# Announcements

- Design Assignment Out today
  - Architecture Design; Class Diagram; Sequence Diagram; Component Diagram

- GCP Assignment- Due Oct 5

- Mid term Feedback Survey- Due Oct 5

CS3300 Introduction to Software Engineering

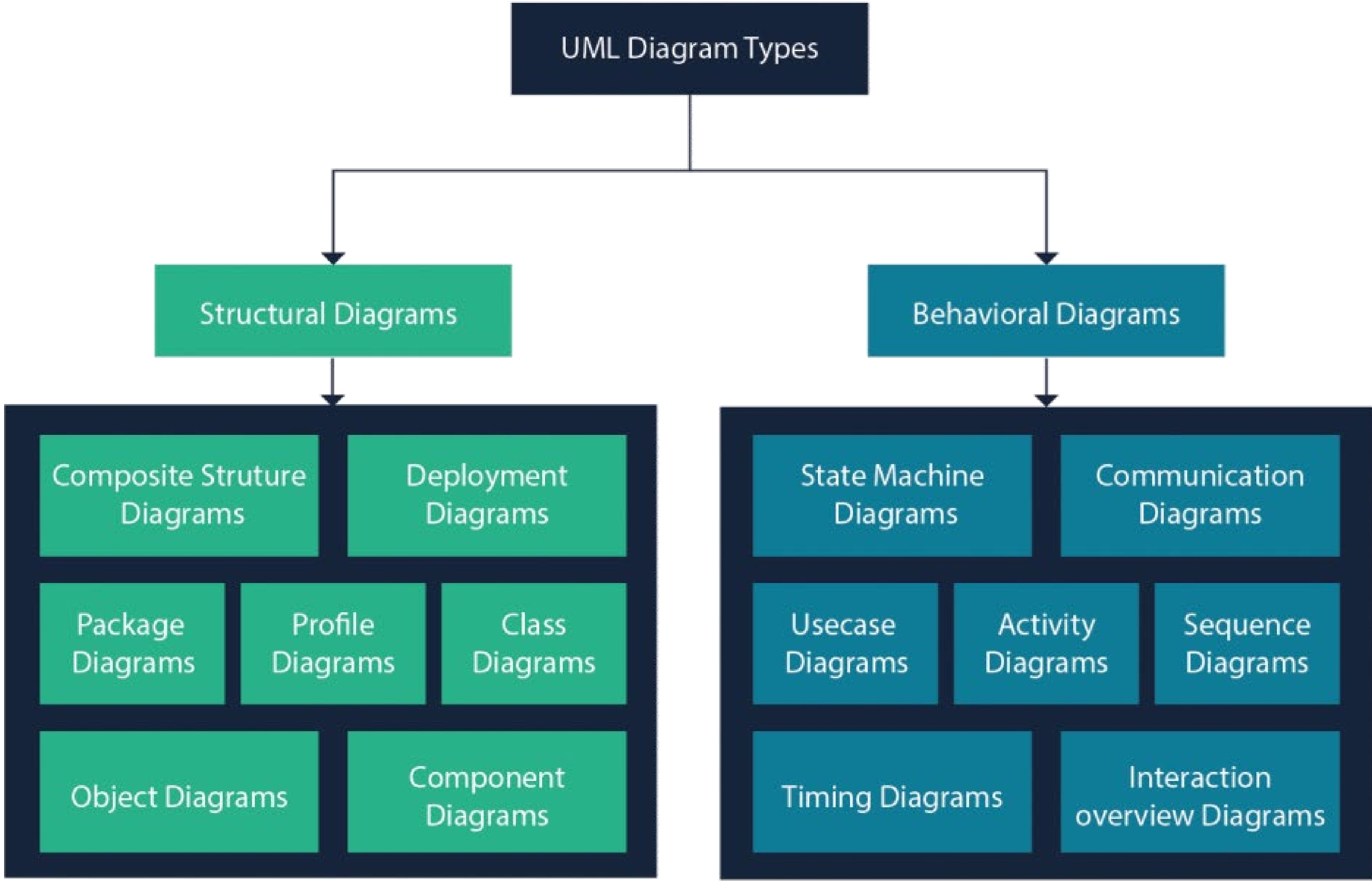
# Lecture 11: Software Design: Unified Modeling Language

Nimisha Roy ▸ nroy9@gatech.edu

Slides adapted from Alessandro Orso

# Unified Modeling Language

Intended to provide a standard way to visualize the design of a system.



