

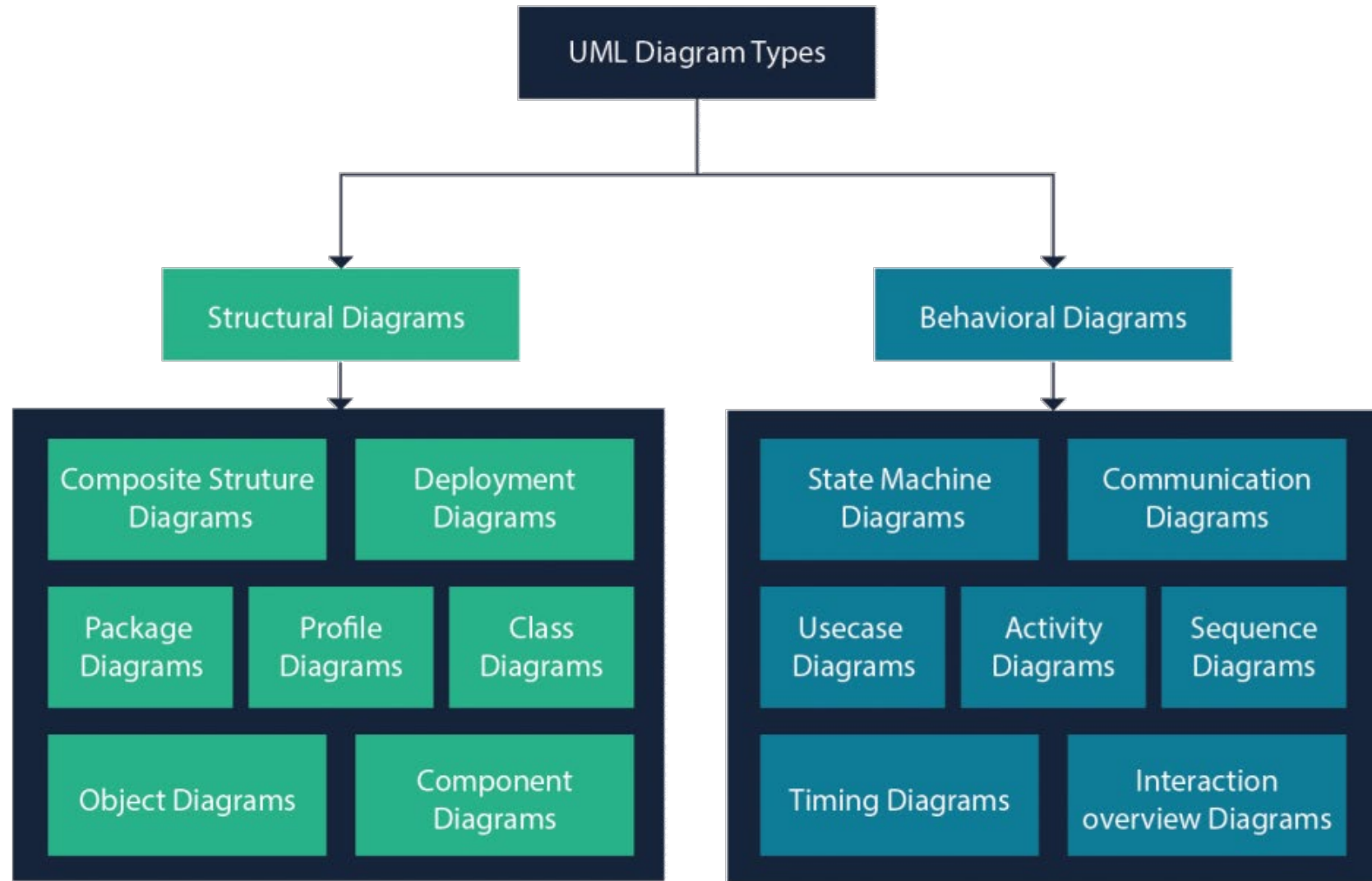
CS3300 Introduction to Software Engineering

