrival Sensor

(2.1.2) Static Modeling: System Context Diagram

- Depicts the actors that interact with the system
- For every external device, there is a corresponding software device interface object. (☛ Problem: Domain analysis and architecture design not clearly separated.)


«external input device» Elevator Button
«external output device» Elevator Lamp
«external input device» Arrival Sensor
1..*
1..*
1..*
Inputs to
Outputs to
Inputs to
1
1
1
«external output device» Motor
Outputs to
1..*
1
«system» Elevator ControlSystem
Outputs to
1
1..*
«external output device» Door
1
1
1
Outputs to
Outputs to
Inputs to
1..*
1..*
1..*
«external output device» FloorLamp
«external output device» DirectionLamp
«external input device» FloorButton

(2.3) Dynamic Modeling

- Define the object interactions that correspond to each use case. (Sequence diagrams can be used instead.)

- Each state-dependent object will have its statechart.

(2.3.1) Determine the objects participating in the use cases

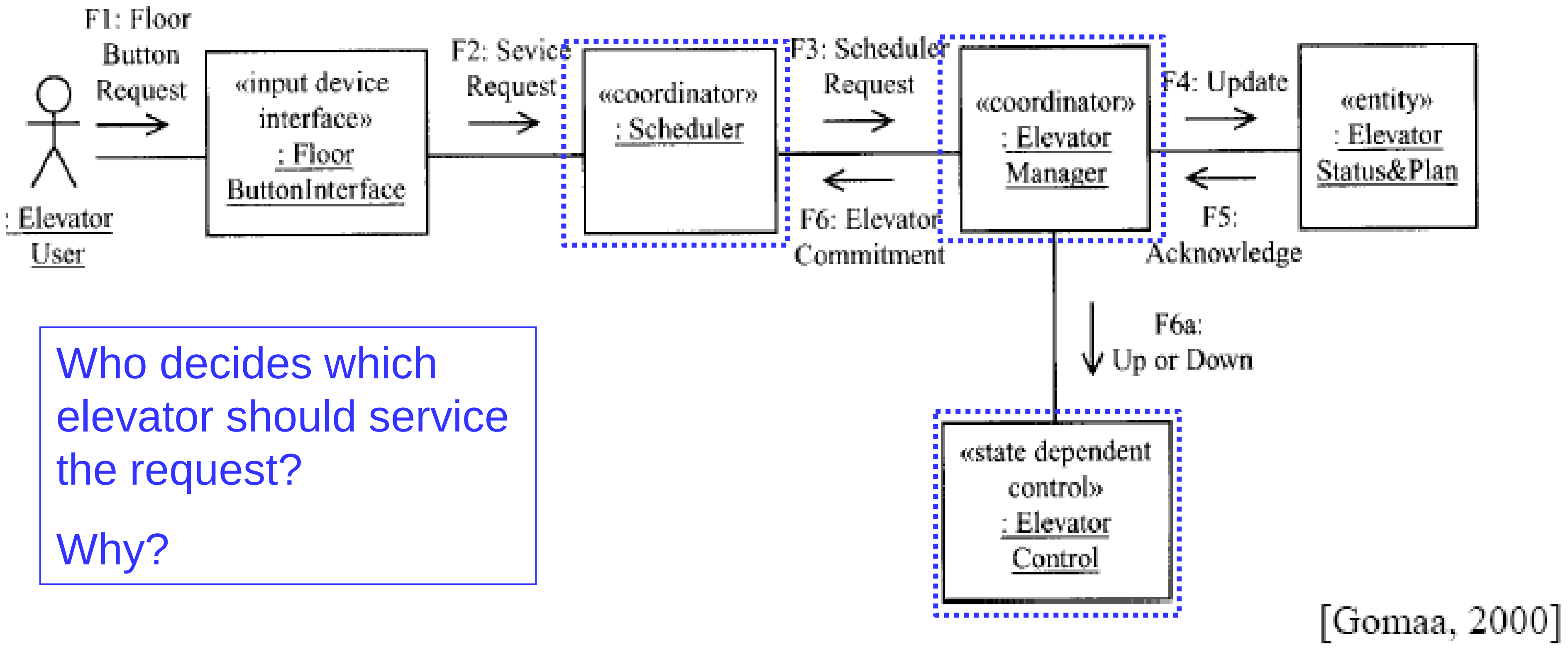
- Further we need:
  - Entity Object
    - For each elevator, `Elevator Status & Plan`
      - (up, down, idle), current floor, a list of floors to visit
  - Control Object:
    - For each elevator, `Elevator Control`
      - Control the elevator motor and door
      - State-dependent
    - For each elevator, `Elevator Manager`
      - Receive incoming requests for the elevator
      - Update Elevator Status & Plan
    - `Scheduler`
      - Select an elevator to service a floor request

$\Big\}$ Each object may be viewed as the result of application of a pattern.

(☛ Problem: Domain analysis and architecture design not clearly separated.)

(2.3.2) Dynamic Modeling – Collaboration Diagram

- UC4) Request Elevator use case



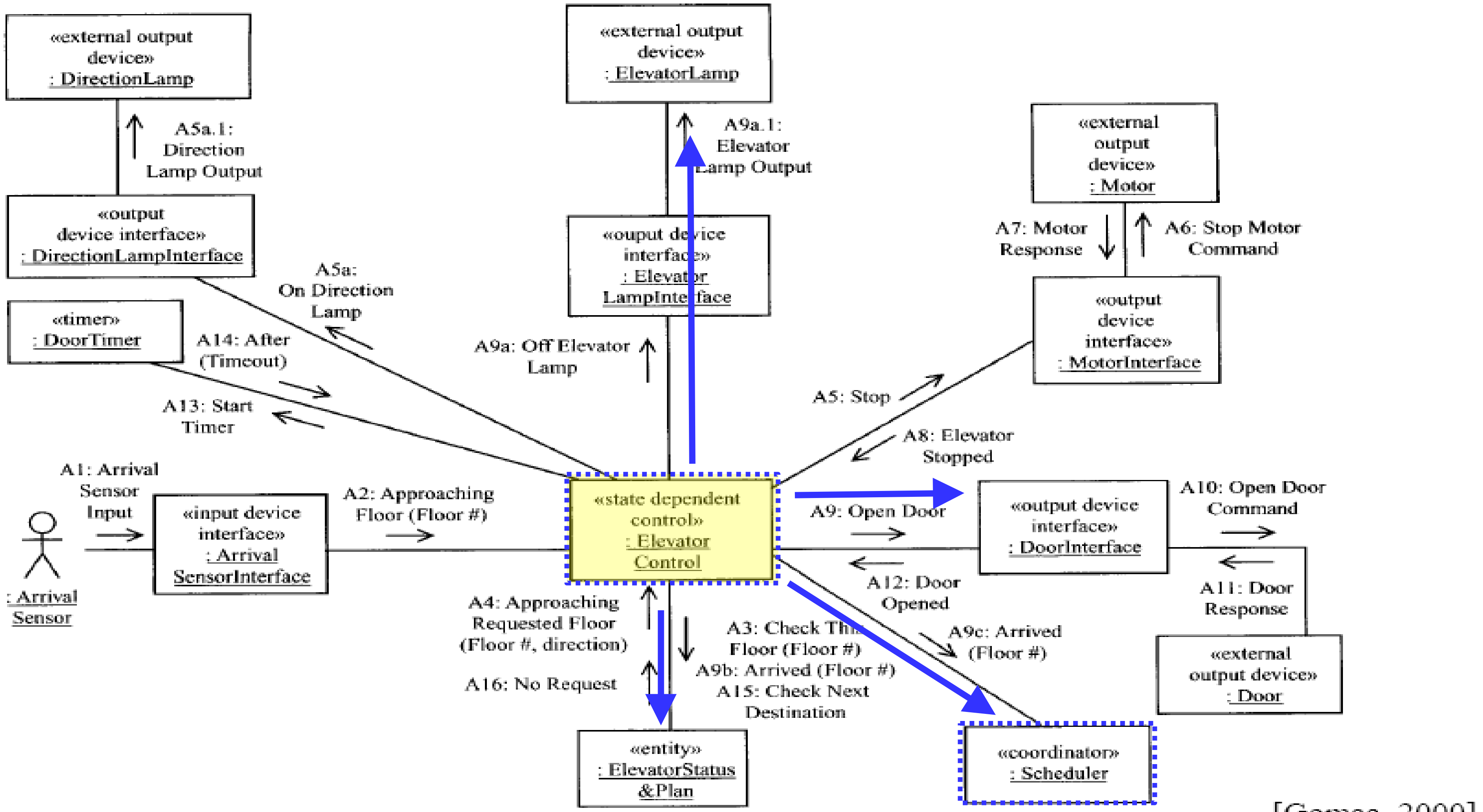
F1: Floor Button Request
: Elevator User
«input device interface» : Floor ButtonInterface
F2: Sevice Request
«coordinator» : Scheduler
F3: Scheduler Request
F6: Elevator Commitment
«coordinator» : Elevator Manager
F4: Update
F5: Acknowledge
«entity» : Elevator Status&Plan
F6a: Up or Down
«state dependent control» : Elevator Control
[Gomaa, 2000]

Who decides which elevator should service the request?

Why?

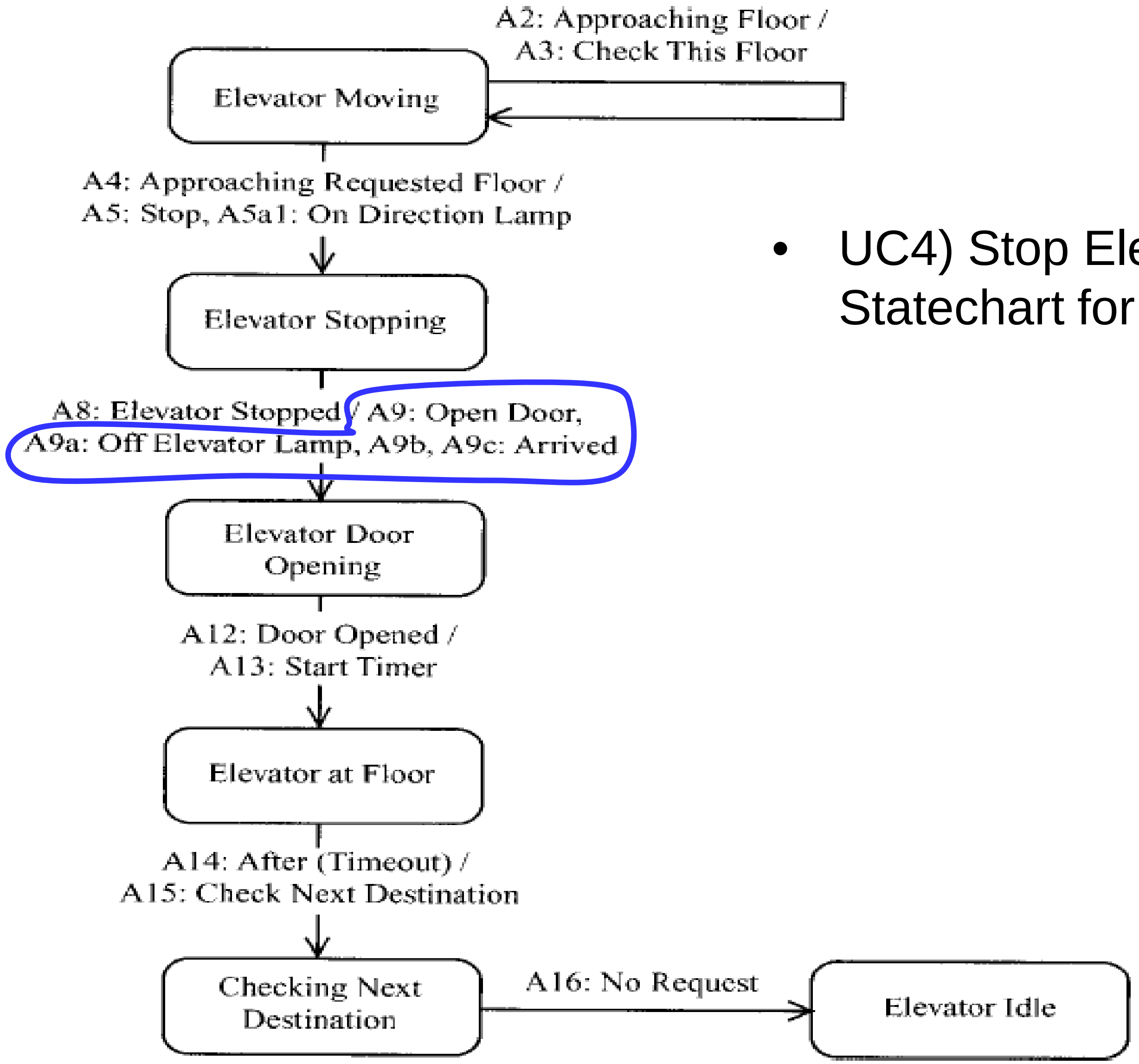
(2.3.2) Dynamic Modeling : Collaboration Diagram

- UC4) Stop Elevator at Floor use case: ***state-dependent !***


«external output device» : DirectionLamp
A5a.1: Direction Lamp Output
«output device interface» : DirectionLampInterface
«external output device» : ElevatorLamp
A9a.1: Elevator Lamp Output
«ouput device interface» : Elevator LampInterface
«external output device» : Motor
A7: Motor Response
A6: Stop Motor Command
«output device interface» : MotorInterface
A5a: On Direction Lamp
«timer» : DoorTimer
A14: After (Timeout)
A13: Start Timer
A9a: Off Elevator Lamp
A5: Stop
A8: Elevator Stopped
A1: Arrival Sensor Input
: Arrival Sensor
«input device interface» : Arrival SensorInterface
A2: Approaching Floor (Floor #)
«state dependent control» : Elevator Control
A9: Open Door
«output device interface» : DoorInterface
A10: Open Door Command
A11: Door Response
A12: Door Opened
«external output device» : Door
A4: Approaching Requested Floor (Floor #, direction)
A16: No Request
A3: Check This Floor (Floor #)
A9b: Arrived (Floor #)
A15: Check Next Destination
A9c: Arrived (Floor #)
«entity» : ElevatorStatus &Plan
«coordinator» : Scheduler

[Gomaa, 2000]