UML Diagram Types

Structural Diagrams

Composite Struture Diagrams

Deployment Diagrams

Package Diagrams

Profile Diagrams

Class Diagrams

Object Diagrams

Component Diagrams

Behavioral Diagrams

State Machine Diagrams

Communication Diagrams

Usecase Diagrams

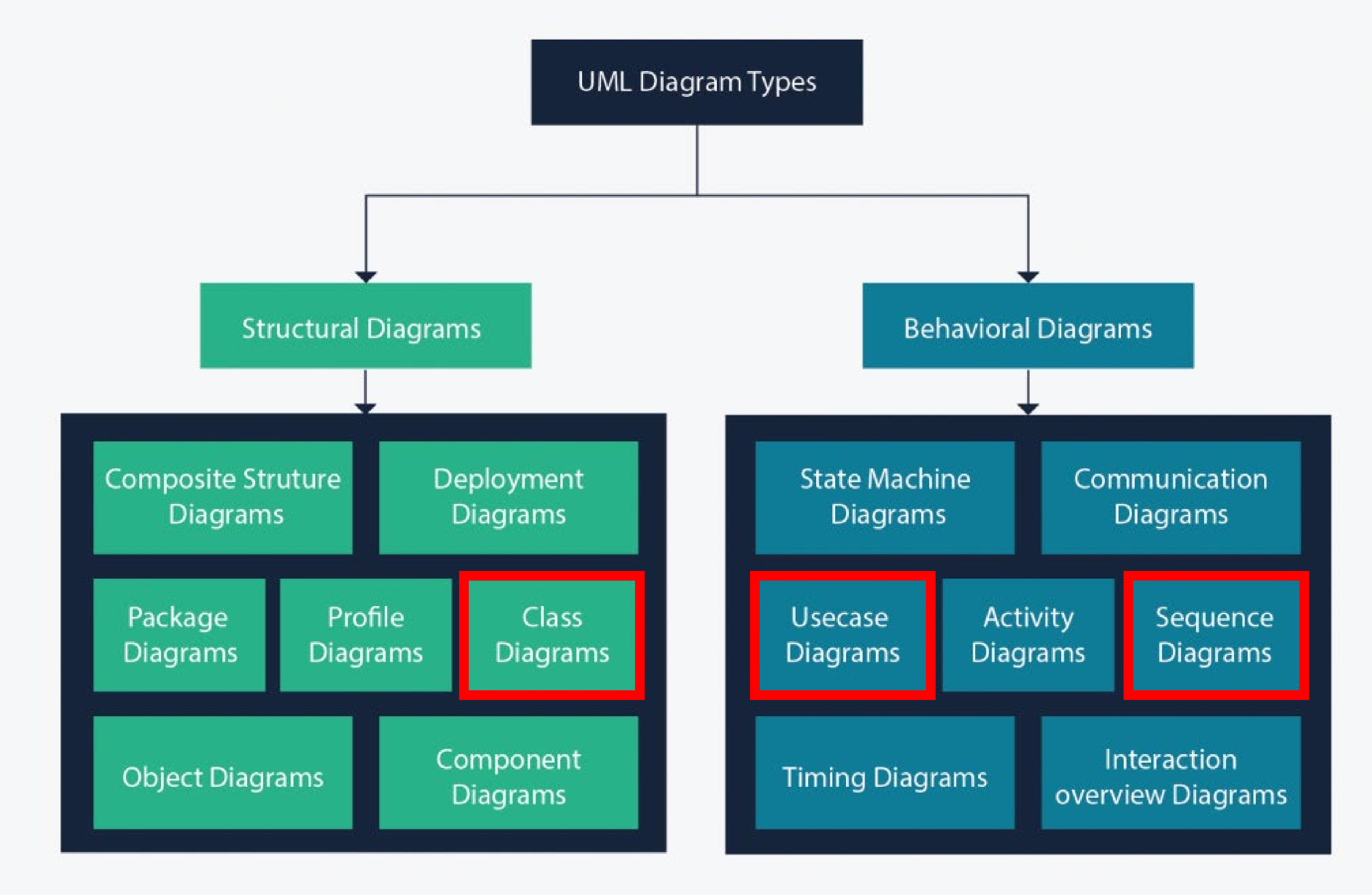
Activity Diagrams

Sequence Diagrams

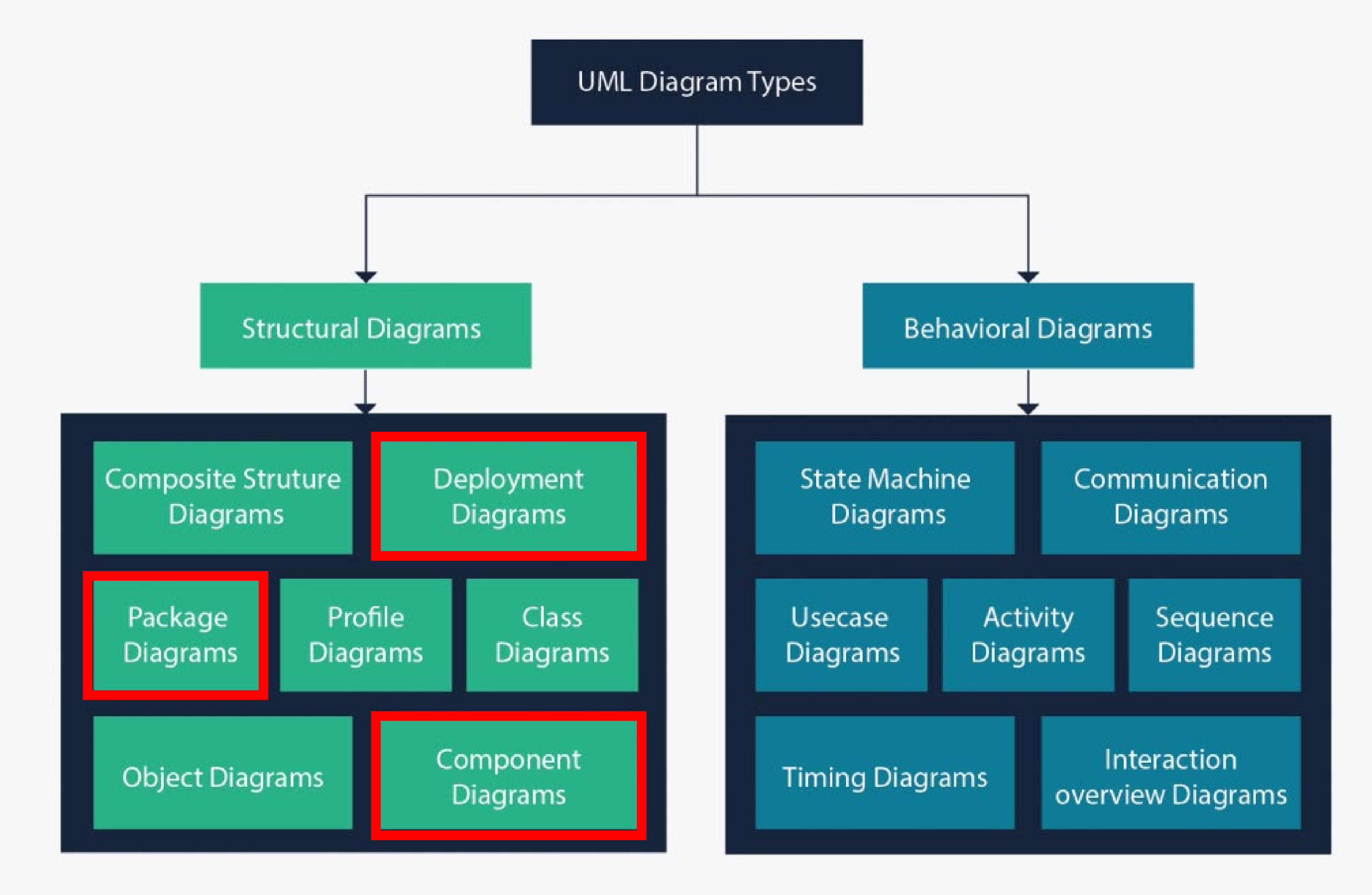
Timing Diagrams

Interaction overview Diagrams

# Unified Modeling Language: You may already know…



# Unified Modeling Language: Architectural View



UML Diagram Types

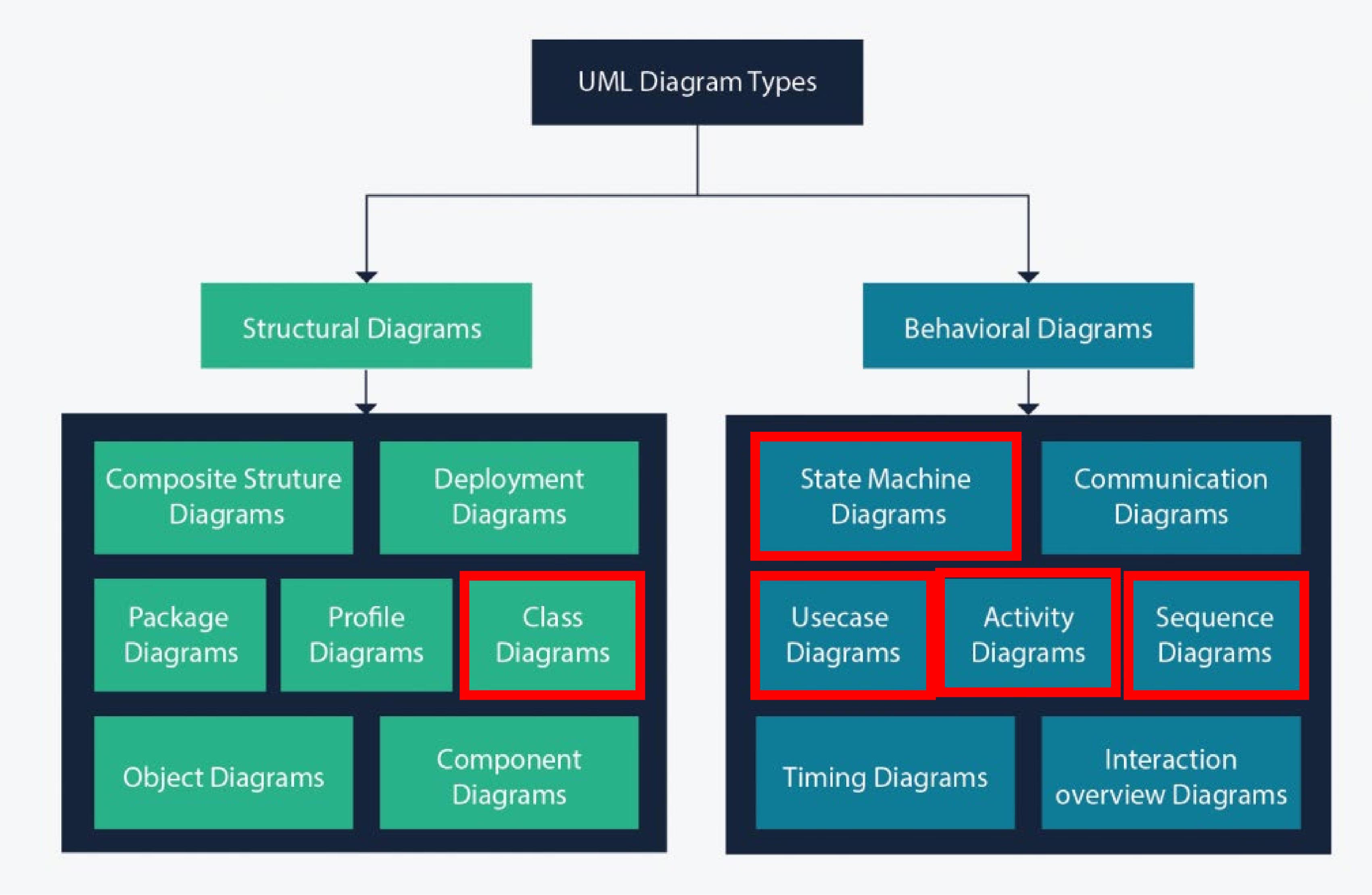
Structural Diagrams

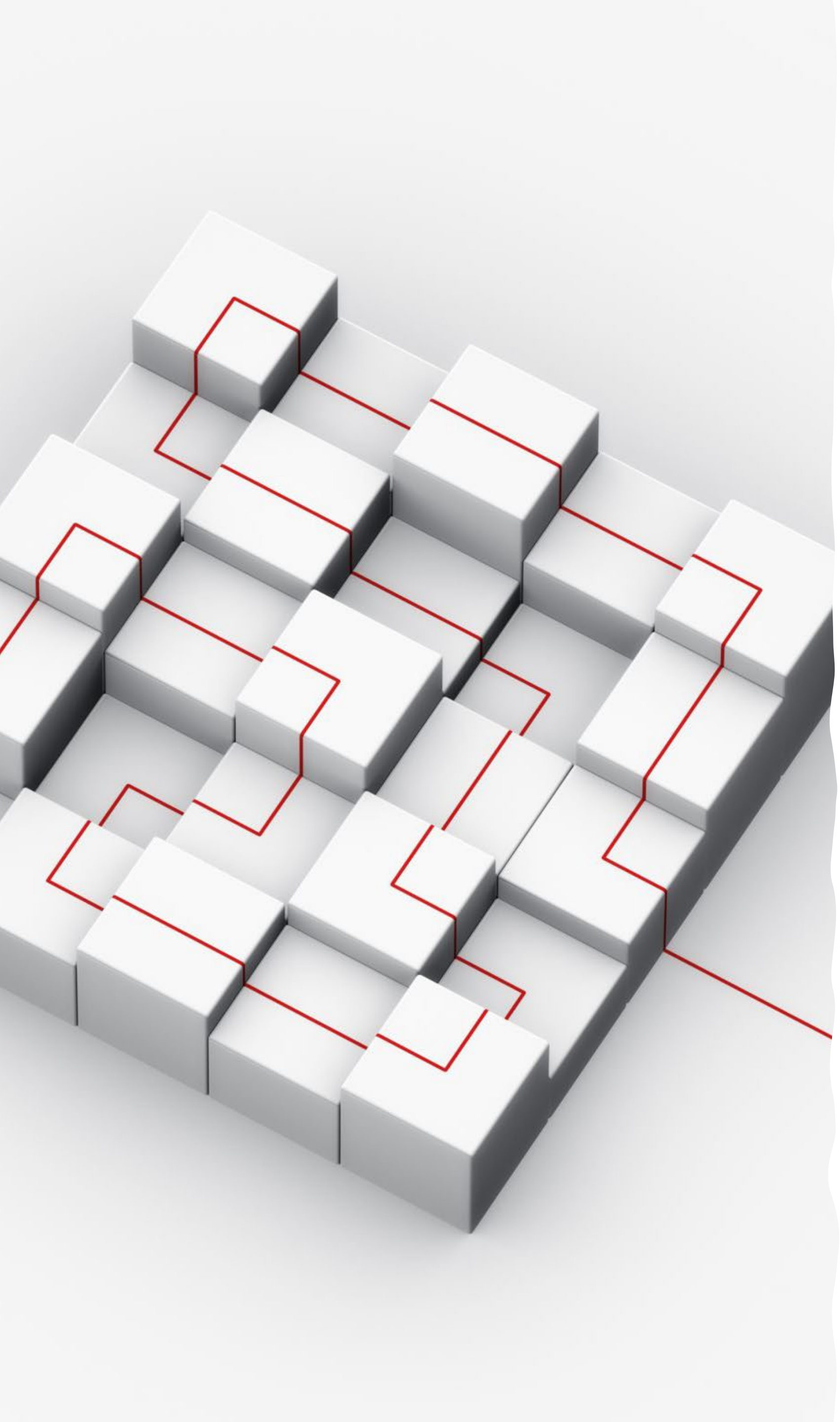
- Composite Struture Diagrams
- Deployment Diagrams
- Package Diagrams
- Profile Diagrams
- Class Diagrams
- Object Diagrams
- Component Diagrams

Behavioral Diagrams

- State Machine Diagrams
- Communication Diagrams
- Usecase Diagrams
- Activity Diagrams
- Sequence Diagrams
- Timing Diagrams
- Interaction overview Diagrams

# Unified Modeling Language: Low Level Design View





# Architectural View – Structural Diagram: Component

# Component Diagram

Static view of components and their relationships

Node = Component
Set of Classes with a well-defined interface

Edge = Relationship
“Uses services of”

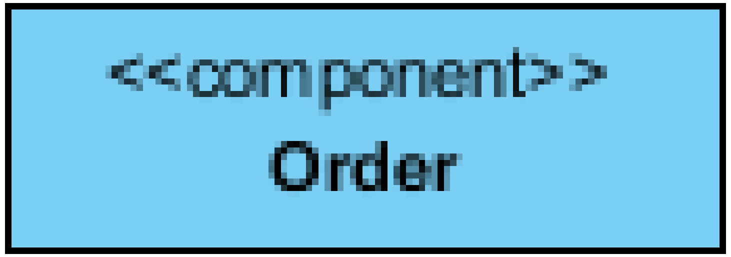
Can be used to represent a software architecture

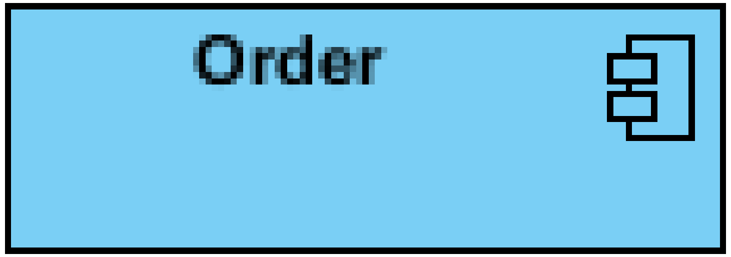
Slide adapted from Alessandro Orso

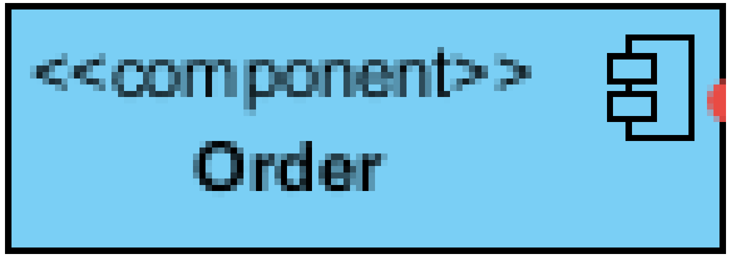
# Component Diagram: Component

Components represented as

1. A rectangle with the component's name
2. A rectangle with the component icon
3. A rectangle with the stereotype text and/or icon



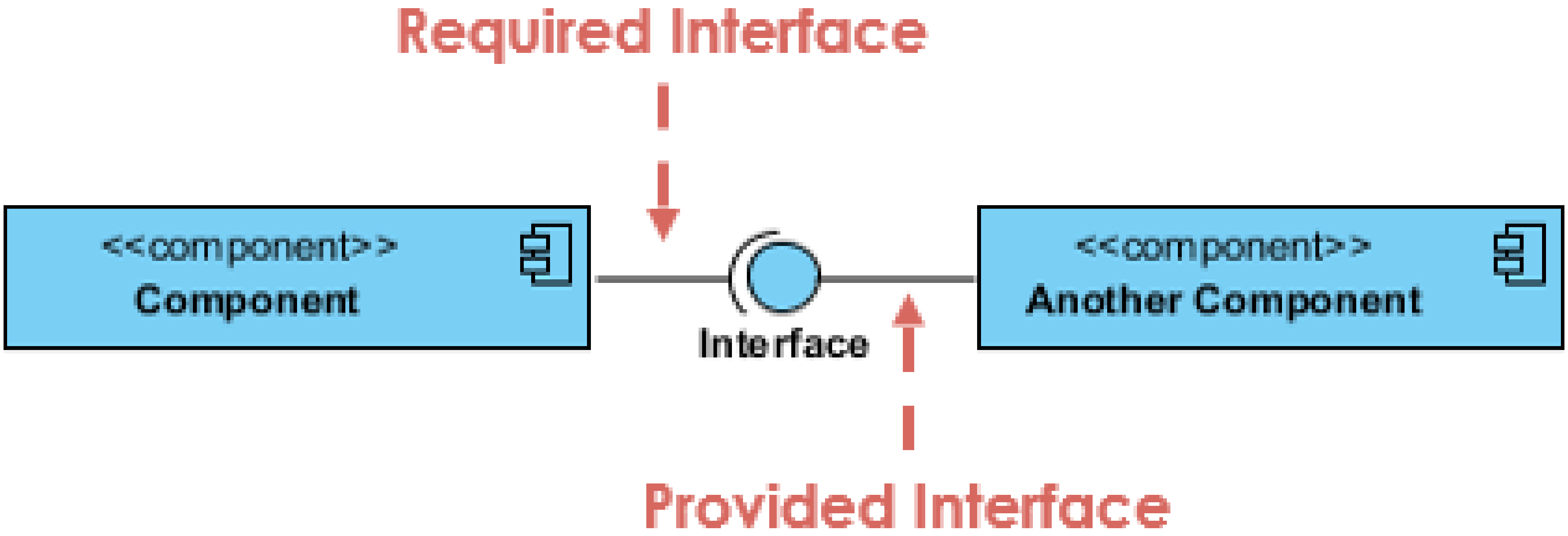




https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/#:~:text=a%20component%20itself).-,Basic%20Concepts%20of%20Component%20Diagram,with%20optional%20compartments%20stacked%20vertically