Lecture 7: Software Design: Unified Modeling Language

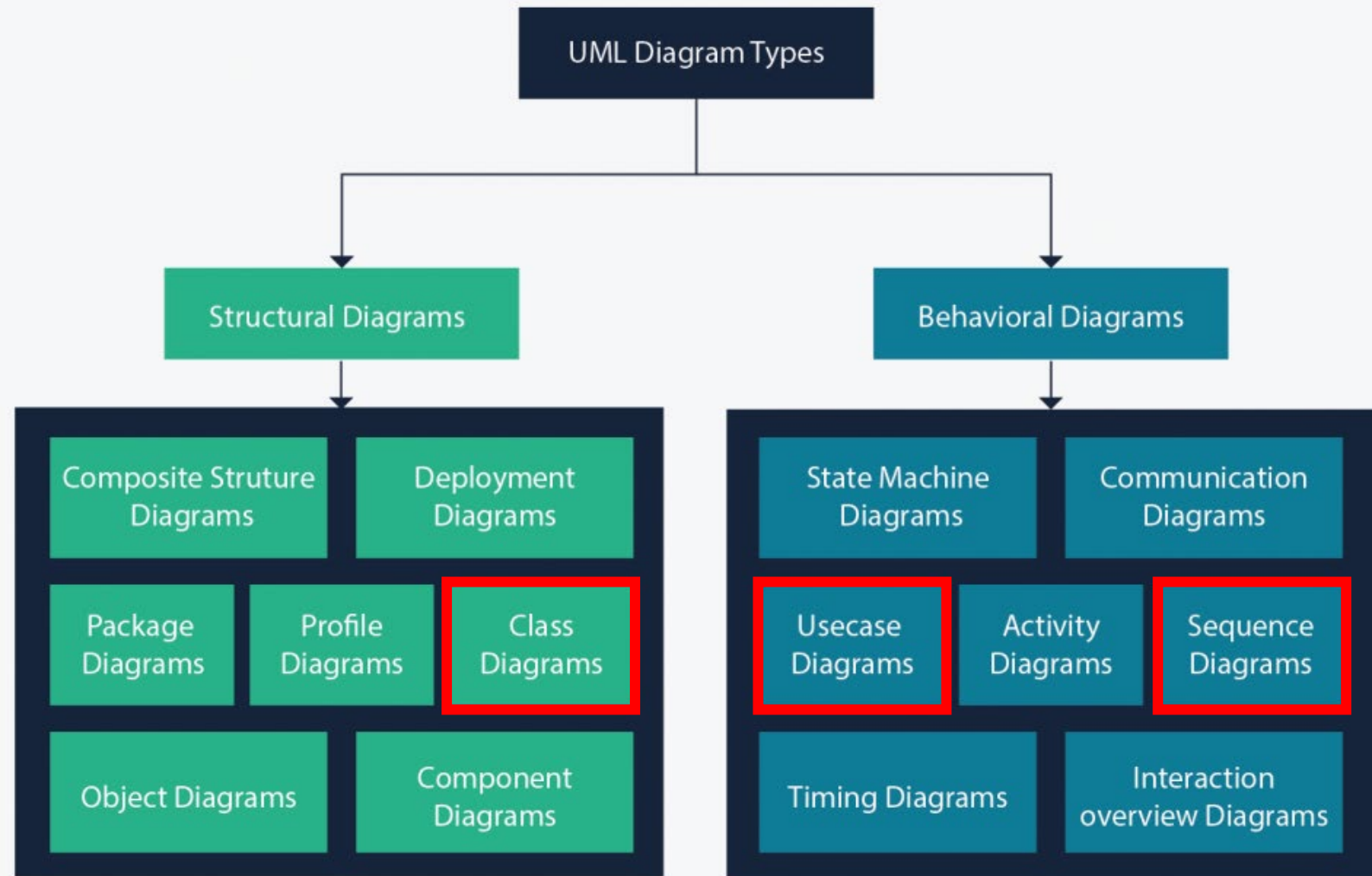
Dr. Nimisha Roy ▶ nroy9@gatech.edu