(2.3.3) Dynamic Modeling : Statechart Diagram

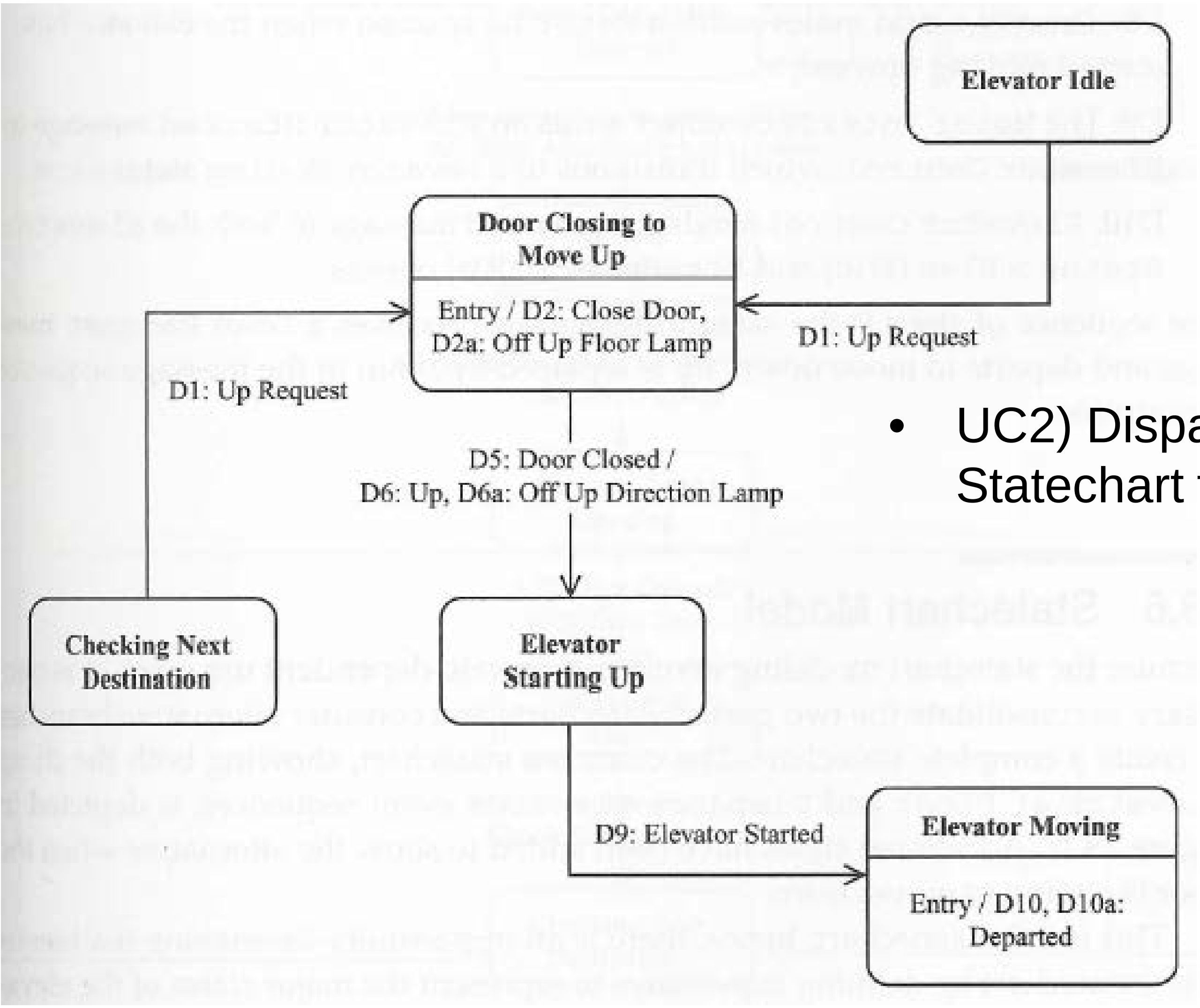


A2: Approaching Floor / A3: Check This Floor
Elevator Moving
A4: Approaching Requested Floor / A5: Stop, A5a1: On Direction Lamp
Elevator Stopping
A8: Elevator Stopped / A9: Open Door, A9a: Off Elevator Lamp, A9b, A9c: Arrived
Elevator Door Opening
A12: Door Opened / A13: Start Timer
Elevator at Floor
A14: After (Timeout) / A15: Check Next Destination
Checking Next Destination
A16: No Request
Elevator Idle

- UC4) Stop Elevator at Floor use case: Statechart for Elevator Control

[Gomaa, 2000]

(2.3.3) Dynamic Modeling : Statechart Diagram

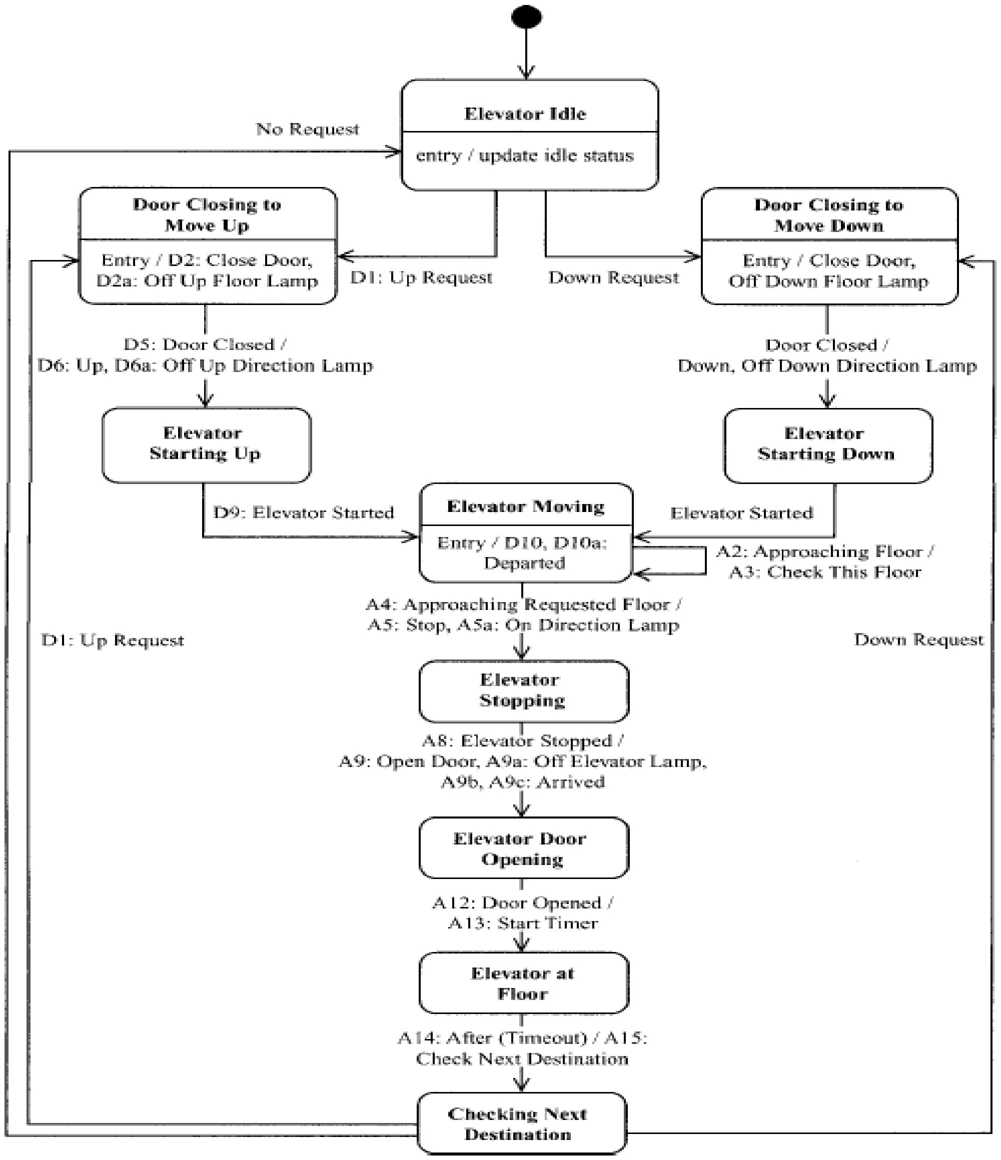

Elevator Idle
D1: Up Request
Door Closing to Move Up
Entry / D2: Close Door, D2a: Off Up Floor Lamp
D1: Up Request
D5: Door Closed / D6: Up, D6a: Off Up Direction Lamp
Checking Next Destination
Elevator Starting Up
D9: Elevator Started
Elevator Moving
Entry / D10, D10a: Departed

- UC2) Dispatch Elevator use case: Statechart for Elevator Control

Phase 3. Design Modeling

3.1 Synthesize analysis model artifacts
3.2 Design overall software architecture
3.3 Design distributed component-base software architecture
3.4 Design the concurrent task architecture for each subsystems
3.5 Analyze the performance of the design
3.6 Design the classes in each subsystem
3.7 Develop the detailed software design for each subsystem
3.8 Analyze the performance of the real-time design in greater detail for each subsystem. If necessary, repeat 3.4-3.7.

(3.1.1) Synthesize statecharts

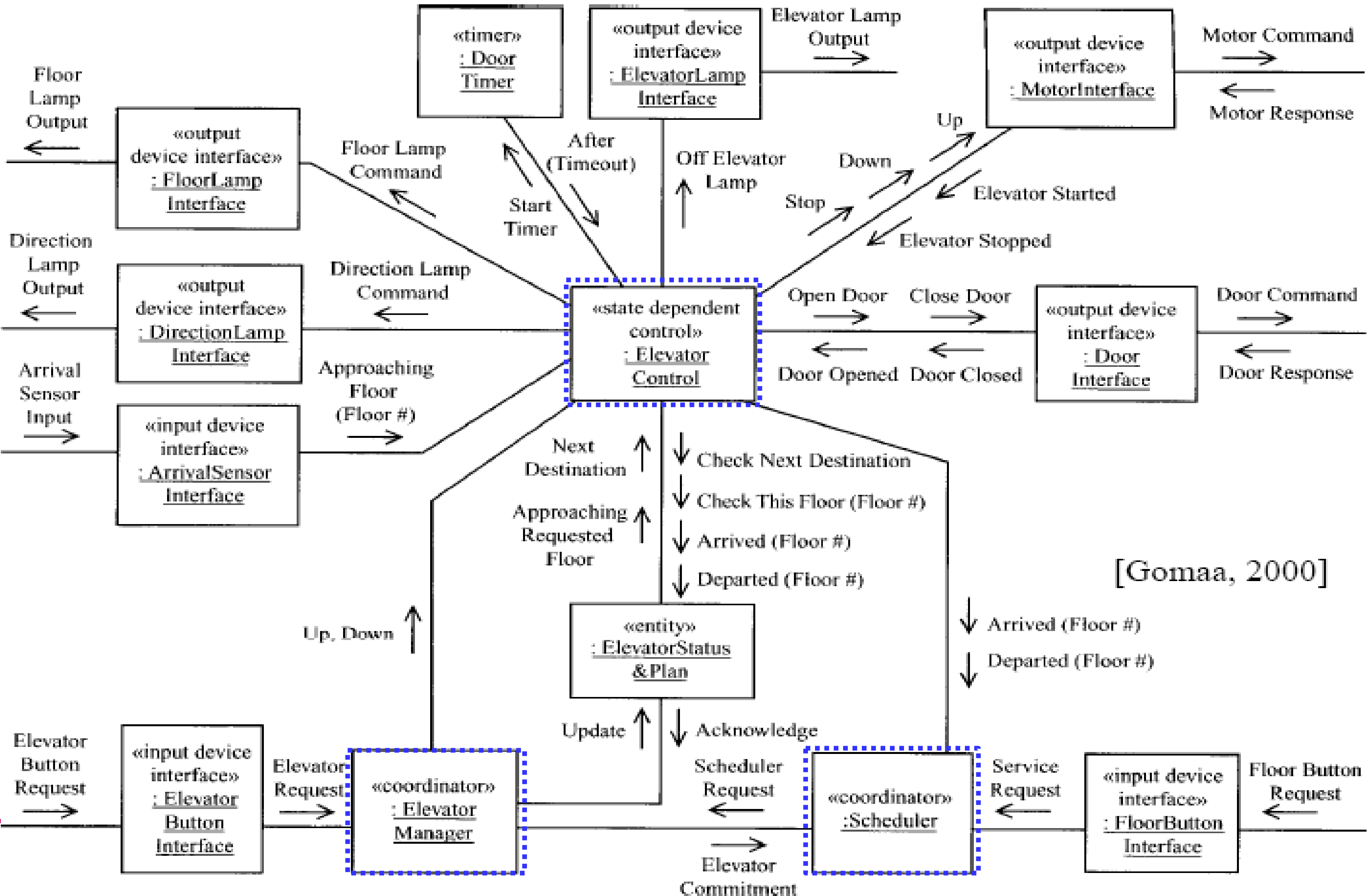


Elevator Idle
entry / update idle status
No Request
Door Closing to Move Up
Entry / D2: Close Door, D2a: Off Up Floor Lamp
D1: Up Request
Down Request
Door Closing to Move Down
Entry / Close Door, Off Down Floor Lamp
D5: Door Closed / D6: Up, D6a: Off Up Direction Lamp
Door Closed / Down, Off Down Direction Lamp
Elevator Starting Up
Elevator Starting Down
D9: Elevator Started
Elevator Started
Elevator Moving
Entry / D10, D10a: Departed
A2: Approaching Floor / A3: Check This Floor
A4: Approaching Requested Floor / A5: Stop, A5a: On Direction Lamp
D1: Up Request
Down Request
Elevator Stopping
A8: Elevator Stopped / A9: Open Door, A9a: Off Elevator Lamp, A9b, A9c: Arrived
Elevator Door Opening
A12: Door Opened / A13: Start Timer
Elevator at Floor
A14: After (Timeout) / A15: Check Next Destination
Checking Next Destination

- Flat statechart
- Can make it a hierarchical statechart by defining superstates

[Gomaa, 2000]

(3.1.2) Synthesize a collaboration model
(from collaboration diagrams in 2.3.2)



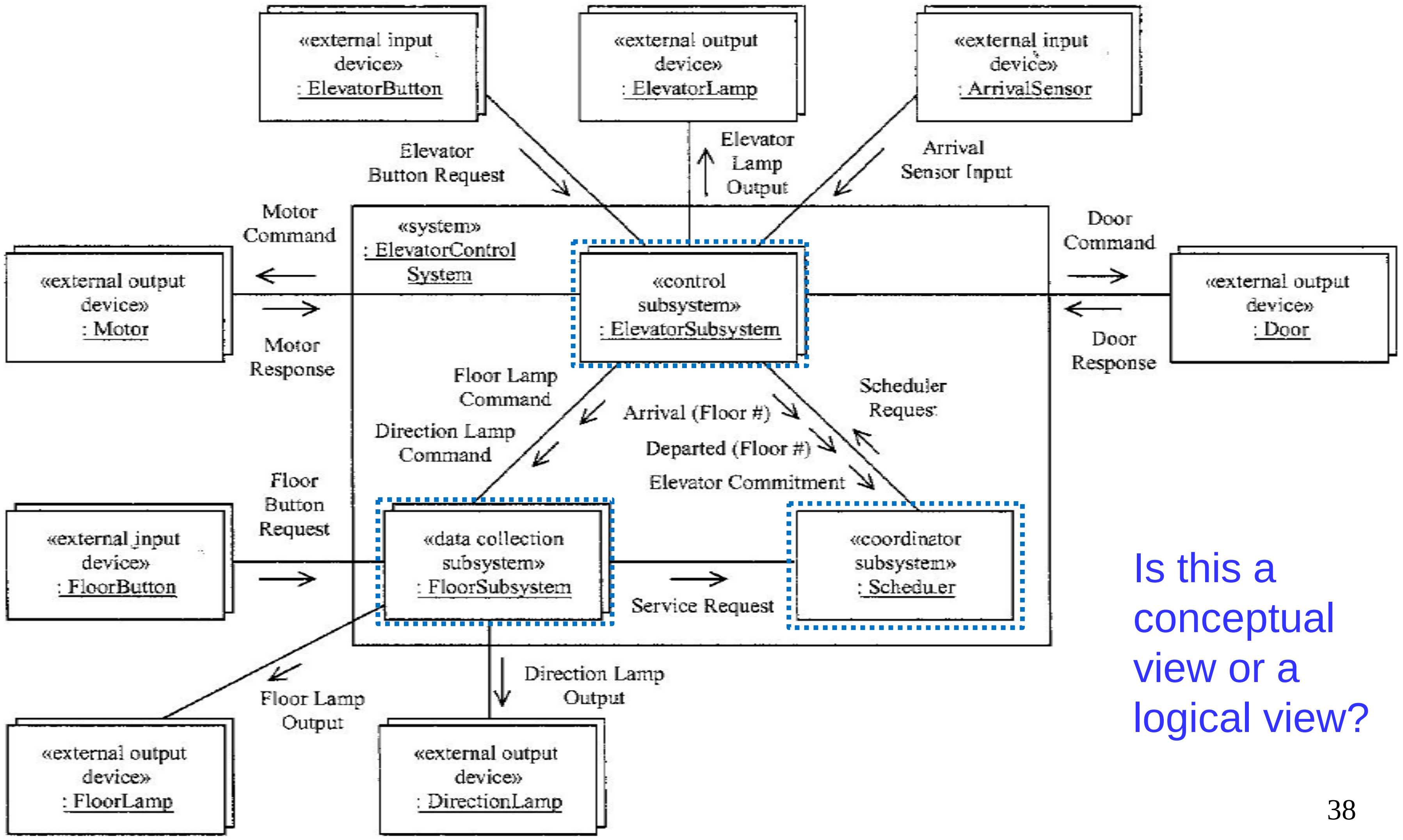
[Gomaa, 2000]

(3.2) Overall Software Architecture Design

- The system is structured into subsystems.
  - For distributed application, the geographical location takes precedence.
- The overall system architecture consists of
  - Multiple instances of the Elevator Subsystem (one per Elevator)
  - Multiple instances of the Floor Subsystem (one per floor)
  - One instance of the Scheduler Subsystem