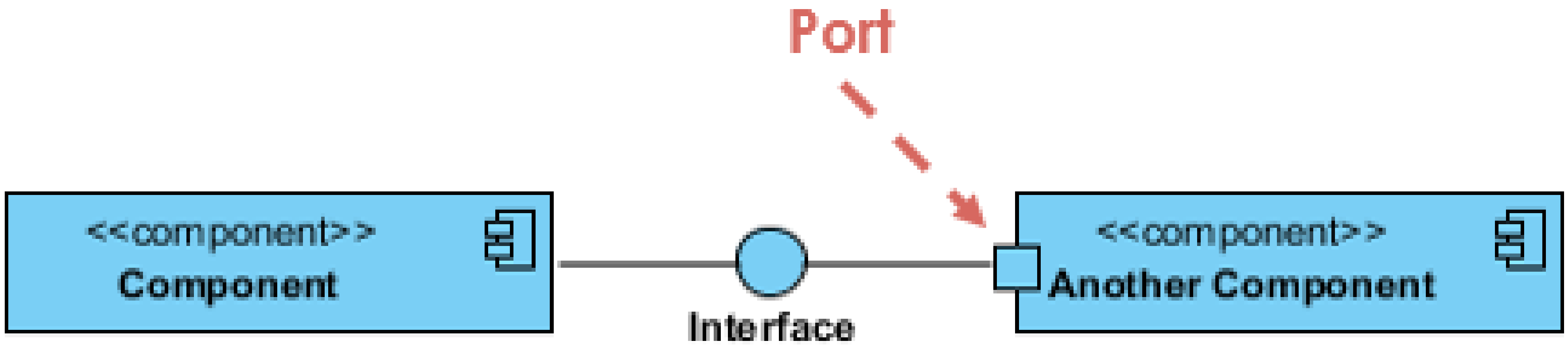
# Component Diagram: Interface

**Provided interface** symbols with a complete circle at their end represent an interface that the component provides - "lollipop" symbol
**Required Interface** symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires.



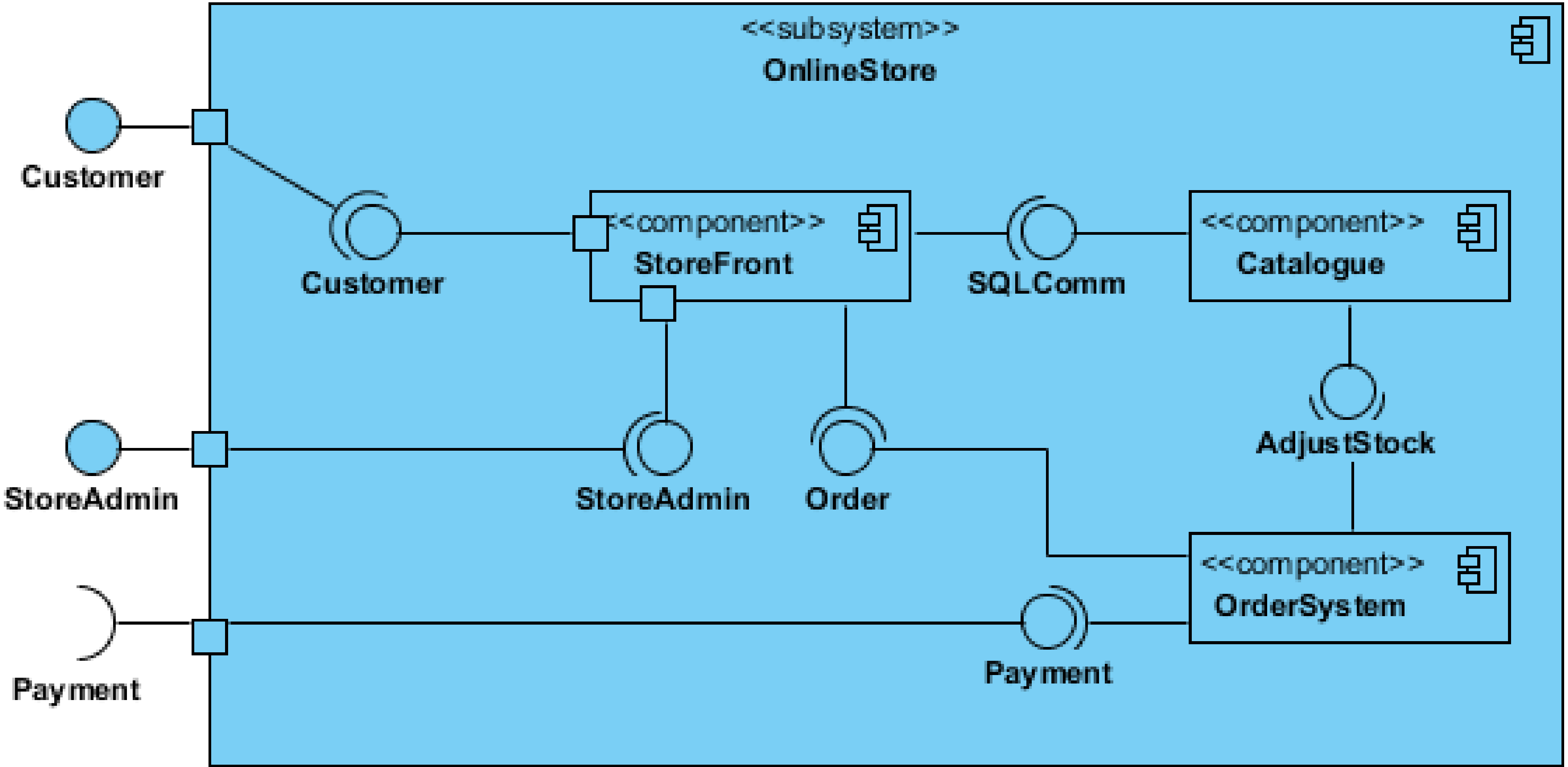
# Component Diagram: Port

Ports are represented using a square along the edge of the system or a component. A port is often used to help expose required and provided interfaces of a component.

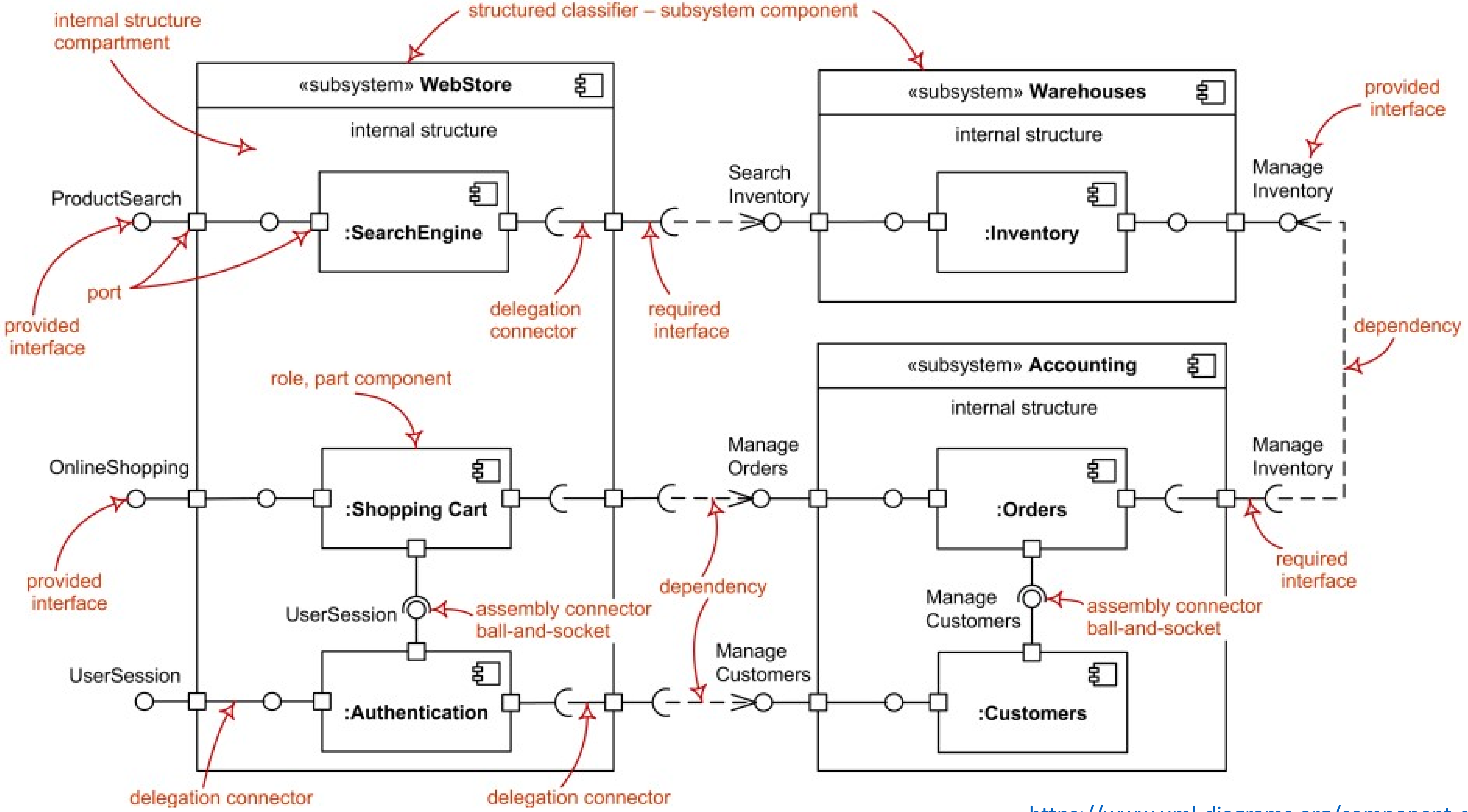


# Component Diagram: Subsystem

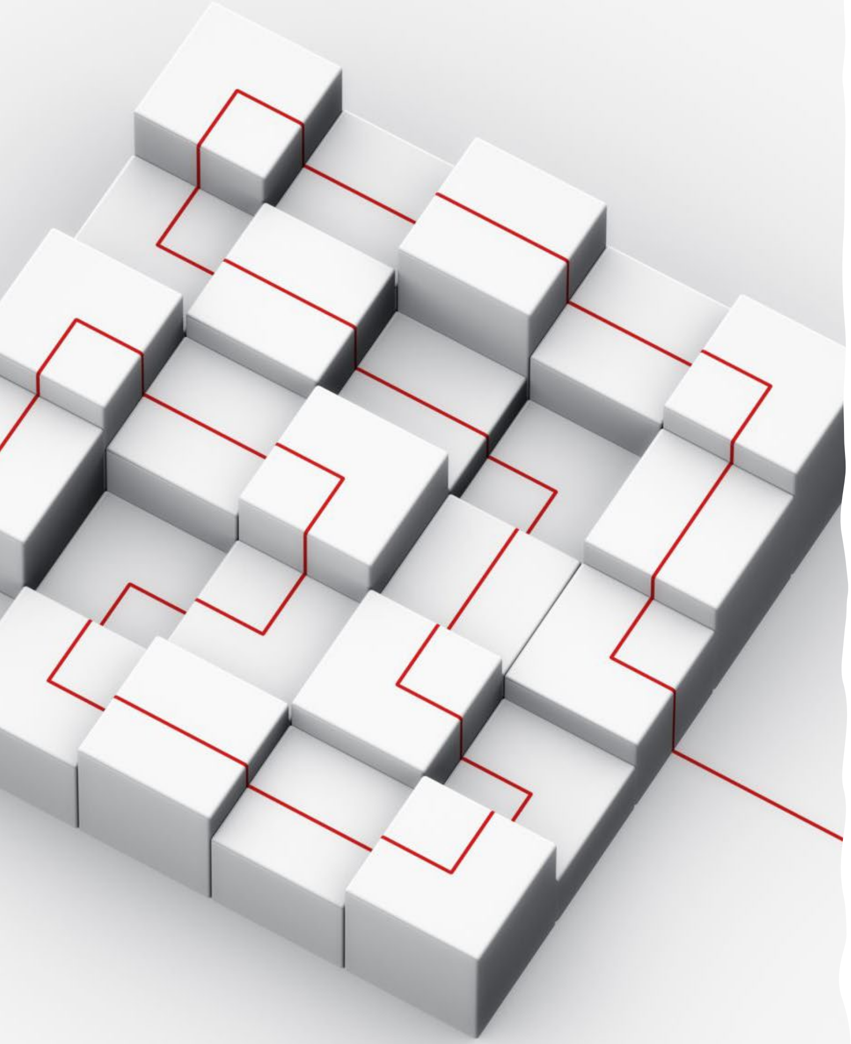
The subsystem classifier is a specialized version of a component classifier. The only difference is that a subsystem notation element has the keyword of subsystem instead of component.



