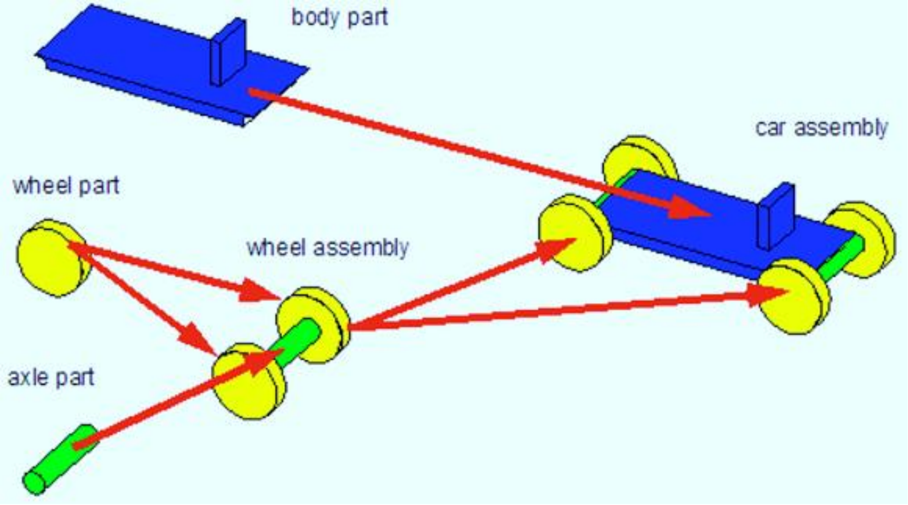
Composite Architecture, PDOM, Class Model

Lecture 10

5 June 2025

Slides created by
Prof Amir Tomer
tomera@cs.technion.ac.il



Picture Source: http://www.techsoft3d.com/developers/technical-documentation/siemens-parasolid/

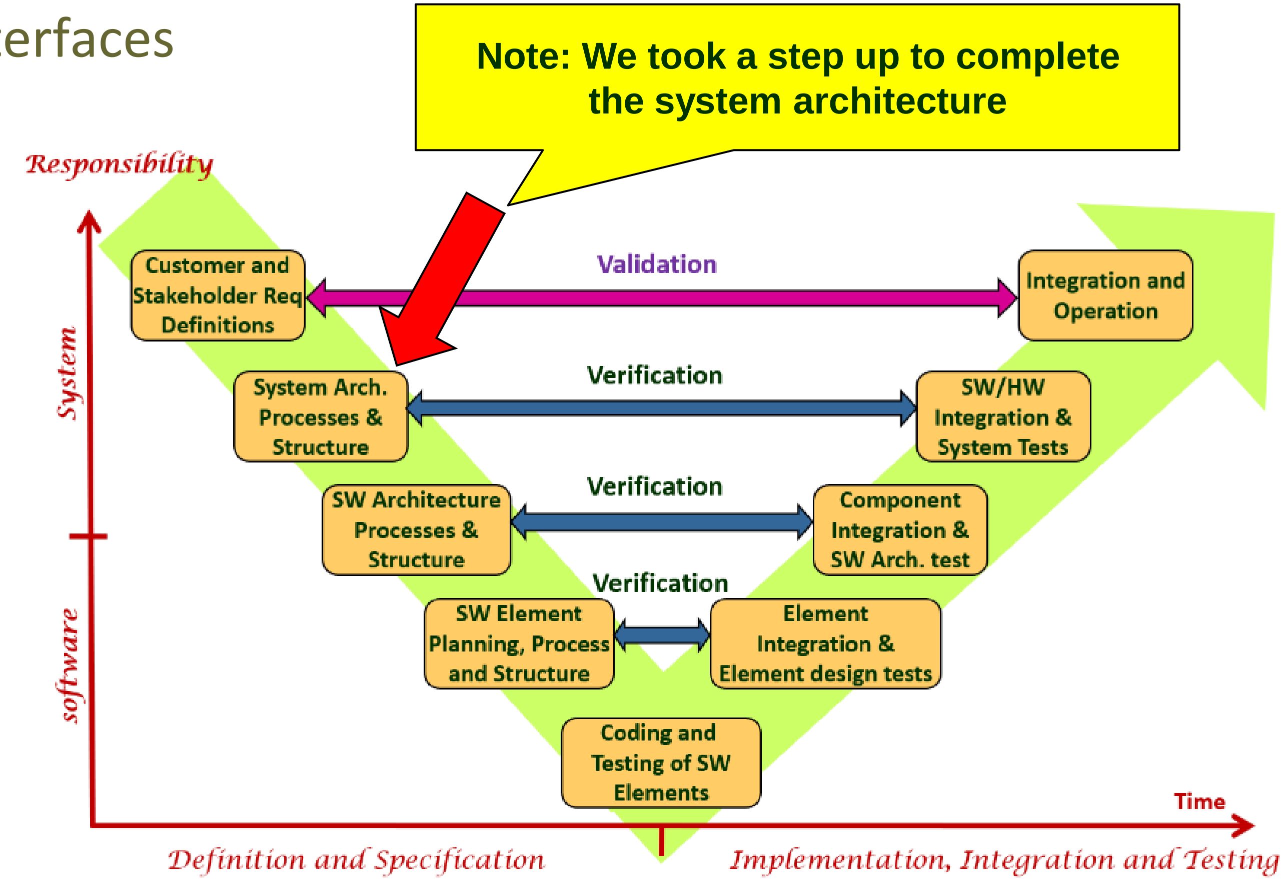
Topics for Today

- Composite Architecture
- PDOM
- Class Model

Building Composite Architecture

[System architecture: Structure]

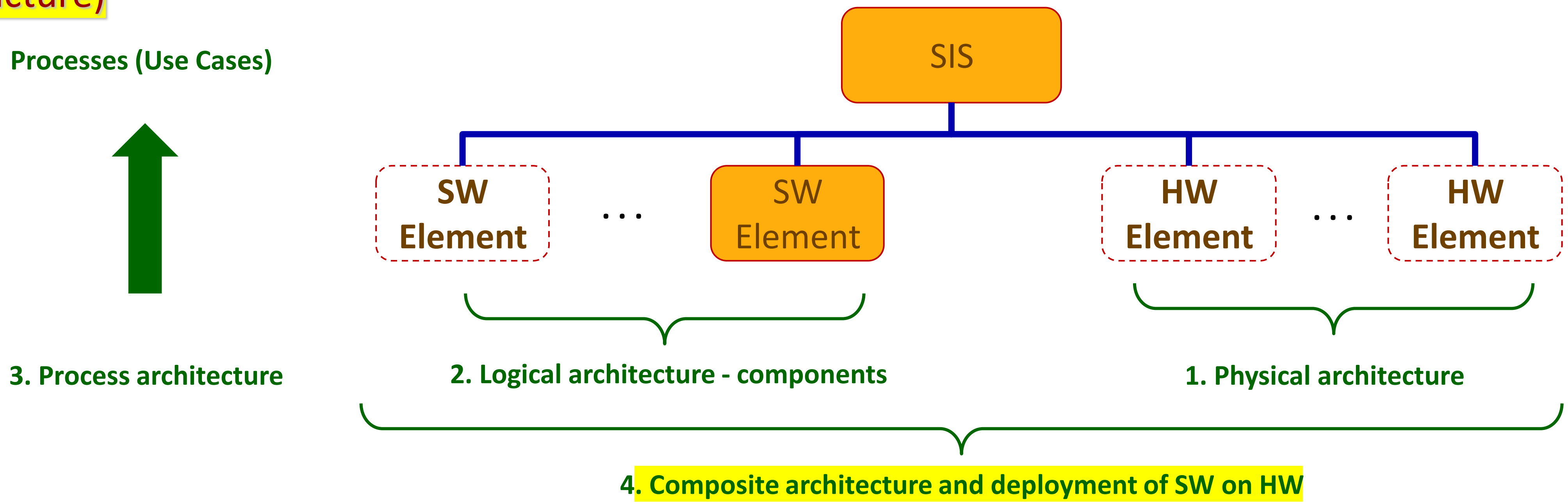
- Our goals:
  - Integrate the software and hardware architectures
  - Connect physical interfaces to logical interfaces
- Inputs:
  - Component Diagram
  - Deployment diagram
- Outputs:
  - Composite diagram
  - Build and deployment plans for software on the hardware



System Architecture: SIS architecture includes

Reminder

1. Physical architecture: Hardware components and physical connections – static model (structure)
2. Logical architecture: Software components and logical connections – static model (structure)
3. Process architecture: Implementation of processes via interaction between components - dynamic model (behavior)
4. Composite architecture: Implementation of logical connections via physical connections – static model (structure)

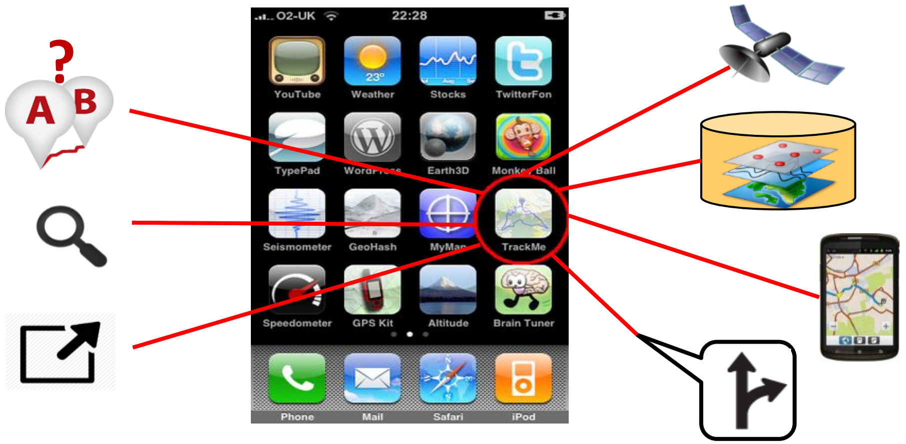


Functional interfaces via physical interfaces

Smartphone physical interfaces

Functional (logical) of navigation app

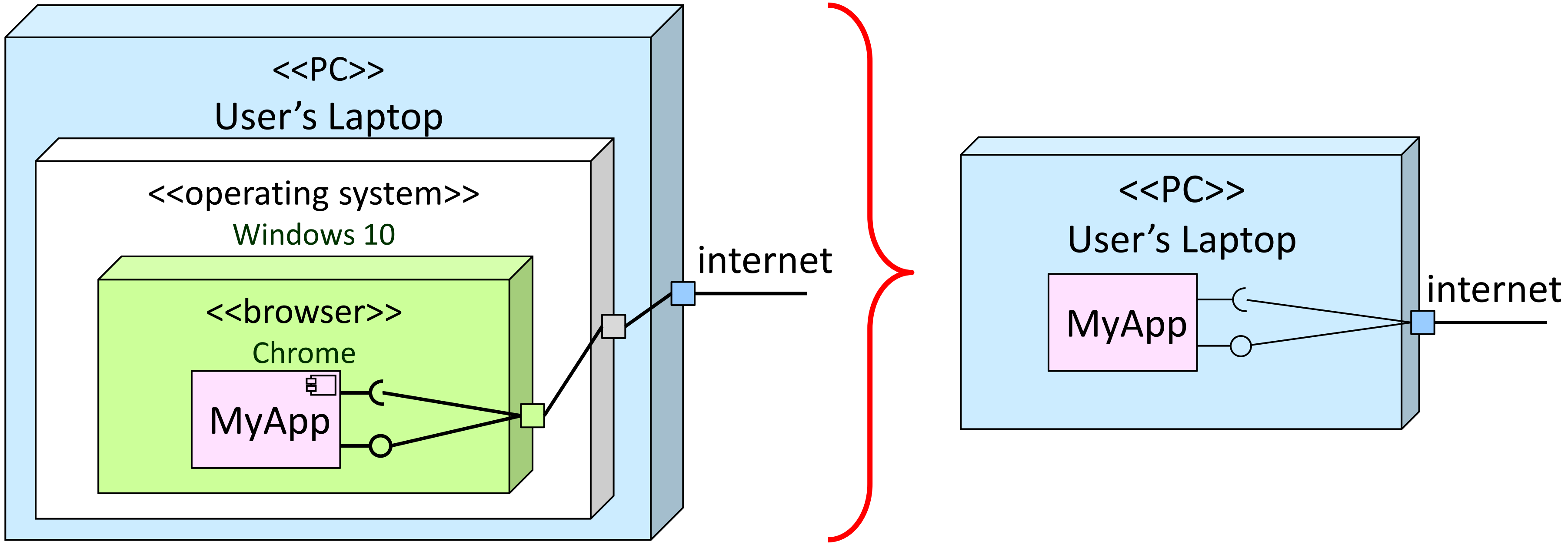
Provided *Interfaces*



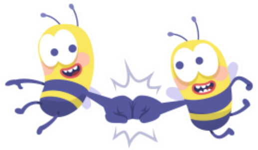
Required Interfaces

Connection between functional and physical interfaces

- Sometimes one or more execution environment layers stand between functional and physical interfaces
  - Passage through each layer requires a specialized mechanism
  - The passage point is called a "port" and is shown with a small square
  - For simplicity, we might compress a few ports into a single general one



Delegation



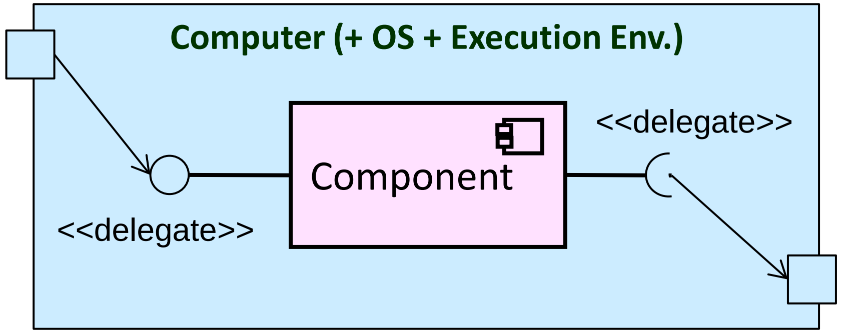
An internal interface can't directly contact the outside environment → it needs a **port** to represent it

Delegation from Provided interface

- The component provides the service to the external environment via the port
- Requests arrive at the port and are forwarded to the appropriate interface

Delegation from a Required interface

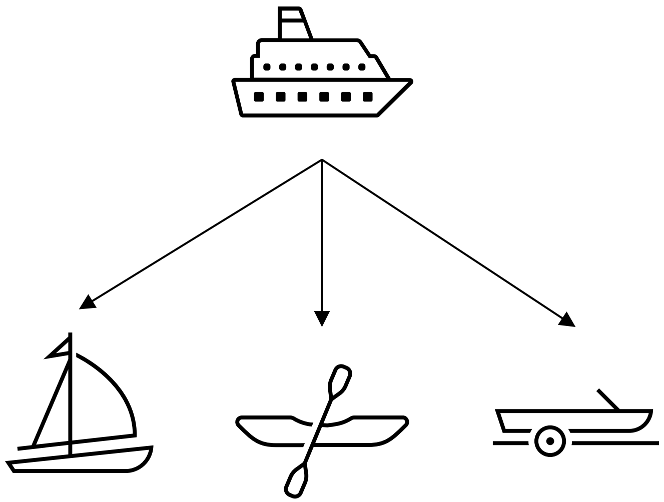
- The component receives the service from the external environment via the port
- Requests for the service leave via the port to the external environment



Difference between Logical and Physical architectures

Physical

- Physical architecture is built for a specific system
- Each configuration has its own physical architecture

