CESE4015 Software Systems

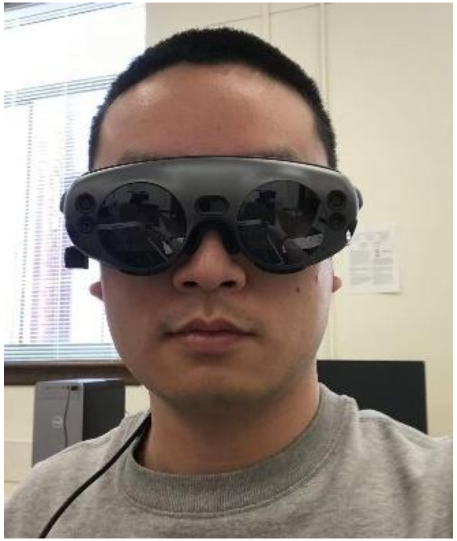
# Unified Modeling Language: An Introduction

**Guohao Lan**

Embedded Systems Group

December 13th 2022

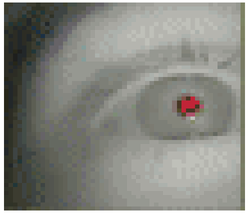
# Brief Intro

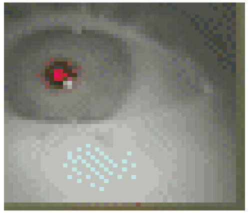


**Guohao Lan,**
**Assistant Professor**
**Embedded Systems Group**

**Doing research in:**

**Mobile Computing, Eye Tracking, and Deep Learning**





# Learning objectives

- At the end of the course, you should be able to:
  - Understand:
    - The purpose of UML (unified modeling language)
    - Three categories of UML diagrams:
      - Structural, behavioral, and interactional.
  - When and how to apply basic UML diagrams to model software systems.

- Assessment:
  - Modeling assignments using UML diagrams. [Group of two]
  - Reflection document on UML-based modeling. [Individual]

# Agenda for UML

- Week 5 Lecture:
  - Background of UML
  - Use Case, Component, Deployment
- Week 5 Lab:
  - Modeling with UML diagrams (part 1)
- Week 6 Lecture:
  - Class, Sequence
- Week 6 Lab:
  - Modeling with UML diagrams (part 2)

# Acknowledgements

- Slides materials are built from different sources:
  - Slides created by Marty Stepp, CSE403 @ U Washington.
  - *UML Distilled, 3rd edition* by Martin Fowler.
  - *The Unified Modeling Language Reference Manual, 2nd edition* by James Rumbaugh, Ivar Jacobson, and Grady Booch.
  - *Practical UML: A Hands-On Introduction for Developers* by Randy Miller.
  - *IBM Rational Software Architect Documentation:* https://www.ibm.com/docs/en/rational-soft-arch/9.5

- Lab platform:
  - PlantUML: https://plantuml.com/
  - A tutorial will be given by TAs during the lab sessions.

# Agenda for UML

- Week 5 Lecture:
  - Background of UML
  - Use Case, Component, Deployment
- Week 5 Lab:
  - Modeling with UML diagrams (part 1)
- Week 6 Lecture:
  - Class, Sequence
- Week 6 Lab:
  - Modeling with UML diagrams (part 2)

# Background

- **Discussion**:
  - Did you use any models in the Rust part of the course?
  - Could you understand of each other's designs/codes easily?

**Think → Pair → Share**

# Background (cont.)

- What is the UML?
  - **UML**: A family of standardized graphical notations that helps in describing and designing software systems at a high level of abstraction.
  - It is a graphical design notation:
    - More clear than natural language and code.
    - Simplifies system design process and avoid a lot of details.
  - Help communicating ideas about a system design.
  - It is language and technology independent.
  - It is a unified/standardized language.

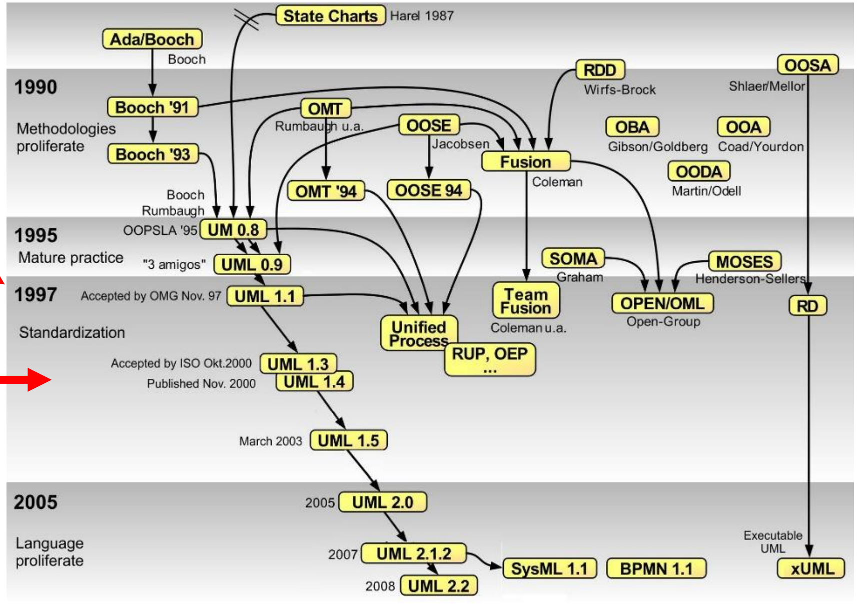
# Background (cont.)

- UML is based on many earlier software design approaches:
  - Evolving since 1990s and highly related to object-oriented programming:
    - The Booch method, the Object-modeling Technique (OMT), the Object-oriented Software Engineering (OOSE) and more.

- Driving force:
  - Programming languages do not provide a high enough level of abstraction to facilitate the design.

UML was adopted as a standard by the Object Management Group (OMG)

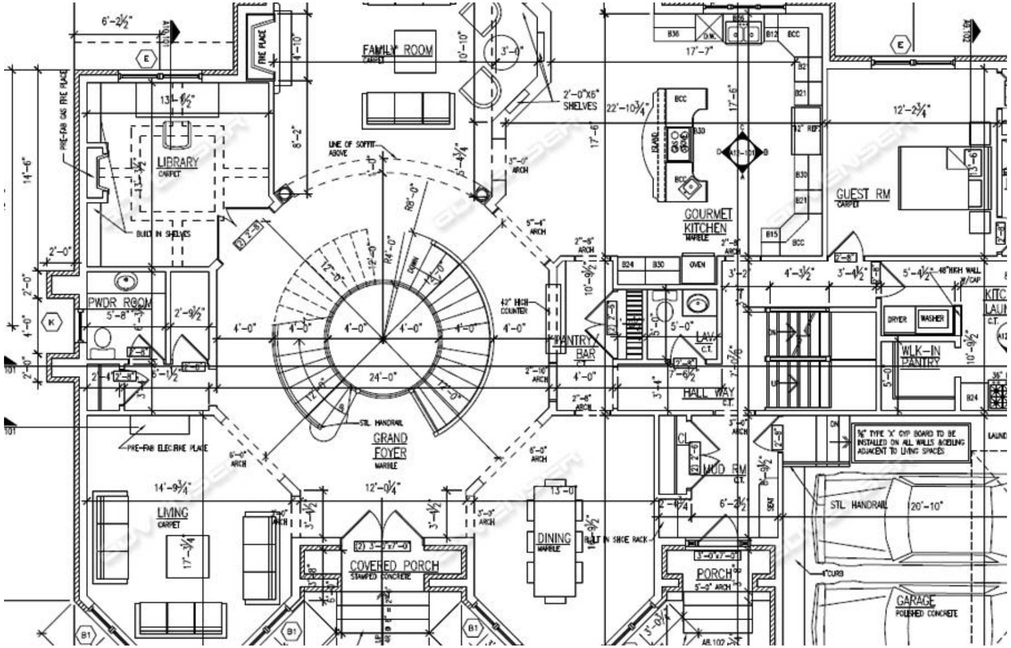
Accepted by IOS as a standard and been periodically revised.



# Why bother with the UML?

From the view of building construction:

A unified standard that can be understood by architects and builders.