# Component Diagram Example



https://www.uml-diagrams.org/component-diagrams.html



# Architectural View – Structural Diagram: Deployment

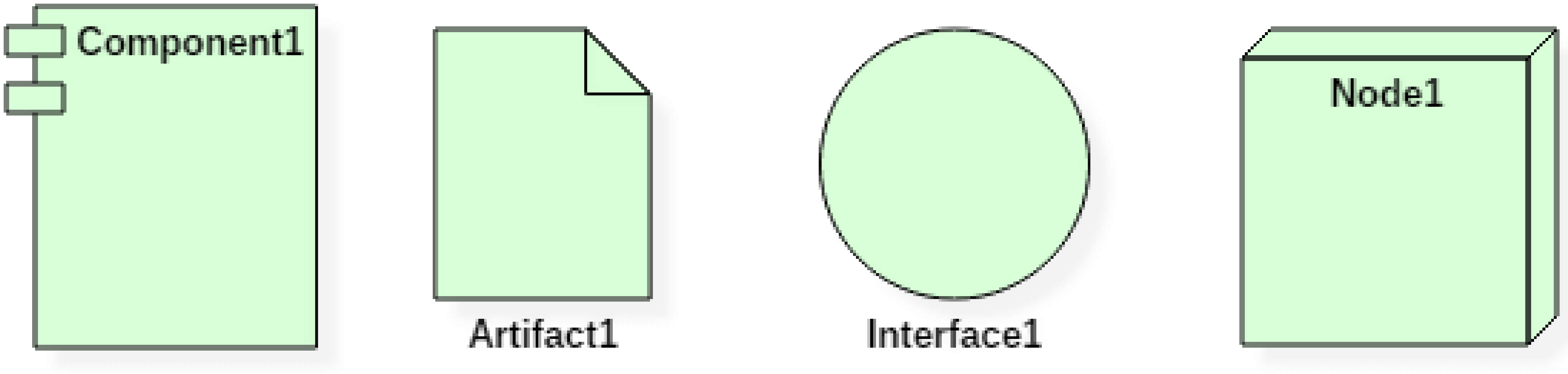
# Deployment Diagram

Specifies the physical hardware on which the software system will execute. It also determines how the software is deployed on the underlying hardware.

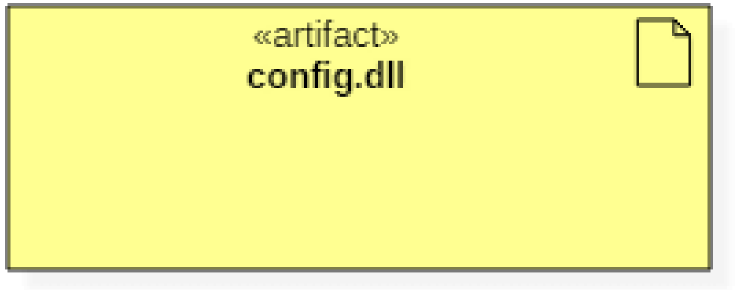
Physical Allocation of components to computational units

Maps the software architecture created in design to the physical system architecture that executes it. In distributed systems, it models the distribution of the software across the physical nodes.

# Deployment Diagram: Notations



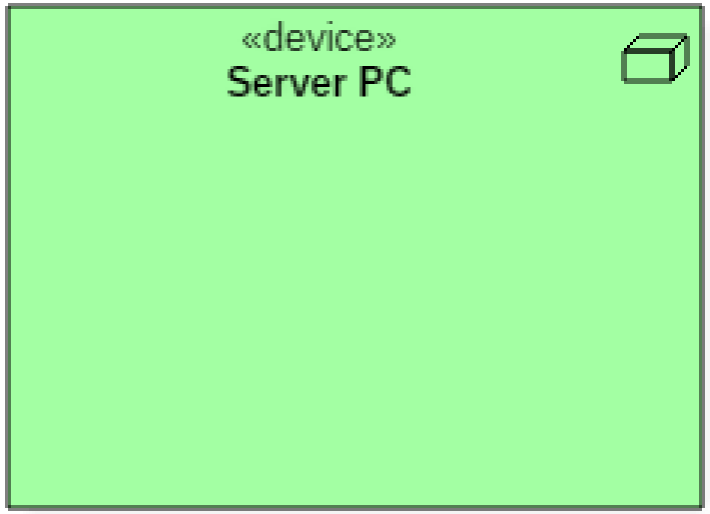
# Deployment Diagram: Artifact

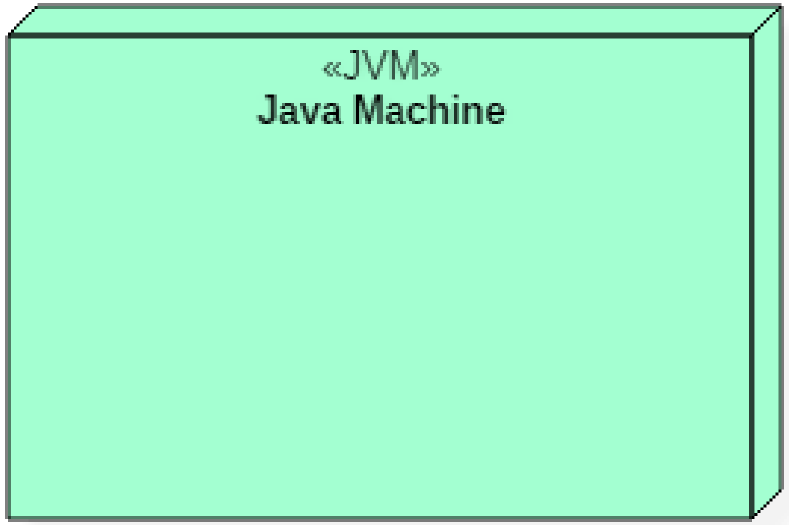


- Represent physical entities that are used or produced in a software development process
- Artifacts are deployed on the nodes. The most common artifacts are as follows:
  - Source files
  - Executable files
  - Database tables
  - Scripts
  - DLL files
  - User manuals or documentation
  - Output files
- Artifacts are labeled with the stereotype **<<artifact>>,** and it may have an artifact icon on the top right corner.
- Each artifact has a filename in its specification

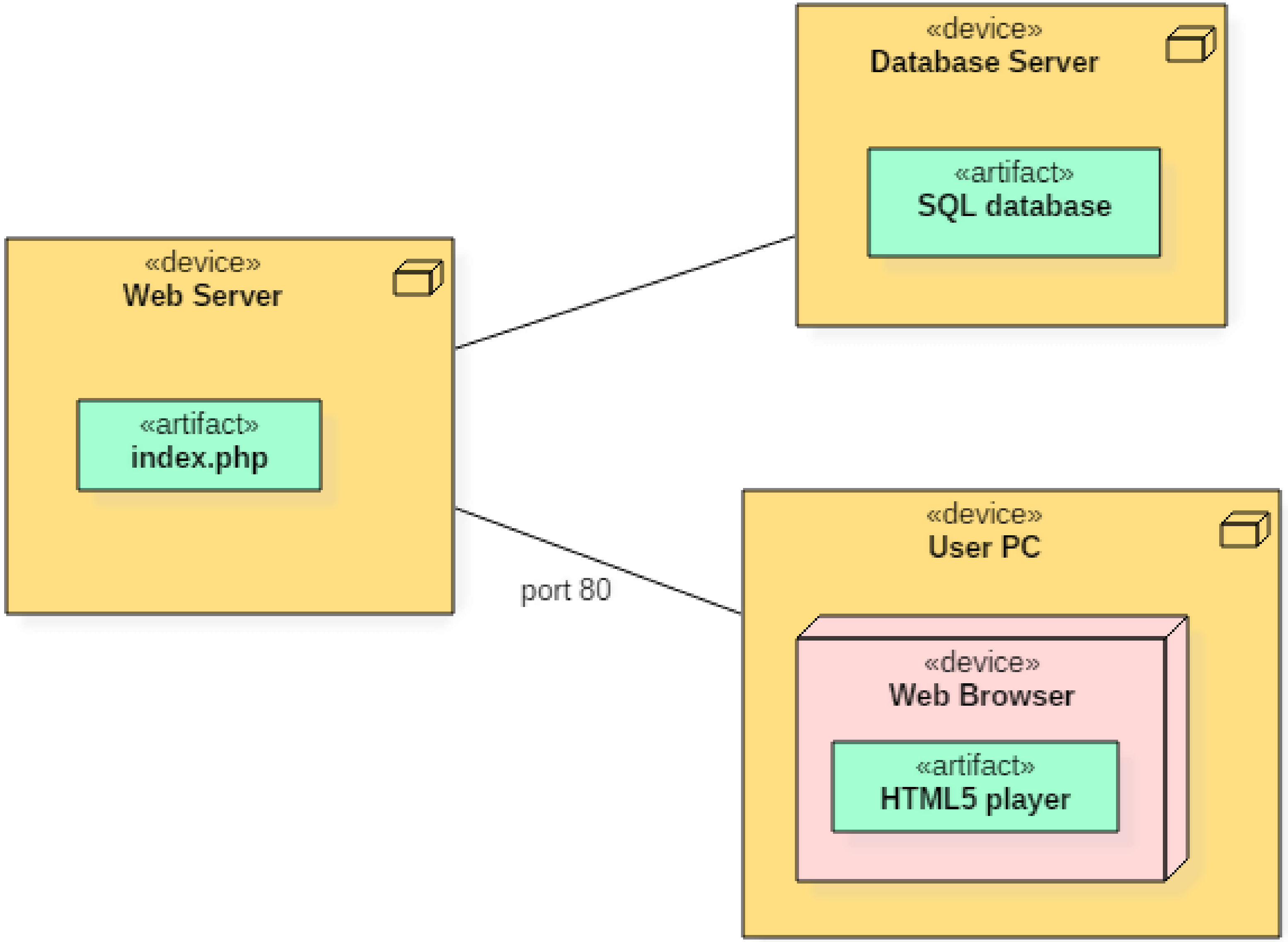
# Deployment Diagram: Node

- Node is a computational resource upon which artifacts are deployed for execution.
- A node is a physical thing that can execute one or more artifacts.
- Shown using the stereotype **<<device>> or <<execution environment>>**