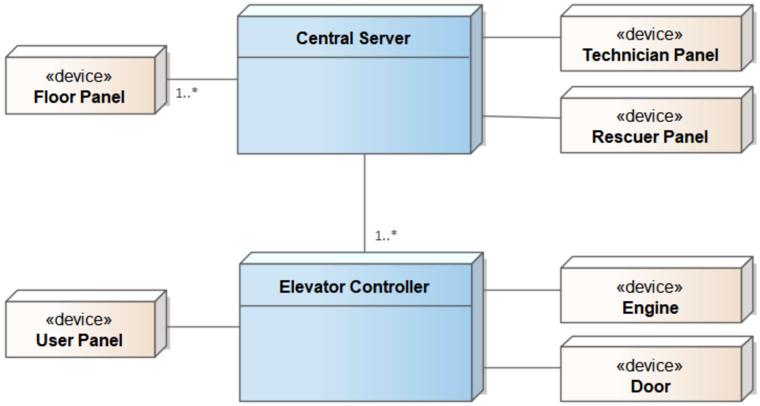


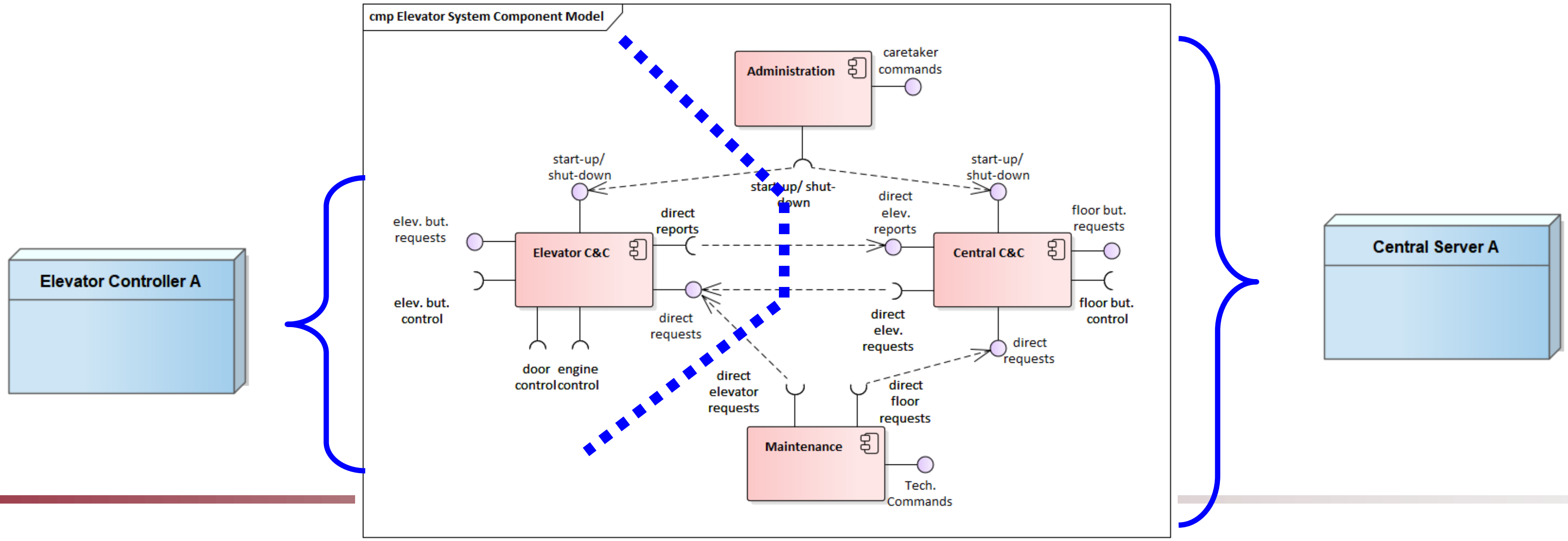
Logical

- Logical architecture is more generic
- Can use the same components for many systems
  - Many "builds"
- Connections between interfaces are implemented based on its physical location
  - Direct connection: components in same software (e.g. included, compiled)
  - Dependency: different software, same computer (e.g. DLL)
  - Via port: different computers, different software (e.g. client-server)

Building composite architecture: Step 1

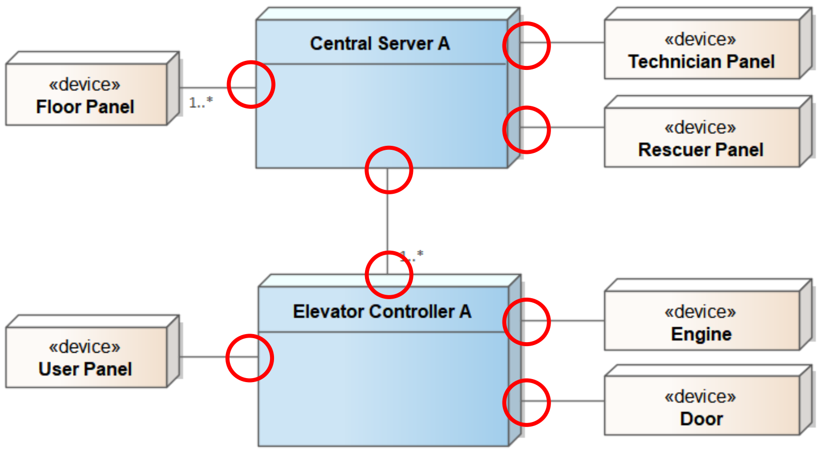
- Choose a physical architecture and decide how to break the components between the computers
- Elevator physical architecture A

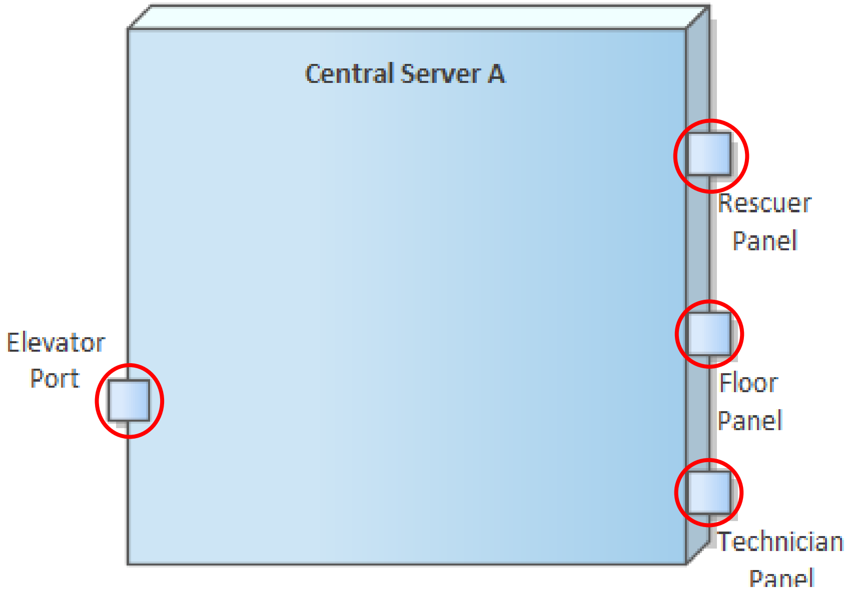


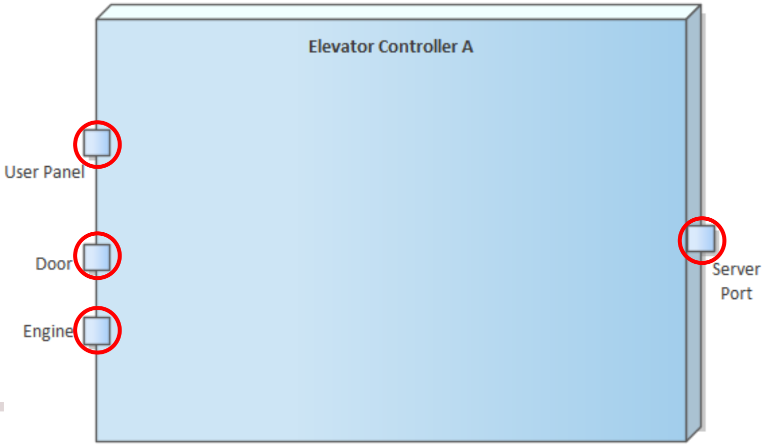


Building composite architecture: Step 2

- Create the composite diagram (can also do one diagram per computer)
  - Put the computers from the architecture in
  - For each physical connection, add a port

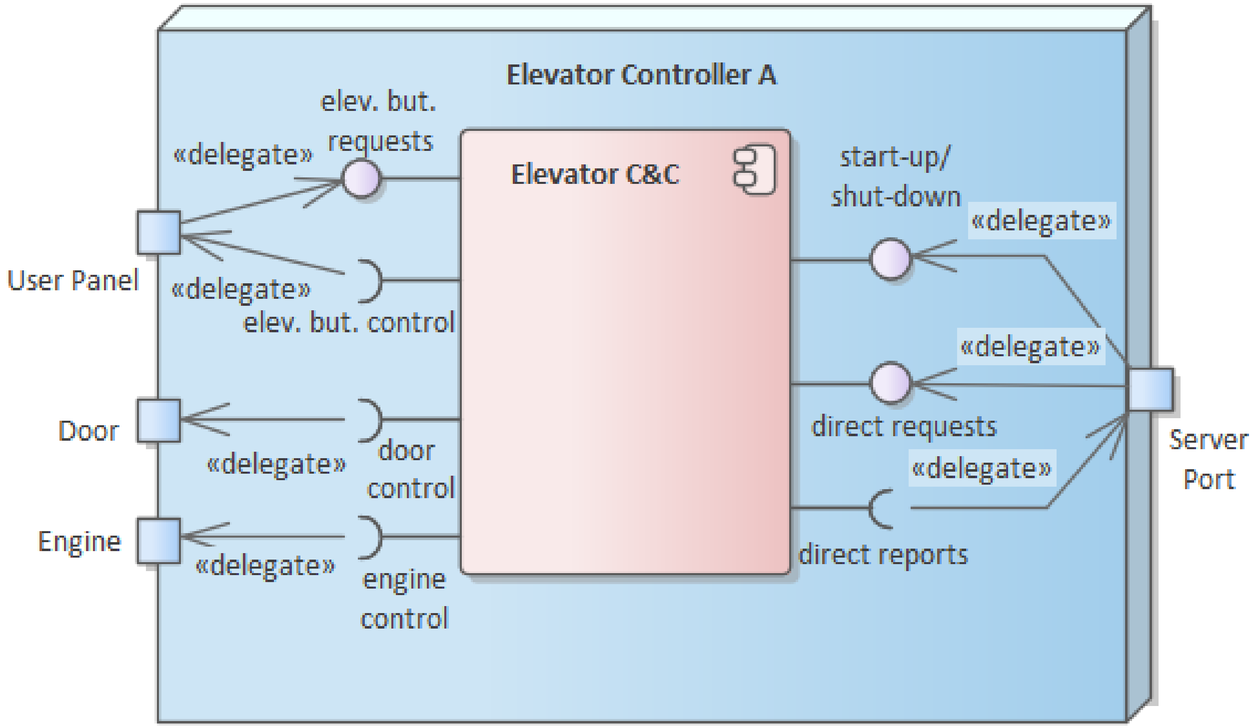




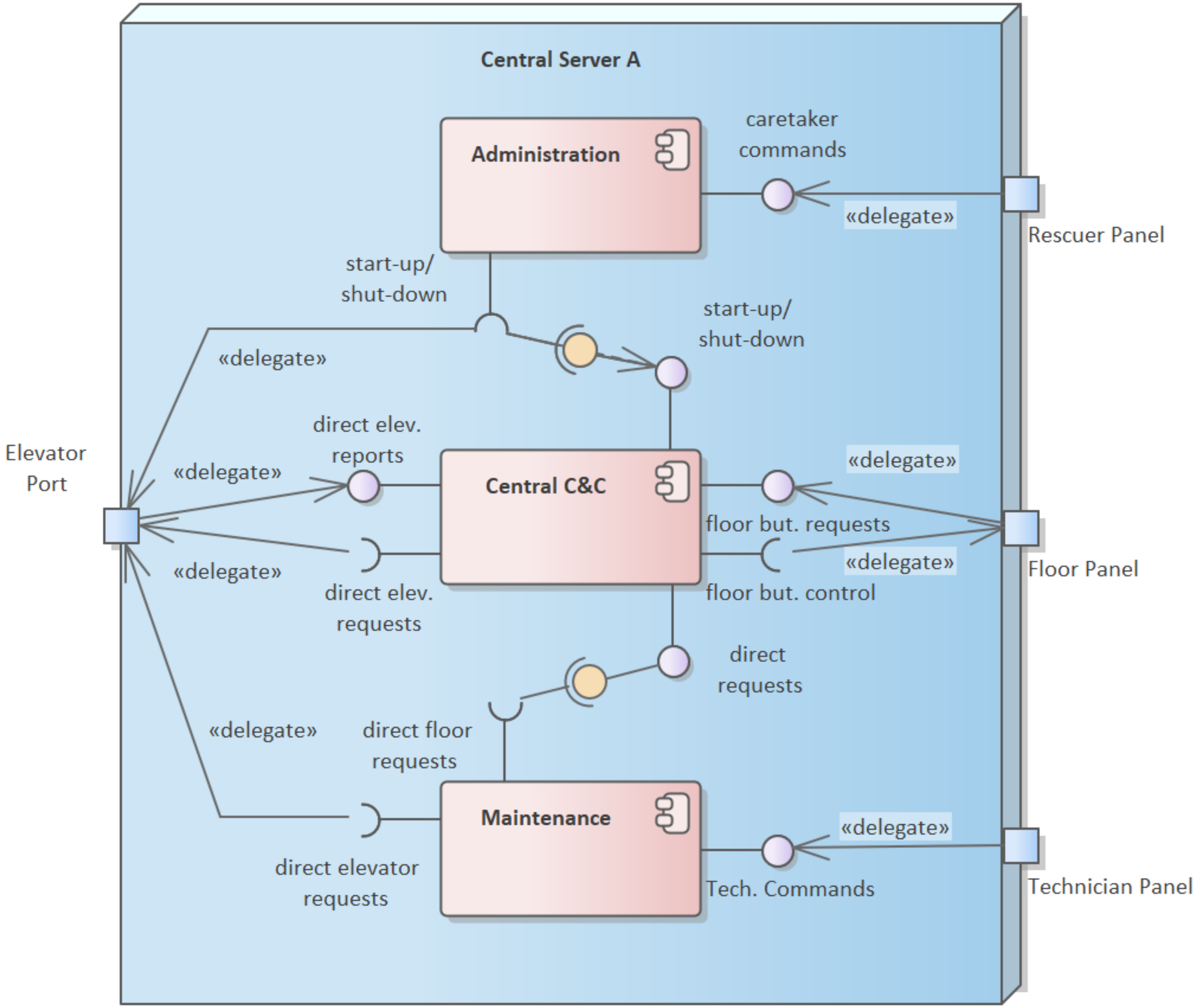


Building composite architecture: Step 3.1

- Place the functional components in the physical computers
- Connect the free interfaces to ports via <<delegate>> connections
- Connect internal dependencies with assembly connector



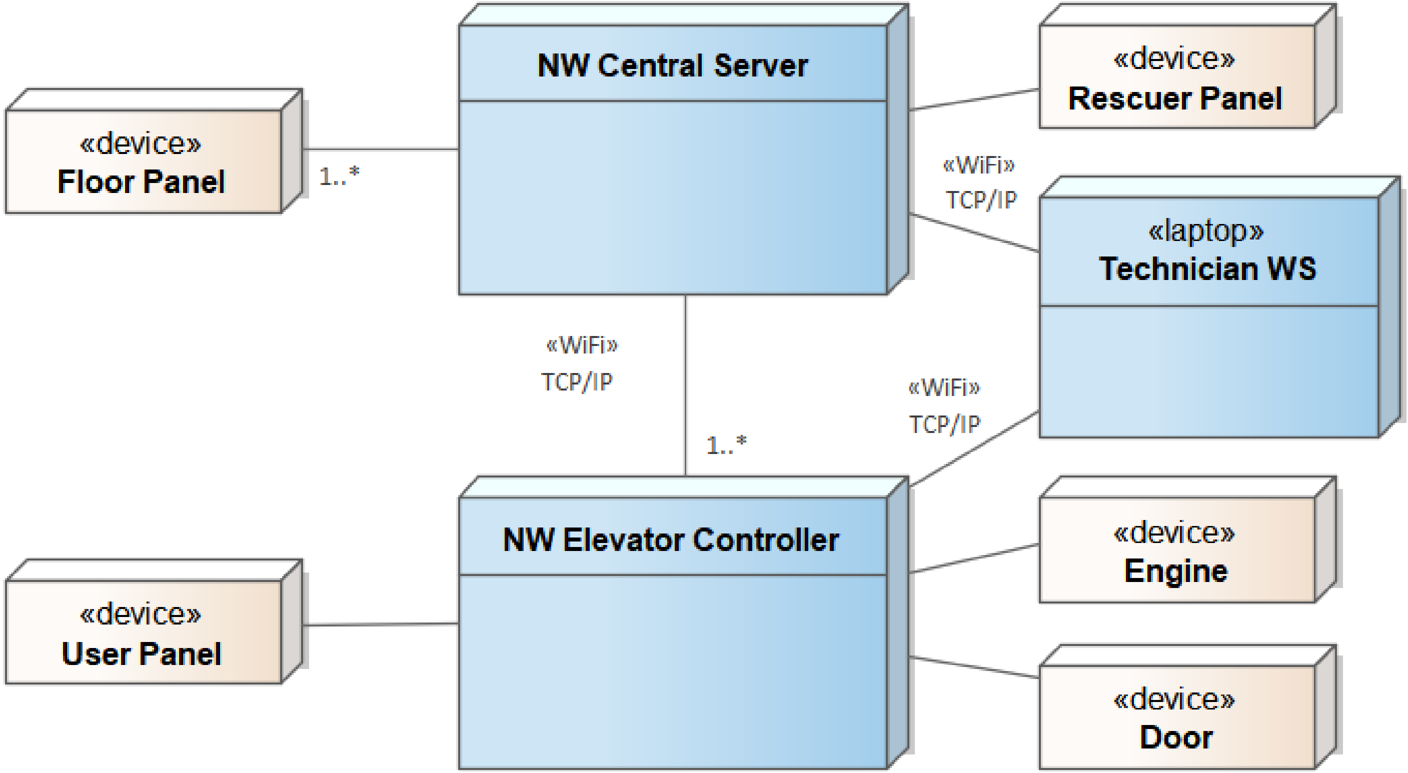
Building composite architecture: Step 3.1