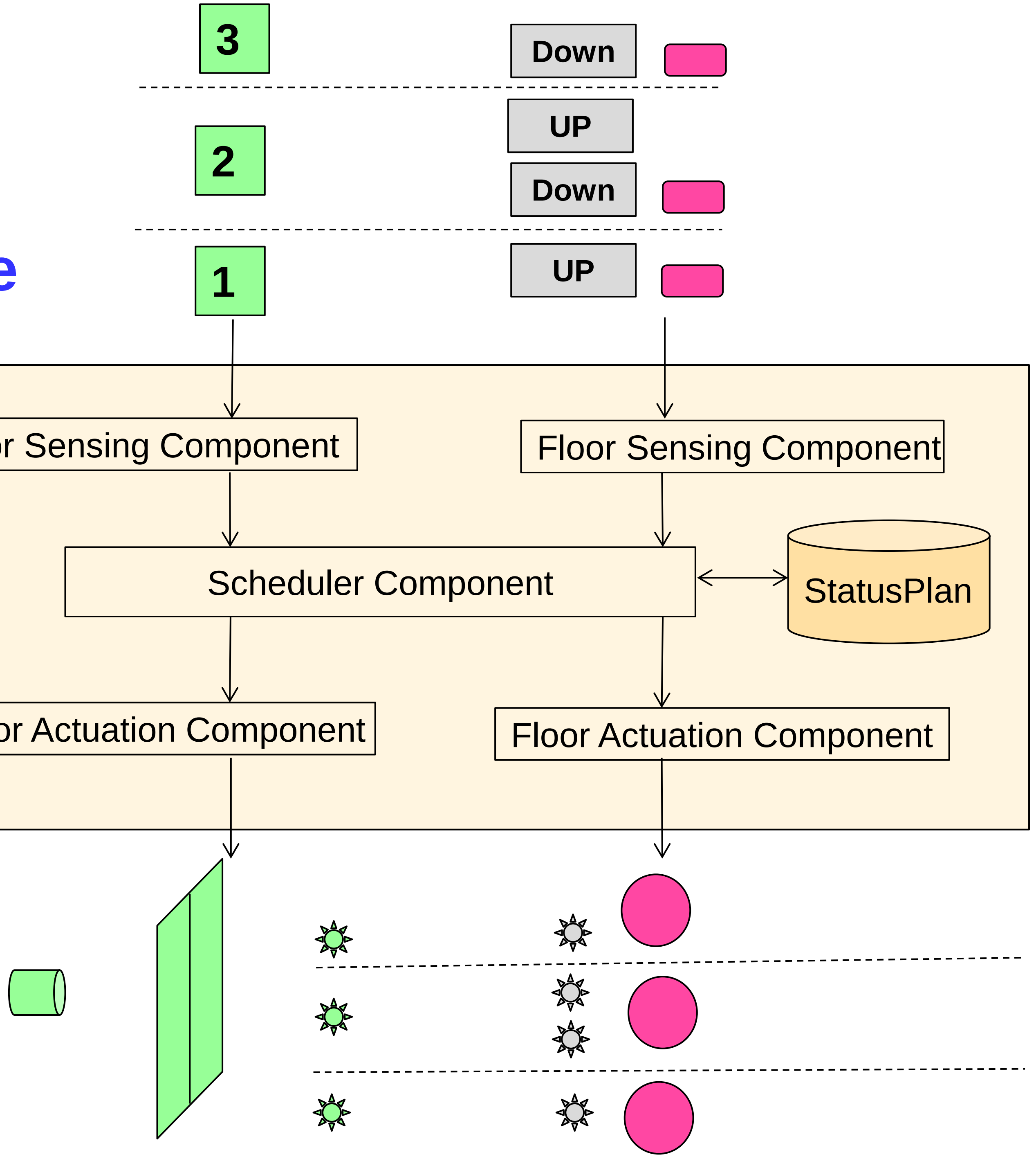
(3.2) Overall Software Architecture Design

- **Overall system architecture**


«external input device» : ElevatorButton
«external output device» : ElevatorLamp
«external input device» : ArrivalSensor
Elevator Button Request
Elevator Lamp Output
Arrival Sensor Input
Motor Command
«system» : ElevatorControl System
Door Command
«external output device» : Motor
«control subsystem» : ElevatorSubsystem
«external output device» : Door
Motor Response
Door Response
Floor Lamp Command
Arrival (Floor #)
Scheduler Request
Direction Lamp Command
Departed (Floor #)
Floor Button Request
Elevator Commitment
«external input device» : FloorButton
«data collection subsystem» : FloorSubsystem
«coordinator subsystem» : Scheduler
Service Request
Direction Lamp Output
Floor Lamp Output
«external output device» : FloorLamp
«external output device» : DirectionLamp

Is this a conceptual view or a logical view?

Conceptual view of an Alternative Architecture

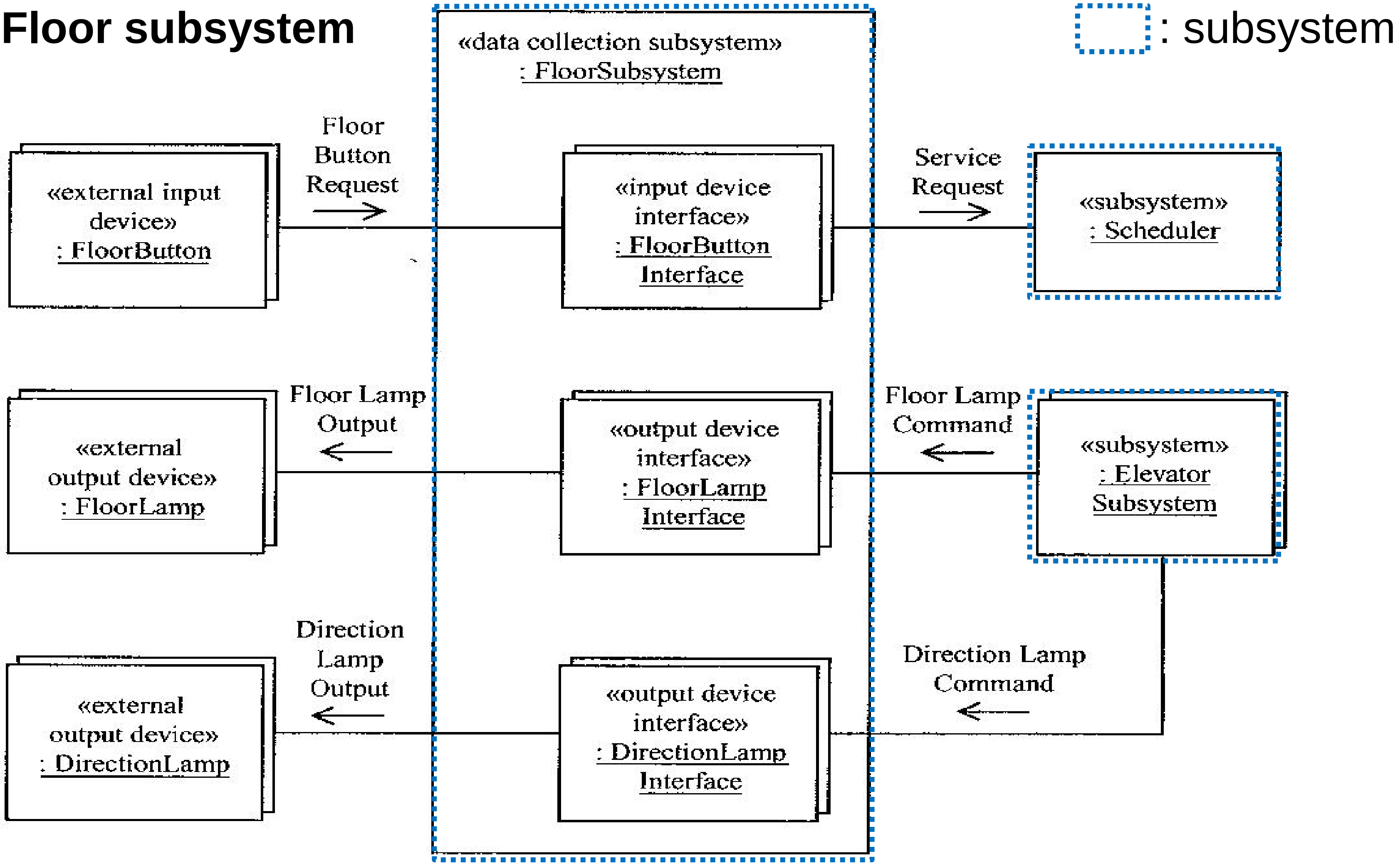

3
Down
UP
2
Down
1
UP
Elevator Sensing Component
Floor Sensing Component
Scheduler Component
StatusPlan
Elevator Actuation Component
Floor Actuation Component

Comparison of the Two Architectures

Perspectives	Gomaa	Alternative
Architecture Style		SAC
Buildability (Will the approach make development easy?)	Scheduler complexity appears to be the same.	
	Elevator subsystem looks complicated.	
Scalability(No. of Floors, No. of Elevators)	Appears to be the same	
Collaborative Development (Can be viewed as buildability)	Decomposing FloorSubsystem and ElevatorSubsystem further for work assignment is not easy. => Each subsystem needs to understand the other subsystems.	Except for Scheduler, teams developing other components need to understand only one adjacent component.
Design Process	• Various input and output objects are introduced as the result of analysis. • Patterns are used mostly for task determination	• Various input and output objects are introduced after the overall architecture is determined • Patterns are used to introduce such objects

(3.2) Overall Software Architecture Design

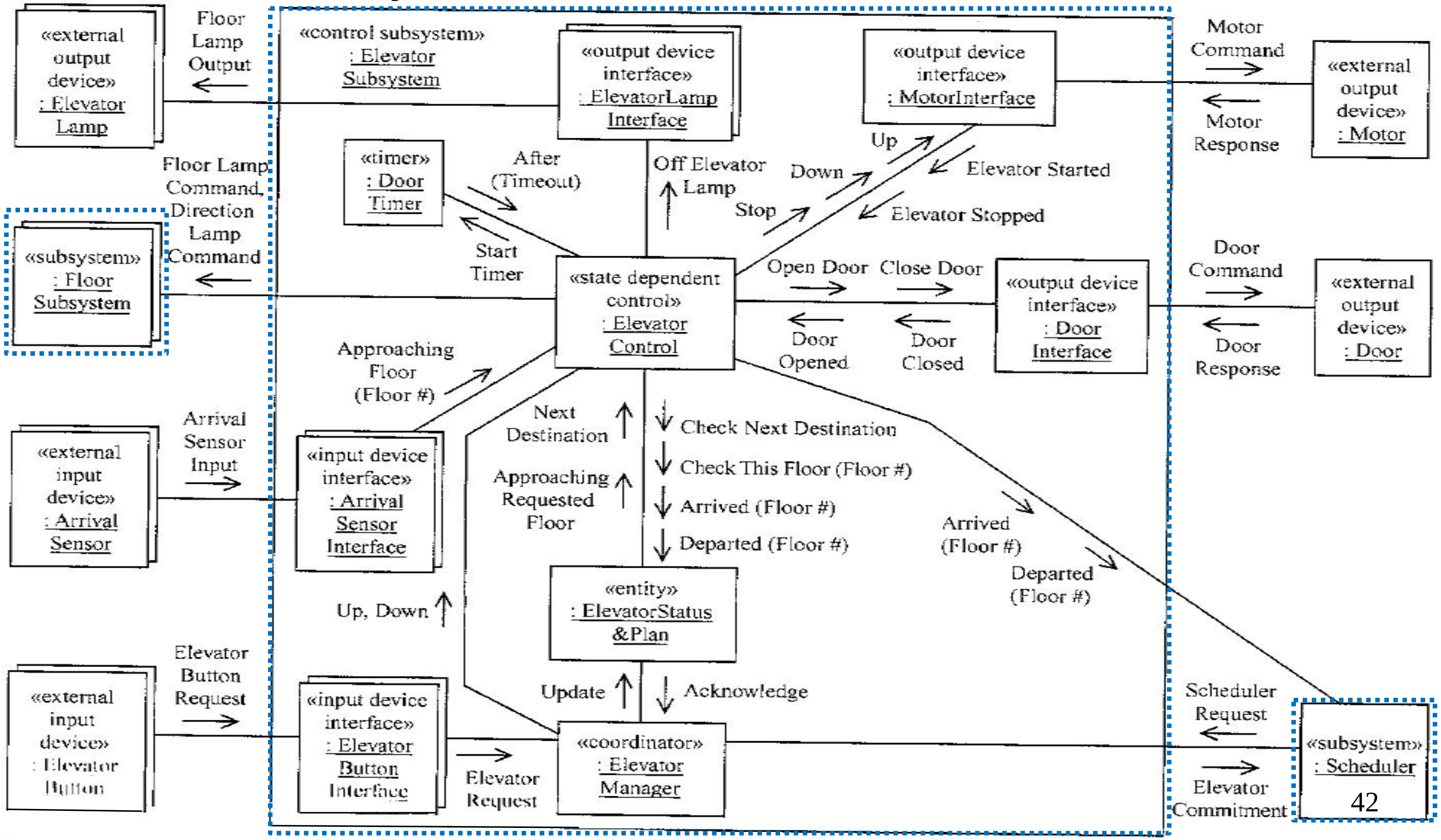
- **Floor subsystem**



: subsystem
«data collection subsystem» : FloorSubsystem
«external input device» : FloorButton
Floor Button Request
«input device interface» : FloorButton Interface
Service Request
«subsystem» : Scheduler
«external output device» : FloorLamp
Floor Lamp Output
«output device interface» : FloorLamp Interface
Floor Lamp Command
«subsystem» : Elevator Subsystem
«external output device» : DirectionLamp
Direction Lamp Output
«output device interface» : DirectionLamp Interface
Direction Lamp Command