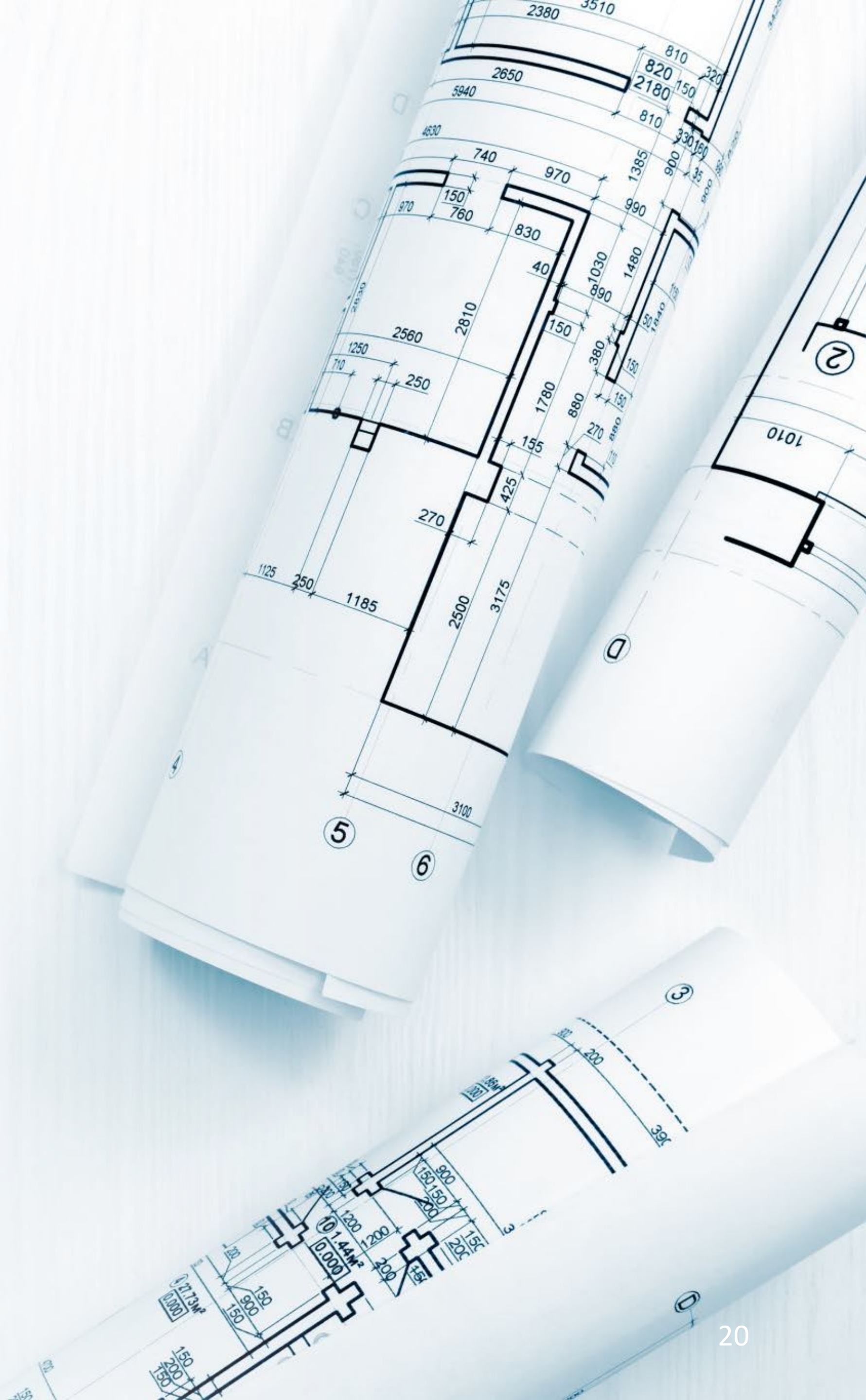


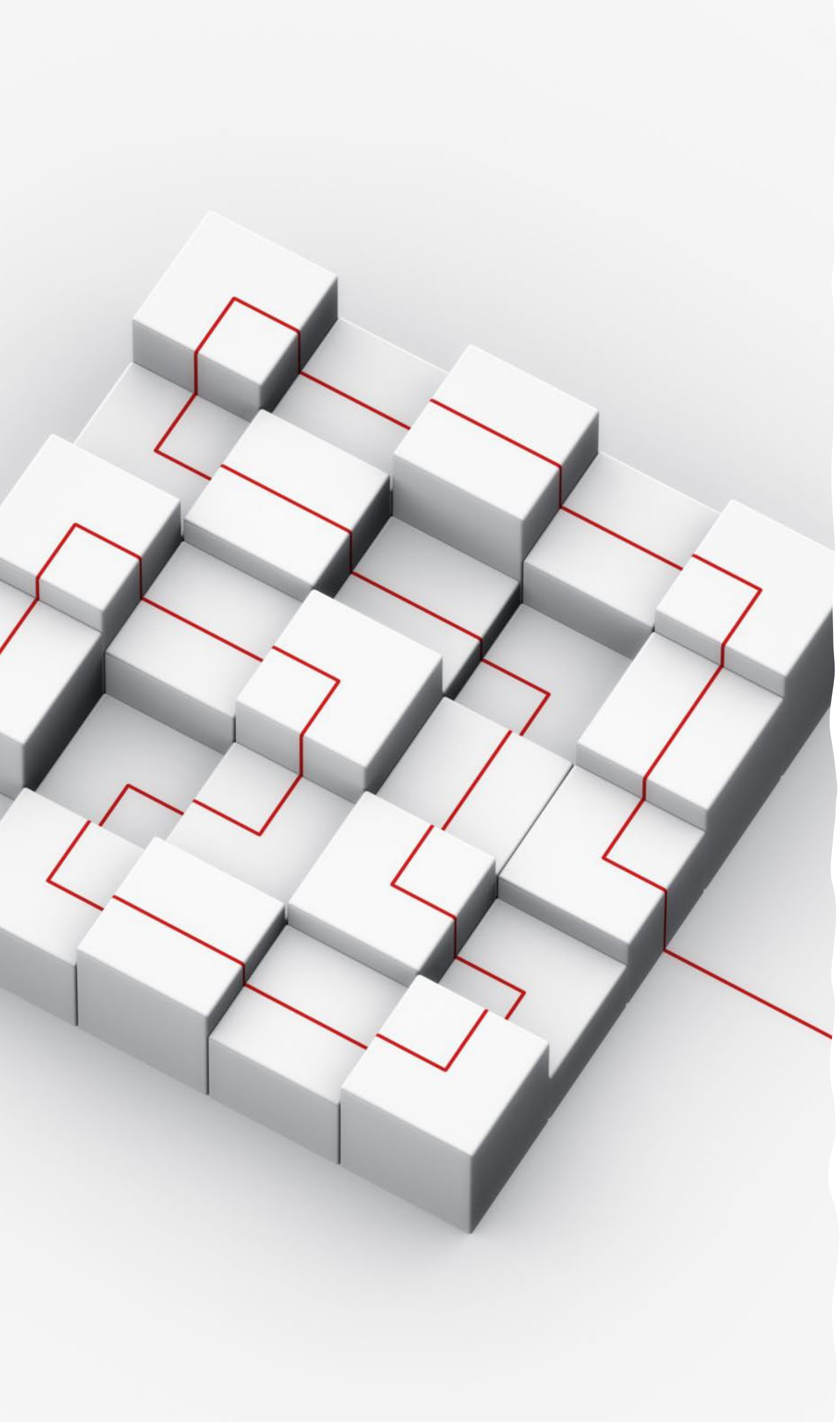


# Deployment Diagram Example: Working of HTML5 video player



# You need to create a component(or package diagram) and a deployment diagram for your project 1.





# Low Level Design View – Structural Diagram: DCD

# Design Class Diagram

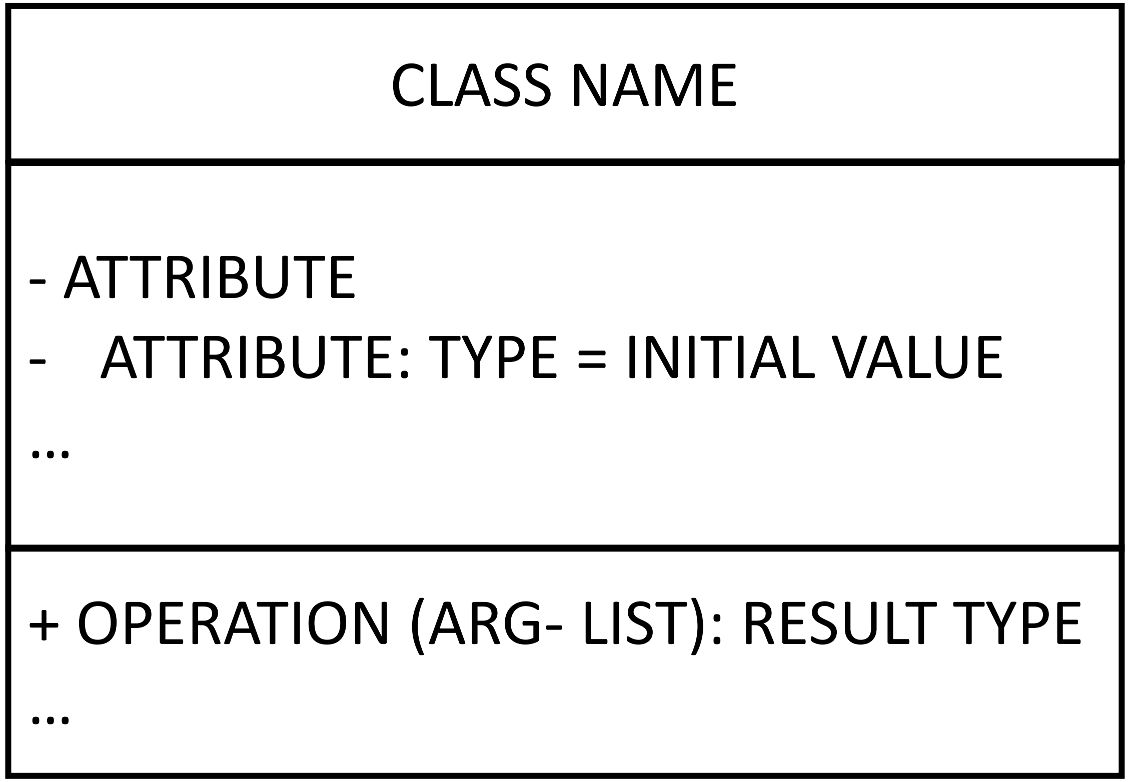
Static, Structural View of the System

Describes

- Classes and their Structure
- Relationships among classes

Slide adapted from Alessandro Orso

# Class Diagram: Class



CLASS NAME

- ATTRIBUTE
- ATTRIBUTE: TYPE = INITIAL VALUE

…

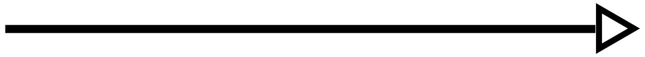
+ OPERATION (ARG- LIST): RESULT TYPE

…

Class names are identified as potential nouns from the requirements

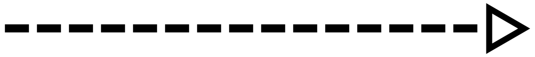
# Relationships in DCD

Generalization: X **is a** Y



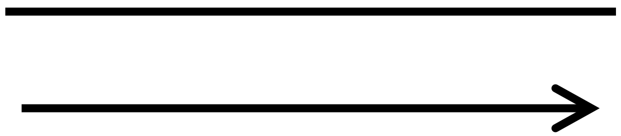
Inheritance between classes or interface implementation

Realization: X **is a** Y



interface implementation

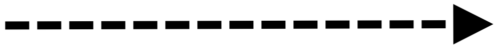
Associations: X and Y are related (**has**)



Can be dependency, aggregation, or composition association

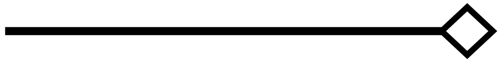
Can be unidirectional

Dependencies: X **uses** Y



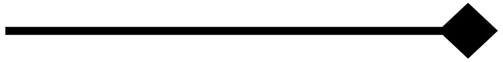
methods of a class that use another class's object as a parameter .