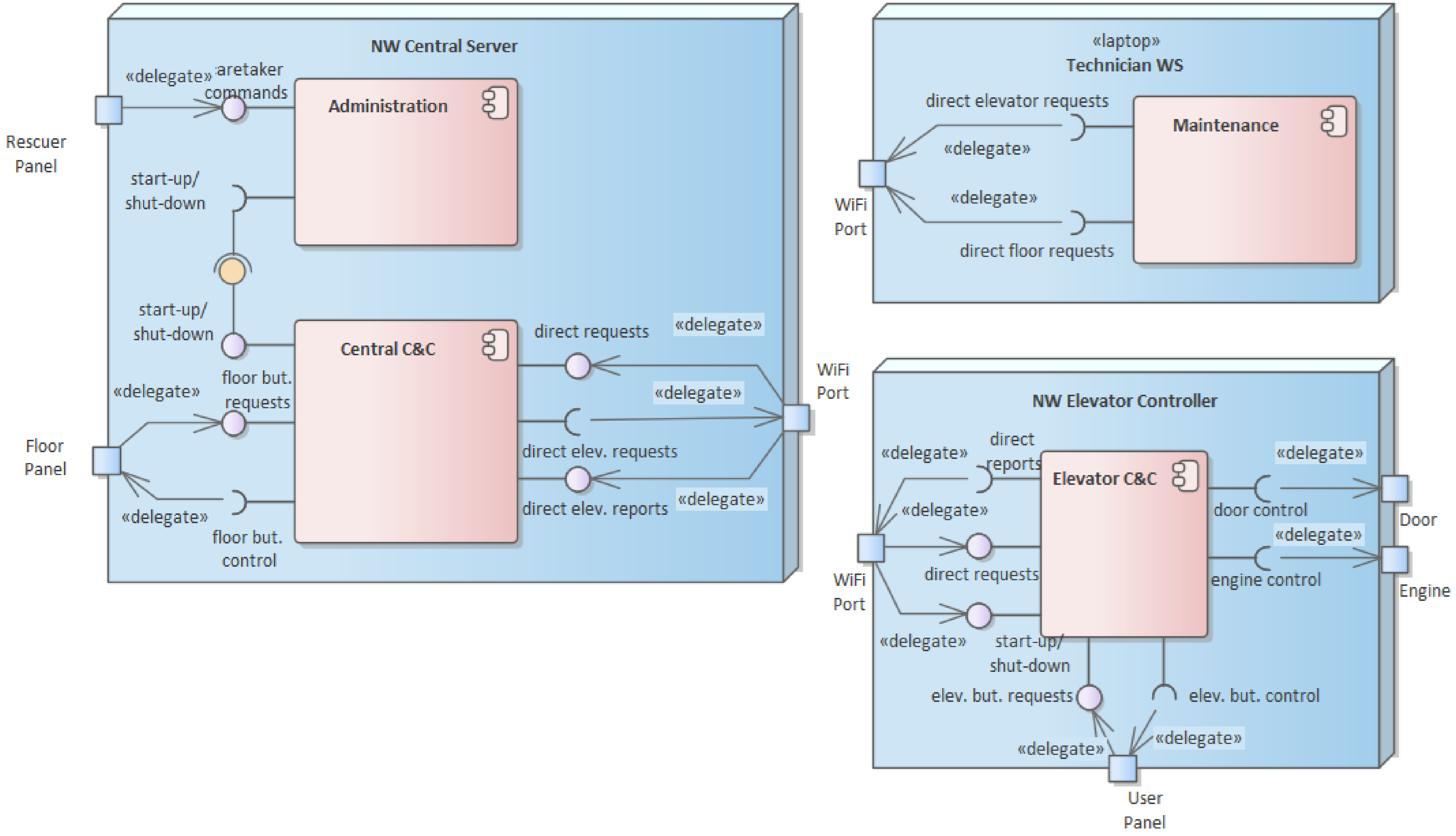
Elevator System Physical Architecture (v2)

Reminder

- Network architecture
  - Central server and elevators connected via a wireless network
  - Technician comes with a laptop and connects to the system via the network



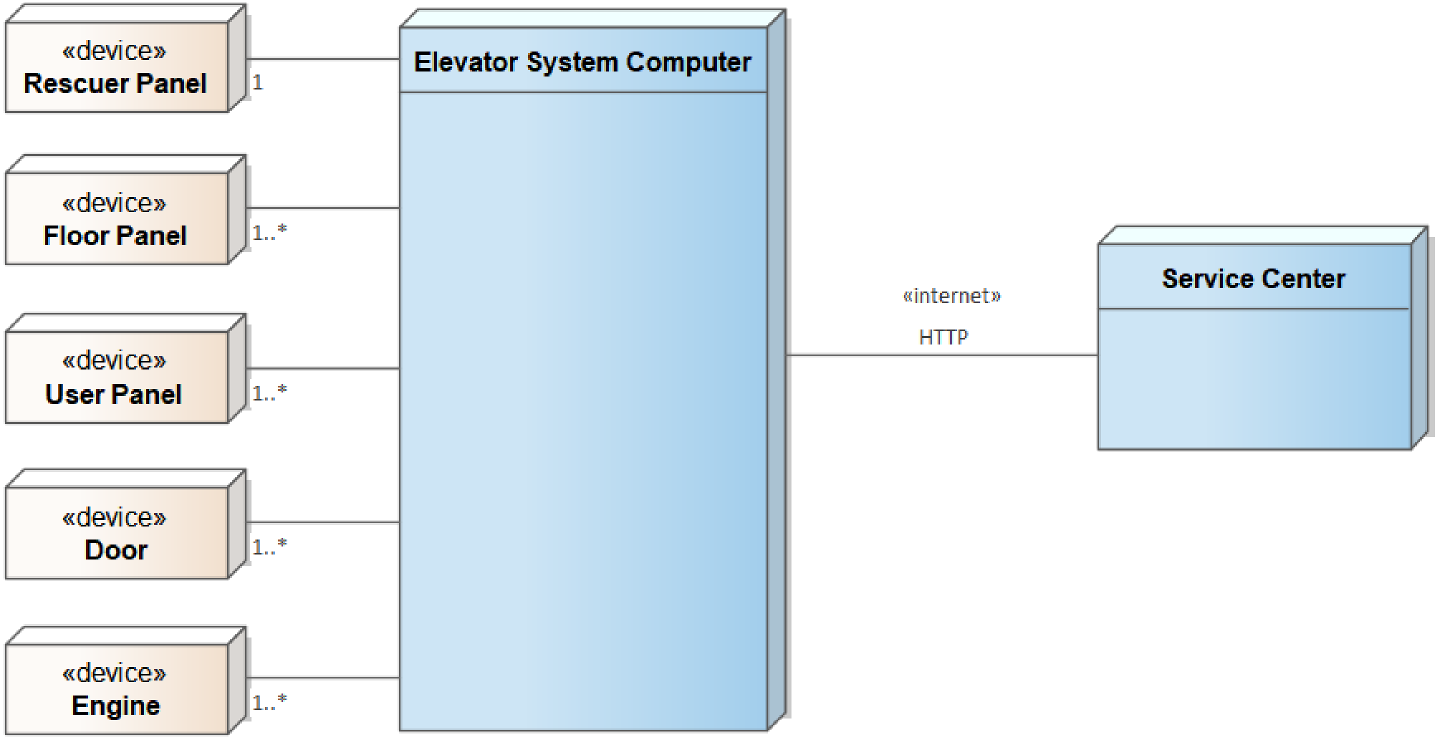
Elevator composite architecture B



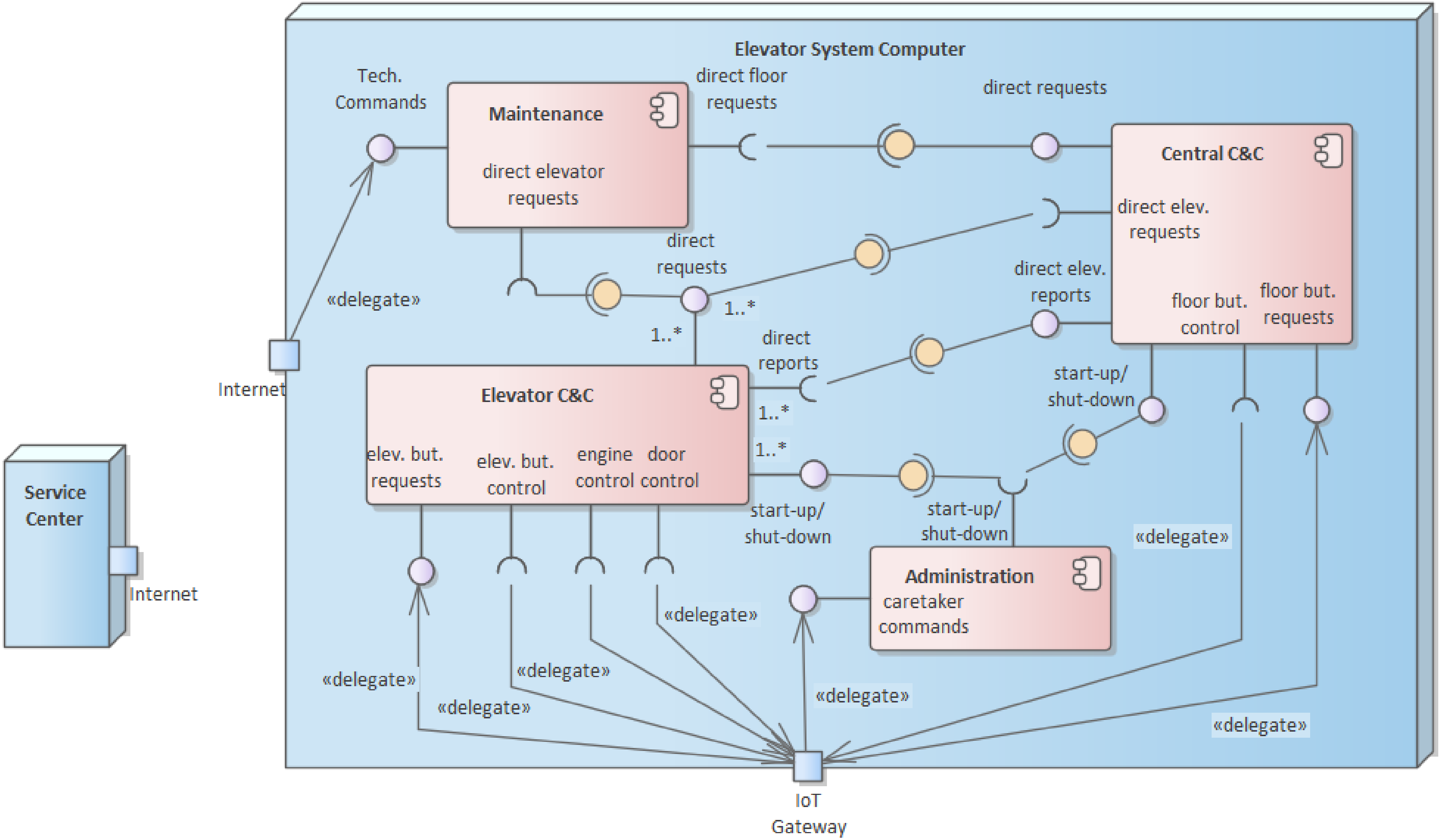
Elevator System Physical Architecture (v3)

Reminder

- Centralized architecture
  - The whole system is controlled and operated by a single computer with IoT connections to all devices
  - External services (operation and control) are offered remotely via the internet



Elevator composite architecture C



Physical architecture in UML: Software components and deployment

Artifact

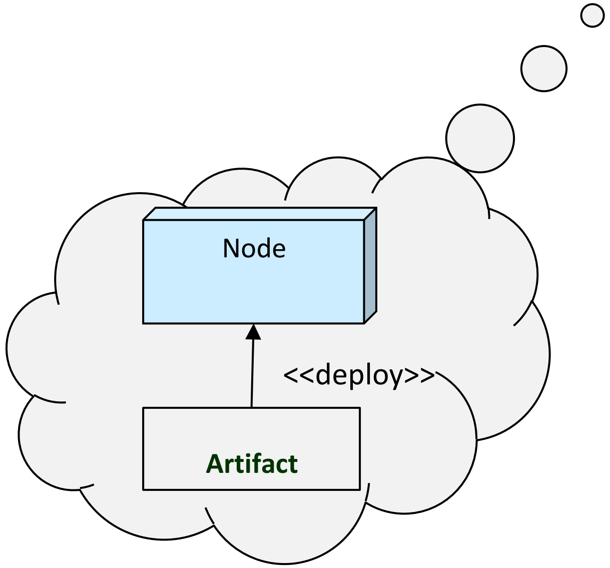
- Two kinds
- Artifacts that exist in the given computational environment (e.g., SendMsg.dll)
- Artifacts created during development (e.g., MyProg.exe)