(3.2) Overall Software Architecture Design

- **Elevator subsystem**

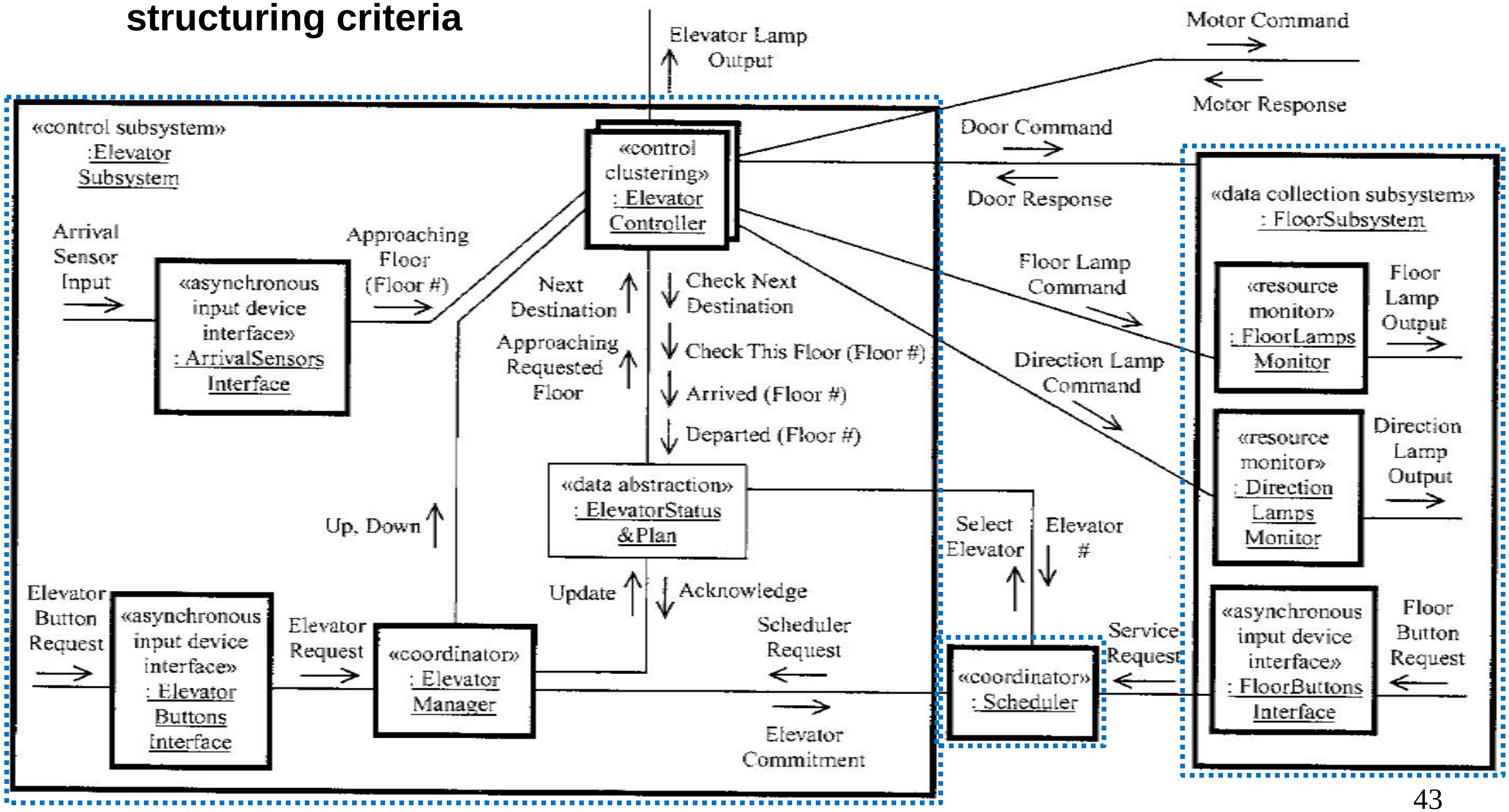


«external output device» : Elevator Lamp
Floor Lamp Output
«control subsystem» : Elevator Subsystem
«output device interface» : ElevatorLamp Interface
«output device interface» : MotorInterface
Motor Command
Motor Response
«external output device» : Motor
«timer» : Door Timer
After (Timeout)
Start Timer
Off Elevator Lamp
Up
Down
Stop
Elevator Started
Elevator Stopped
Floor Lamp Command, Direction Lamp Command
«subsystem» : Floor Subsystem
«state dependent control» : Elevator Control
Open Door
Close Door
Door Opened
Door Closed
«output device interface» : Door Interface
Door Command
Door Response
«external output device» : Door
Approaching Floor (Floor #)
Arrival Sensor Input
«external input device» : Arrival Sensor
«input device interface» : Arrival Sensor Interface
Next Destination
Approaching Requested Floor
Check Next Destination
Check This Floor (Floor #)
Arrived (Floor #)
Departed (Floor #)
Arrived (Floor #)
Departed (Floor #)
«entity» : ElevatorStatus &Plan
Up, Down
Update
Acknowledge
Elevator Button Request
«external input device» : Elevator Button
«input device interface» : Elevator Button Interface
Elevator Request
«coordinator» : Elevator Manager
Scheduler Request
Elevator Commitment
«subsystem» : Scheduler

(3.4) Concurrent Task Architecture Design: Task Architecture

Exercise: Explain what criterion has been used for each task.

- **Analyze all objects on the collaboration diagrams and apply task structuring criteria**



Elevator Lamp Output
Motor Command
Motor Response
«control subsystem» :Elevator Subsystem
«control clustering» : Elevator Controller
Door Command
Door Response
«data collection subsystem» : FloorSubsystem
Arrival Sensor Input
«asynchronous input device interface» : ArrivalSensors Interface
Approaching Floor (Floor #)
Next Destination
Approaching Requested Floor
Check Next Destination
Check This Floor (Floor #)
Arrived (Floor #)
Departed (Floor #)
Floor Lamp Command
Direction Lamp Command
«resource monitor» : FloorLamps Monitor
Floor Lamp Output
«resource monitor» : Direction Lamps Monitor
Direction Lamp Output
«data abstraction» : ElevatorStatus &Plan
Up, Down
Select Elevator
Elevator #
Update
Acknowledge
Elevator Button Request
«asynchronous input device interface» : Elevator Buttons Interface
Elevator Request
«coordinator» : Elevator Manager
Scheduler Request
Elevator Commitment
«coordinator» : Scheduler
Service Request
«asynchronous input device interface» : FloorButtons Interface
Floor Button Request

Figure 18.19 *Non-distributed Elevator Control System: task architecture*

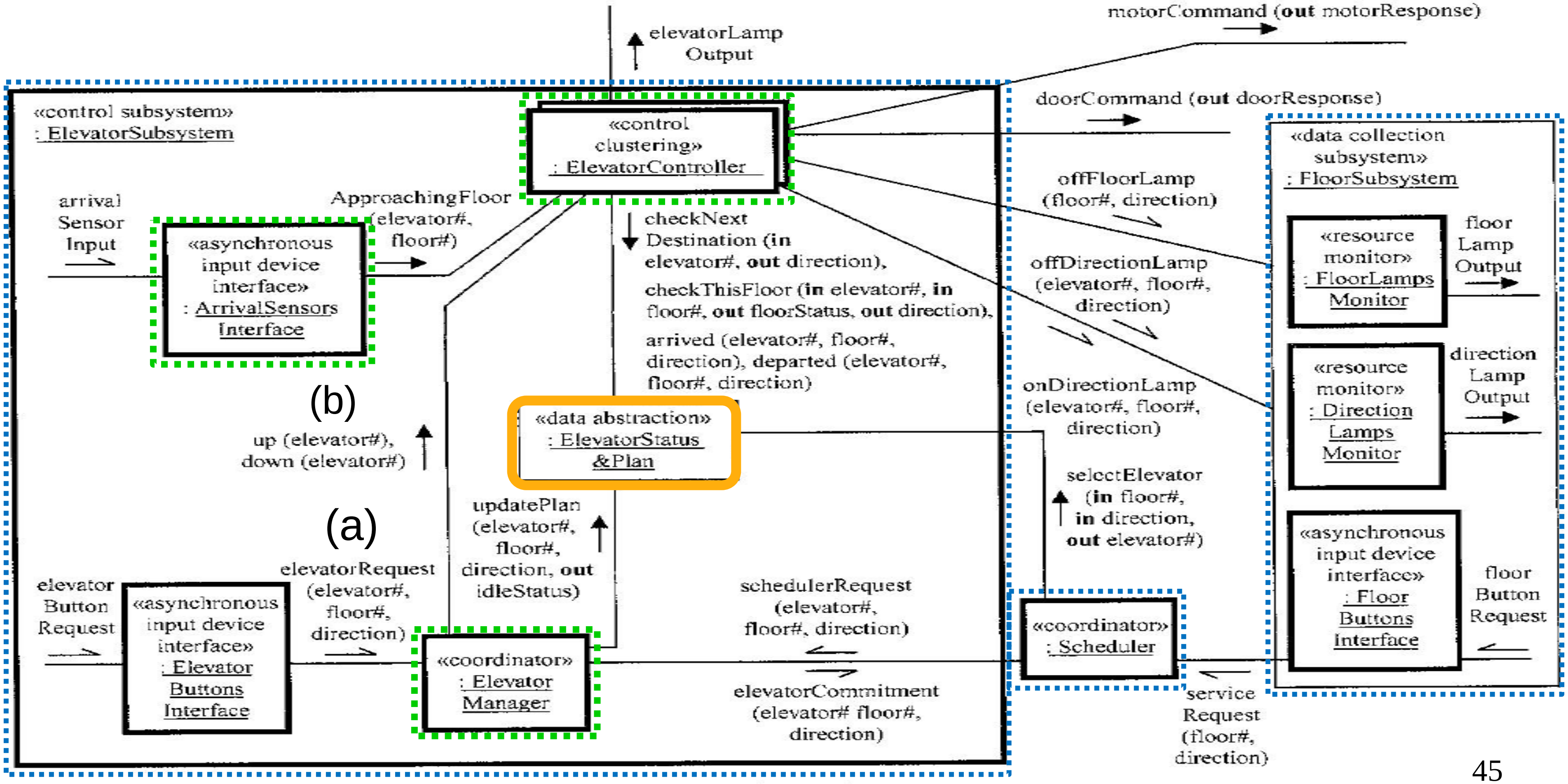
Task Clustering Criteria

- Analysis model introduces lots of objects that can potentially map to tasks that will execute concurrently.
- However, a large number of small tasks will increase system complexity and deteriorate performance.
- See Appendix I for task clustering criteria

(3.4) Concurrent Task Architecture Design: Task Interfaces

Exercise: Explain what criterion has been used for each task.

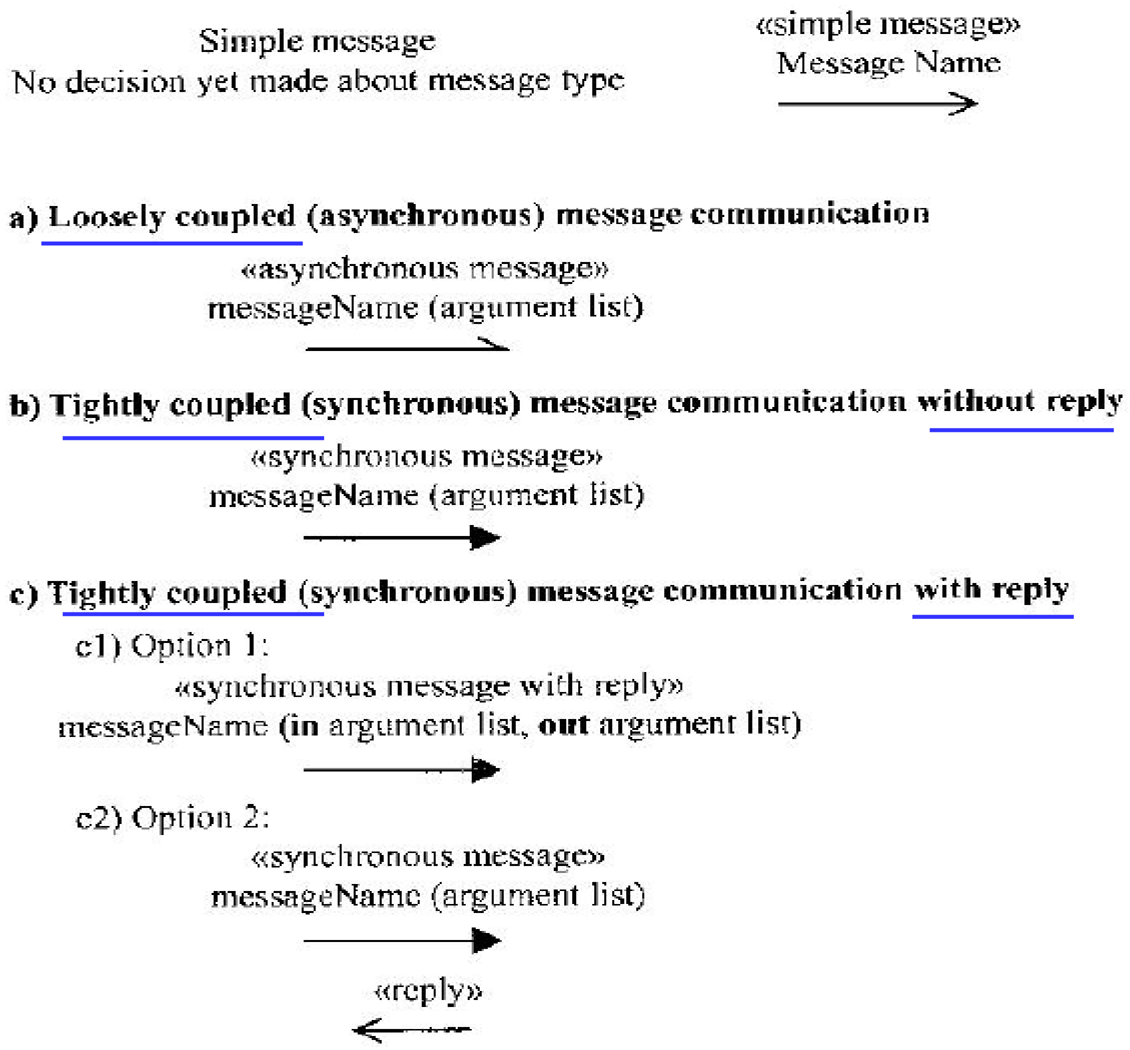
- Message interface between Tasks is considered: (Why between tasks, not modules?)
  (a) Loose coupling: E.g. ElevatorButtonsInterface and ElevatorManager tasks
  (b) Tight coupling: E.g. ElevatorManager and ElevatorController tasks



motorCommand (out motorResponse)
elevatorLamp Output
doorCommand (out doorResponse)
«control subsystem» : ElevatorSubsystem
«control clustering» : ElevatorController
«data collection subsystem» : FloorSubsystem
arrival Sensor Input
ApproachingFloor (elevator#, floor#)
«asynchronous input device interface» : ArrivalSensors Interface
checkNext Destination (in elevator#, out direction),
checkThisFloor (in elevator#, in floor#, out floorStatus, out direction),
arrived (elevator#, floor#, direction), departed (elevator#, floor#, direction)
offFloorLamp (floor#, direction)
offDirectionLamp (elevator#, floor#, direction)
onDirectionLamp (elevator#, floor#, direction)
«resource monitor» : FloorLamps Monitor
floor Lamp Output
«resource monitor» : Direction Lamps Monitor
direction Lamp Output
(b)
up (elevator#), down (elevator#)
«data abstraction» : ElevatorStatus &Plan
(a)
updatePlan (elevator#, floor#, direction, out idleStatus)
selectElevator (in floor#, in direction, out elevator#)
elevator Button Request
«asynchronous input device interface» : Elevator Buttons Interface
elevatorRequest (elevator#, floor#, direction)
«coordinator» : Elevator Manager
schedulerRequest (elevator#, floor#, direction)
elevatorCommitment (elevator# floor#, direction)
«coordinator» : Scheduler
service Request (floor#, direction)
«asynchronous input device interface» : Floor Buttons Interface
floor Button Request

Figure 18.20 *Non-distributed Elevator Control System: task interfaces*

UML Message Notation



Simple message
No decision yet made about message type

«simple message»
Message Name

a) Loosely coupled (asynchronous) message communication

«asynchronous message»
messageName (argument list)

b) Tightly coupled (synchronous) message communication without reply

«synchronous message»
messageName (argument list)

c) Tightly coupled (synchronous) message communication with reply

c1) Option 1:

«synchronous message with reply»
messageName (**in** argument list, **out** argument list)

c2) Option 2:

«synchronous message»
messageName (argument list)

«reply»

Appendix I. Task Structuring

Task Structuring

- During the task structuring phase, a task architecture (= task view) is developed in which

  (1) the system is structured into concurrent tasks and

  (2) the task interfaces and interconnections are defined.

- Task structuring criteria (= guidelines or patterns): assist in mapping an OO analysis model to a concurrent tasking architecture.