Aggregations: X **has** a Y



The relationship between the whole and the part (parent and child), but the part can exist independent of the whole

Composition: X **has** a Y

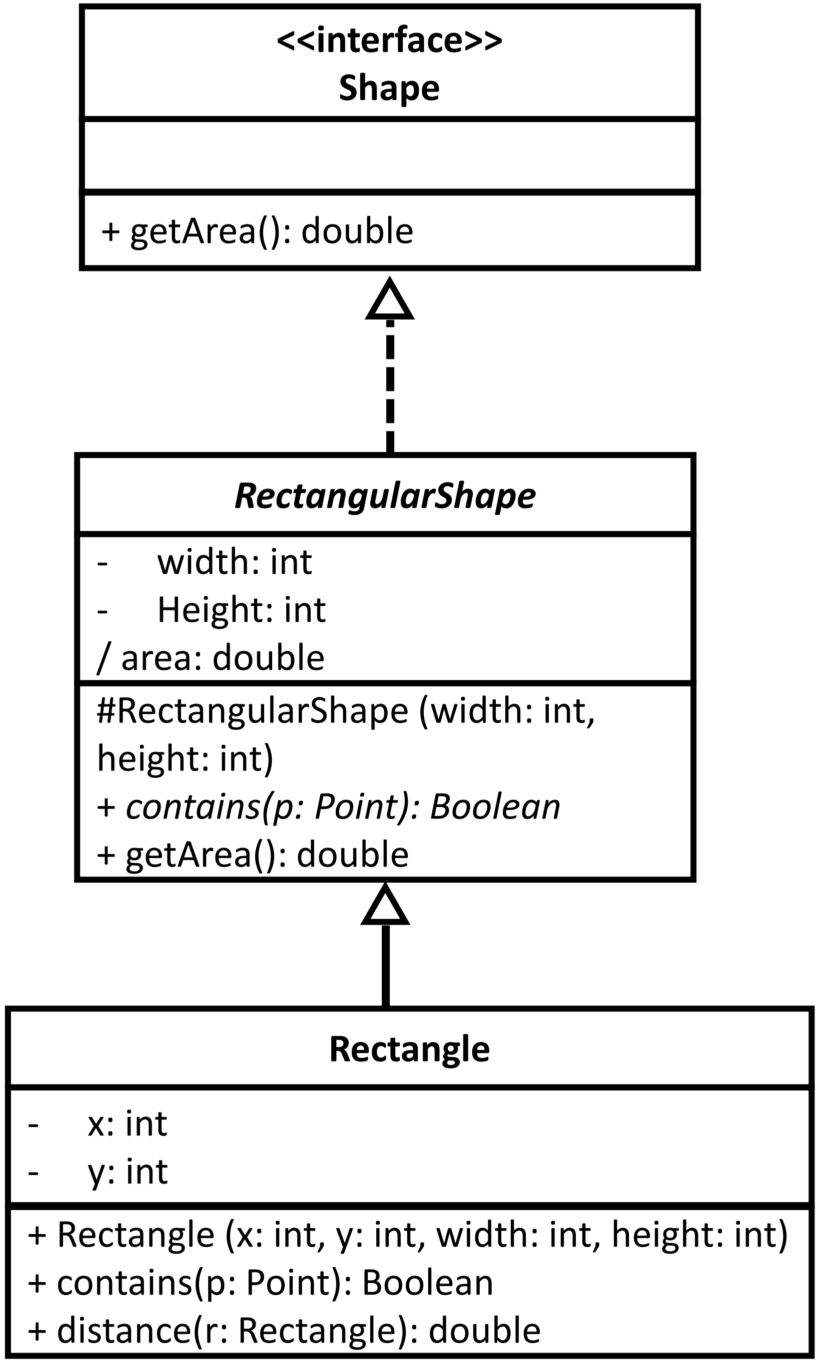


The relationship between the whole and the part, but the part cannot exist independent of the whole.

# Generalization (Realization)

- hierarchies drawn top-down
- arrows point upward to parent
- line/arrow styles indicate whether parent is:
  - class:
    solid line, black arrow
  - interface:
    dashed line, white arrow (also called **realization relationship**)

- often omit trivial / obvious generalization relationships, such as drawing the Object class as a parent



**<<interface>>**
**Shape**

+ getArea(): double

***RectangularShape***

- width: int
- Height: int
/ area: double

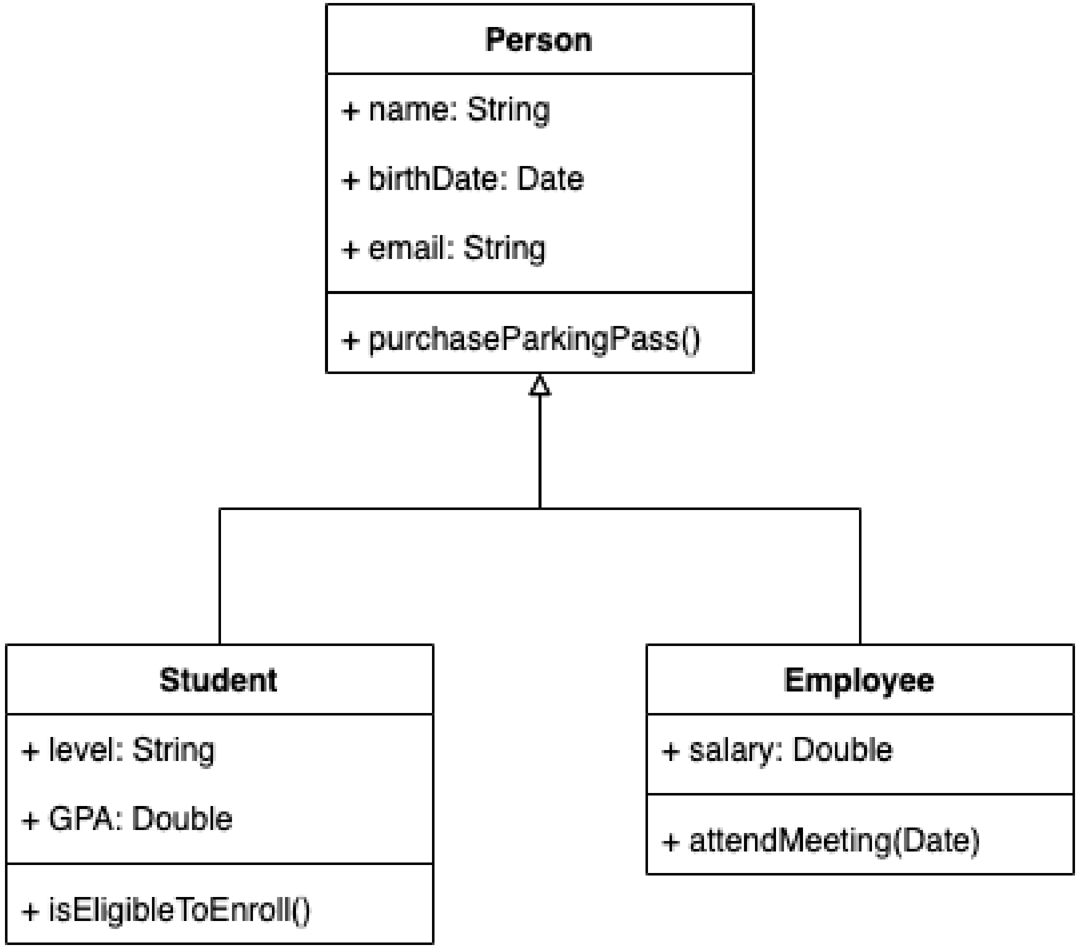
#RectangularShape (width: int, height: int)
+ *contains(p: Point): Boolean*
+ getArea(): double

**Rectangle**

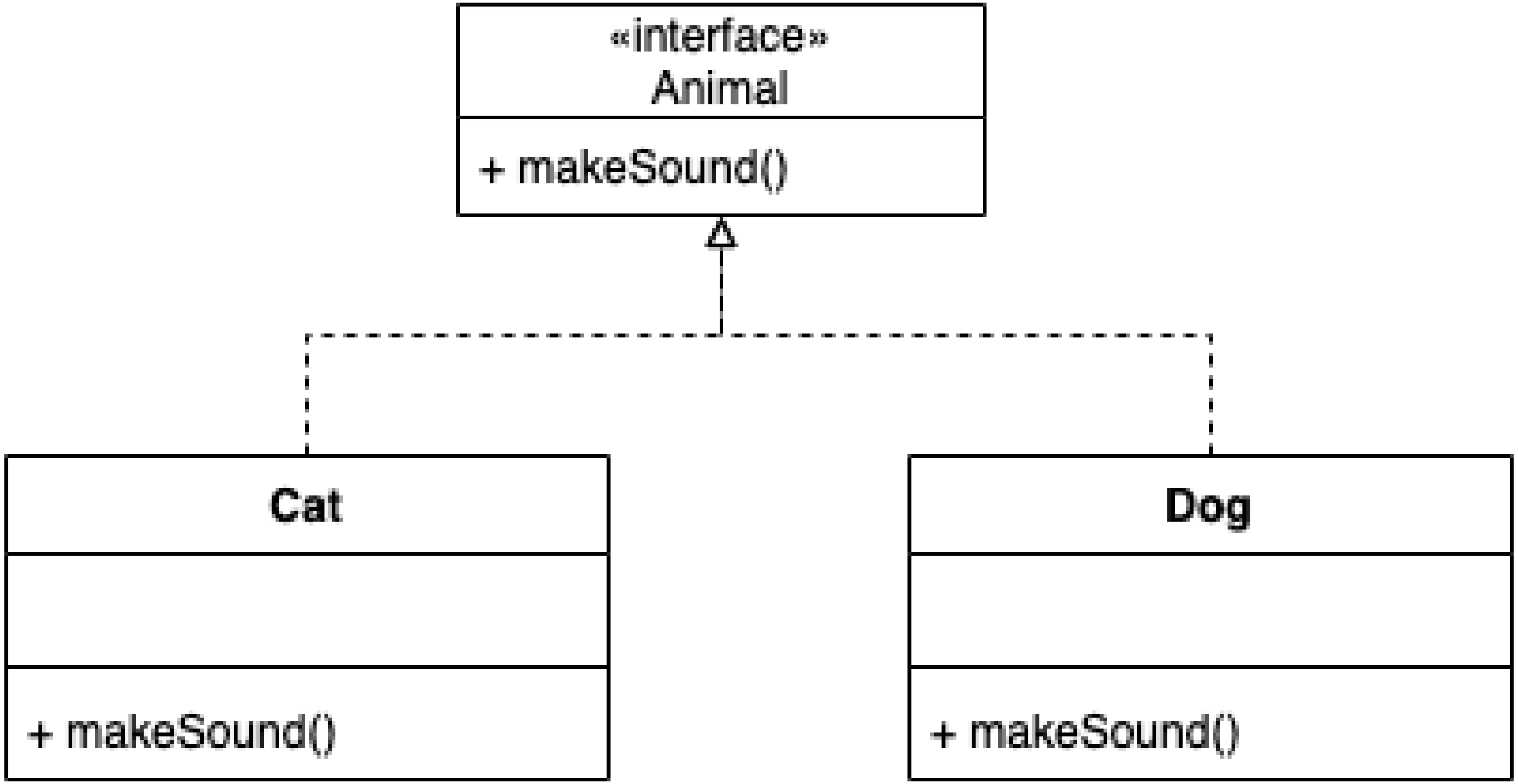
- x: int
- y: int

+ Rectangle (x: int, y: int, width: int, height: int)
+ contains(p: Point): Boolean
+ distance(r: Rectangle): double