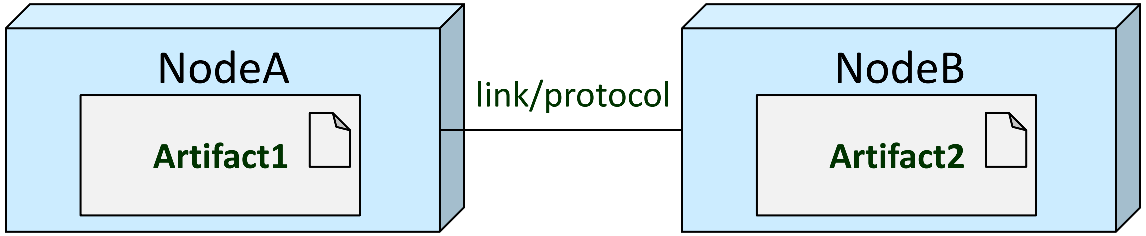
Deployment

- A dependency between a software artifact and the node it's installed on

Deployment Diagram

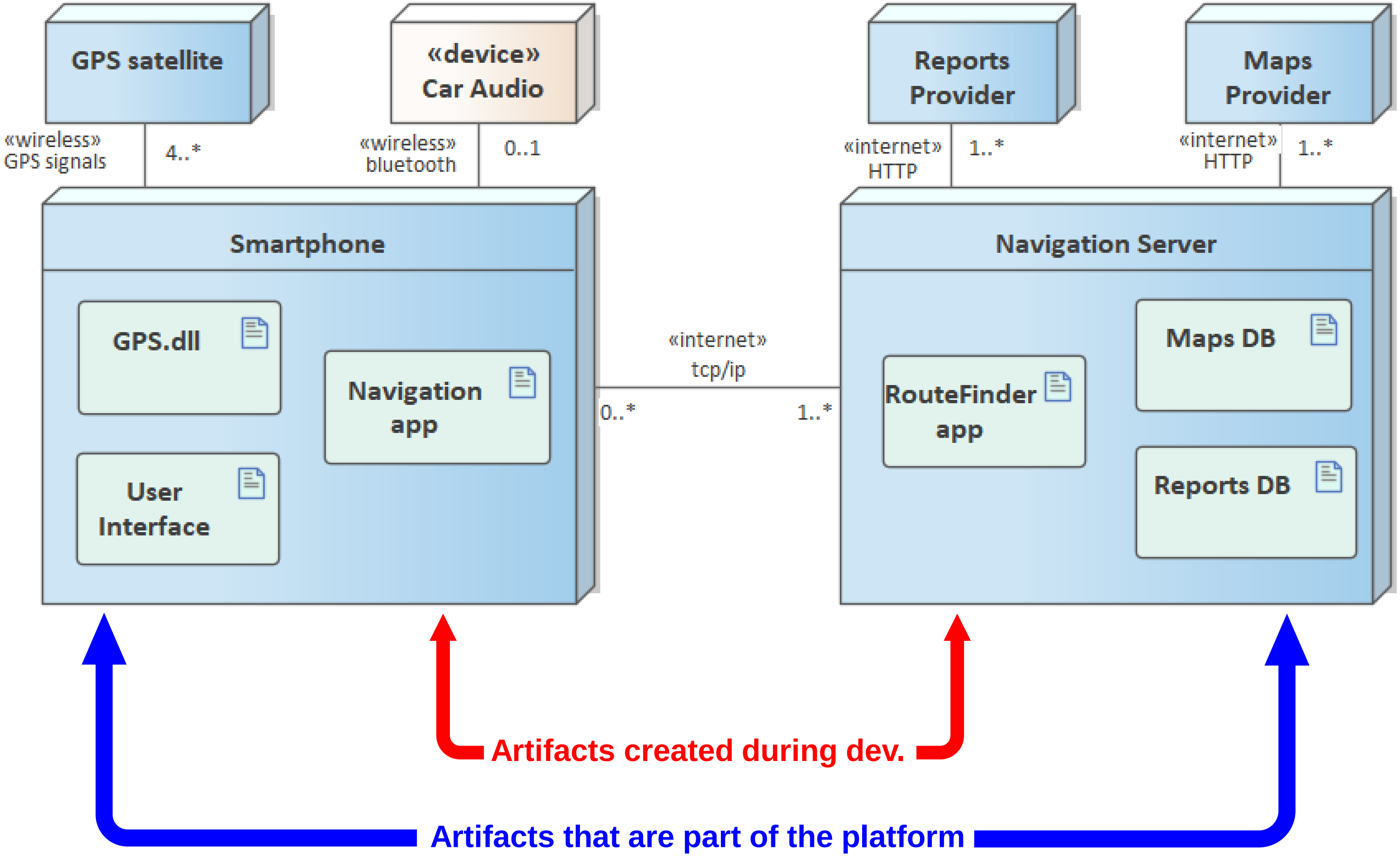
- Diagram that shows the deployment of software on hardware
- In practice, the hardware diagram showing where software is





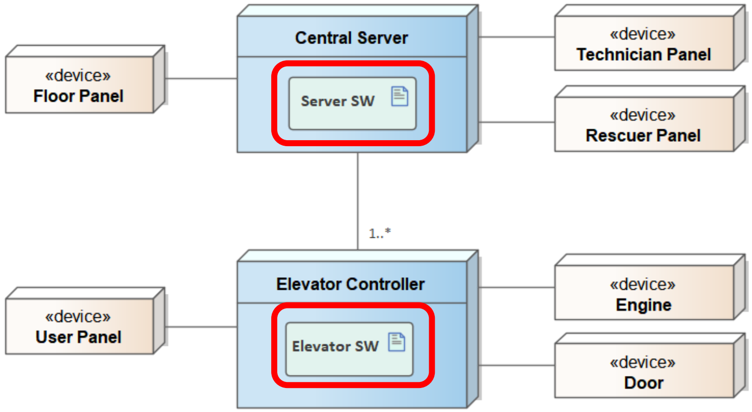
Deployment diagram with installed artifacts

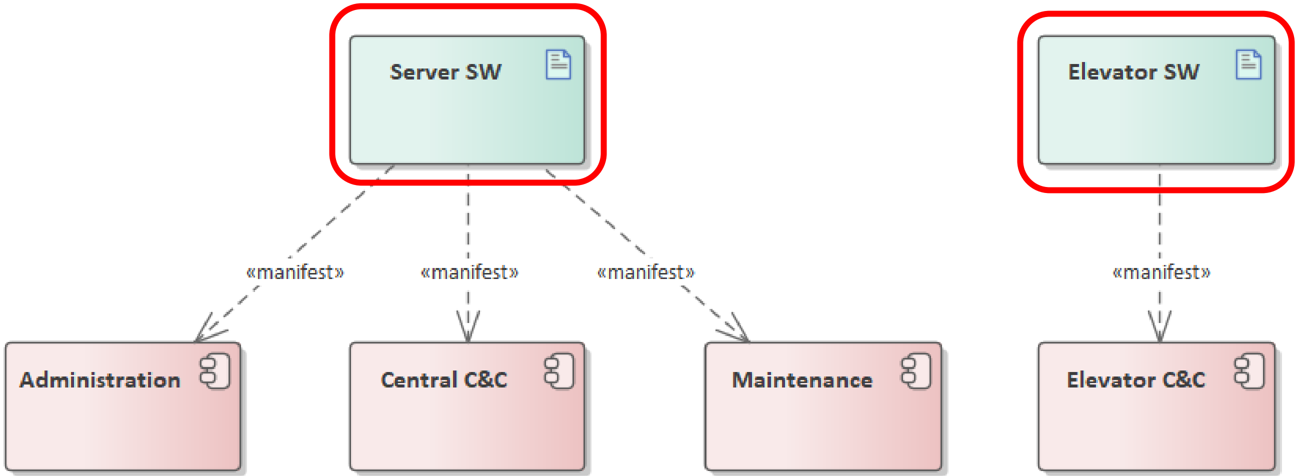
- Example: In-car navigation



Connection between artifacts and components

- Components = Development code modules
- Artifacts = installed files (.exe)
- Components are "built" into artifacts
  - Artifacts offer the components they are built of
  - The offer is reflected by the <<manifest>> dependency
- Artifacts are installed on computers in the physical architecture
  - Example elevator architecture A





In class assignment: Composite Architecture

- Create a new Composite Architecture diagram for ePark
  - Drag components and hardware nodes from your existing physical and logical architecture diagrams
  - Add ports for physical connections
  - Add “User Port” for user interfaces (GUI)
  - Connect external interfaces to ports using Delegate arrows

In class assignment: Building artifacts and installation

- Create a new Deployment Diagram called “Build Allocation”
  - Drag into it the components from the Component View
  - Based on the composite architecture, define new artifacts
  - Create <<manifest>> relationships between components and artifacts
  - Install the artifacts on the on the physical architecture

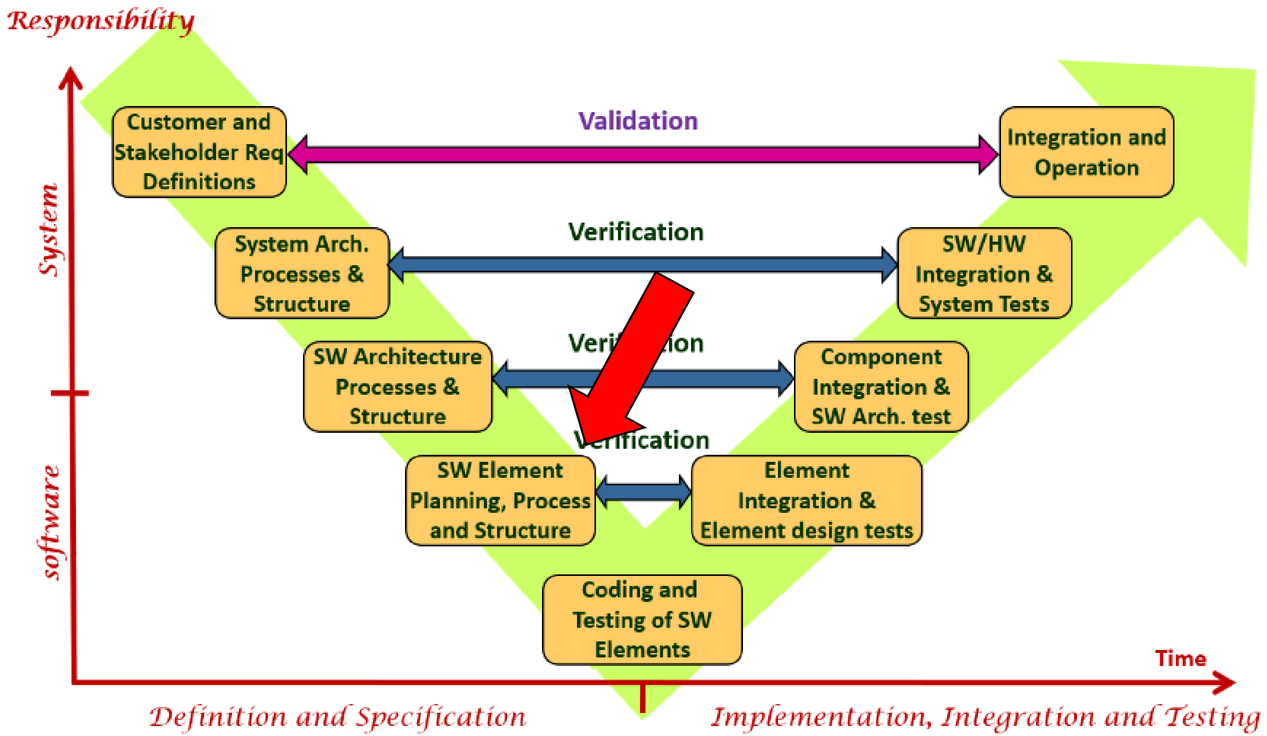
- Perform the steps above on the ePark architecture
  - Work using the steps shown in the slides before

So Far

- Composite Architecture
- PDOM
- Class Model

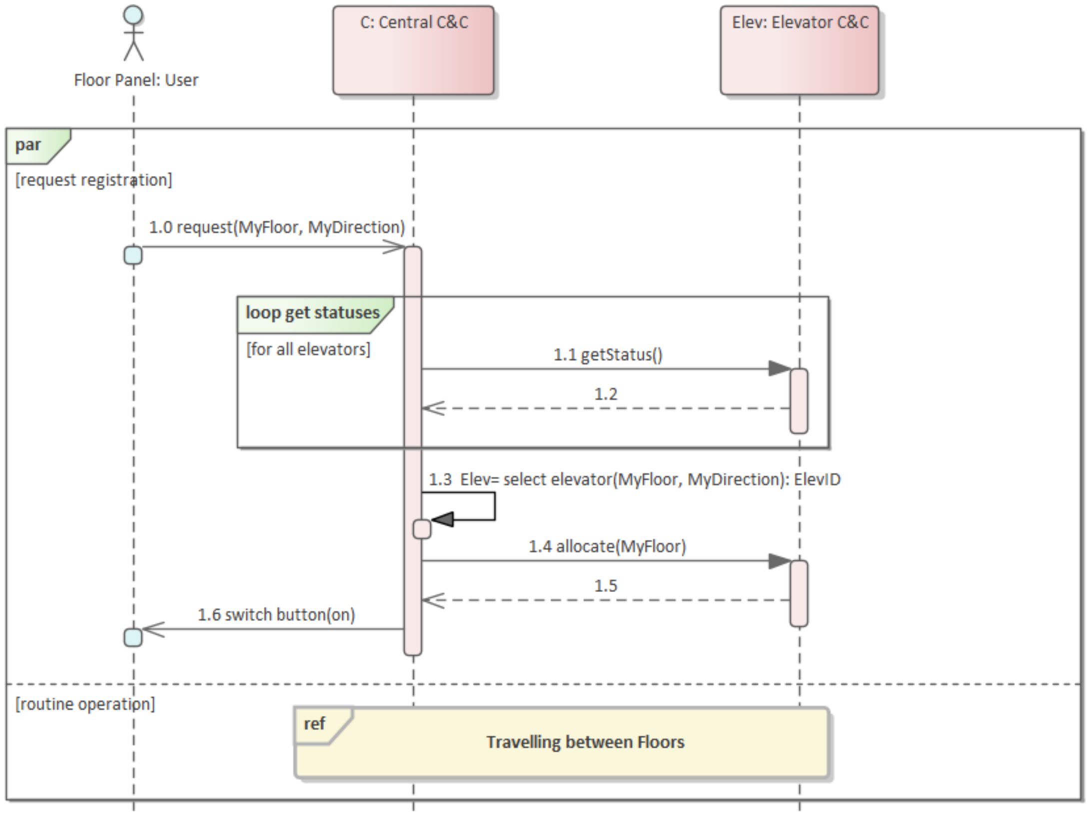
Planning software structure

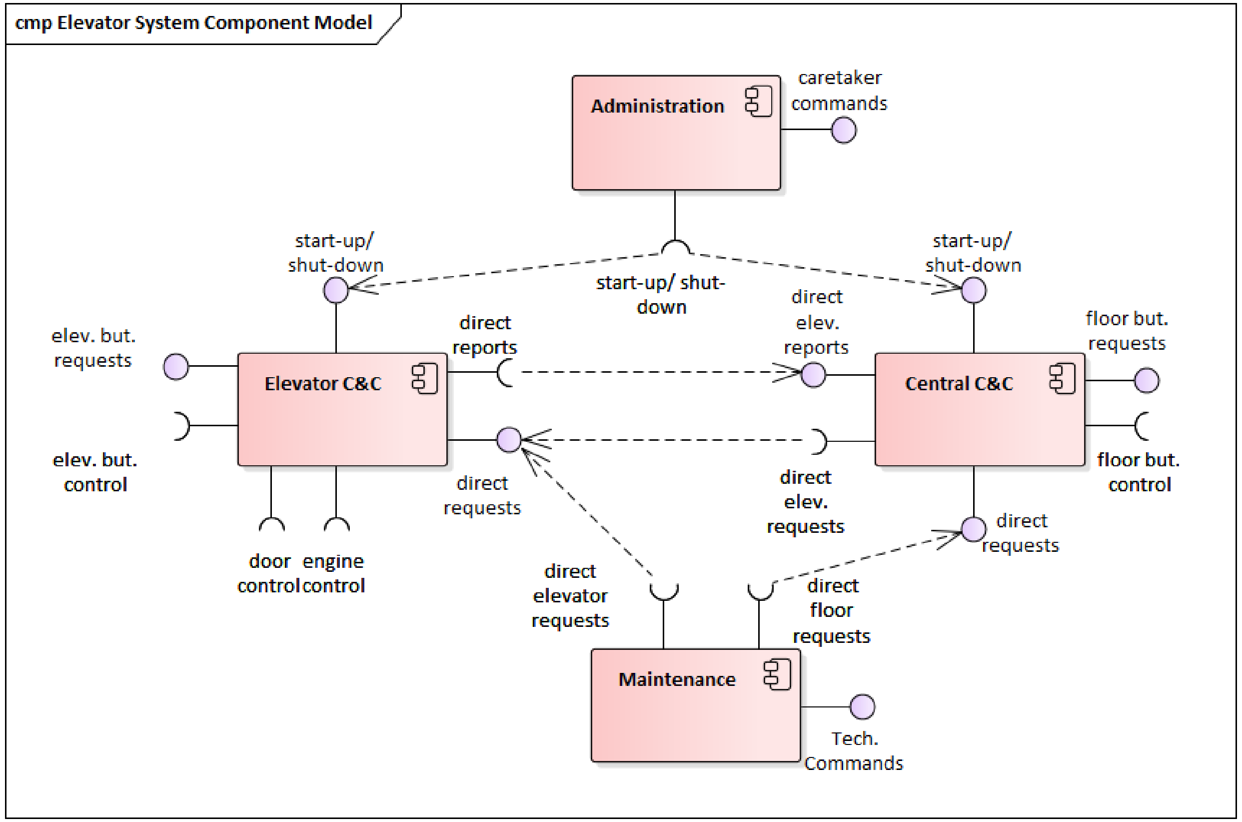
- Our goals:
  - Define the modules (classes that yield objects) that comprise the software
  - Assign functionality to classes (attributes and methods)
- Inputs:
  - Component Diagram
  - Component-level sequence diagrams
- Outputs:
  - Class diagram



What we have so far

- We built a software architecture that includes
  - **Structure**: the internal and external connections between parts – component diagram
  - **Behavior**: the interaction between components to implement tasks – sequence diagrams