- Task: an active object with its own thread of control

  (<–> A passive object)

Task Structuring Categories

(1) I/O task structuring criteria

- Address how device interface objects are mapped to I/O tasks and when an I/O task is activated

(2) Internal task structuring criteria

- Address how internal objects are mapped to internal tasks and when an internal task is activated

(3) Task priority criteria

- Address the importance of executing a given task relative to others

(4) Task clustering criteria

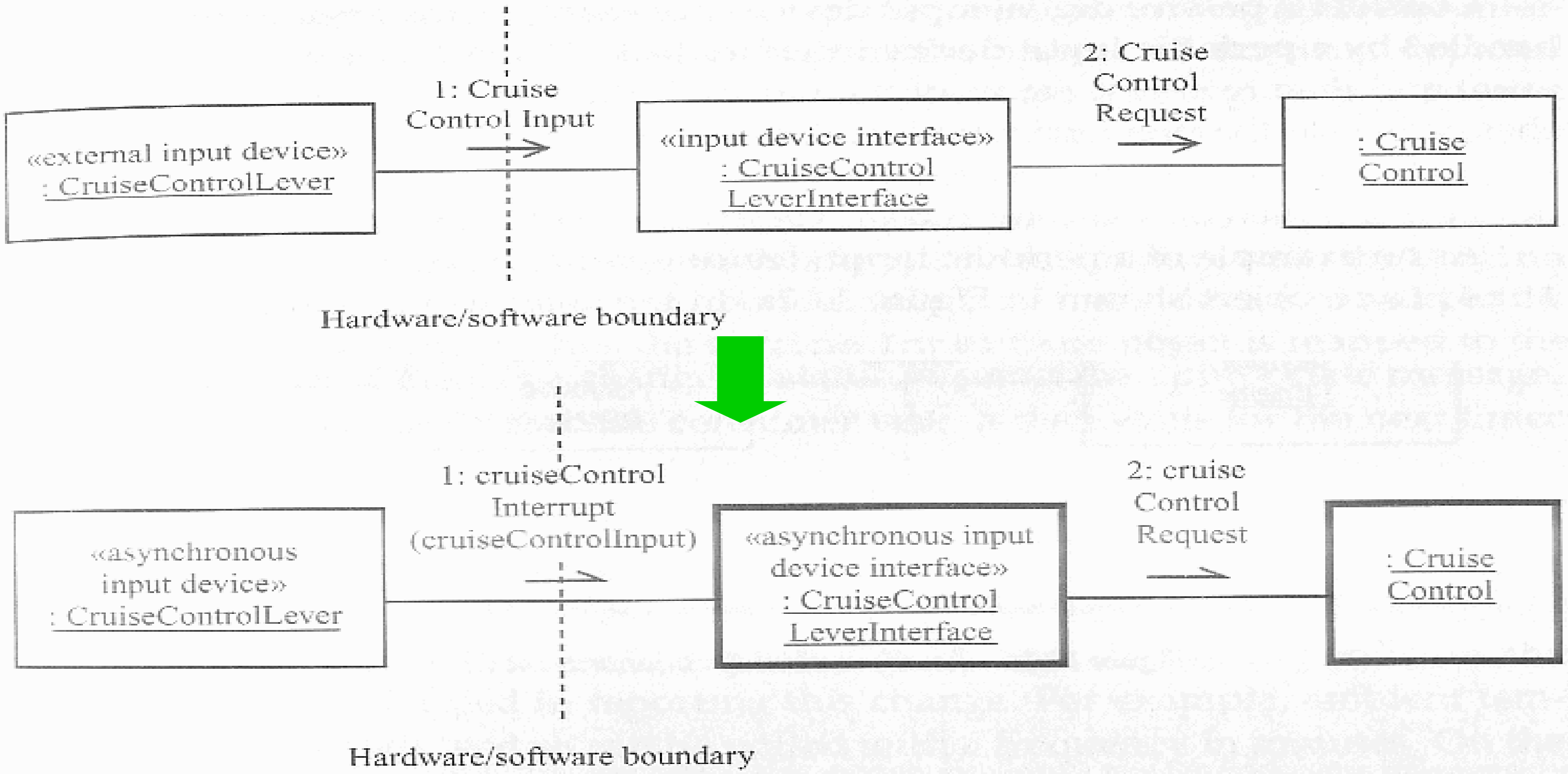
- Address whether and how objects should be grouped into concurrent tasks

(5) Task inversion criteria

- Used for merging tasks to reduce task overhead either during initial task structuring or during design restructuring

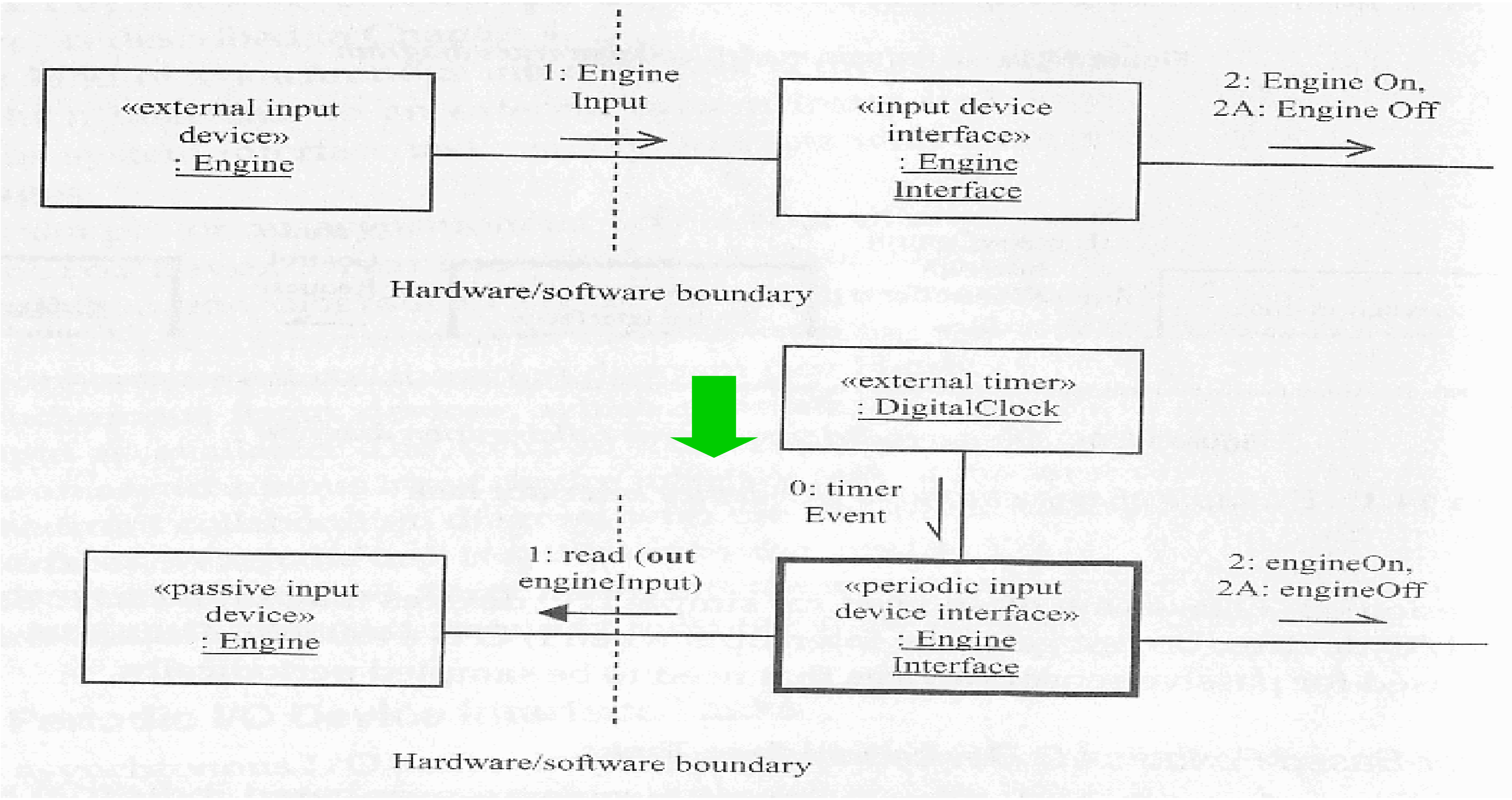
(1) I/O Task Structuring Criteria

Asynchronous (= Aperiodic) I/O Device Interface Tasks


«external input device»
: CruiseControlLever
1: Cruise Control Input
«input device interface»
: CruiseControl LeverInterface
2: Cruise Control Request
: Cruise Control
Hardware/software boundary
«asynchronous input device»
: CruiseControlLever
1: cruiseControl Interrupt (cruiseControlInput)
«asynchronous input device interface»
: CruiseControl LeverInterface
2: cruise Control Request
: Cruise Control
Hardware/software boundary

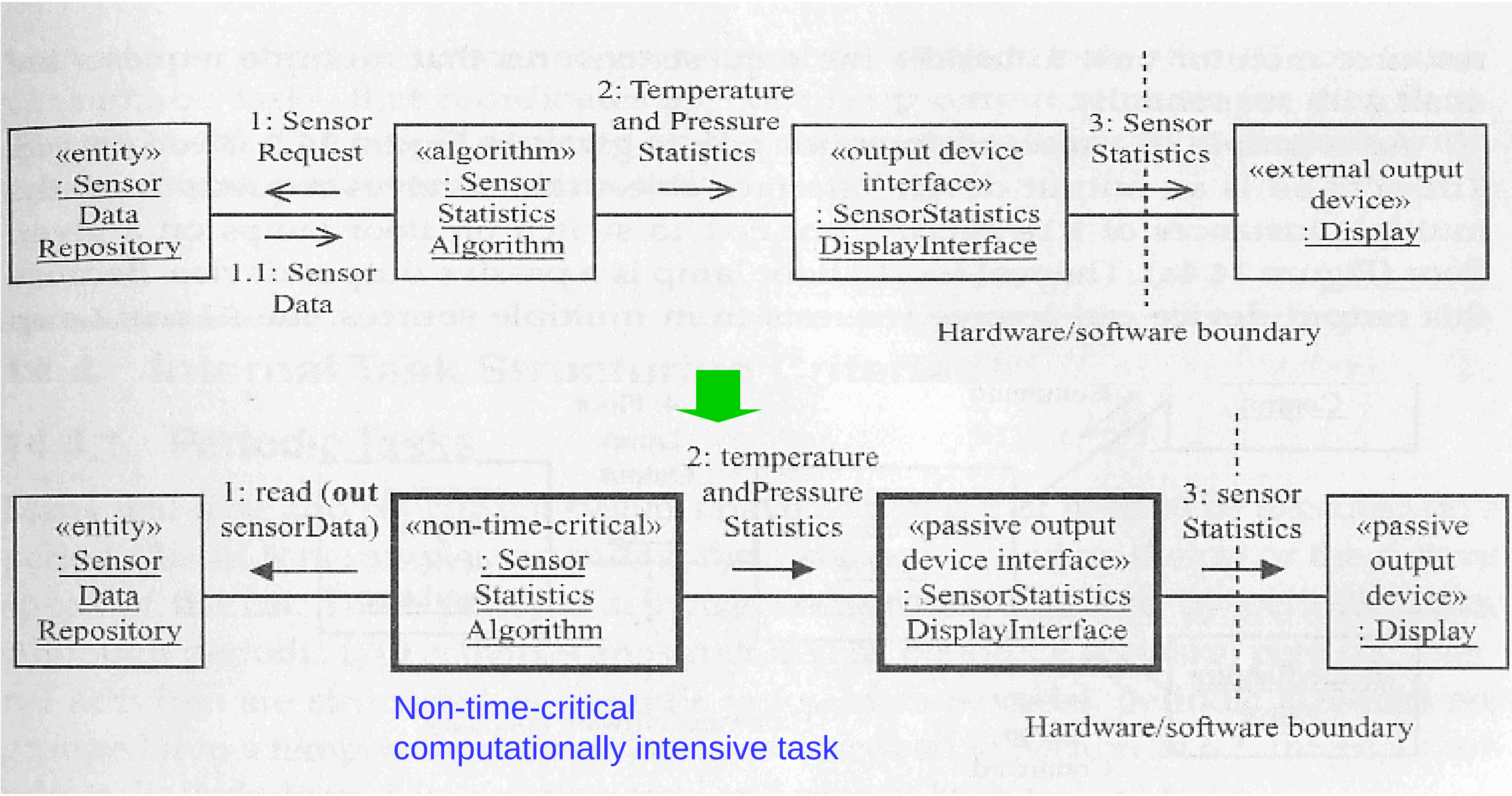
- Same as interrupt handling

Periodic I/O Device Interface Tasks


«external input device» : Engine
1: Engine Input
«input device interface» : Engine Interface
2: Engine On, 2A: Engine Off
Hardware/software boundary
«external timer» : DigitalClock
0: timer Event
«passive input device» : Engine
1: read (out engineInput)
«periodic input device interface» : Engine Interface
2: engineOn, 2A: engineOff
Hardware/software boundary

- Sensor-based industrial systems have many sensors.

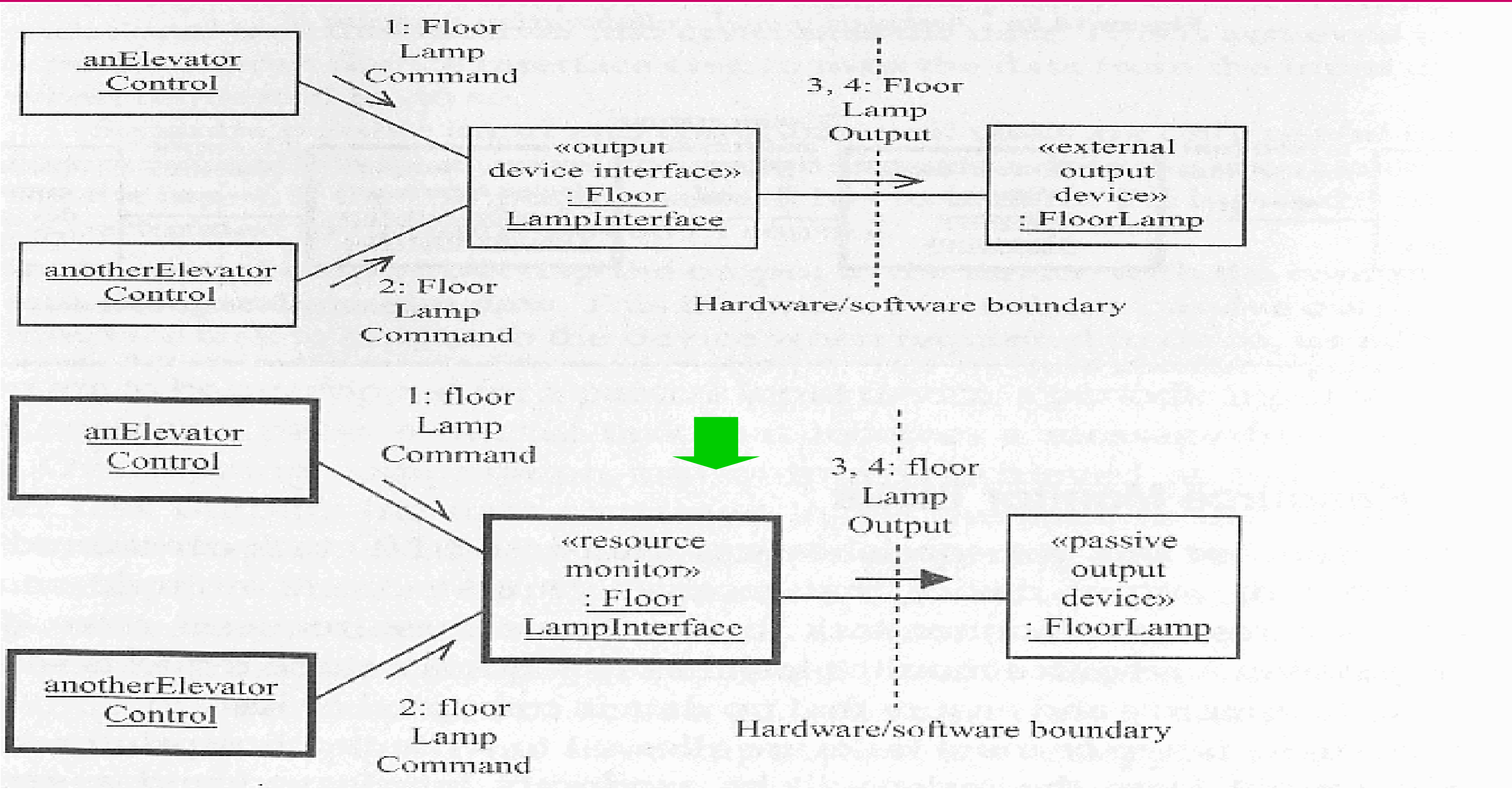
Passive I/O Device Interface Tasks



1: Sensor Request
«entity» : Sensor Data Repository
1.1: Sensor Data
«algorithm» : Sensor Statistics Algorithm
2: Temperature and Pressure Statistics
«output device interface» : SensorStatistics DisplayInterface
3: Sensor Statistics
«external output device» : Display
Hardware/software boundary
1: read (out sensorData)
«entity» : Sensor Data Repository
«non-time-critical» : Sensor Statistics Algorithm
2: temperature andPressure Statistics
«passive output device interface» : SensorStatistics DisplayInterface
3: sensor Statistics
«passive output device» : Display
Hardware/software boundary
Non-time-critical computationally intensive task

- Concurrency used: DisplayInterface displays sensor statistics while StatisticsAlgorithm is computing the next set of values to display.