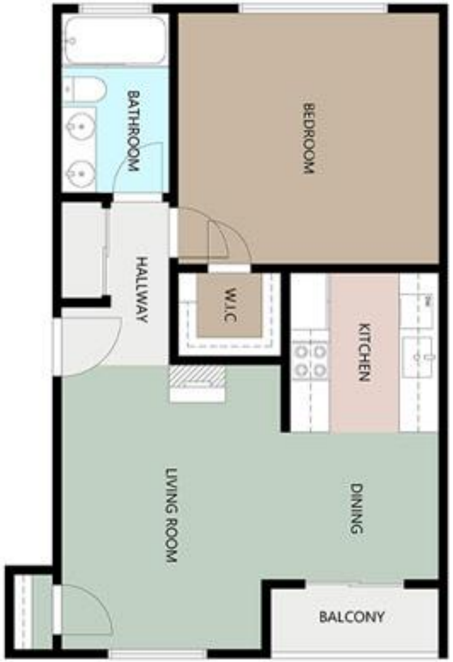


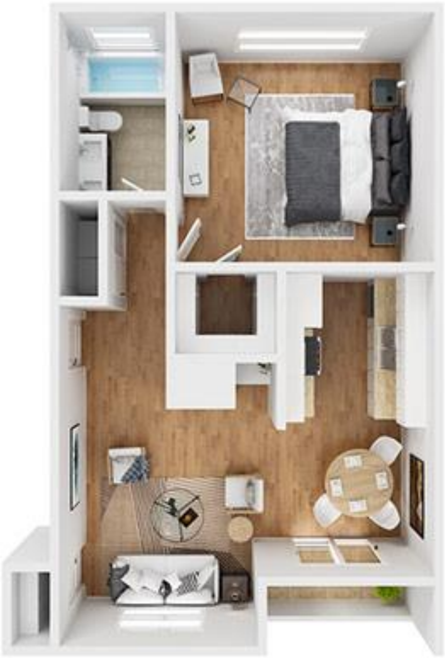
UML is programming language and technology independent and is a unified/standardized language that has been widely used.

# Why bother with the UML? (cont.)

From the view of building construction:

Providing different views (and levels of abstraction) of the design based on the needs.





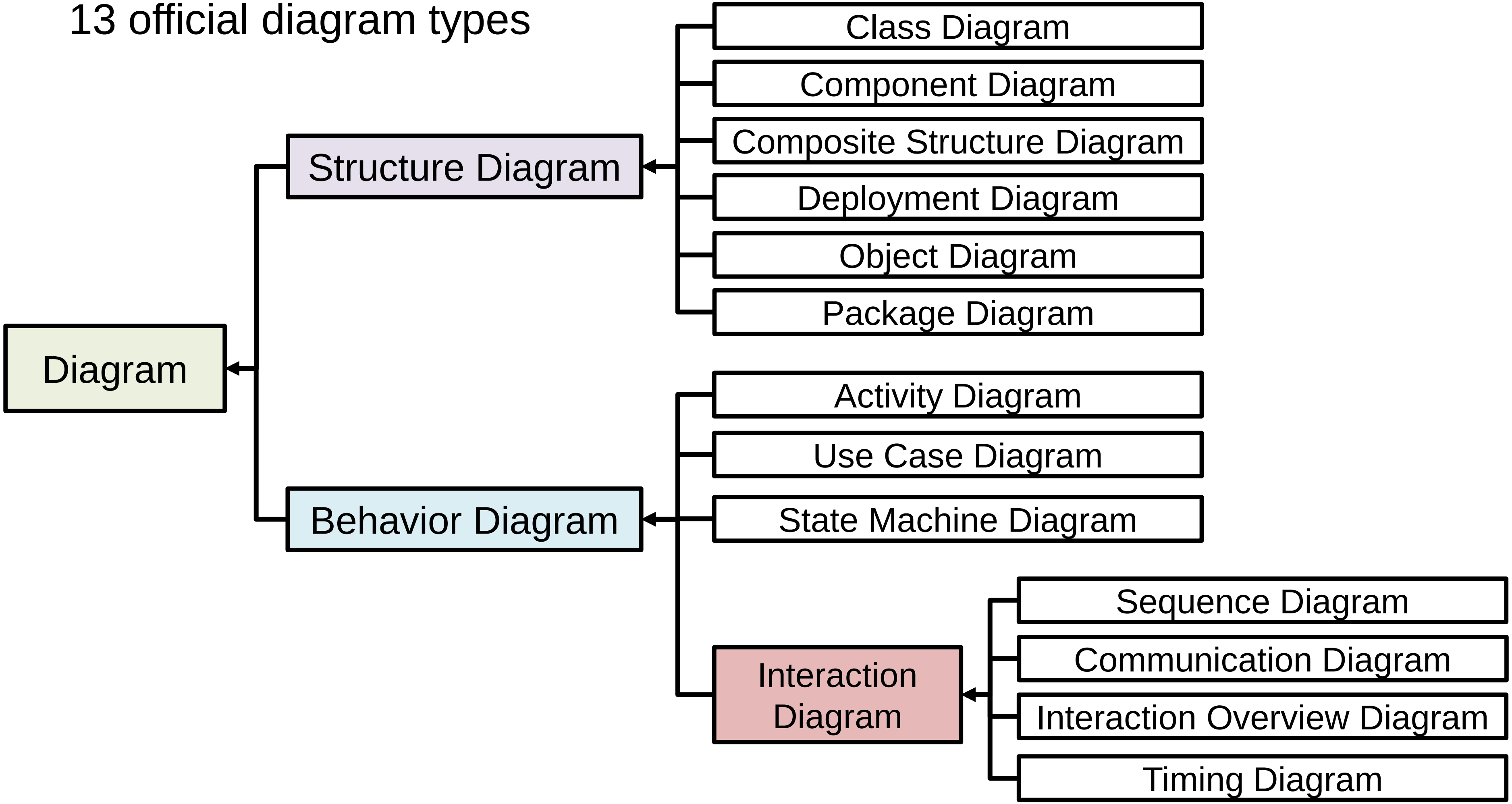
# Why bother with the UML? (cont.)

- Ways of using the UML:
  - Three modes [1]:
    - UML as **sketch**:
      - Use UML to help communicate high-level aspects of a system.
    - UML as **forward engineering**:
      - Draw a complete UML diagram before you write codes. The design covers sufficient design decisions for the programmer to code up.
    - UML as **reverse engineering**:
      - Build UML diagrams from existing code in order to help understand it.