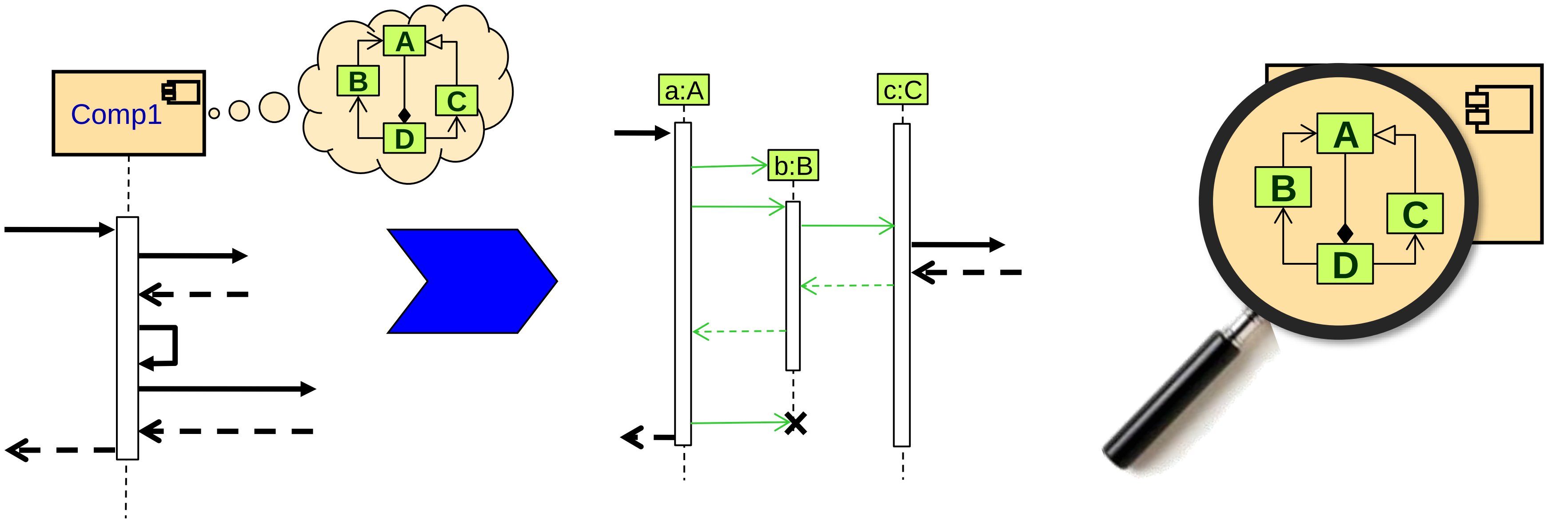




Object Oriented Design

- We need to break down each software component next

Goals	Ingredients	Structure	Behavior
• Implement each component's functionality	• Software modules (classes) that will comprise them	• The connections between various objects	• The interaction between the objects & between them and the environment that implements functionality



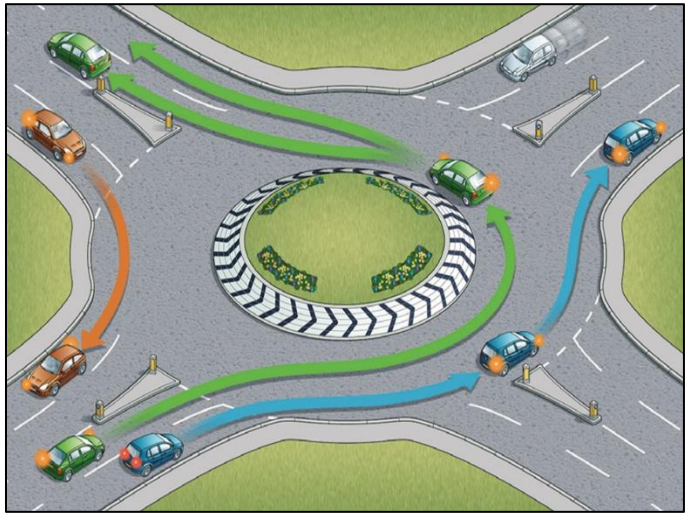
Traffic light vs. Roundabout

Traffic light intersection

- Vehicles are passive
- The algorithm is known only to the traffic light
- Traffic light manages all vehicles

Roundabout intersection

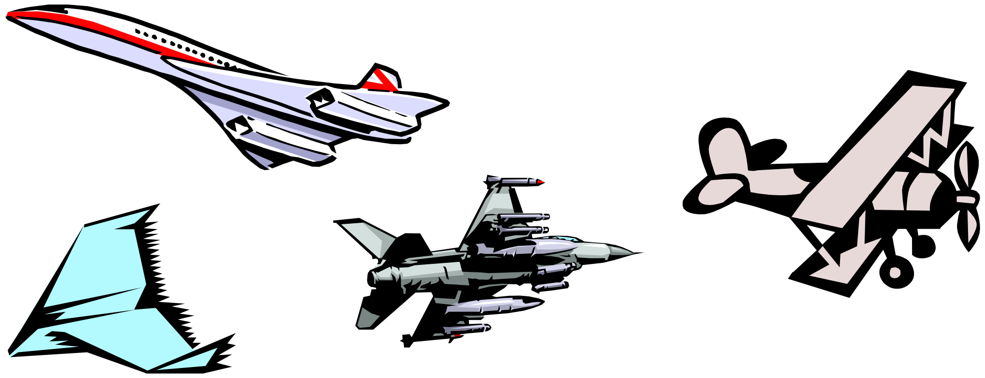
- Vehicles are active
- Each vehicle knows the algorithm
- Each vehicle manages itself and its interactions with other vehicles



Object Oriented Paradigm

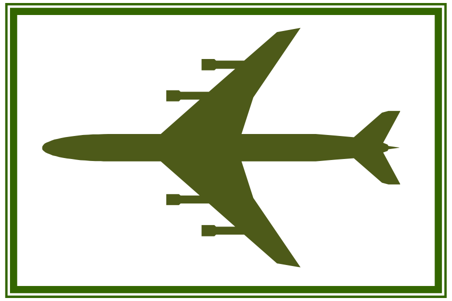
Object

- Specific entity
- Borders and identity defined
- Encapsulates state and behavior
  - State = data members, data structures
  - Behavior = member methods, functions



Class

- Descriptor of a set of objects with shared attributes
  - Properties, actions, relationships, behavior

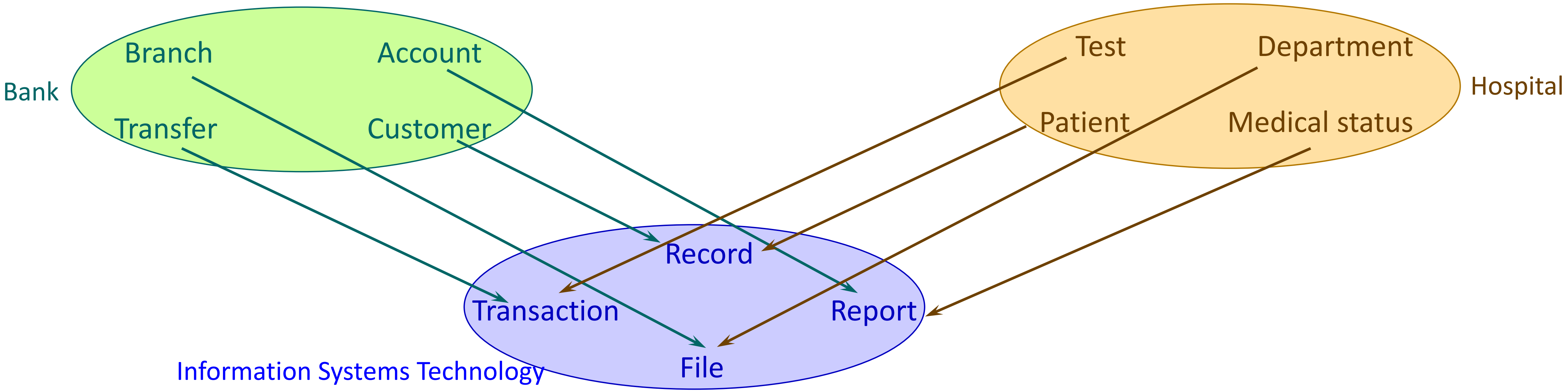


Classes exist in code only at code time
Objects exist in memory only at run time

Problem space and Solution Space

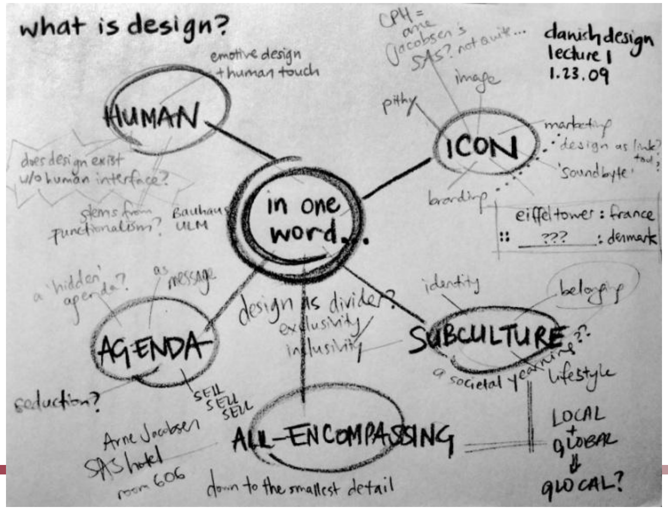
Reminder

- A development process solves a problem or a need by mapping from the problem space to the solution space
  - Solution space: Technology, engineering results
  - Problem space varies from system to system
- Example: Information Systems Technology
  - Solution space: Data records, files, reports, transactions
  - Problem space 1: A bank = {branch, customer, account, transfers}
  - Problem space 2: A hospital = {department, test, patient, diagnosis}



OO Modeling of Problem Space

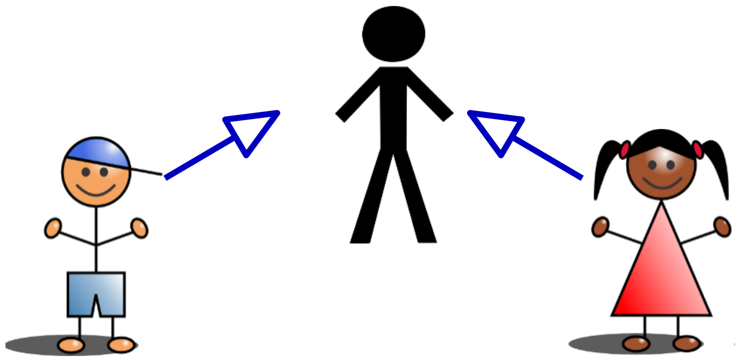
- OO allows us to build software that matches the problem environment
  - We later add solution-specific elements
- During system analysis we can build a structured models of objects that reflect the problem space
  - Problem domain object model (PDOM)



- Goal: Clarify and specify concepts and their relationships
  - Create a common language for stakeholders
- Uses:
  - Resolve contradictions and gray areas
  - System glossary
  - Basis for software objects
- Technique: Semantic network

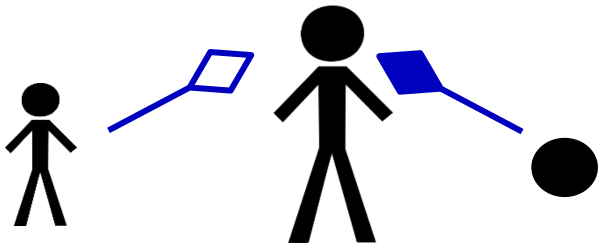
Semantic network

- Collection of concepts and relationships between them
- UML recognizes three kinds of relationships
  - Kind of
  - Part of/Has a
  - Relates to
- A is a B
  - A Boy is a kind of Person
  - A Girl is a kind of Person



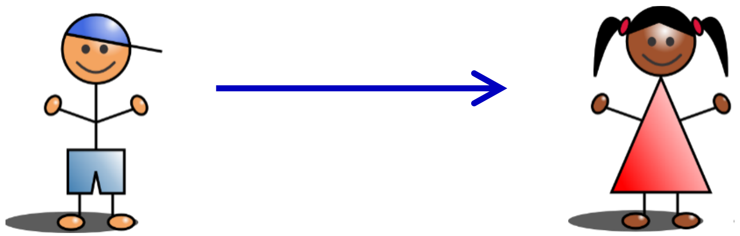
Semantic network

- Collection of concepts and relationships between them
- UML recognizes three kinds of relationships
  - Kind of
  - Part of/Has a
  - Relates to
- A has a B
  - A Head is part of a Person
  - A Person has a Child



Semantic network

- Collection of concepts and relationships between them
- UML recognizes three kinds of relationships
  - Kind of
  - Part of/Has a
  - Relates to
- A <relates to> B
  - A Boy works with a Girl



Operational Specification: Elevator System

Reminder

A passenger who is on a particular floor and wants to call an elevator presses the appropriate button for the direction of travel (up or down). If the button was not already lit, the button lights up after being pressed. An elevator car traveling in the requested direction will arrive at the floor within one minute at most. When the car arrives, the door opens and the light on the button turns off.

A passenger who is in an elevator car and wants to travel to a particular floor presses on the button associated with the desired floor. If the button wasn't already lit, the button lights up and a new stop request for the floor is registered. The door closes. After a short pause, the car continues moving. It stops at every floor for which there is a stop request. When the car stops at a floor, the door opens and the button for the floor turns off.

A passenger can stop the elevator car when its moving by pressing on the emergency stop button. In that case, the elevator stops immediately and all registered stop requests are erased. Afterwards, the elevator can resume operation by pressing a button for any floor.

Operational Specification: Elevator System

Reminder

If the elevator gets stuck while in operation, passengers can call for help using the emergency rescue button. The rescuer (the building maintenance engineer) will access the emergency panel in the machine room and perform operations to move the elevator car to the bottom floor and open the door.

The maintenance engineer is responsible to start up the elevators at the beginning of the day and to shut it down at the end of the day. The technician comes once every 6 months and performs a complete check of the system and fixes problems using the technician panel in the machine room.

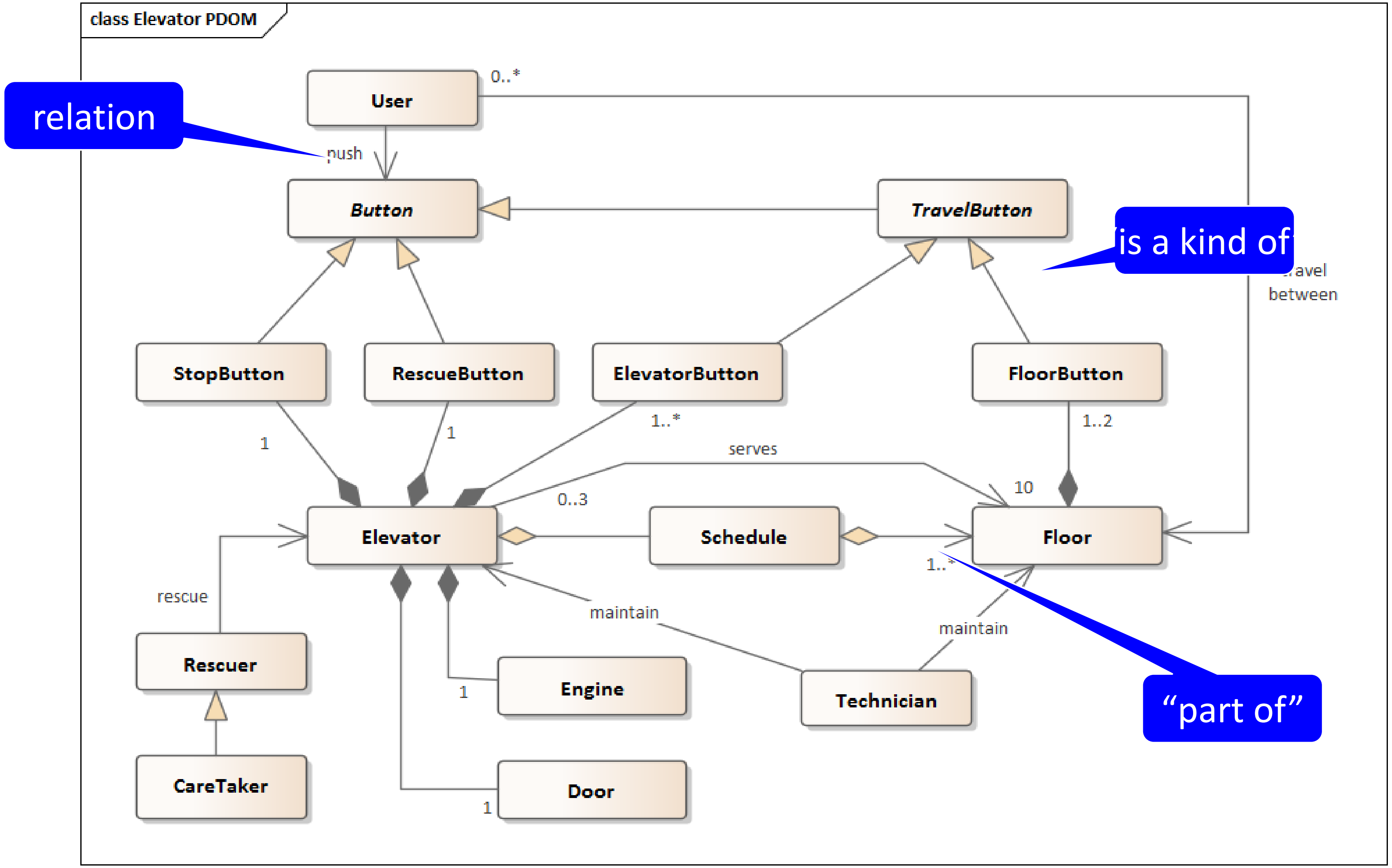
The elevator system must meet all applicable safety rules.