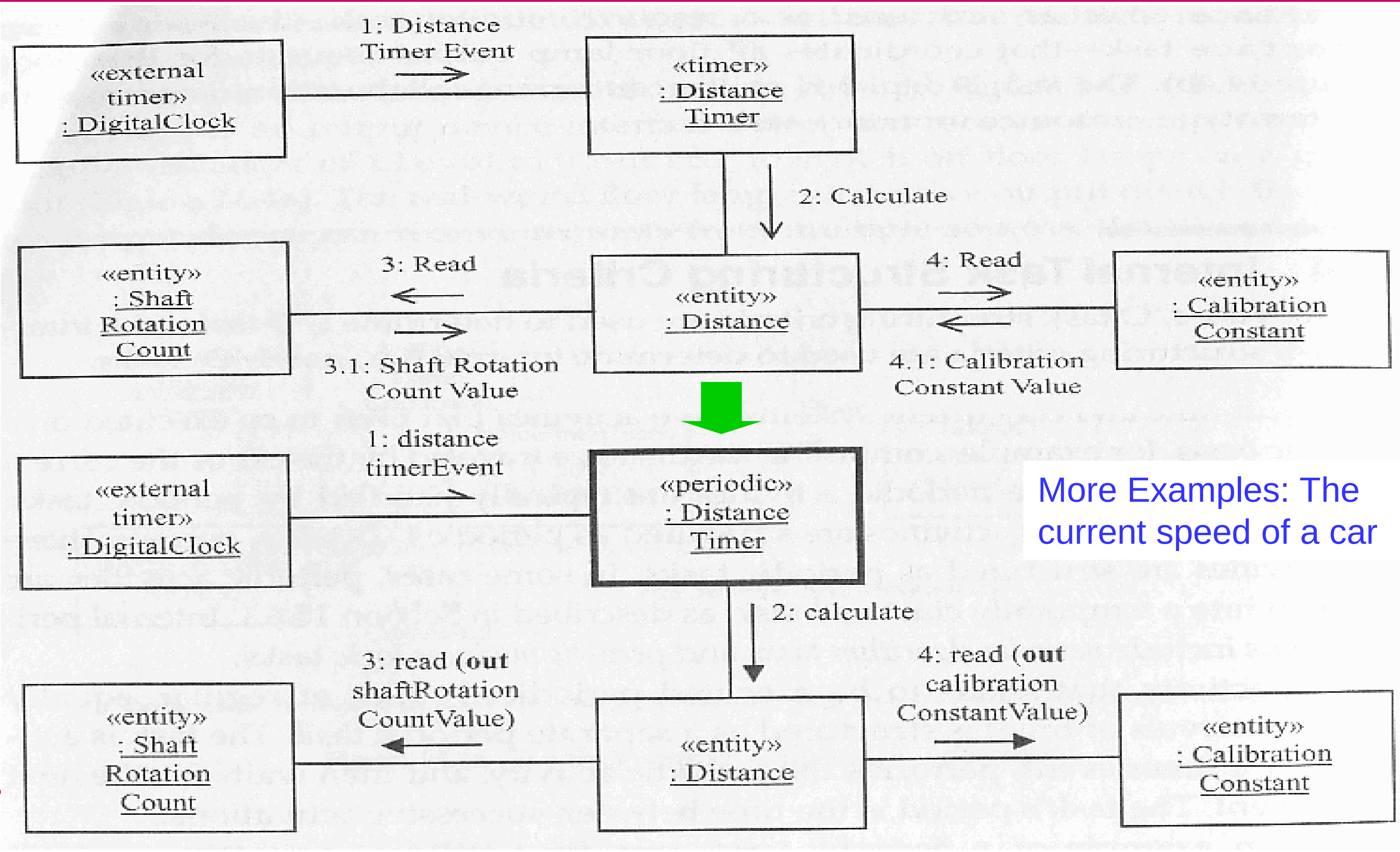
Resource Monitor Task
(A special case of Passive I/O Task)



- Requests from multiple sources are coordinated by Resource Monitor.

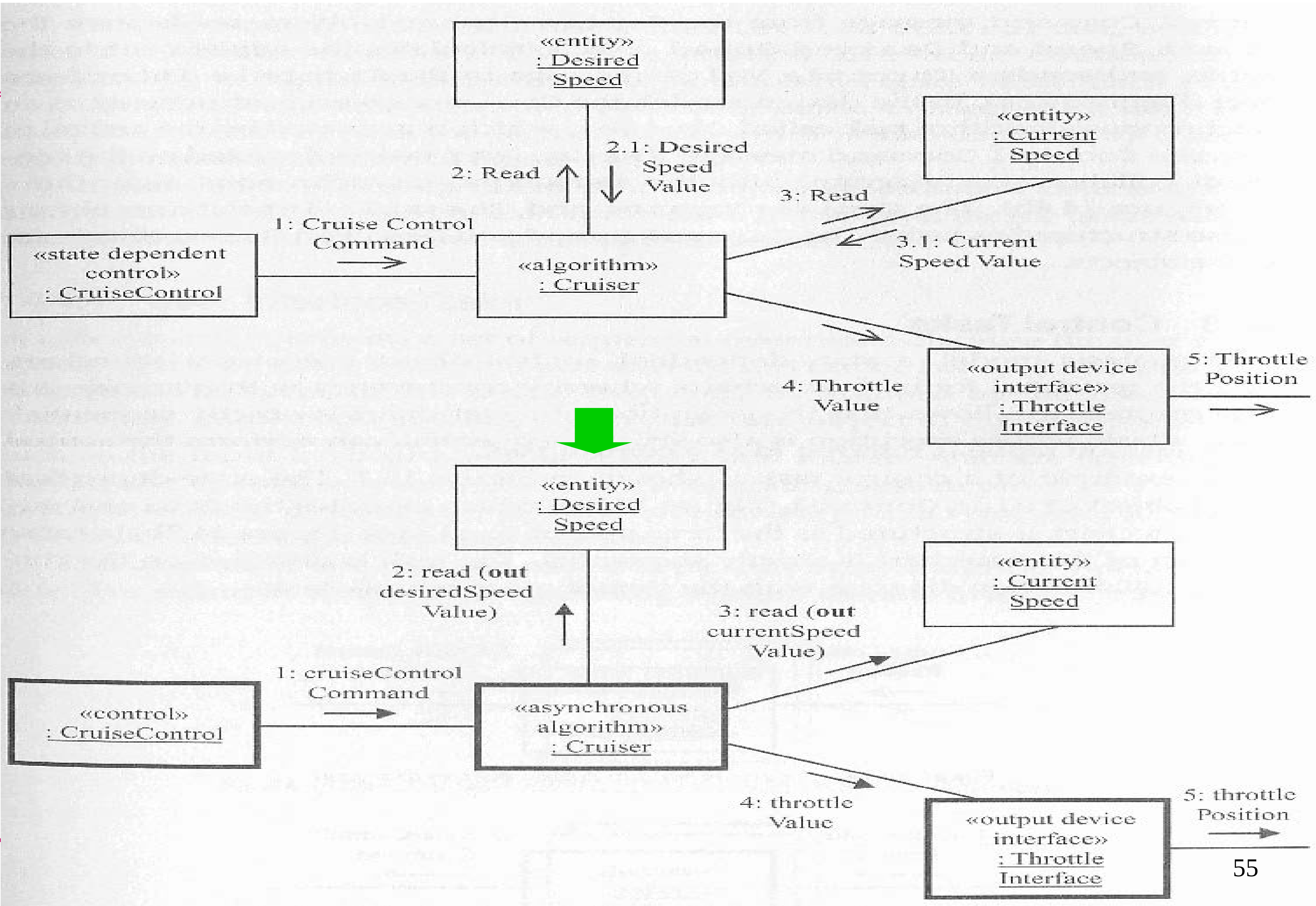
(2) Internal Task Structuring Criteria

Internal Periodic Task


1: Distance Timer Event
«external timer» : DigitalClock
«timer» : Distance Timer
2: Calculate
«entity» : Shaft Rotation Count
3: Read
3.1: Shaft Rotation Count Value
«entity» : Distance
4: Read
4.1: Calibration Constant Value
«entity» : Calibration Constant
1: distance timerEvent
«external timer» : DigitalClock
«periodic» : Distance Timer
2: calculate
3: read (out shaftRotation CountValue)
4: read (out calibration ConstantValue)
«entity» : Shaft Rotation Count
«entity» : Distance
«entity» : Calibration Constant

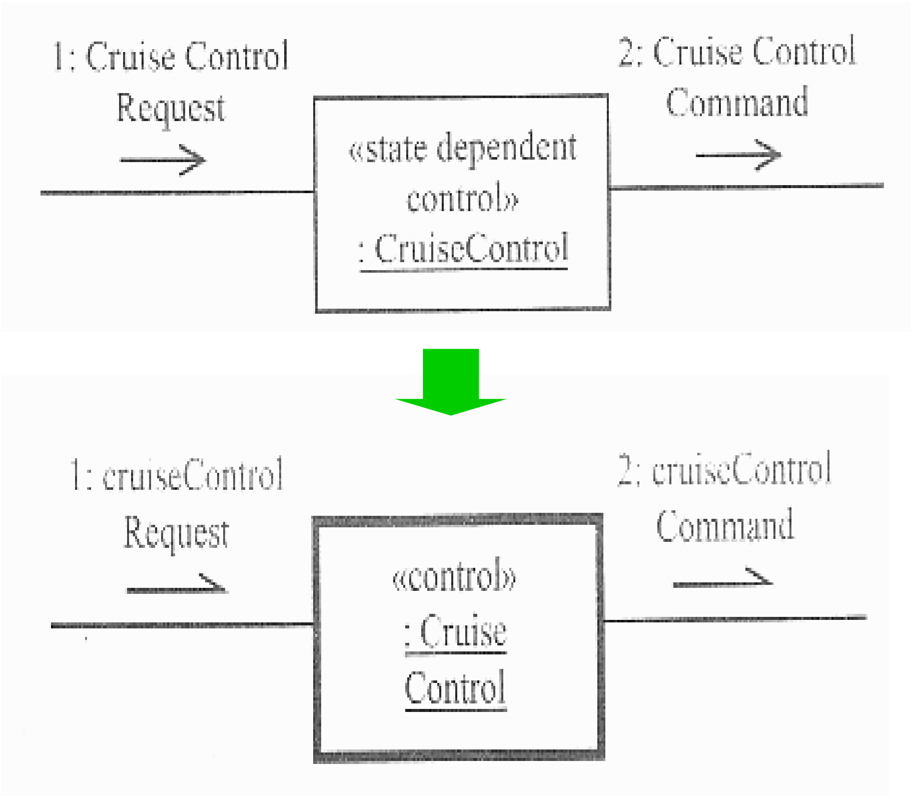
More Examples: The current speed of a car

Internal Asynchronous (= Aperiodic) Task

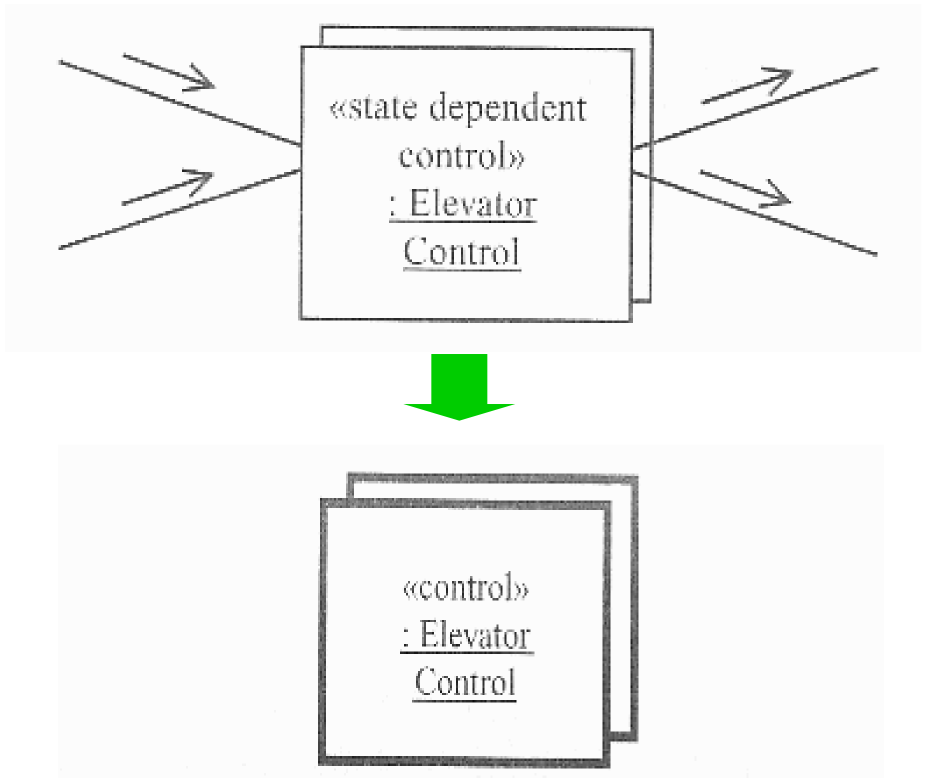

«entity» : Desired Speed
«entity» : Current Speed
2.1: Desired Speed Value
2: Read
3: Read
1: Cruise Control Command
3.1: Current Speed Value
«state dependent control» : CruiseControl
«algorithm» : Cruiser
«output device interface» : Throttle Interface
5: Throttle Position
4: Throttle Value
«entity» : Desired Speed
«entity» : Current Speed
2: read (out desiredSpeed Value)
3: read (out currentSpeed Value)
1: cruiseControl Command
«control» : CruiseControl
«asynchronous algorithm» : Cruiser
4: throttle Value
«output device interface» : Throttle Interface
5: throttle Position

Control Task

- Control Task


1: Cruise Control Request
2: Cruise Control Command
«state dependent control» : CruiseControl
1: cruiseControl Request
2: cruiseControl Command
«control» : Cruise Control

- Multiple Control Tasks of the same type

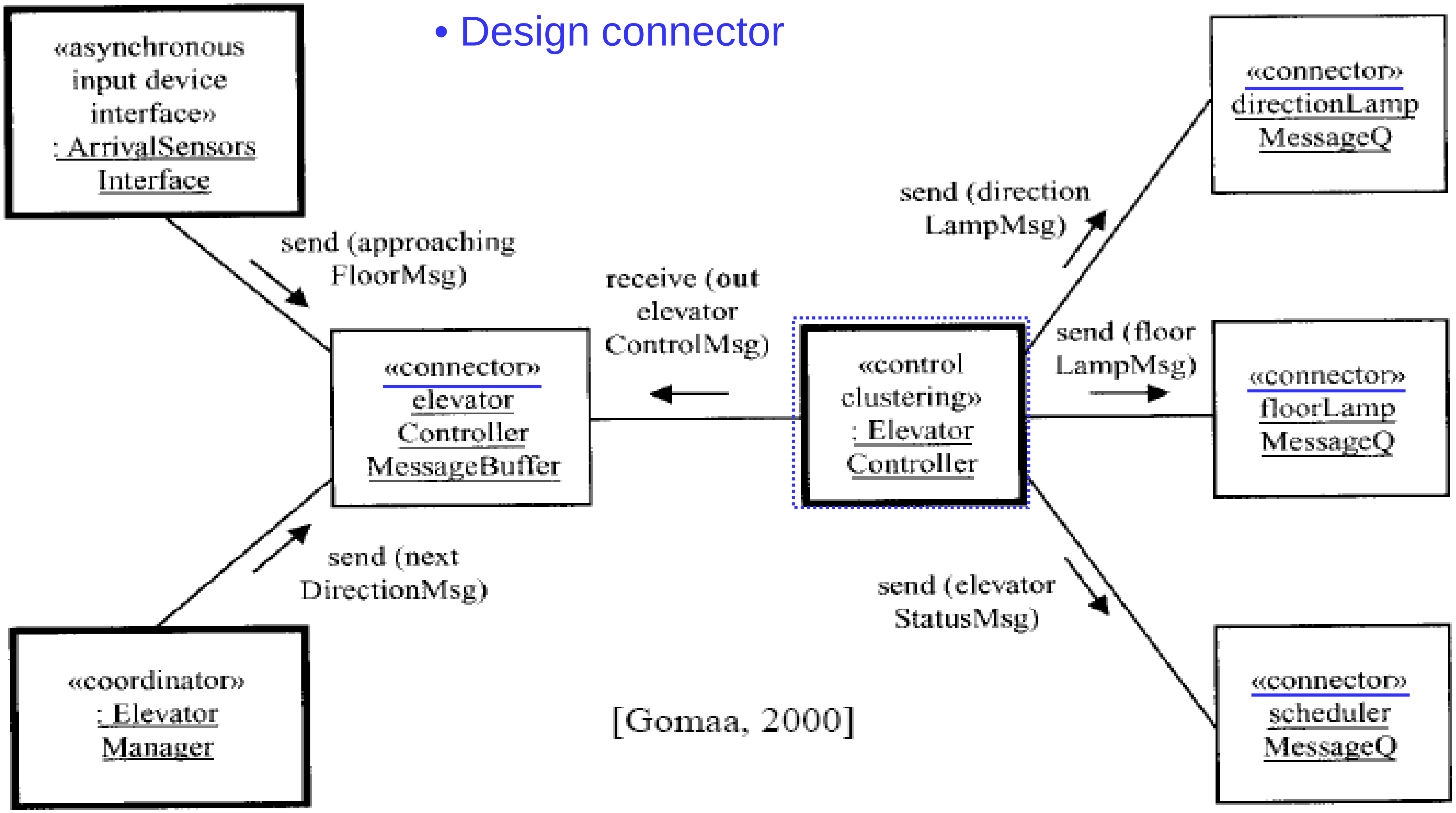

«state dependent control» : Elevator Control
«control» : Elevator Control

- Complicated state dependent component is mapped to a task.