[1] UML Distilled, 3rd edition by Martin Fowler

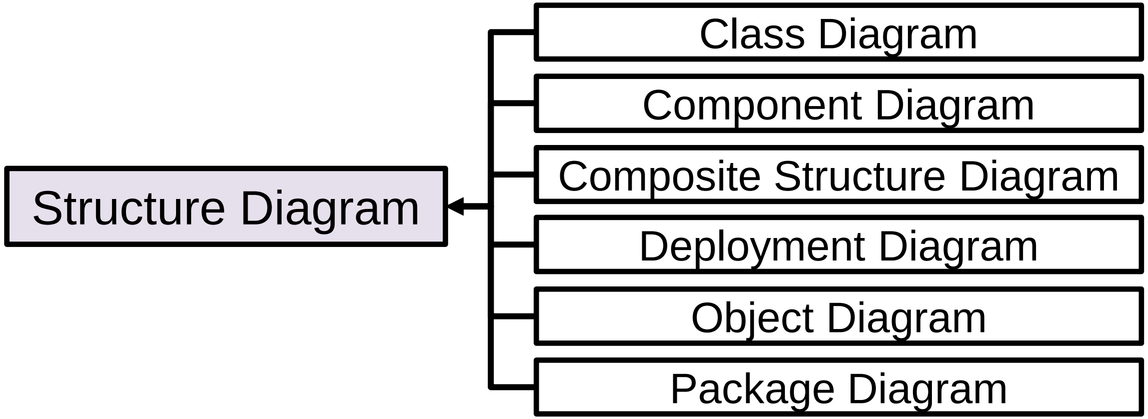
# Overview of UML Diagrams

13 official diagram types



# Overview of UML Diagrams (cont.)

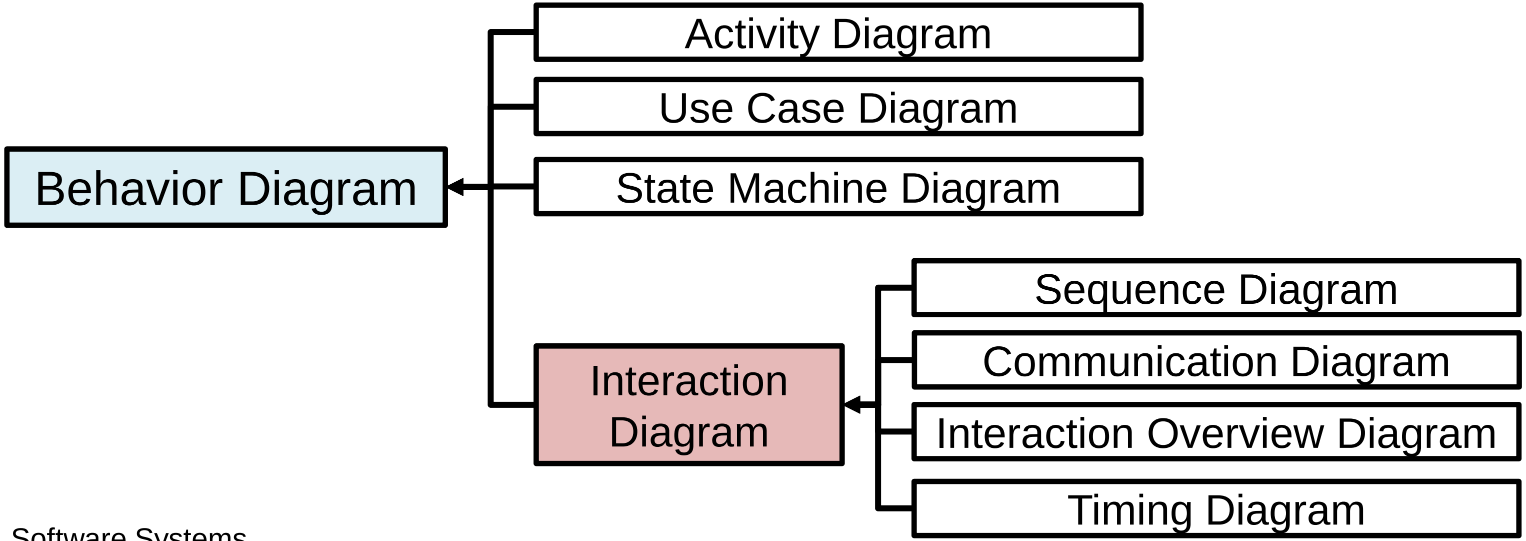
- Three types of diagrams:
  - Structural diagrams:
    - Emphasizes the static structure of the system and the things that must be presented in the system, including objects, attributes, operations, components, and relationships.
    - Used extensively in documenting the architecture of the software systems.



# Overview of UML Diagrams (cont.)

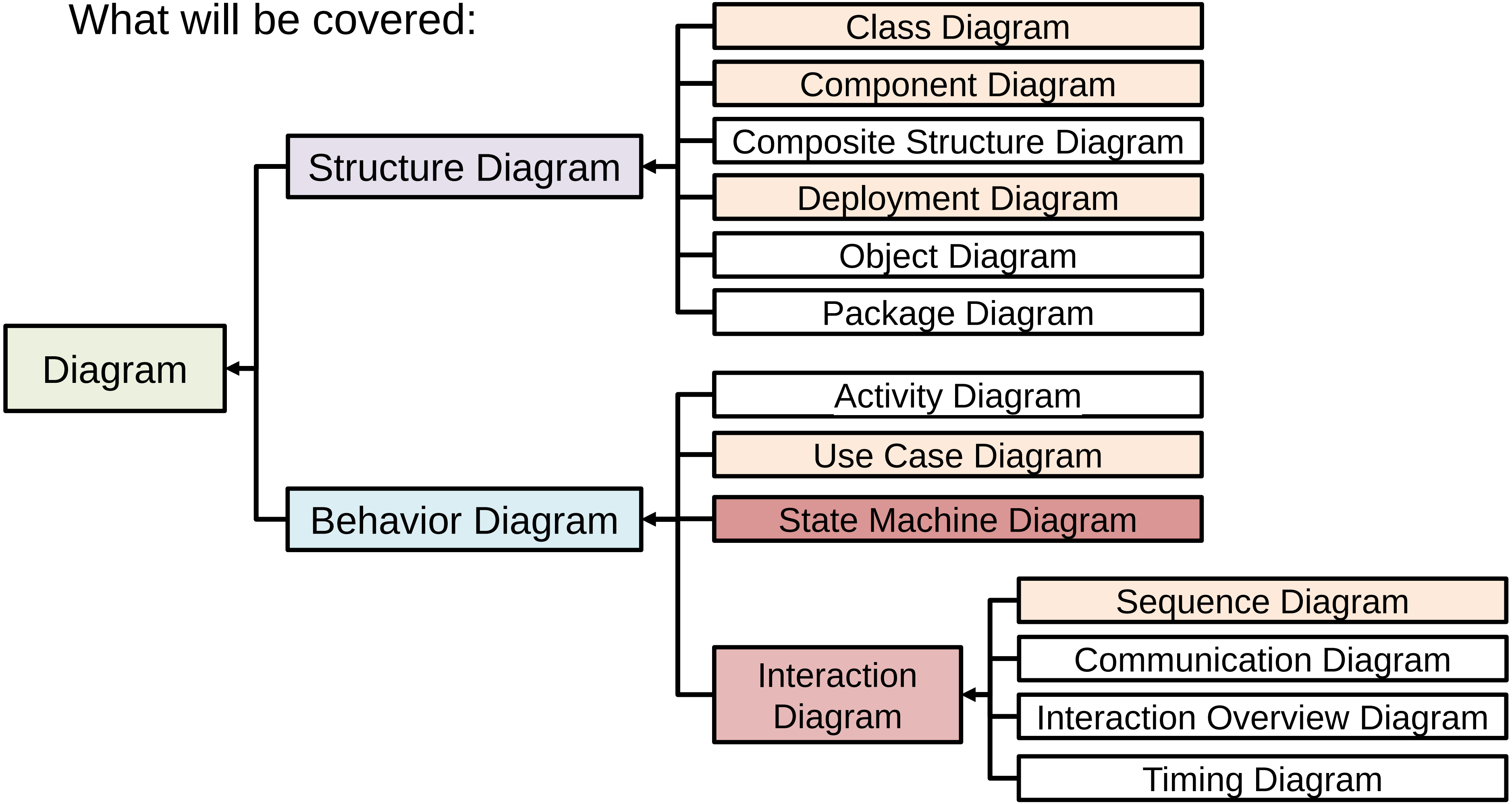
– Behavioral diagrams:

- Focuses on the dynamic behavior of the systems and changes to the internal states of objects.
  - Behavior: how data moves; how does the system change in time; how system behaves with different events.
- Interaction diagrams:
  - Interaction: emphasize the flow of control, showing collaborations among objects; how objects communicate;

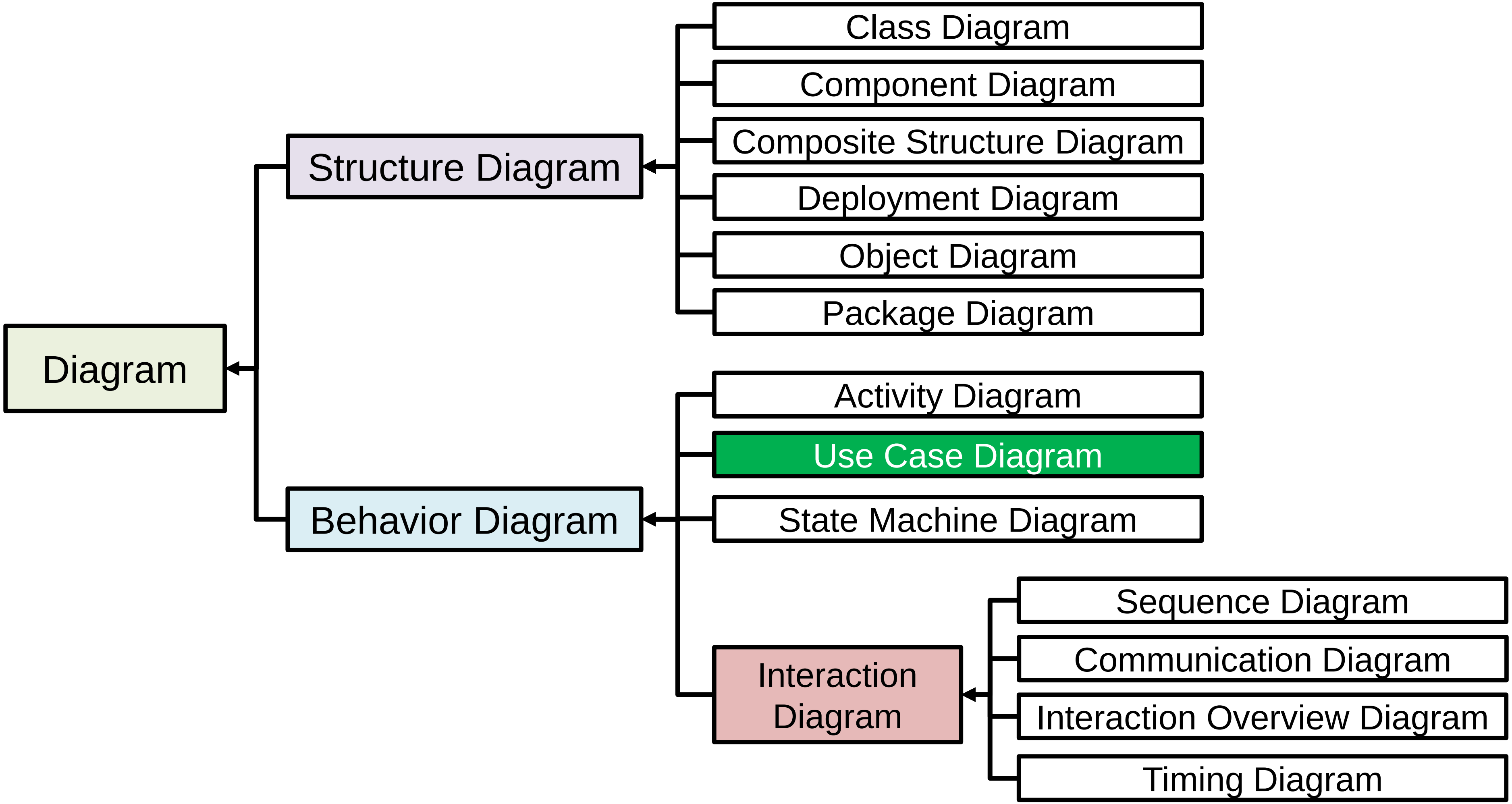


# Overview of UML Diagrams (cont.)

What will be covered:



# Use Case Diagram



# Use Case Diagram (cont.)

- What is the Use Case Diagram?

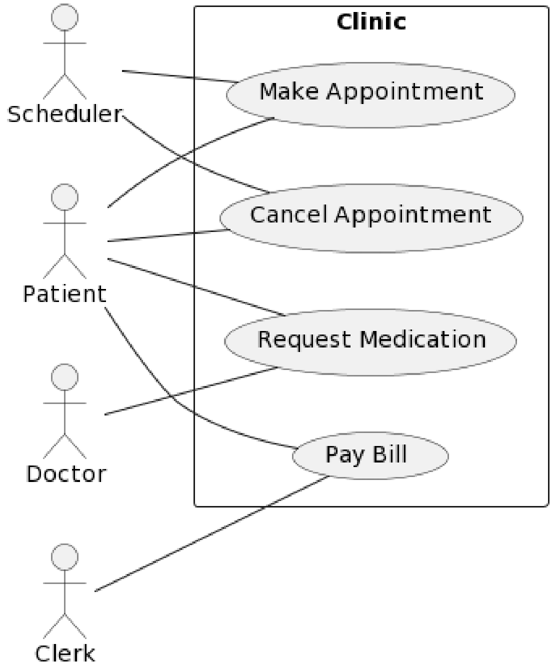
▪ **Use Case Diagram:** a collection of actors, use cases, and their associations that describes what a system does from the standpoint of an external observer.

  - It presents the users of the system and their interactions with the system.
  - Show high-level overview of relationship between use cases, actors, and the system.
  - Does not provide a lot of details.

# Use Case Diagram (cont.)

- **Discussion**:
  - What do you see in this diagram?
  - What are the elements in this diagram?
  - What message(s) this diagram may try to deliver?

**Clinic management system**

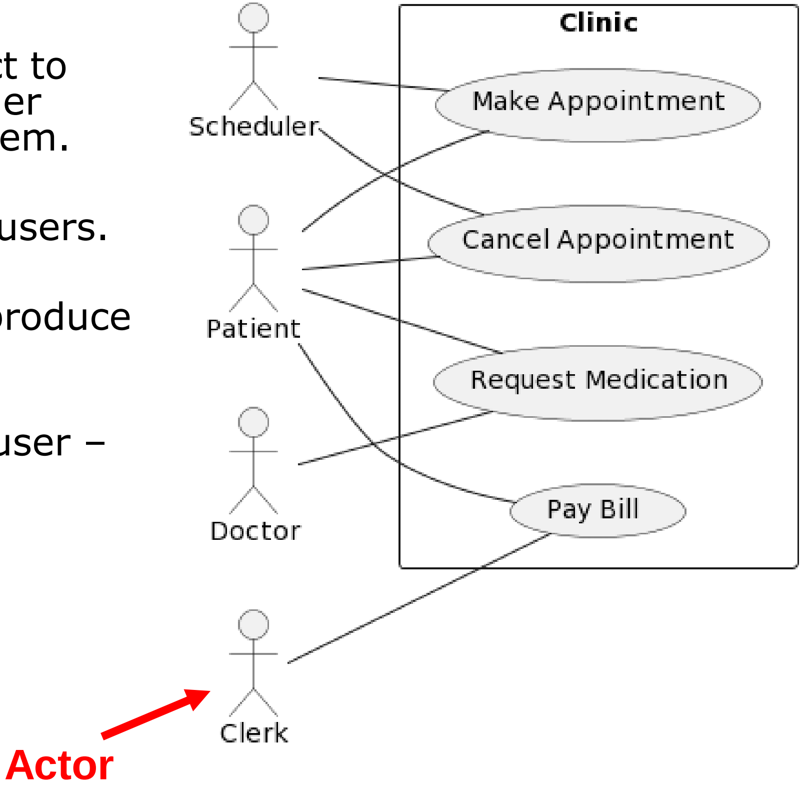


Think → Pair → Share

# Use Case Diagram (cont.)

- Common elements in the Use Case Diagram:

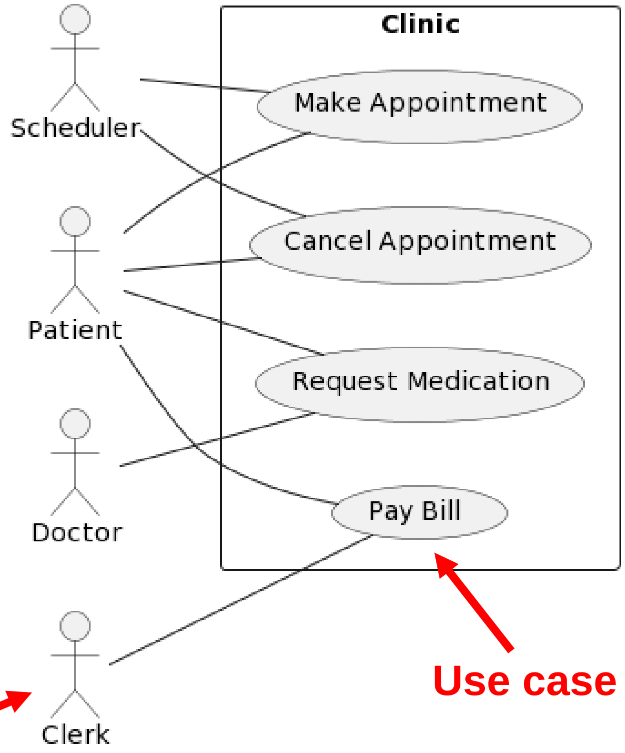
  - **Actor**: a role that a user plays with respect to the system. Actor could be a user or another system that interacts with the current system.
    - Stick figures that represents external users.
    - Actors must be external objects that produce or consume data.
    - Actor is different from the concept of user – a user can act as different actors.



# Use Case Diagram (cont.)

- Common elements in the Use Case Diagram:

  - **Use case:** is a summary of scenarios that describes the typical interaction between the users of a system and the system itself.
    - Horizontally shaped ovals
    - Represent different uses/interactions that a user might have.
    - Typically represents system function.



Use case

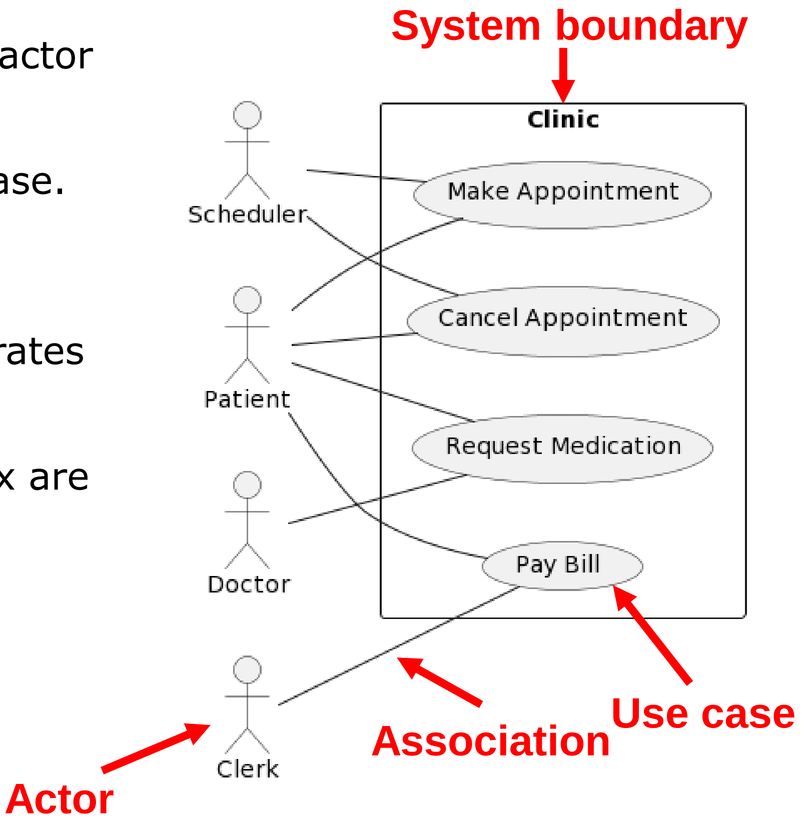
Actor

# Use Case Diagram (cont.)

- Common elements in the Use Case Diagram:

  - **Association**: communication between an actor and a use case.
    - A solid line between actor and user case. **[No arrow!]**

  - **System boundary:** a rectangle that separates the system from the external actors.
    - All use cases outside the boundary box are outside the scope of the system.
    - For large and complex systems, each module may be the system boundary.