# Generalization Example

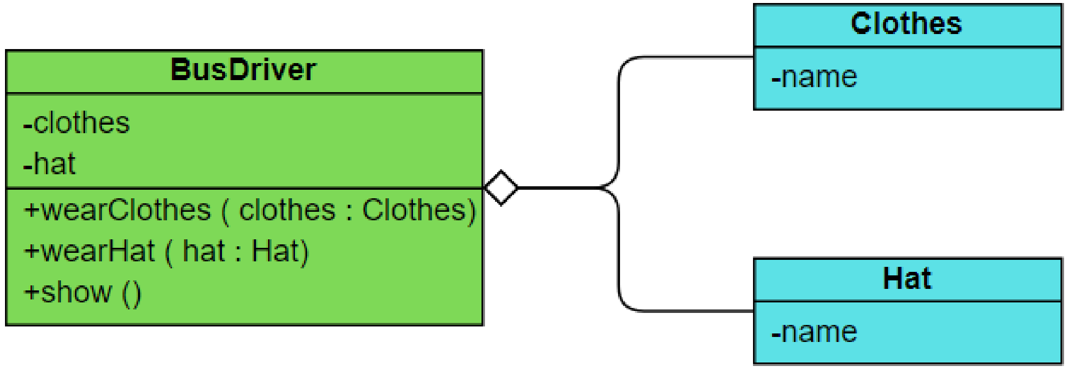


# Generalization (Realization) Examples: Interface

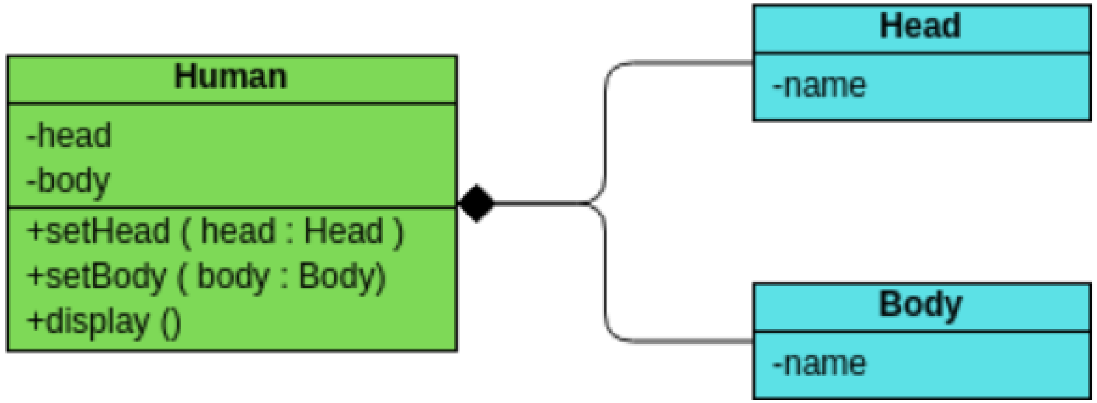


# Association Types

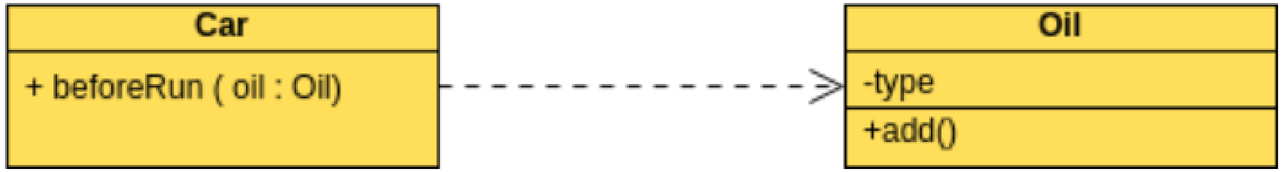
- aggregation: “is part of”
  - child can exist independently of the parent



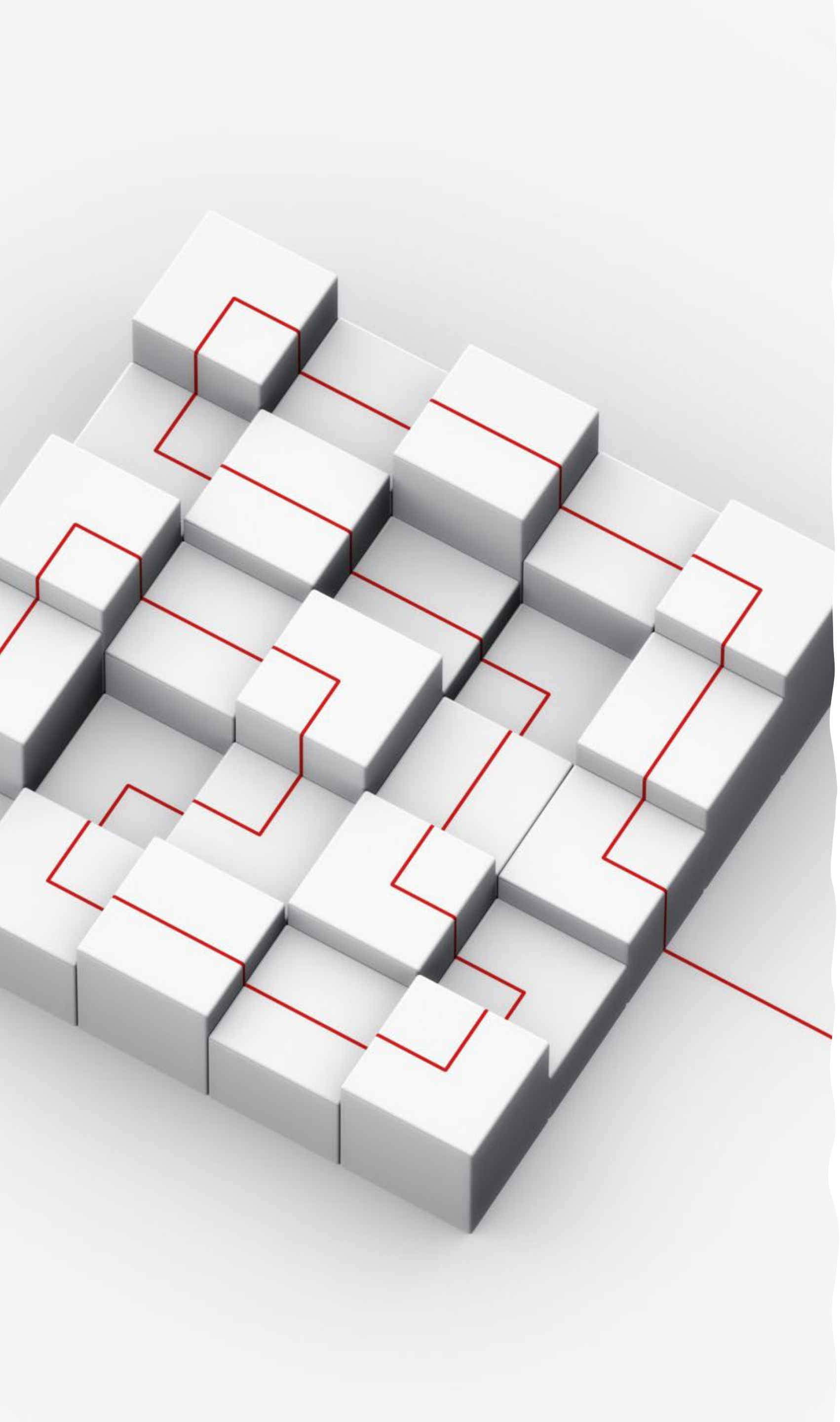
- composition: “is entirely made of”
  - stronger version of aggregation
  - the parts live and die with the whole
  - child cannot exist independent of the parent
  - symbolized by a black diamond



- dependency: “uses temporarily”
  - symbolized by dotted line
  - often is an implementation detail, not an intrinsic part of that object's state



https://blog.visual-paradigm.com/what-are-the-six-types-of-relationships-in-uml-class-diagrams/

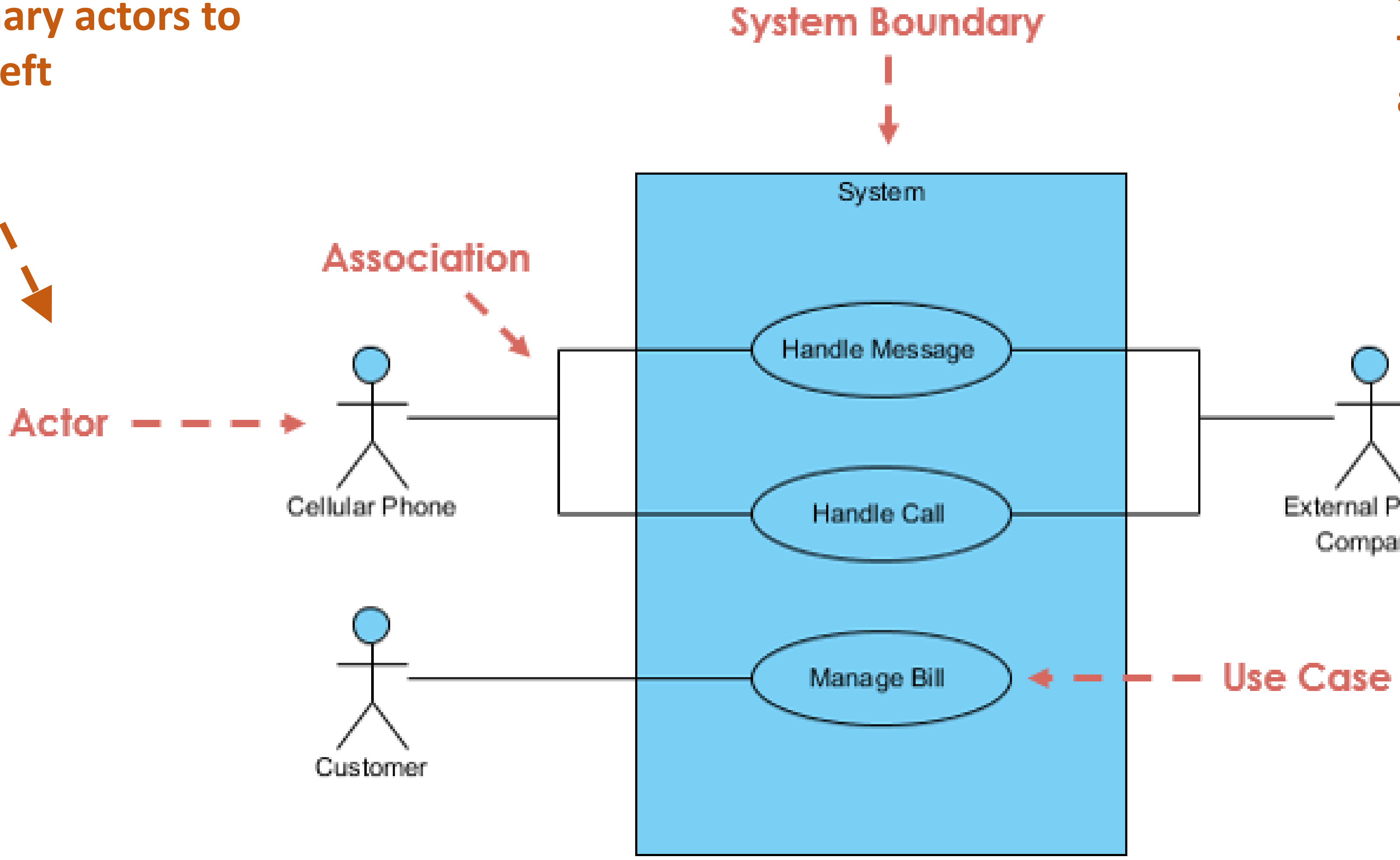


# Low Level Design View – Behavioral Diagram: Use Case Diagram

# Use Case Diagram: Structure

- actors as stick-figures, with their names (nouns)
- use case goals as ellipses with their names (verbs)
- line associations, connecting an actor to a use case in which that actor participates

Primary actors to the left



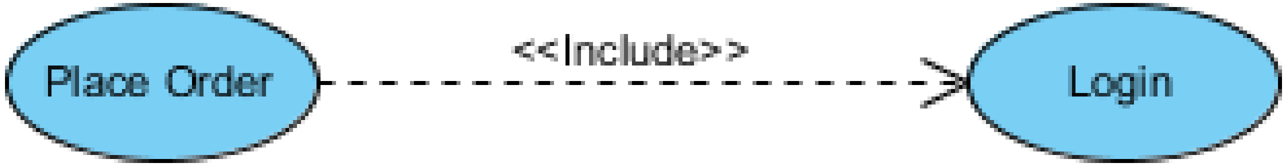
Supporting actors to the right: they provide a service.

Offstage actor towards the bottom

# Use Case Relationships

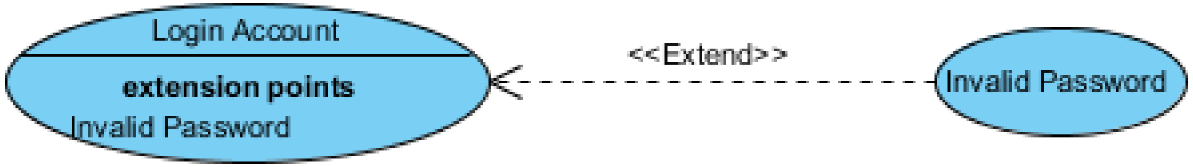
## <<Include>>

- Used when one use case is **used by** another use case
- Typically used if a sub-use case or series of steps is used by **several use cases**
- A **uses** relationship from base use case to child use case indicates that an instance of the base use case will ***include*** the behavior in the child use case.
- Depicted with a directed arrow having a dotted line. The tip of arrowhead **points to the child use case** and the parent use case connected at the base of the arrow.