Appendix II. Detailed Software Design

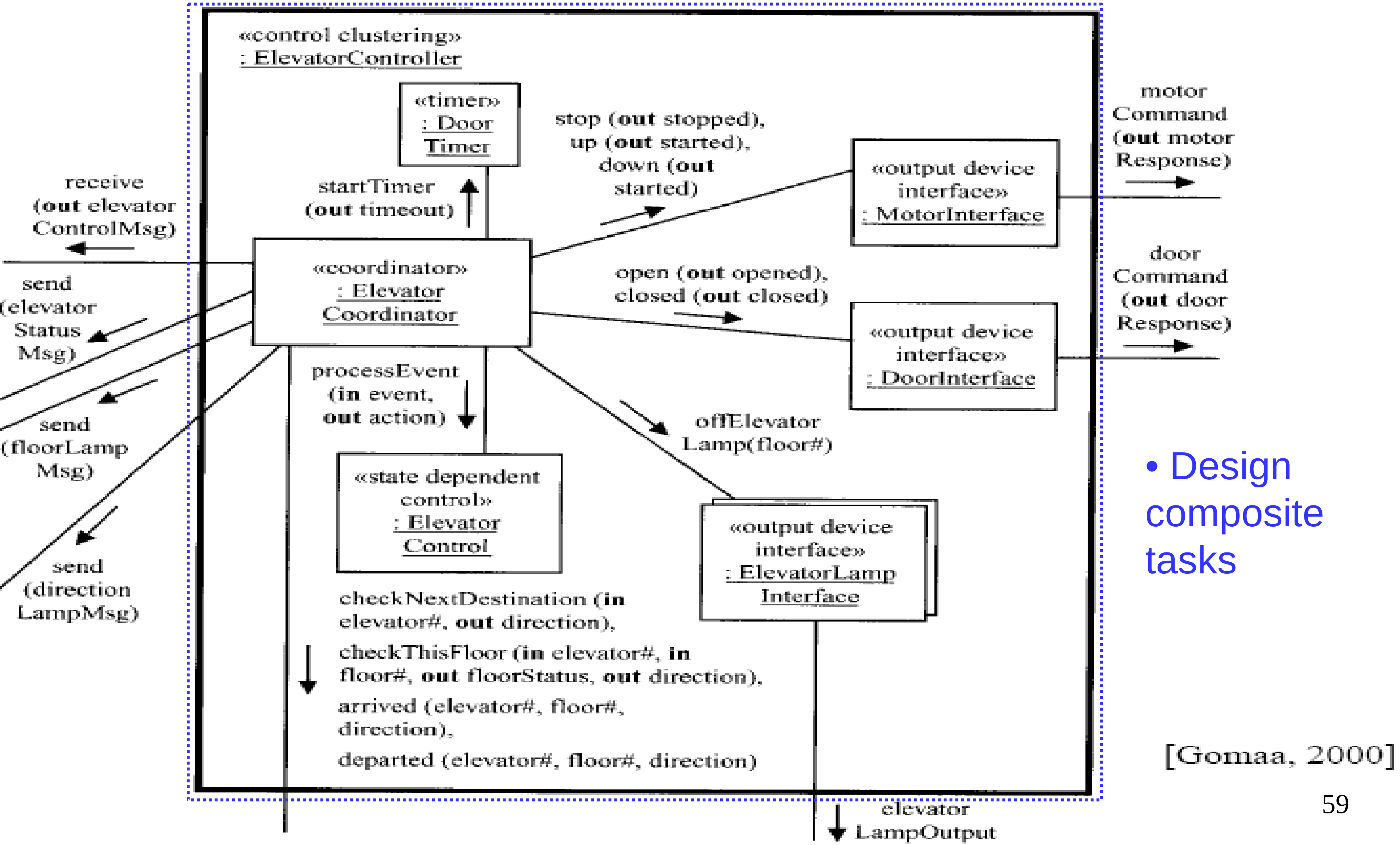
(3.7) Detailed Software Design

- Design connector


«asynchronous input device interface» : ArrivalSensors Interface
send (approaching FloorMsg)
«connector» elevator Controller MessageBuffer
receive (out elevator ControlMsg)
«control clustering» : Elevator Controller
send (direction LampMsg)
«connector» directionLamp MessageQ
send (floor LampMsg)
«connector» floorLamp MessageQ
send (elevator StatusMsg)
«connector» scheduler MessageQ
send (next DirectionMsg)
«coordinator» : Elevator Manager

[Gomaa, 2000]

(3.7) Detailed Software Design



«control clustering»
: ElevatorController
«timer»
: Door Timer
stop (out stopped), up (out started), down (out started)
startTimer (out timeout)
receive (out elevator ControlMsg)
«output device interface»
: MotorInterface
motor Command (out motor Response)
«coordinator»
: Elevator Coordinator
send (elevator Status Msg)
open (out opened), closed (out closed)
«output device interface»
: DoorInterface
door Command (out door Response)
processEvent (in event, out action)
offElevator Lamp(floor#)
send (floorLamp Msg)
«state dependent control»
: Elevator Control
«output device interface»
: ElevatorLamp Interface
send (direction LampMsg)
checkNextDestination (in elevator#, out direction),
checkThisFloor (in elevator#, in floor#, out floorStatus, out direction),
arrived (elevator#, floor#, direction),
departed (elevator#, floor#, direction)
elevator LampOutput
[Gomaa, 2000]

- Design composite tasks

Appendix III. Task Scheduling

1. Periodic Task vs. Aperiodic Task
2. Definitions for Realtime Properties
3. Task Scheduling

1. Periodic Task vs. Aperiodic Task

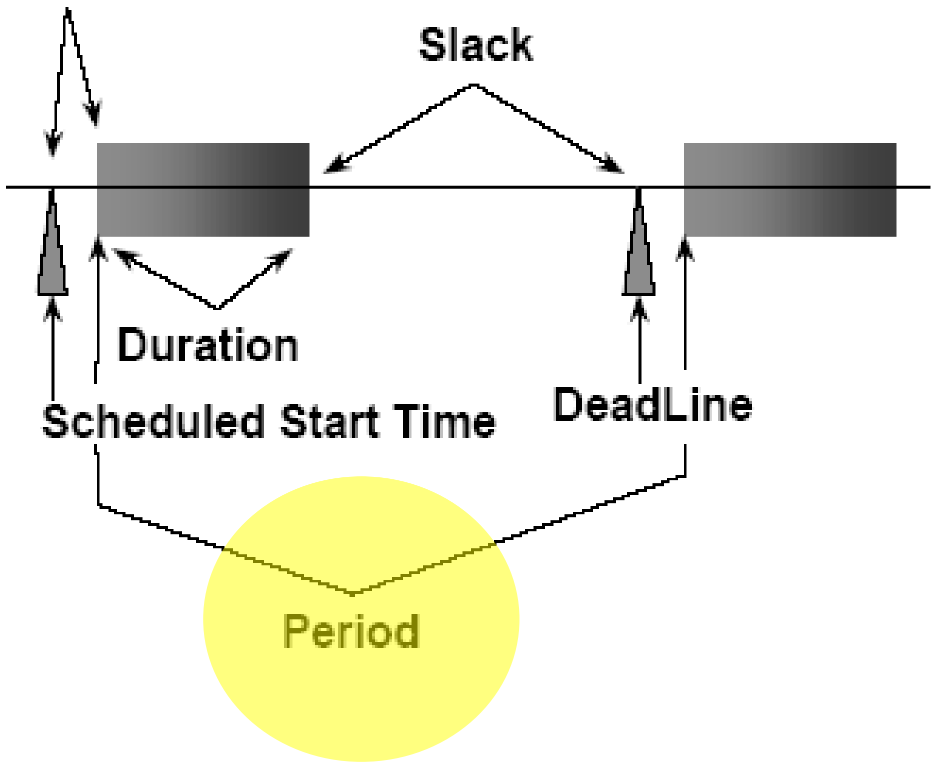
- Periodic tasks
  - activated by a timer, or timer built into the task
  - the task's period is the time between successive activations
  - periodic interrupts

- Aperiodic tasks
  - waits to be activated by an aperiodic event, message, or request for service

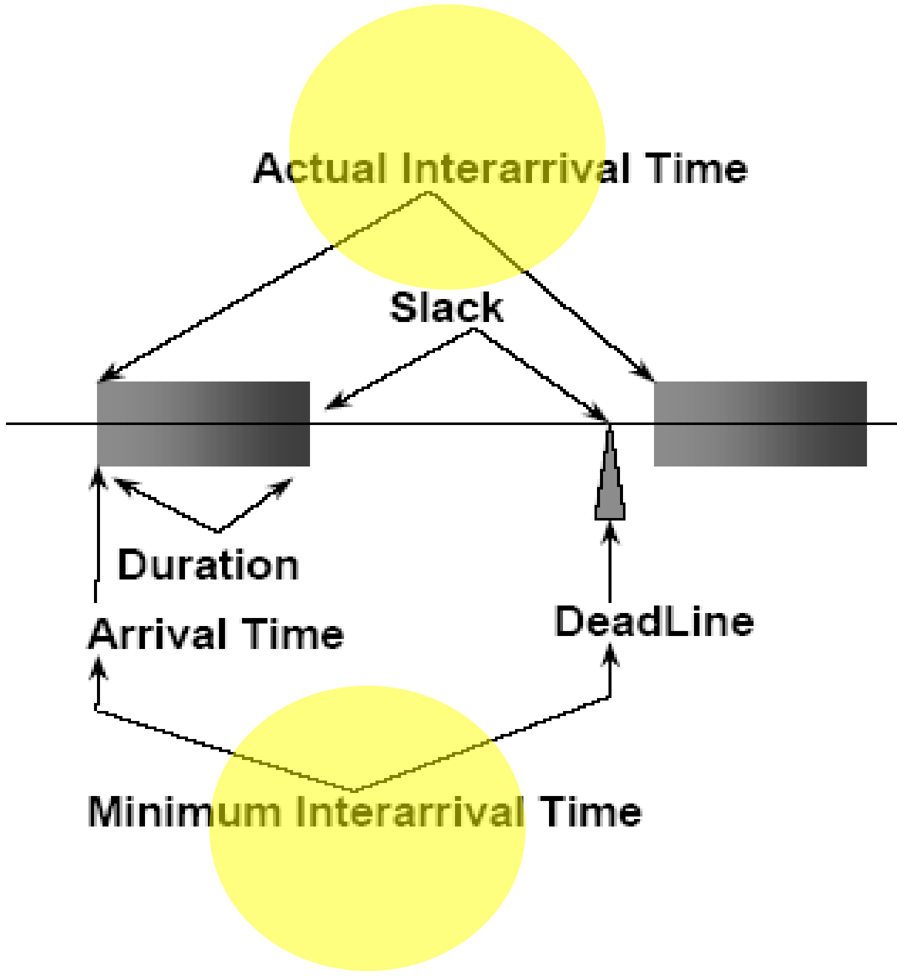
2. Definitions for Realtime properties (1/2)

- Task Duration
  - Time it takes for a task to execute, not including context switch time
- Deadline
  - Time that the task must complete by
- For periodic tasks
  - Period: Time between task starts
  - Jitter: Small variation in schedule task start time and actual task start time
  - Schedule Start Time: Time that the task should start
- For aperiodic task
  - Interarrival Time: Time between event arrivals/task starts
  - Arrival Time: The time that an event arrives and the servicing task begins

2. Definitions for Realtime properties (2/2)



(a) Periodic Tasks



(b) Aperiodic Tasks

3. Task Scheduling (1/6)

- Goals of scheduling algorithms
  - Determine if we can schedule our tasks
    - Make sure that each task executes on time
    - Make sure each task finishes on time
    - Avoid task overflow
- Priorities
  - Static Priorities
    - Priority fixed before runtime
  - Dynamic Priorities
    - Priorities change at runtime

      (=> Most difficult scheduling problem)
- A task is schedulable if all its deadlines are met, i.e. if the task completes its execution before its period elapses.

3. Task Scheduling (2/6)

- How would you order the execution of your tasks so they can all meet their deadlines?

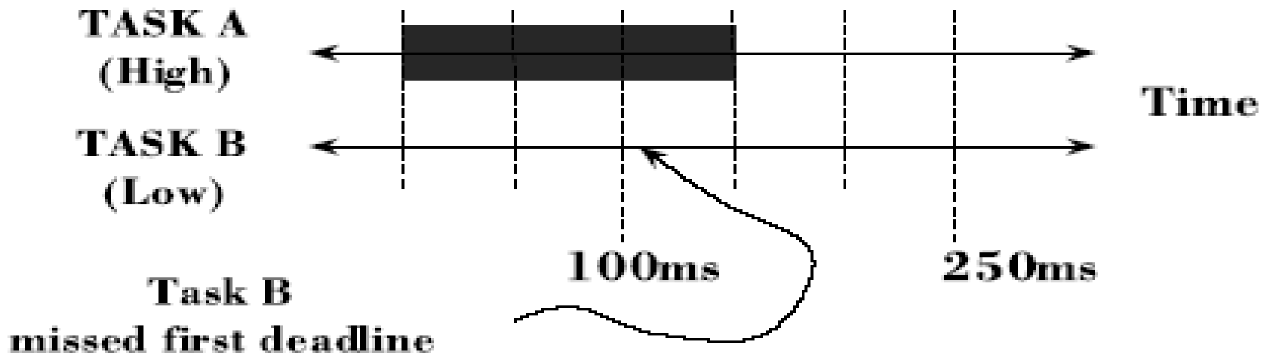
Task A	Task B
Most Important	Least Important
Period: 250 ms	Period: 150 ms
Duration: 150 ms	Duration: 50 ms

- We could schedule the most important task with the highest priority

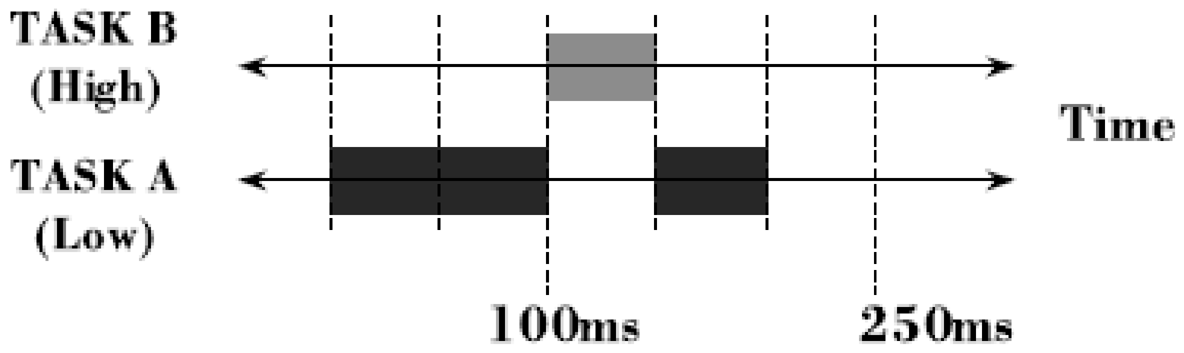
- An alternative is to schedule the task with the highest frequency (shortest period) first

3. Task Scheduling (3/6)

- Highest priority first



- Highest frequency (shortest period) first



3. Task Scheduling (4/6)

- **Rate Monotonic Algorithm (RMA)**
  - Scheduling that assign higher priorities to tasks with shorter periods
  - **Pre-condition for using RMA**
    - **N** independent periodic tasks
  - **Utilization Bound Theorem:** always meet deadlines if :
    - $C1/T1 + ... + Cn/Tn <= n \times (2^{1/n} - 1) = U(n)$

      Ci: Execution time of Task i

      Ti: Period of Task i

      U(n): Utilization bound (1 if n=1, 0.69 if n = ω)
  - **Completion Time Theorem:** meet the deadlines regardless of the start time if :
    - when all are started at the same time,

      each task meets its first deadline.