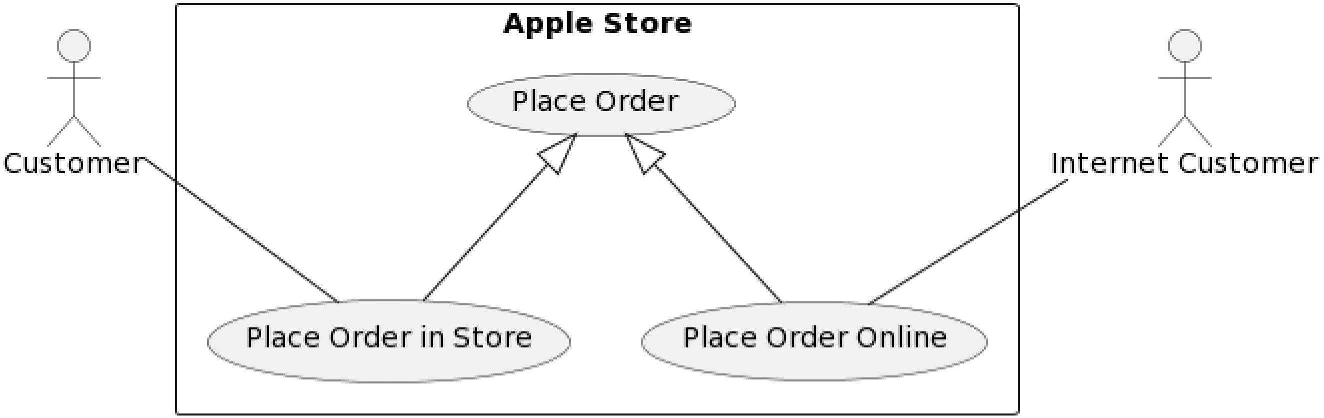


# Use Case Diagram (cont.)

- Use Case Relationship:

  - **Generalization**: indicates one use case is a special kind of another.

    - Represented by a directed arrow with a triangle arrowhead.
    - Indicates a **parent-child relationship** between use cases.
    - The child use case is connected at the base of the arrow, while the tip of the arrow is connected to the parent use case.
    - Generalization is used when we find **two or more use cases that have commonalities** in behavior, structure, and purpose.

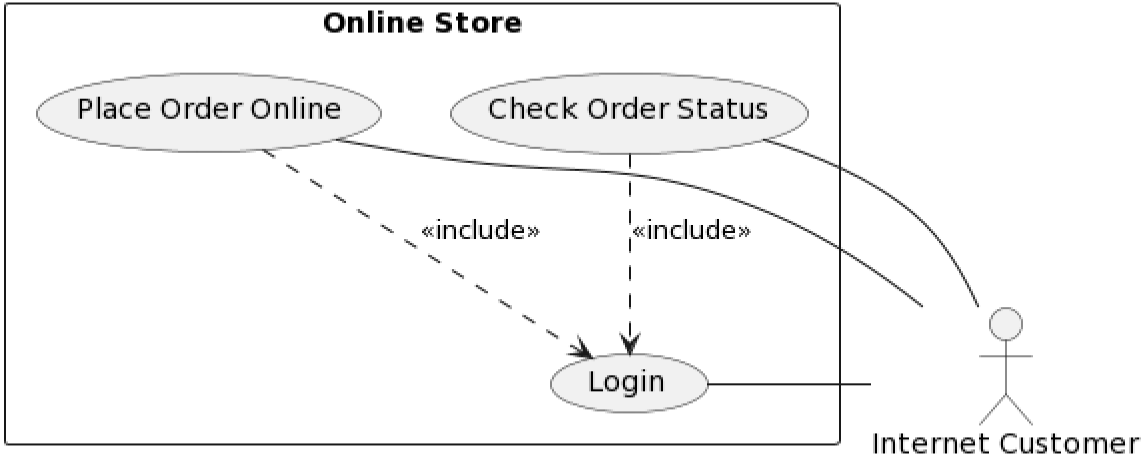


# Use Case Diagram (cont.)

- Use Case Relationship:

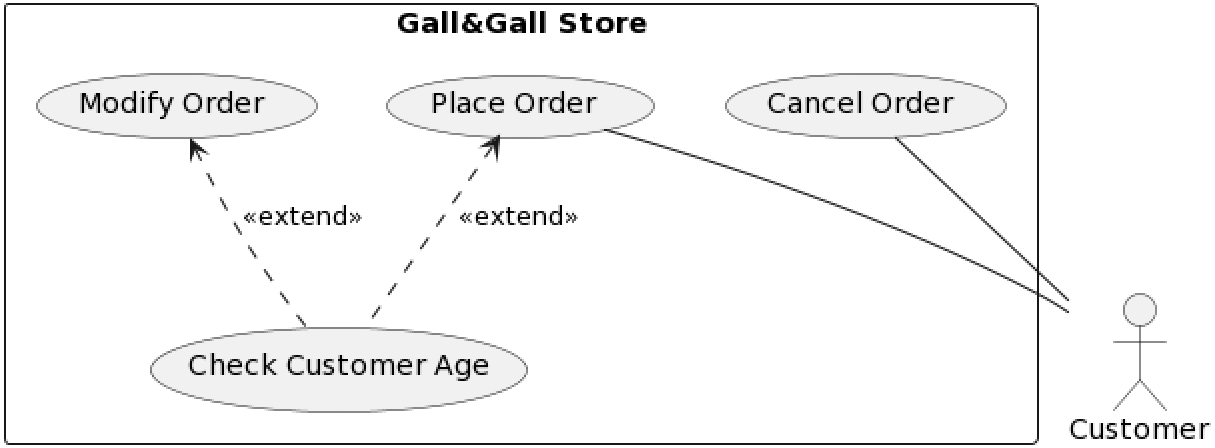
  - **Include**: indicates one use case (the base use case) is using the functionality of another use case (the inclusion use case).

    - Represented by a directed arrow with dotted line.
    - The stereotype "<<include>>" identifies the include relationship, where the base use case includes the functionality of the inclusion use case.
    - Include relation is used to support the reuse of functionality in a use-case model.