The system must be accessible to the handicapped.

List of Elevator Concepts

- Passenger
- Floor
- Elevator
- Directional button (floor)
- Door
- Floor button (in car)
- Stop request
- Emergency stop
- Rescue panel
- Maintenance engineer
- Check/Test
- Problems/faults
- Technician

Elevator PDOM



In class assignment: PDOM

- Create a PDOM for the ePark system
  - Use the Class Diagram Toolbox in Enterprise Architect
  - Include the following concepts (you can add your own if you need)

1. Child’s Guardian
2. Account
3. Child
4. eTicket
5. Bracelet
6. Entry
7. Attraction (device)
8. Supervisor

So Far

- Composite Architecture
- PDOM
- Class Model

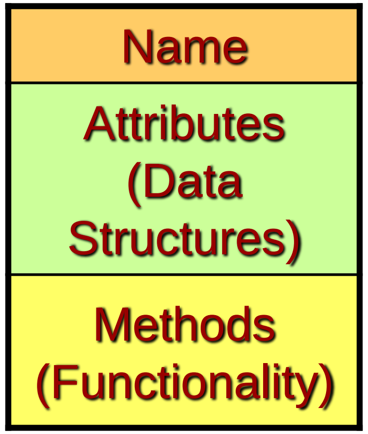
Software Elements: Objects and Classes

Objects

- Basic elements of software
- Every object manages its information via internal functionality (methods)
- Objects exist in memory while the program is running
  - Can create and destroy them dynamically
  - Constructor, Destructor
- Every object has at least one handle or pointer to it in memory

Classes

- Forms out of which objects are created
- Contain three elements



- Defined in code by the programmer
- Objects are instances of classes

Creating and Operating Objects

Create a new object	`theBlueCar = new Car()`
Initialize car details	`theBlueCar.make = “Mazda”` `theBlueCar.model = “CX-8”`
Licensing and registration	`theBlueCar.licenseplate = “12-345-67”` `theBlueCar.registrationDate = 08/09/2021`
Selling	`theBlueCar.sellTo(Lior)`

	`Function sellTo (X){`
	`owner = X;`
	`}`

Car

+ maker: string
+ model: string
+ licensePlate: string
+ testDate: Date
- owner: Person

+ sellTo(Person) : void
+ getOwner(int) : Person
+ testIsValid(Date) : boolean

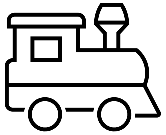
Object name

Class name

<u>theBlueCar : Car</u>
maker = “Mazda”
model = “CX-8”
licensePlate = “12-345-67”
registrationDate = 08/09/2020
owner = Lior

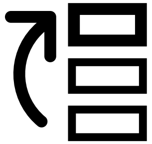
Candidates for Objects in Software

Objects that represent physical entities



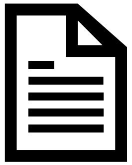
- Door, Engine, Workstation
- Attributes:
  - Parameters and data about the entity
  - Input/output
- Methods: Physical functionality
- Serve as a stand-in or interface to physical object

Objects that represent logical entities



- Process, Service
- Attributes:
  - Parameters and data about the entity
  - Input/Output
  - Methods:
    - Operations the entity can do

Objects that represent data entities

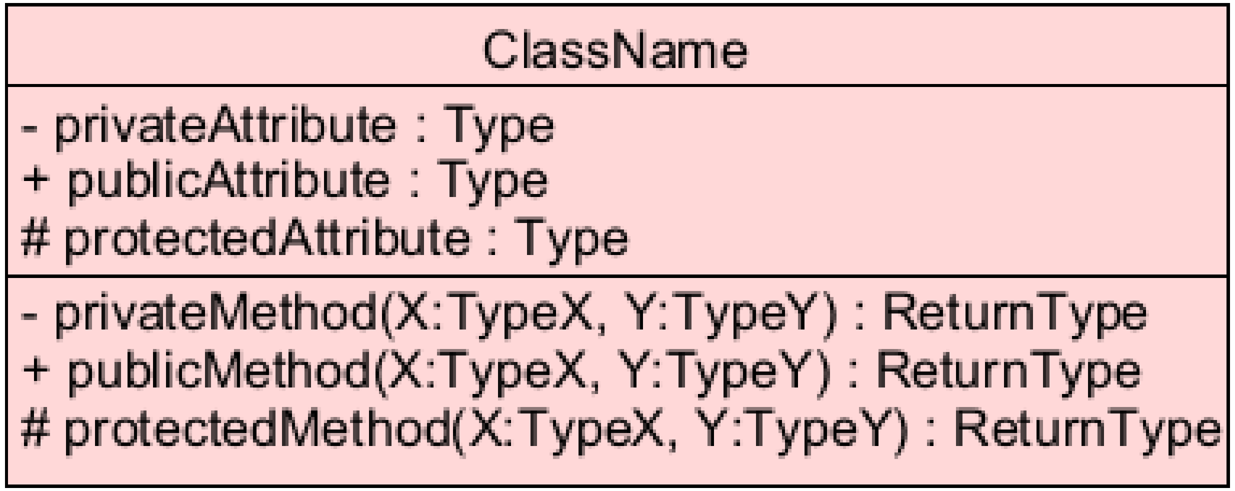


- Databases, data stores, lists, queues
- Attributes:
  - Data elements managed by the object
- Methods:
  - Data operations (store, retrieve, modify)

Class Diagram - Syntax

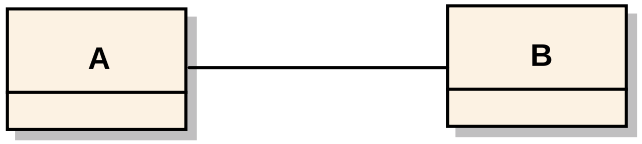
Class

- Name
- Attributes (variables)
  - Private (-): Can only be accessed from within the class
  - Public (+): Can be accessed also externally
  - Protected (#): Can be accessed within the package or by sub-classes
- Methods (Functions)
  - Private (-): Can only be called from within the class
  - Public (+): Can be called also externally
  - Protected (#): Can be called within the package or by sub-classes

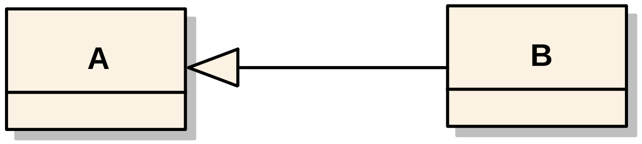


Class Diagram - Syntax

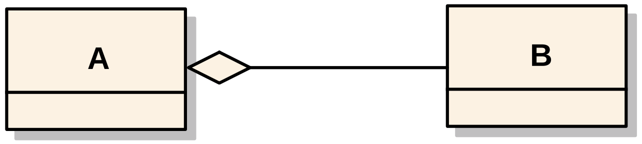
Class diagrams are based on the principles of semantic networks



Association



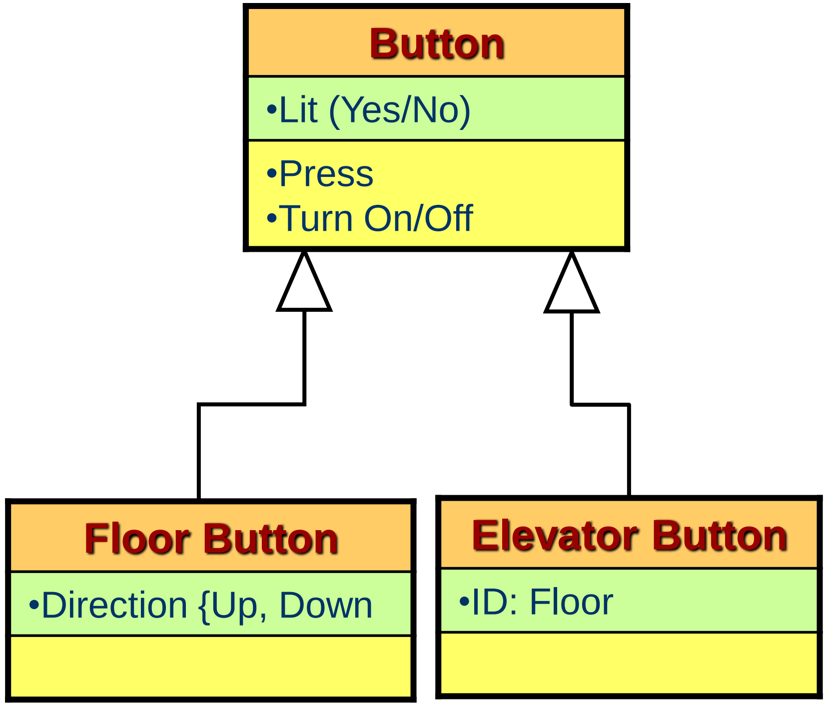
Inheritance



Aggregation

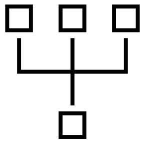
Inheritance and Generalization

- When class B inherits from Class A
  - B has all of A's attributes
  - B has all of A's methods
  - B can also do more stuff
- Inheritance is "B is-an A"
- B is a "sub-class" of A
  - B can do more than A
- Inheritance creates hierarchical relationships
- Abstract classes
  - A class you can't make instances of
  - All instances are of subclasses (e.g. vehicle)



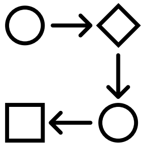
Problems with inheritance

Multiple Inheritance



- One class inherits from multiple classes
- Problem:
  - Might lead to contradictions in attributes or methods
- Solution:
  - Most programming languages don't allow multiple inheritance (force tree-like structure)

Deep Inheritance



- A ◁ – B ◁ – C ◁ – D ◁ – … ◁ – X
- Problem:
  - Keeping track of connections (maintainability)
- Solution:
  - Break the chain where connection is weak

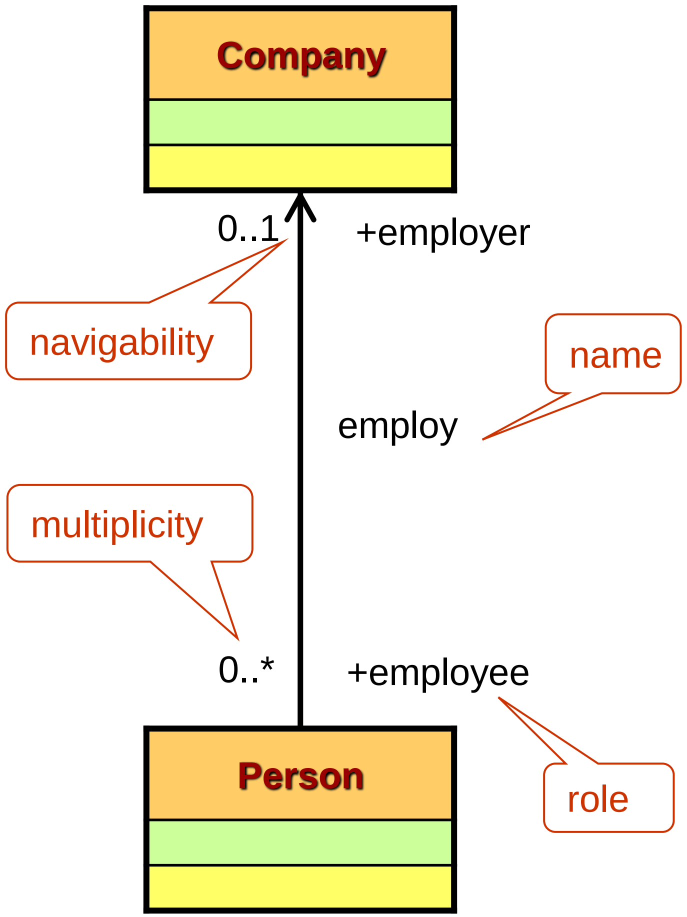
Problems with inheritance

False Inheritance

- Example: A square is a type of rectangle, so Rectangle ◁ — Square
- Problem:
  - Rectangle has two attributes (len, wid)
  - Square has one attribute (edge)
- Solution:
  - Define inheritance based on shared properties (attributes, methods) not just conceptual similarity

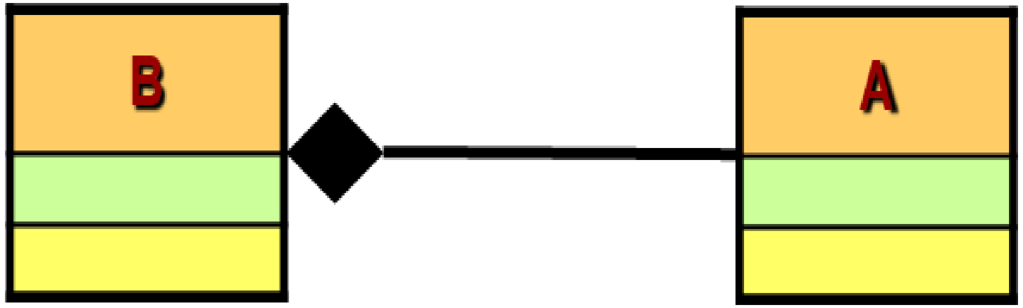
Association

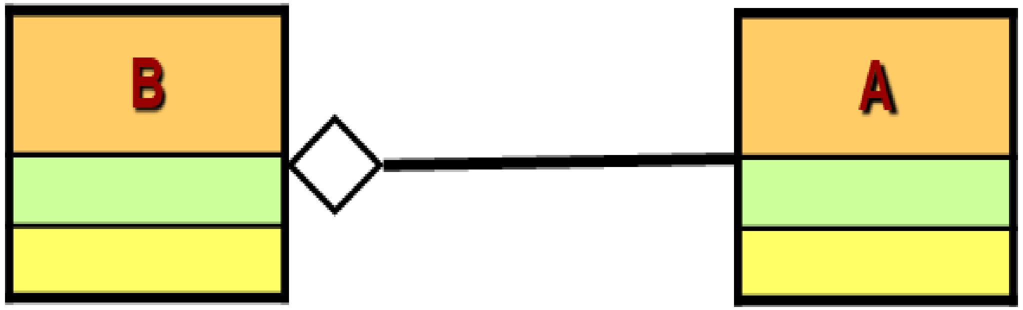
- Relationship between classes that instances of them “know” each other
  - Knowing works via references and pointers
  - Properties specific association
- Name: Can be read both ways
  - Company employs person, Person employed by company
- Role
  - Company is the employer, Person is the employee
- Multiplicity
  - Company employs 0 or more Persons, Person is employed by 0 or 1 Companies
- Navigability
  - Person knows who is its Company, Company does not know its Persons



Aggregation

- A special kind of association (knowing): B has-a A





Composite Aggregation

- A is an integral part of B and only of B
- A exists only due to B
- AKA: Whole-part aggregation, Non-shared aggregation

Shared Aggregation

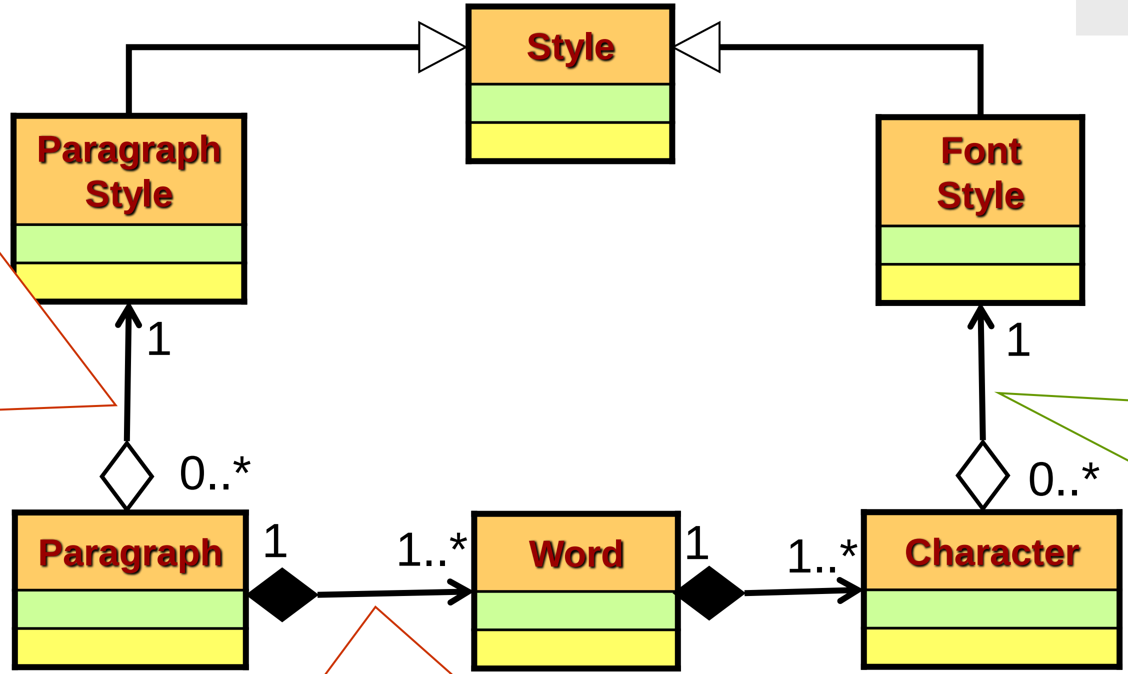
- A is part of B, but
- A can exist without B
- A can be shared between other objects

Aggregation Example

1. A Numbered Title

This is the first *paragraph* of this document. It contains 17 *words* and 80 non-blank characters.