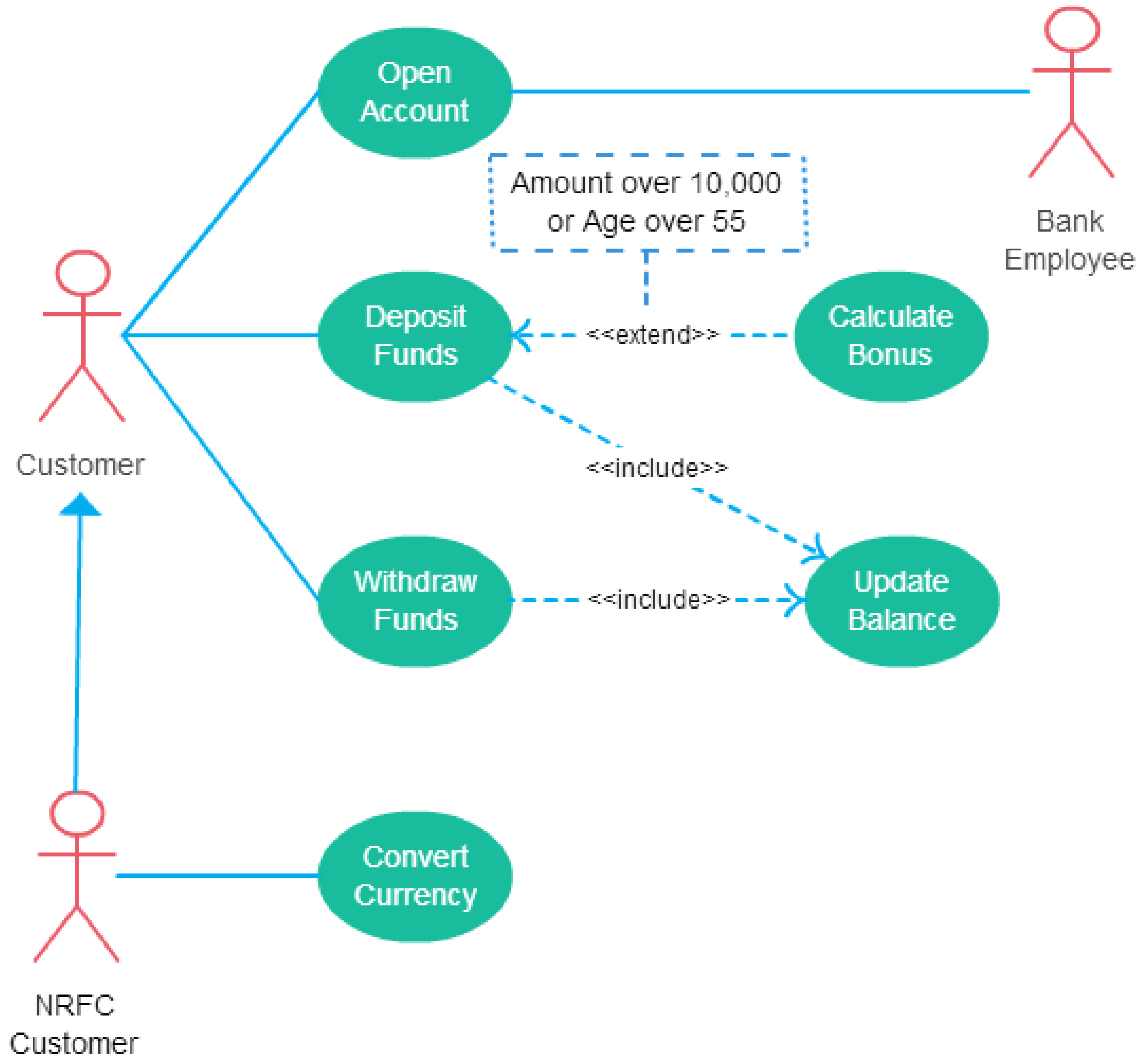


## <<Extend>>:

- Used when a use case may optionally take an alternate path
- You can think of these as exceptions to the typical path in the use case
- Depict with a directed arrow having a dotted line. The tip of arrowhead **points to the base use case** and the child use case is connected at the base of the arrow.
- The stereotype "<<extends>>" identifies as an extend relationship
- The extension point is specified in the base use case



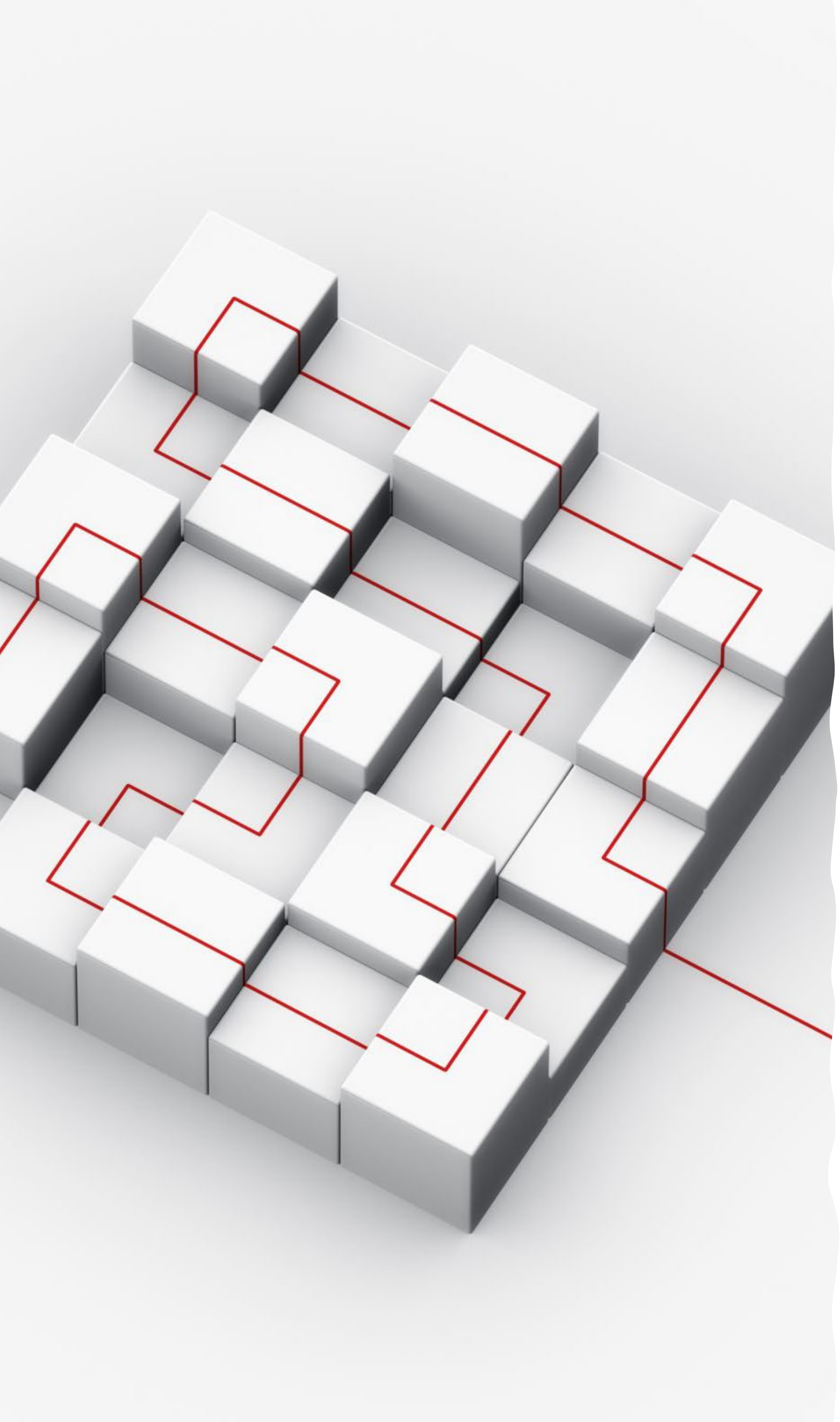
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/



Note that we don’t cover “generalization” in this class

*Includes is usually used to model common behavior*

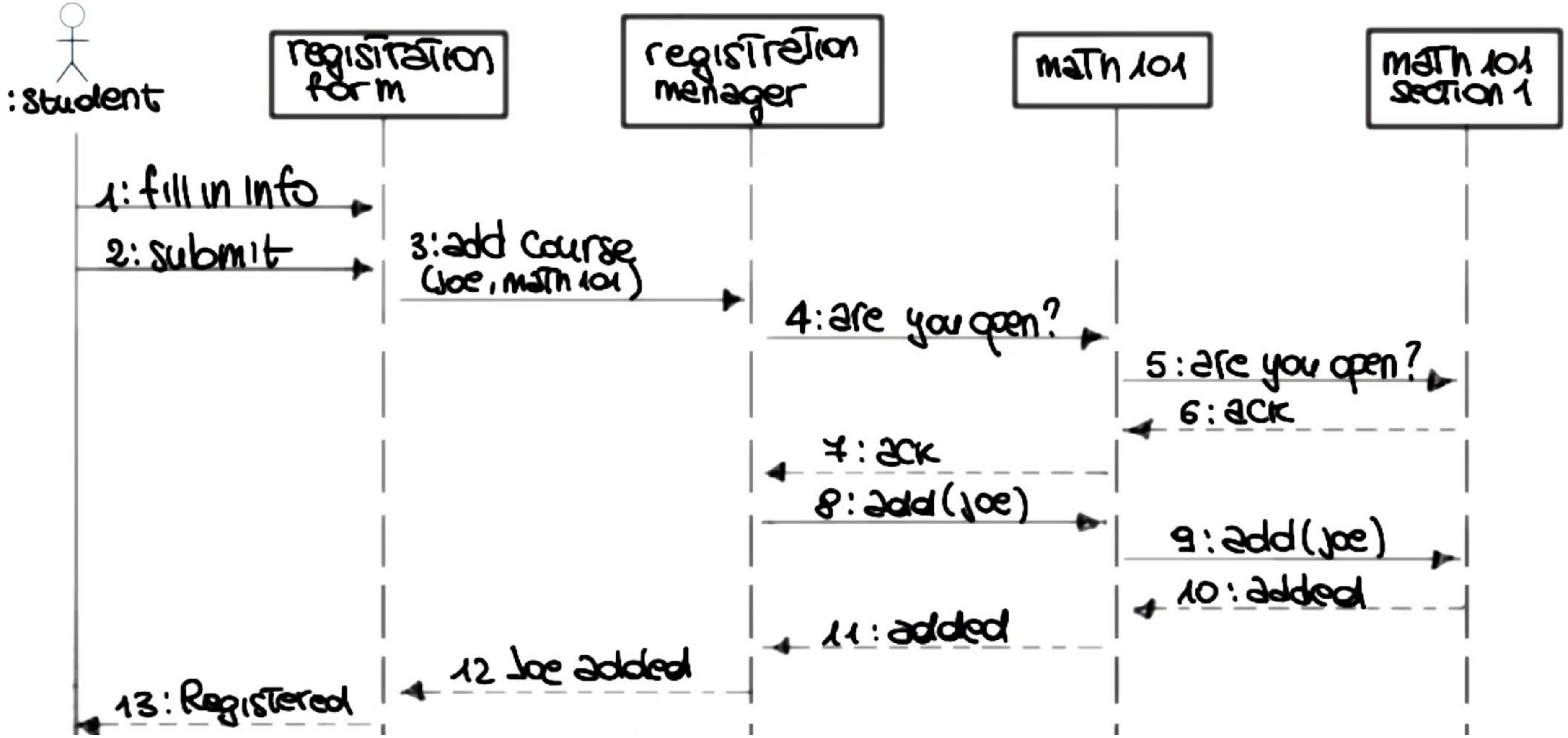
https://creately.com/blog/diagrams/use-case-diagram-relationships/



# Low Level Design View – Behavioral Diagram: SD

# Sequence Diagrams

Diagrams that emphasize time ordering of messages between classes/components



Slide adapted from Alessandro Orso

# You need to create a DCD, UCD and SD for your project 1.