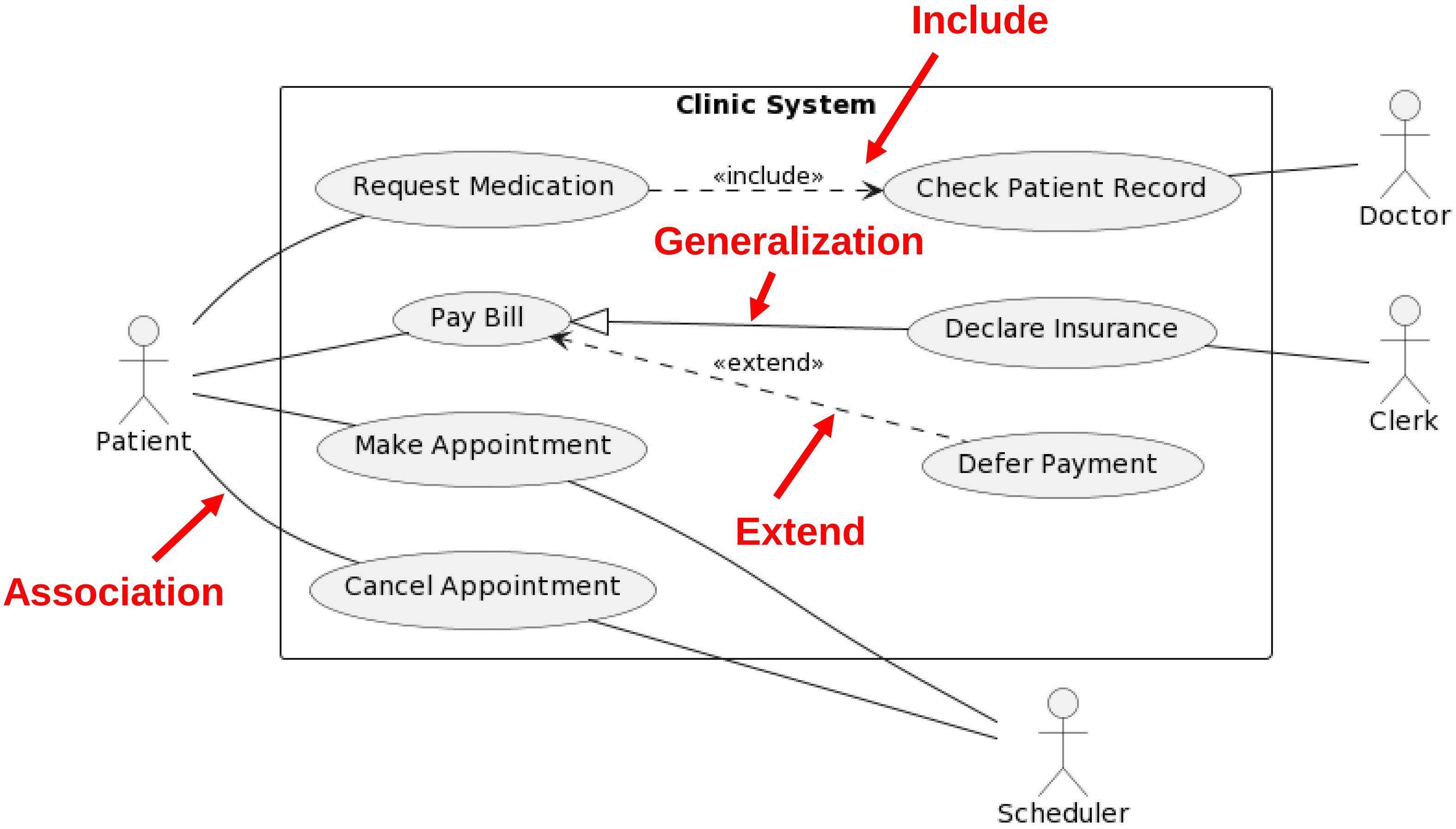
# Use Case Diagram (cont.)

- Use Case Relationship:
  - **Extend**: specify that one use case (extension) extends the behavior of another use case (base).
    - Represented by a directed arrow with dotted line. The stereotype “<<extend>>” identifies the extend relationship.
    - The extension owns the extend relationship, and we can specify several extend relationships for a single base use case.
    - We use extend relationship to show:
      - A use case is an optional system behavior.
      - A use case is executed only under certain conditions.



# Use Case Diagram (cont.)

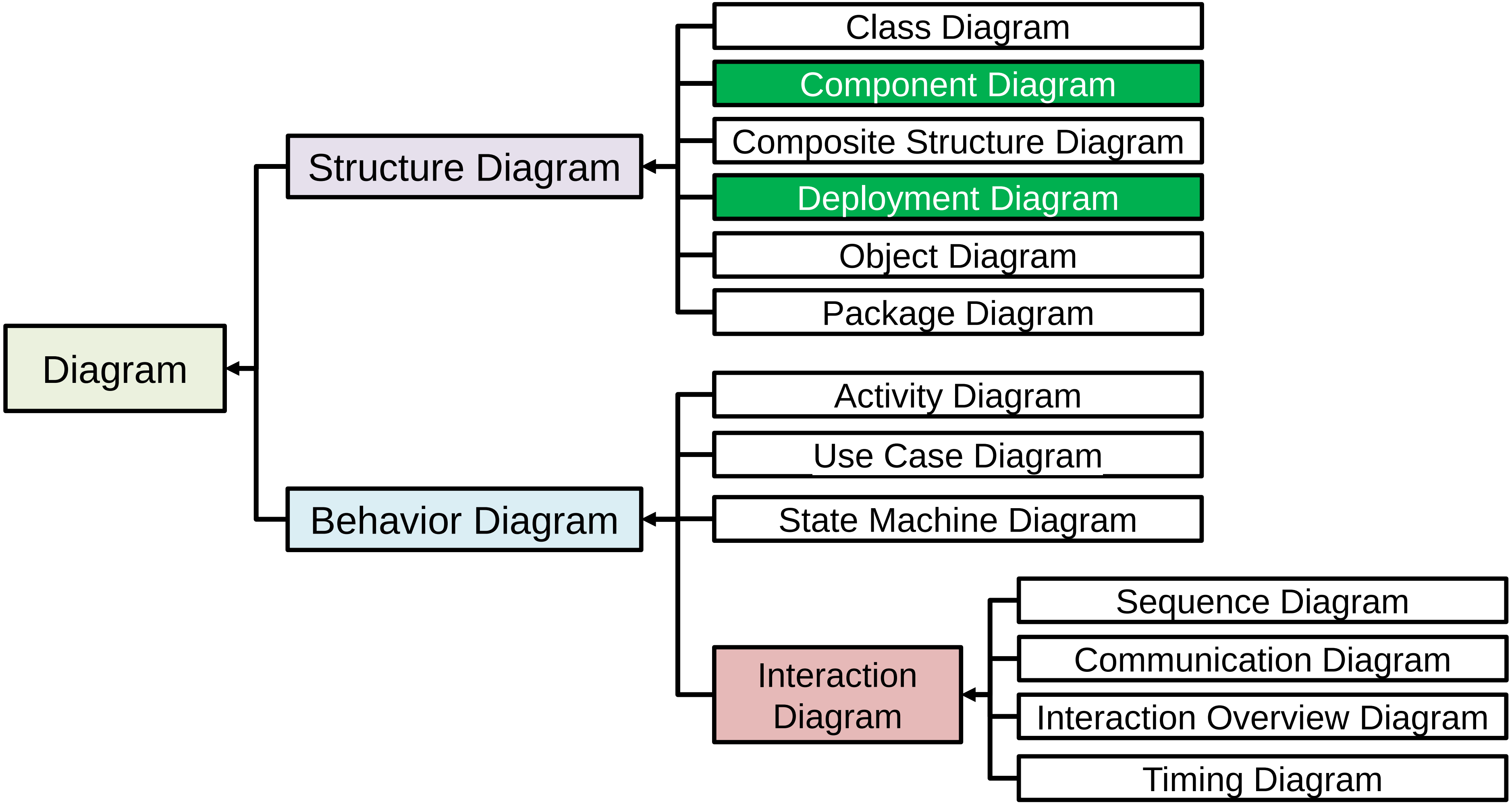
- An overall example:



# Use Case Diagram (cont.)

- When to use the Use Case Diagram?
  - To represent the system-user interactions.
  - To define and organize the functional requirements of a system.
  - Is typically used in the early phase in system design.

# Component and Deployment Diagrams



# Component Diagram

- What is the Component Diagram?

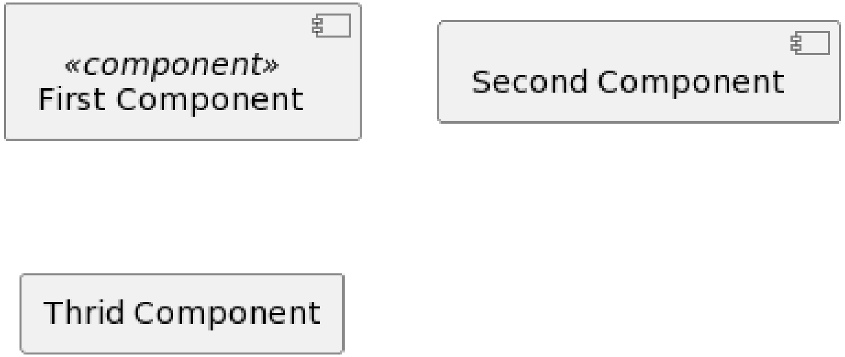
  ■ **Component Diagram:** divides a complex system into multiple components and shows the inter-relationships between the components.

  ■ The term ‘**component**’: a module of classes that represents independent system or subsystem with the ability to interface with the rest of a more complex system.

  – Component diagram is useful to:
    - Show the system’s physical structure.
    - Show the system’s static components and their relations .

# Component Diagram (cont.)

- Common elements in the diagram:
  - **Component:** represents a modular part of a system that encapsulates its contents. It can be represented by different ways:
    - A rectangle with the stereotype <<component>> and/or icon.
    - A rectangle with the component icon.
    - A rectangle with the name of the component.