Shared aggregation:
- A paragraph has one Paragraph Style
- Paragraph style can apply to many paragraphs
- Paragraph style is an independent entity and can exist without any paragraphs
- Deleting a paragraph doesn't delete the style



Navigation:
- Character knows its style
- Style doesn't know characters it applies to

Composite aggregation:
- Paragraph has at least one word
- Each word in the paragraph belongs to only one paragraph
- Words exist only within a paragraph
- Deleting a paragraph deletes all the words within

Software level functional analysis

- As with functional analysis before, we need to do software functional analysis (objects, classes)
- As a start, software elements are classes in the PDOM → Then assign methods and functionality
  - We may add classes later as necessary
- Sources of functionality
  - Software architecture: Implement processes found in the sequence diagrams
- When assigning functionality, preserve "Specialization" and "Independence"
  - Tight cohesion: What is the common ground among data and methods in the class?
  - Weak coupling: How much is a class dependent on others?

System level functional analysis

Reminder

- We did the following at the system architecture level

1. Find functionality in use cases

SUC-1	Call Elevator
Actors and Goals	Passenger: To receive an elevator car available for travel
Stakeholders and Interests	None
Pre-conditions	• User is on a floor in the building with an elevator door • System is operational (post-condition of UC: Start-up)
Post-conditions	An elevator car is at the user's floor with the door open (destination floor)
Trigger	**Passenger** pushes the up or down button on the floor
Main Success Sequence (MSS)	1. The system records the button press 2. The button lights up 3. The system finds a car traveling in the desired direction 4. The system assigns a stop for the car 5. The elevator arrives at the floor 6. The door opens 7. The floor button turns off

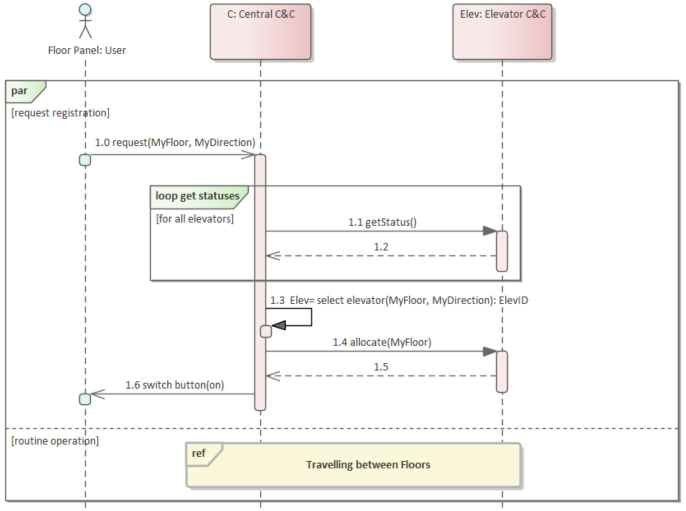
2. Break down into components

Whole system operations

System Command and Control — Central C&C
1. Receive user requests from per-floor up/down buttons
2. Assigning calls to elevator cars
3. Sending calls to elevator cars
4. Identify emergency events and take care of rescue operations

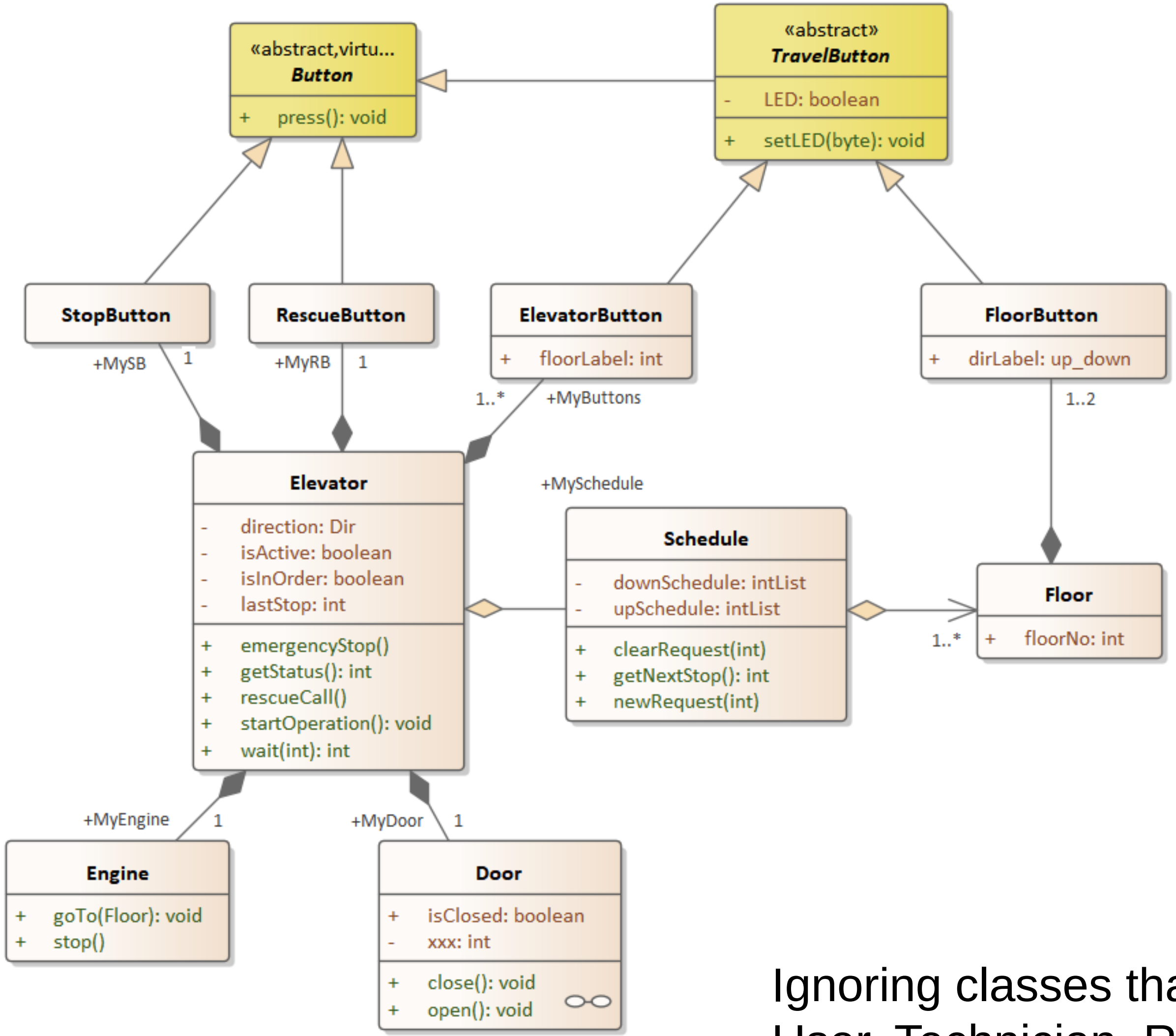
System Maintenance — Maintenance
5. Built in Test (BIT)
6. Support technician tests
7. Support technician repair tasks

Configuration Management — Administration
8. Start up
9. Shut down
10. Sabbath mode



3. Implement processes using sequence diagrams

Elevator system – Base class diagram from PDOM

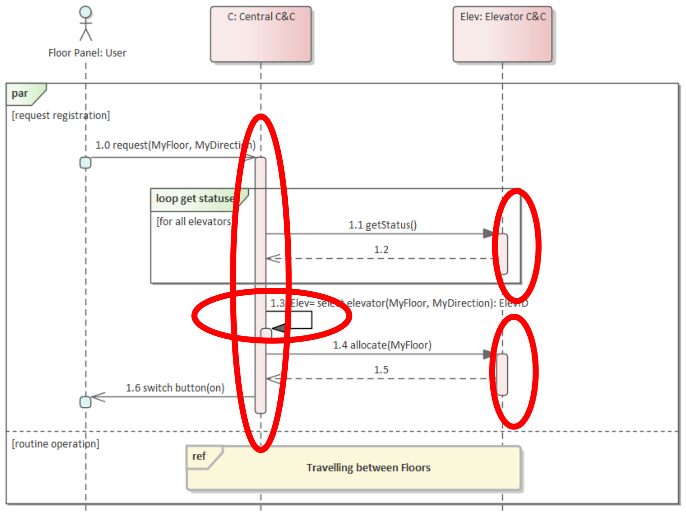


Ignoring classes that represent people:
User, Technician, Rescuer, Maintenance Engineer

Software level functional analysis

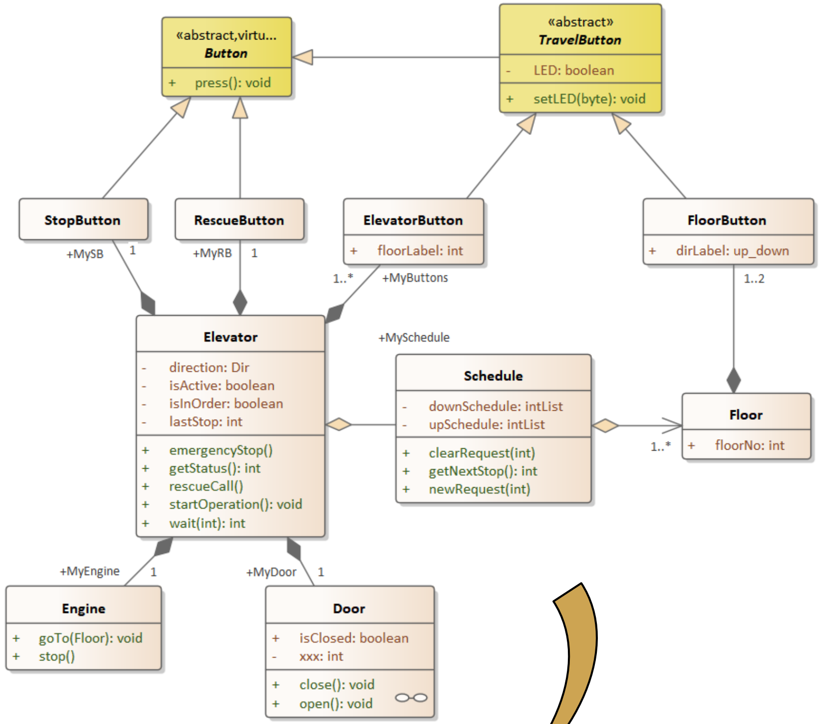
- Perform the steps above on the level below