3. Task Scheduling (5/6)

Example 1. (Utilization Bound Theorem)

Utilization of Ti is Ui = Ci/Ti

T1 = 100, C1 = 20, U1 = 0.2

T2 = 150, C2 = 30, U2 = 0.2

T3 = 200, C3 = 60, U3 = 0.3

$U(3) = 3 \times (2^{1/3} - 1) = 0.779$

U = (6x20 + 4x30 + 3x60)/600 = 420/600 = 0.7

=> Since 0.7 < 0.779, Ti's will always meet the deadlines.

Example 2. (Utilization Bound Theorem)

T1 = 100, C1 = 20, U1 = 0.2

T2 = 150, C2 = 30, U2 = 0.2

T3 = 200, C3 = 90, U3 = 0.45

U = (6x20 + 4x30 + 3x90)/600 = 510/600 = 0.85

Will T1 and T2 always meet deadlines?

U = (20x3 + 30x2)/300=120/300 = 0.4 < 0.69   So T1 and T2 are okay.

3. Task Scheduling (6/6)

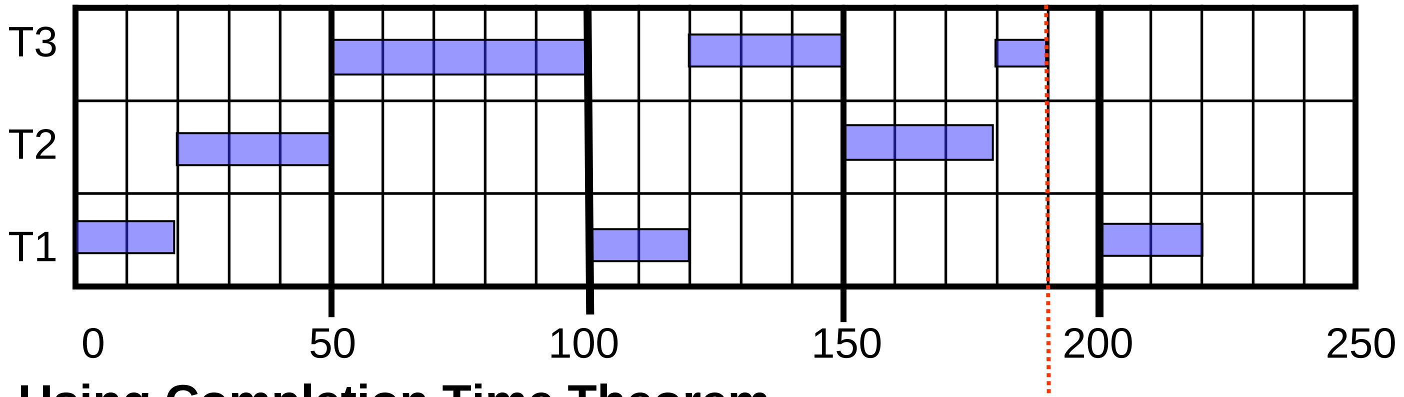
Example Completion Time Theorem

T1 = 100, C1 = 20, U1 = 0.2

T2 = 150, C2 = 30, U2 = 0.2

T3 = 200, C3 = 90, U3 = 0.45

Least Common Period = 600 → U = (6x20+4x30+3x90)/600 = 0.85



- **Using Completion Time Theorem**
  - Since, when all Ti’s start at 0, within 190 Ti's all meet their first deadlines, Ti's will meet deadlines in any case.
  - Initially, utilization is very high with 190/200 = 0.95.
    But at 600, the utilization will average out to 0.85.

Appendix IV. Realtime Design (Elevator Case Study)

Realtime Design

- Realtime Design Approaches

1. Ad hoc
    - Implementation => Testing => If the result Not OK, find reason => Modify design => Implement again
    - Analysis performed after design

2. Design phase analysis
    - Perform task scheduling at design time

      (1) Worst-case scenario development

      (2) Event Sequence Analysis

      (3) Calculate Real-time Scheduling Task parameters

      (4) Assign priorities

Realtime Design

Example

- 10 story building
- 3 elevators
- System exist on a single node (= computer)

Which one is the most demanding timeliness requirement and which one is the least demanding ?

(1) 3 Worst case scenarios

FA. (Stop Elevator at Floor) 3 elevators arrive at the same floor simultaneously. 3 `Floor-arrival interrupts` arrive within 50 ms.

=> `Elevator Controlller` needs to decide whether to stop before passing (☛ 50 ms/3 = 17 ms)

EB. (Select Destination) `Elevator button interrupts` arrive simultaneously within 3 seconds from 10 x 3 buttons (☛ 3000 ms/30 = 100 ms)

FB. (Request Elevator) 2 (up and down) x 8 floors + 1 (up or down) x 2 floors = 18 `Floor button interrupts` arrive simultaneously within 3.6 seconds

(☛ 3600 ms/18 = 200 ms)

Realtime Design

(2) Event Sequence Analysis

FA. (Stop Elevator at Floor) 3 elevators arrive simultaneously at the same floor. 3 floor-arrival interrupts arrive within 50 ms

A1. `ArrivalSensorsInterface` receives and process interrupts

A2. `ArrivalSensorsInterface` sends to `ElevatorController` `approaching Floor` message

A3. `ElevatorController` receives message and enquires `ElevatorStatus&Plan` whether to stop

A4. If should stop, `ElevatorController` calls `StopMotor` action

EB. (Select Destination) `Elevator button interrupts` arrive simultaneously within 3 seconds from 10 x 3 buttons

E1. `ElevatorButtonsInterface` receives and process interrupts

E2. `ElevatorButtonsInterface` sends Elevator Request message to `ElevatorManager`

E3. `ElevatorManager` receives message and registers destination with `Elevator Status&Plan` object

Realtime Design

(2) Event Sequence Analysis (Cont.)

FB. (Request Elevator) 2 (up and down) x 8 floors + 1 (up or down) x 2 floors = 18 Floor button interrupts arrive simultaneously within 3.6 seconds

F1. `FloorButtonsInterface` receives and processes interrupt

F2. `ElevatorButtonsInterface` sends `service Request` message to `Scheduler`

F3. `Scheduler` receives message and enquires `ElevatorStatus&Plan` object whether it is possible to go to the floor. If not, `Scheduler` selects elevator

F4. `Scheduler` sends `scheduler Request` message informing selected elevator to `ElevatorManager`

F5. `Elevator Manager` receives message and registers it with `Elevator Status&Plan` object

Realtime Design

(3) Calculate Real-time Scheduling Task parameters
(4) Assign priorities

TET: Total Elapsed Time
TU: Total Utilization

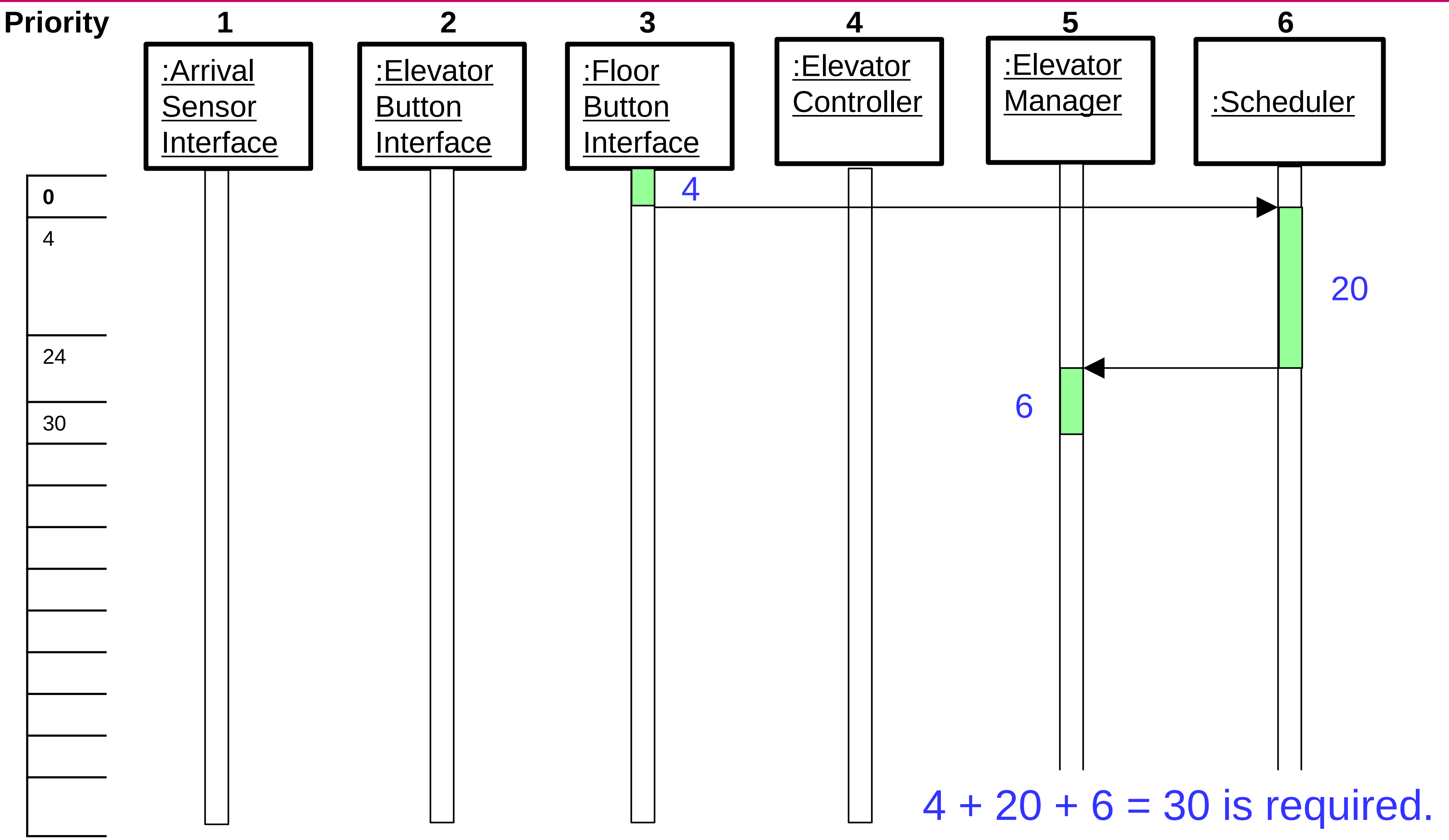
Task	CPU time Ci	Period Ti	Utilization Ui	Assigned Priority
(Stop Elevator at Floor) ArrivalSensors Interface ElevatorController TET = 34 ms	 2 5	 50 50	 0.04 0.10 TU = 0.68	1 4
(Select Destination) ElevatorButtonsInterface ElevatorManager (Case b) TET = 47 ms	 3 6	 100 100	 0.03 0.06 TU = 0.47	2 5
(Request Elevator) **FloorButtonsInterface** **Scheduler** **ElevatorManager(Case c)** **TET = 76 ms**	 **4** **20** **6**	 **200** **200** **200**	 **0.02** **0.10** **0.03** **TU = 0.38**	**3** **6**
Other Tasks Floor Lamps Monitor Direction Lamps Monitor	 5 5	 500 500	 0.01 0.01	7 8

Violate RMA because they require very time critical interrupt handling !

☛ Needs careful analysis !

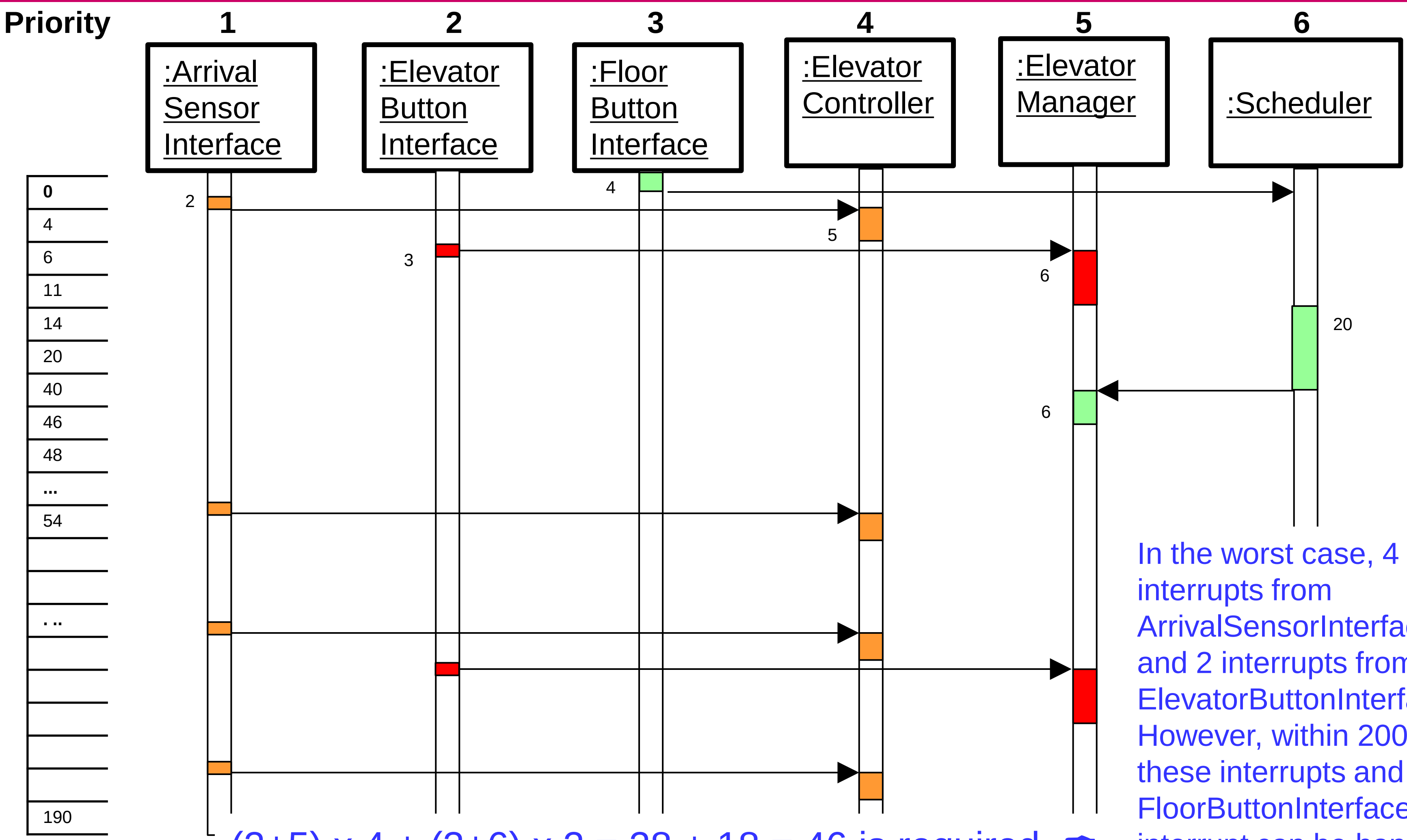
Total utilization 0.4 is less than the worst case utilization 0.69 but

Analysis of Request Elevator event sequence (Best Case)



4 + 20 + 6 = 30 is required.

Analysis of Request Elevator event sequence (Worst Case)



(2+5) x 4 + (3+6) x 2 = 28 + 18 = 46 is required.

In the worst case, 4 interrupts from ArrivalSensorInterface and 2 interrupts from ElevatorButtonInterface. However, within 200 all these interrupts and FloorButtonInterface interrupt can be handled.

Questions?