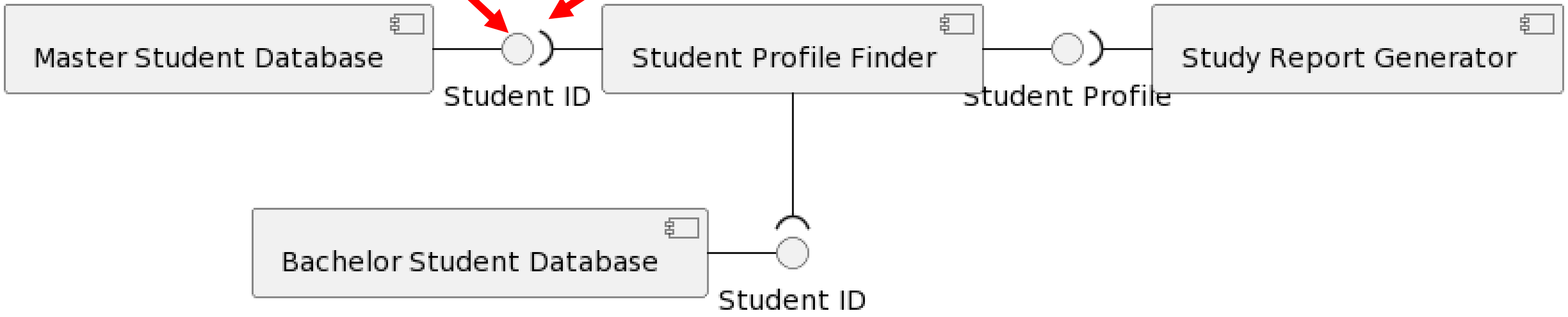


# Component Diagram (cont.)

- Common elements in the Component Diagram:
  - **Assembly:**
    - **Provided interface**: symbols with a complete circle at the end represent an interface
    - **Required interface**: symbols with a half circle at the end represent an interface that the component requires.

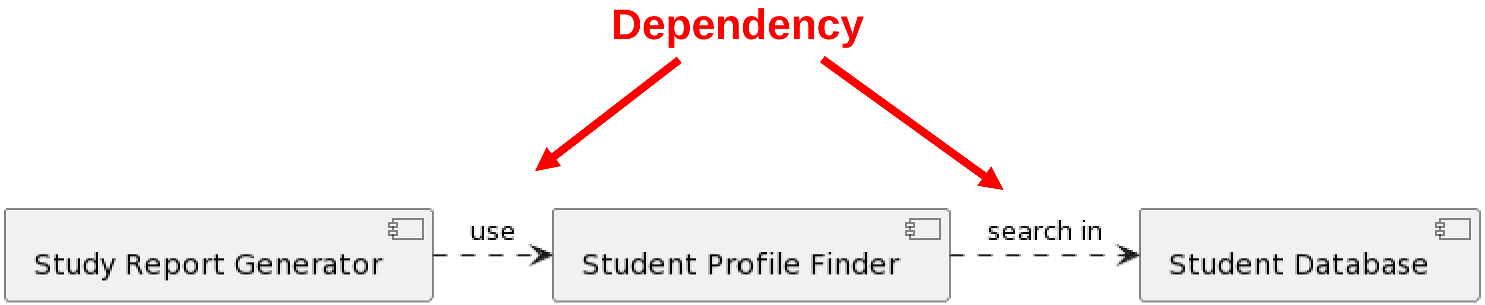
Provided Interface

Required interface



# Component Diagram (cont.)

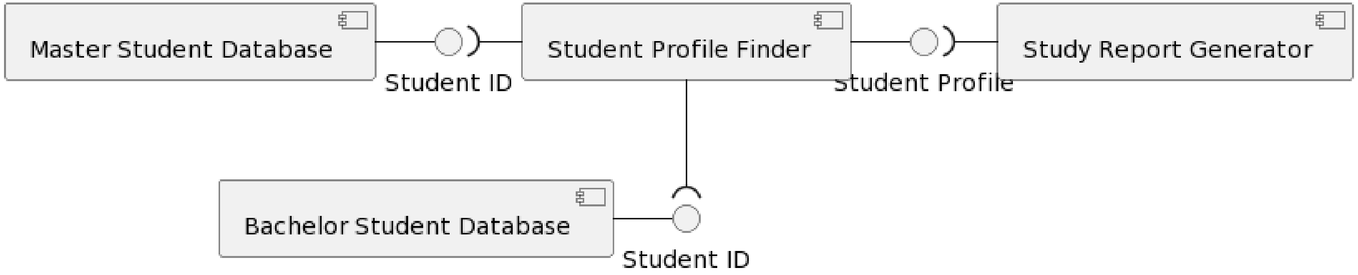
- Common elements in the Component Diagram:
  - **Dependency:**
    - Indicates that the functioning of one element depends on the existence of another element. (Thinking about the *#include* statement)



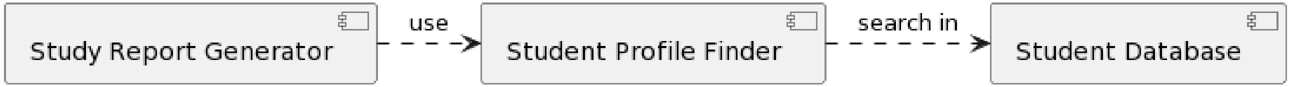
# Component Diagram (cont.)

- **Discussion**:
  - What are the differences between the following two diagrams?
  - What are the differences between the use of assembly and dependency?

**Assembly**



**Dependency**



Think → Pair → Share

# Component Diagram (cont.)

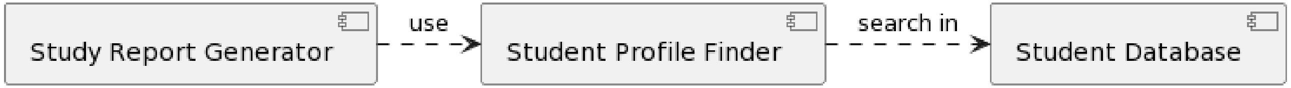
- Differences between dependency and assembly:
  - **Dependency:**
    - Is a *classifier-level* relation between components
  - **Assembly:**
    - Is an *instance-level* relation between two instances of a class (object) that established in the run-time of the system.

- Dependency between two components on the classifier level expresses a potential assembly relationship between the two corresponding instances in system run-time.

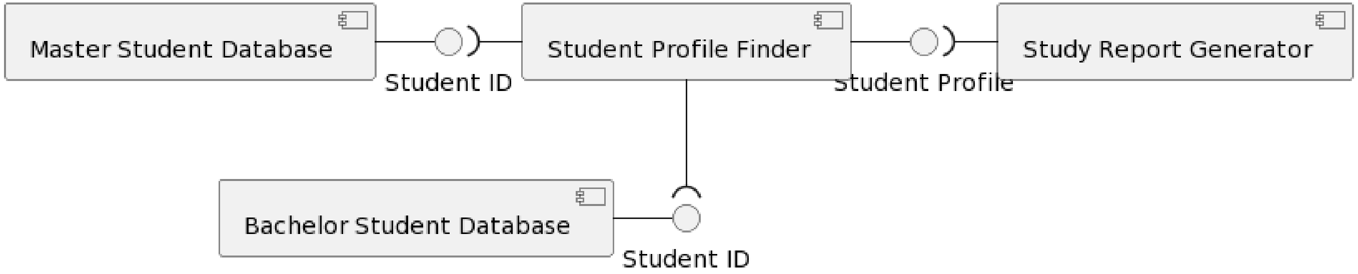
# Component Diagram (cont.)