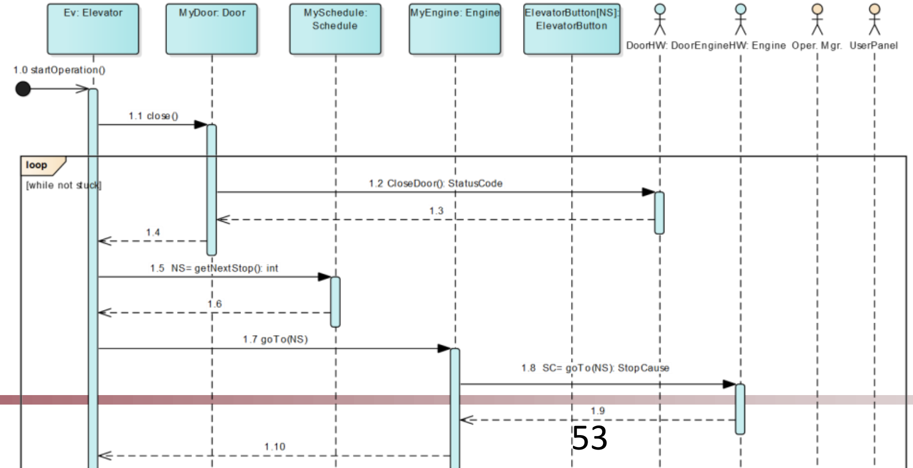
1. Find functionality in seq. diagrams



See next slide

2. Find classes, attributes, and methods



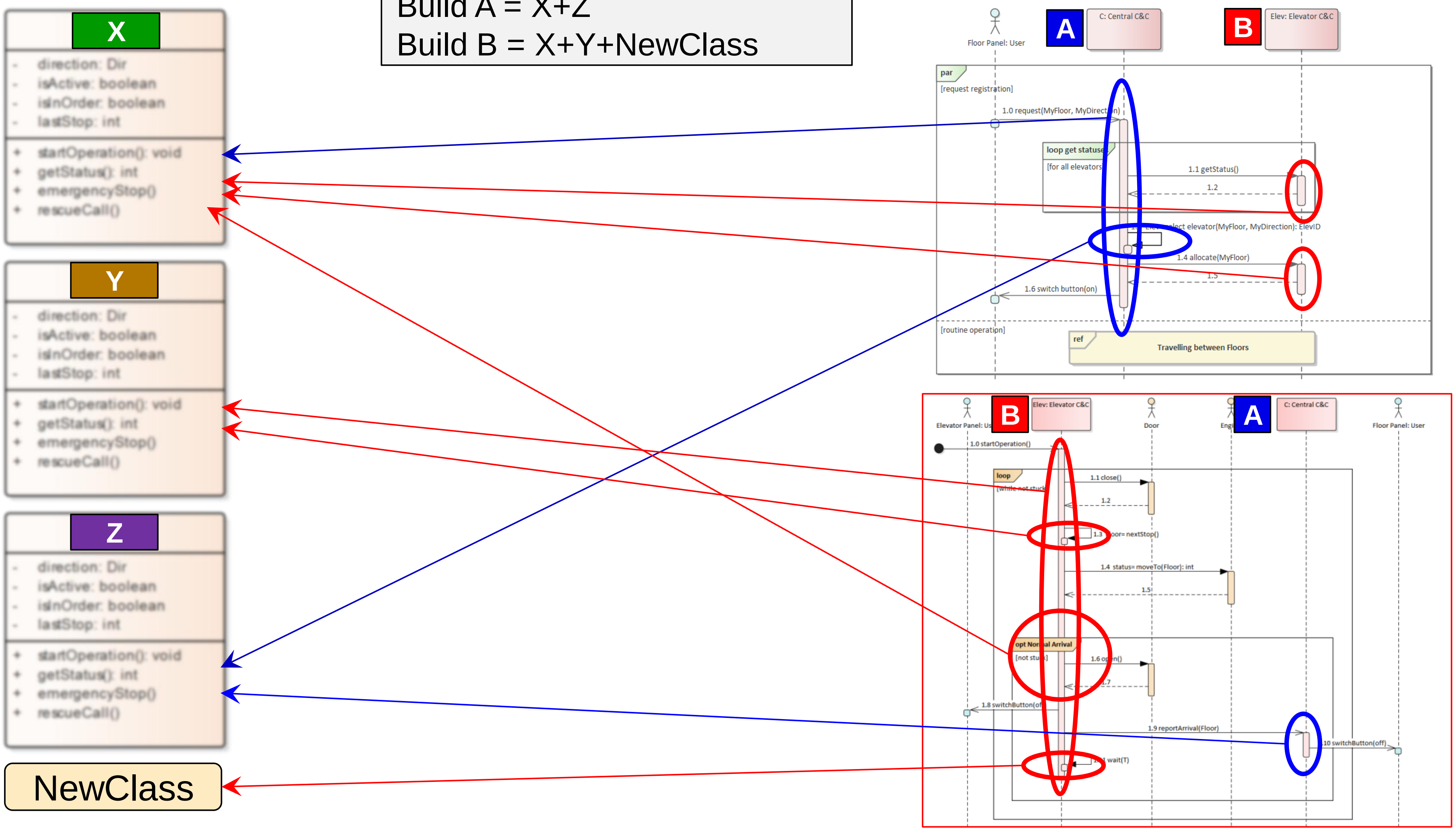


3. Implement functionality via interactions

Assigning component functionality to classes

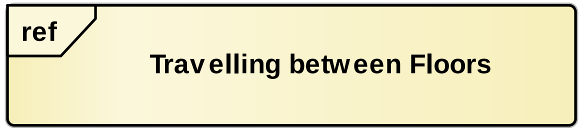
Build A = X+Z
Build B = X+Y+NewClass

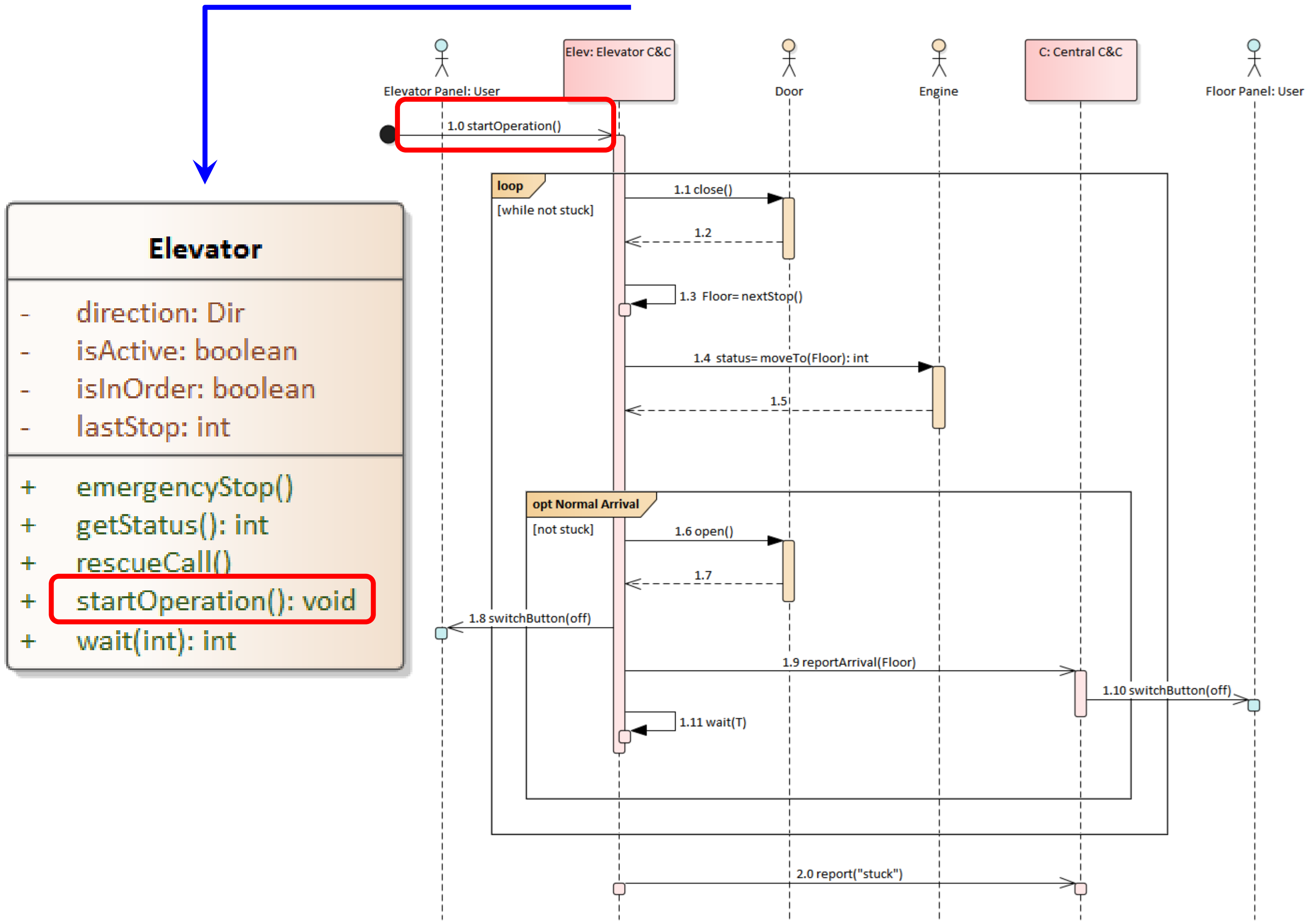
Classes initially identified in the PDOM



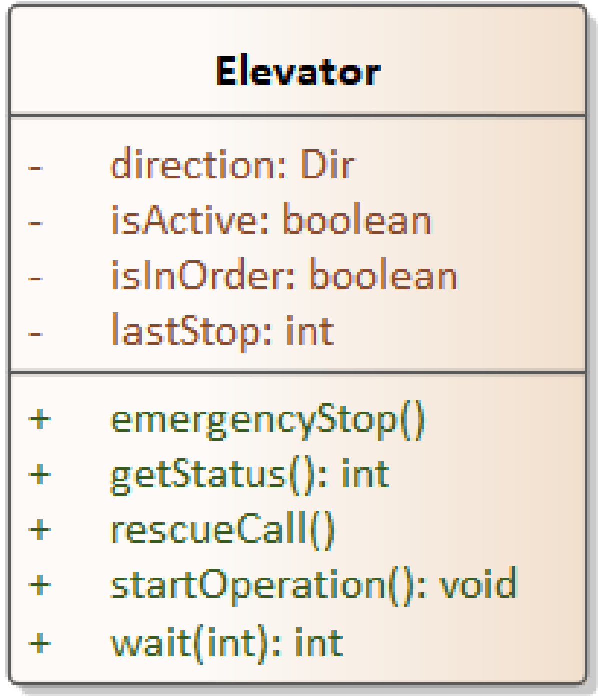
Sequence diagrams with components A and B

Example: Regular Elevator Operations





Elevator System: Elevator Class



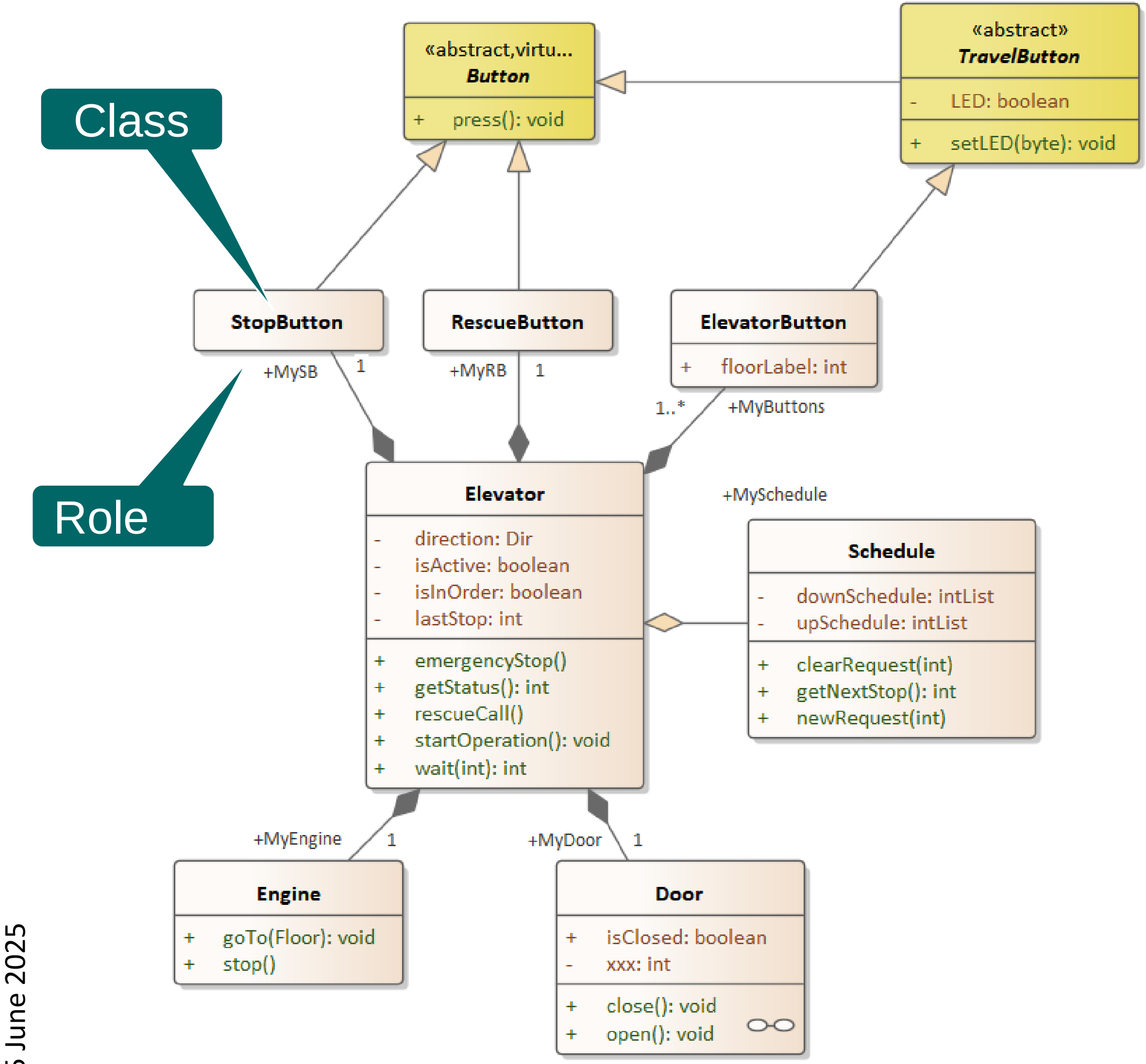
In addition to these attributes, the pointers to other classes that are derived from associations and aggregations

In addition to these attributes and methods, there are inherited attributes and methods

Tracking functional requirements to class diagram

- Classes in the class diagram must fulfill all system functionality
- Every functional requirement (FR) must point to one or more classes that fulfill it
  - Participate in an OR
    - E.g. “If the button wasn’t previously lit, it turns on after being pressed” → Button
  - Provide data structures for DR
    - E.g. “Every floor has two buttons” → Floor
- Each class in the class diagram must refer to the FR related to it

Automatic Static Code from Class Diagram – Attributes




```
public class Elevator {

    private Dir direction;
    private boolean isActive;
    private boolean isInOrder;
    private int lastStop;
    public ElevatorButton MyButtons;
    public Door MyDoor;
    public Schedule MySchedule;
    public RescueButton MyRB;
    public Floor ServedFloors;
    public Engine MyEngine;
    public StopButton MySB;
```

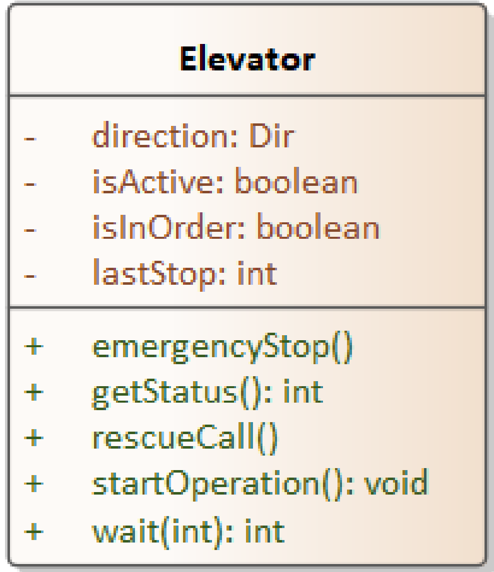
Explicitly defined attributes (the private attributes)

Attributes derived from associations (the public attributes)

Class (points to the type in public StopButton MySB;)

Role (points to the name in public StopButton MySB;)

Automatic Static Code from Class Diagram – Methods



Constructor

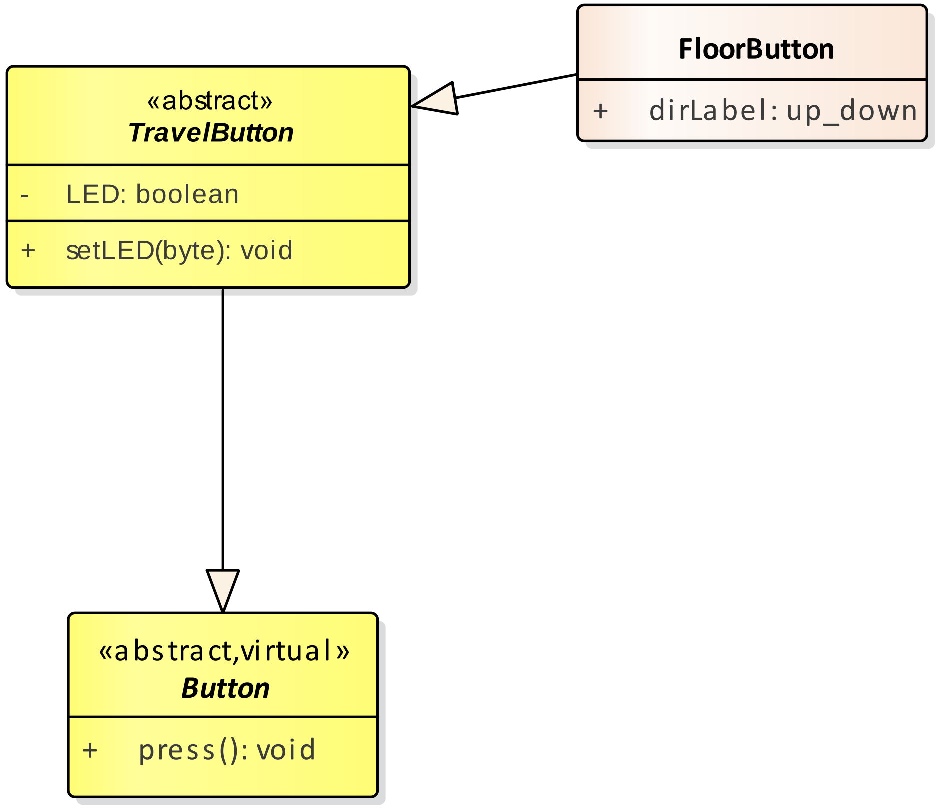
Destructor

Explicitly defined methods

```
public class Elevator {

    public Elevator(){ }
    public void finalize() throws Throwable { }
    public emergencyStop(){}
      public int getStatus(){
            return 0;
    }
    public rescueCall(){}
    public void startOperation(){ }
    public int wait(int Sec){}
```

Automatic Static Code from Class Diagram – Method Inheritance




```java
public class FloorButton extends TravelButton {
        public up_down dirLabel;
        public FloorButton(){ }
        public void finalize() throws Throwable {
                super.finalize();
        }
}
```

```java
public abstract class TravelButton extends Button {

        private boolean LED;
        public TravelButton(){ }
        public void finalize() throws Throwable {
                super.finalize();
        }
        public void setLED(byte on_off){ }
}
```

```java
public abstract class Button {
        public Button(){ }
        public void finalize() throws Throwable { }
        public void press(){ }
}
```

Conclusion

- Composite Architecture
- PDOM
- Class Model