- Differences between dependency and assembly:
  - Dependency between two components on the classifier level expresses a potential assembly relationship between the two corresponding instances in system run-time.
  - They are modeling the system at different abstraction

**Dependency**

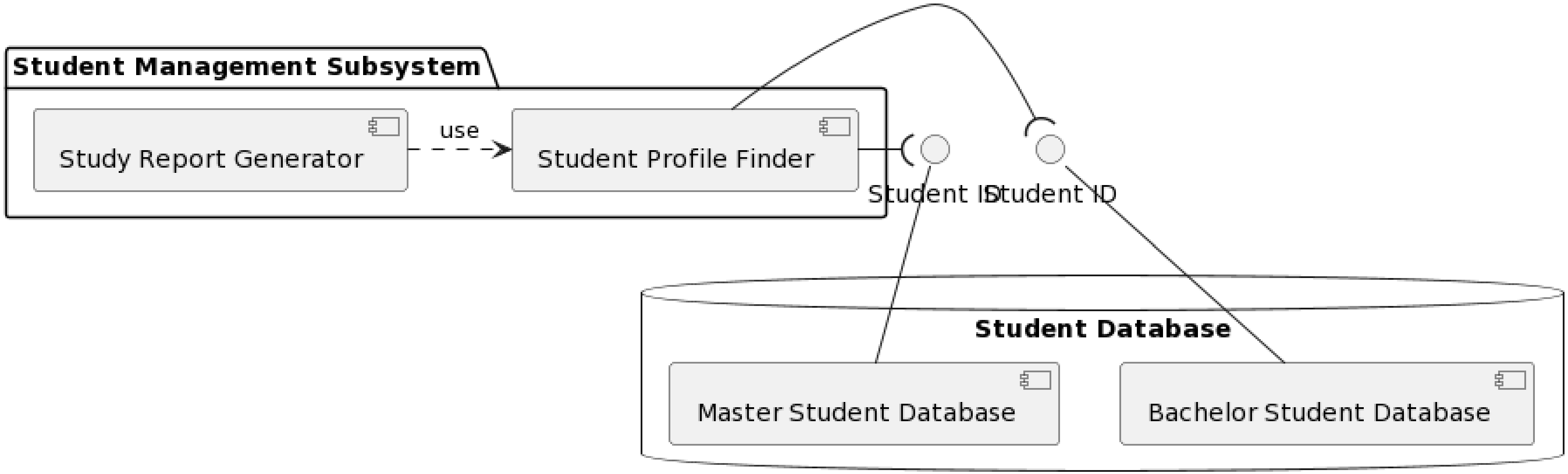


**Assembly**



# Component Diagram (cont.)

- Common elements in the Component Diagram:
  - **Group and package:**



# Deployment Diagram

- What is the Deployment Diagram?

  ▪ **Deployment Diagram:** a type of structural diagram that shows a system’s physical layout, revealing which pieces of software run on what pieces of hardware.

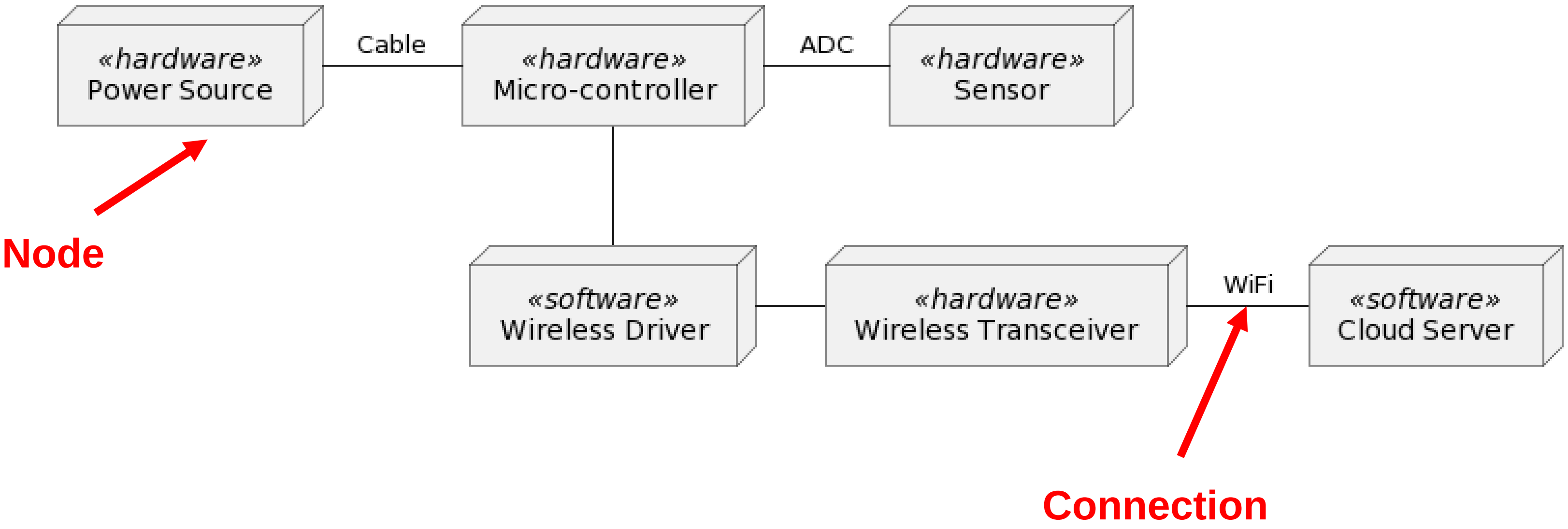
  – It shows the physical deployment of the software elements.

  – It illustrates the runtime processing for hardware.

  – It provides the topology of the hardware system.

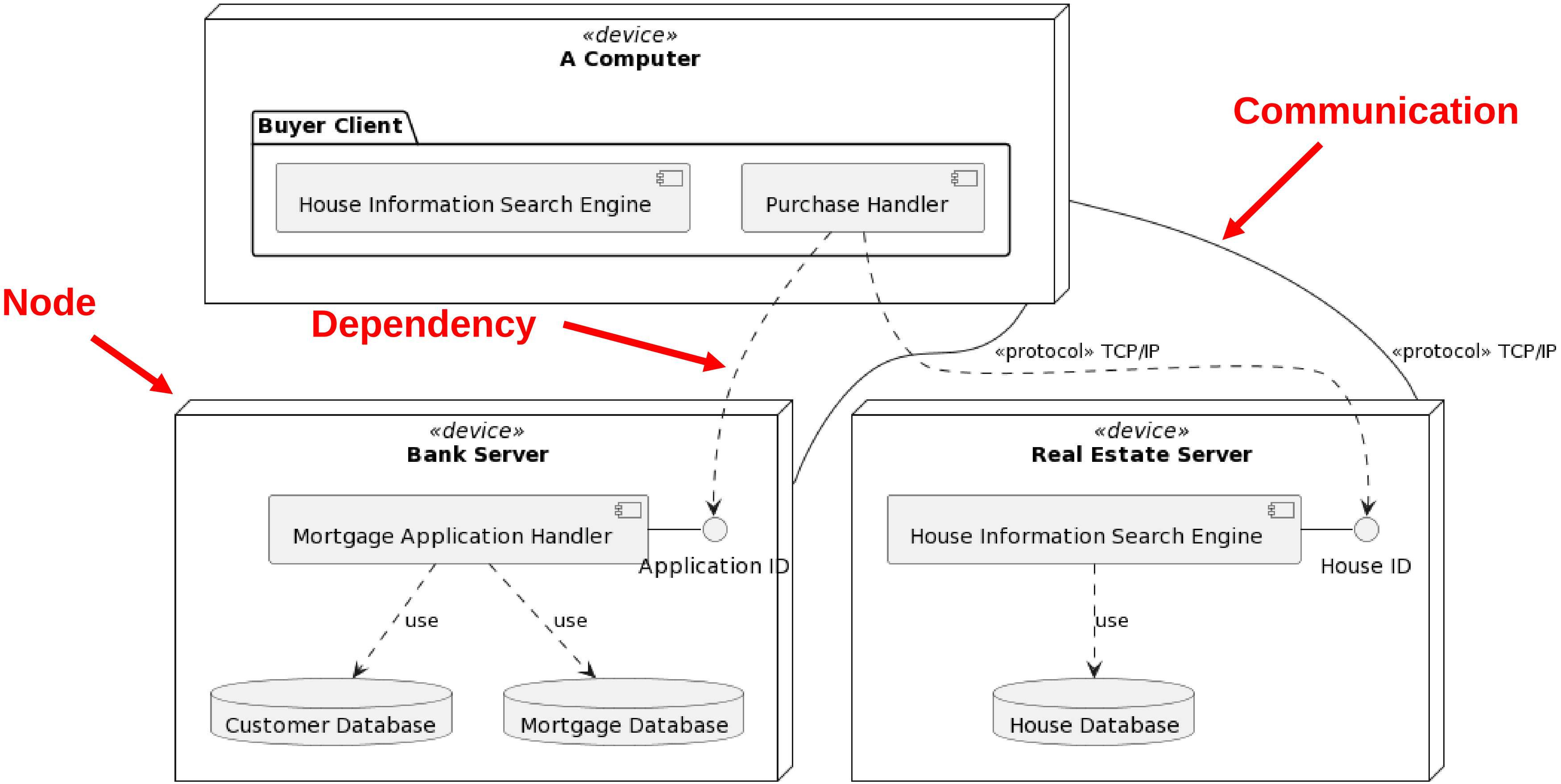
# Deployment Diagram (cont.)

- Modeling a wireless sensor node:



# Deployment Diagram (cont.)

- Another example:



# Closing remarks

- In the Lab session:
  - Download and install PlantUML;
  - Go over the tutorial for the use case and component diagrams:
    - URL: https://software-fundamentals.pages.ewi.tudelft.nl/software-systems/website/part-2/Tutorials/Summary.html
  - Get familiar with the system mentioned in the modeling assignments;
  - Work on the component diagram for the modeling assignment.