Unified Modeling Language

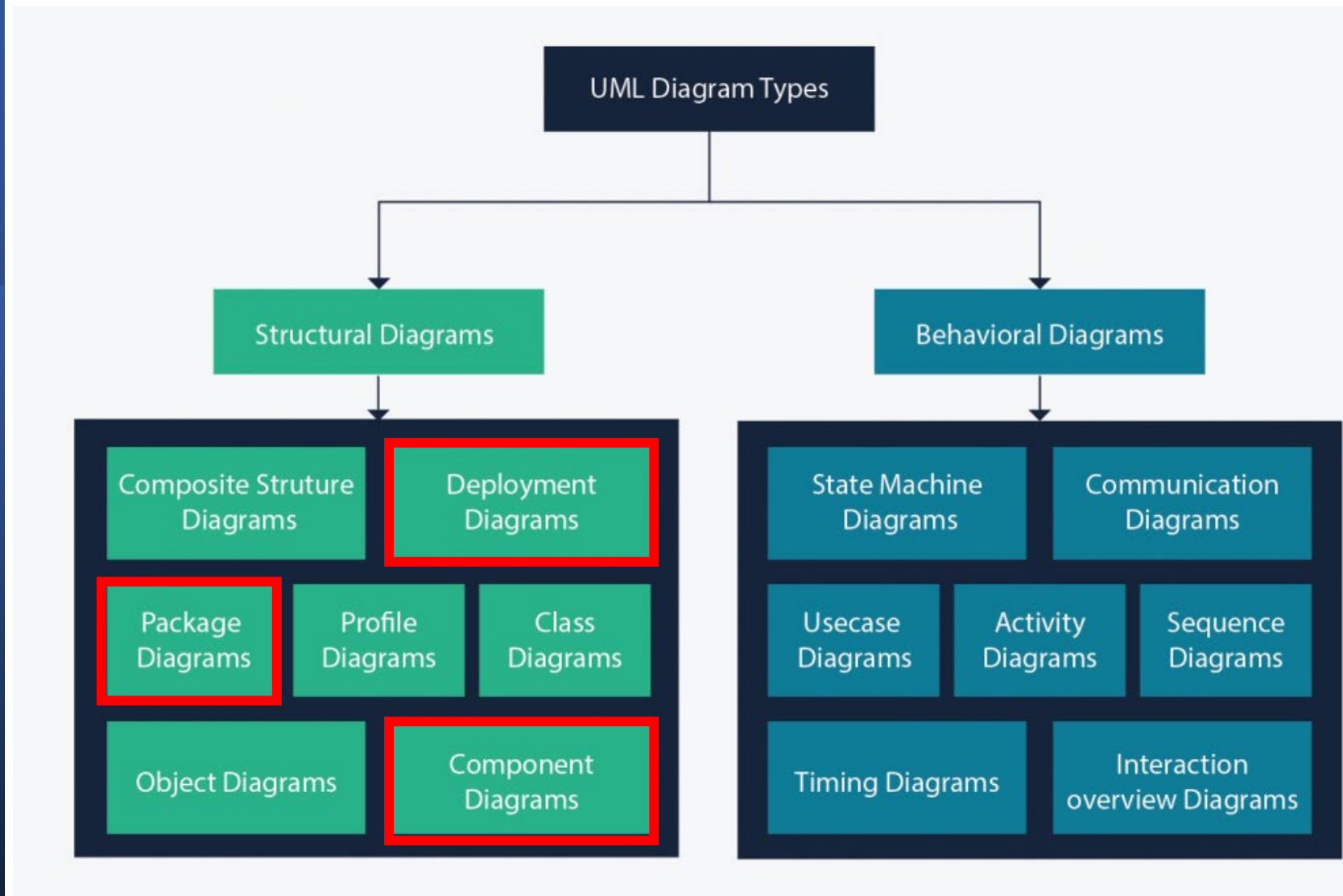
Intended to provide a standard way to visualize the design of a system.



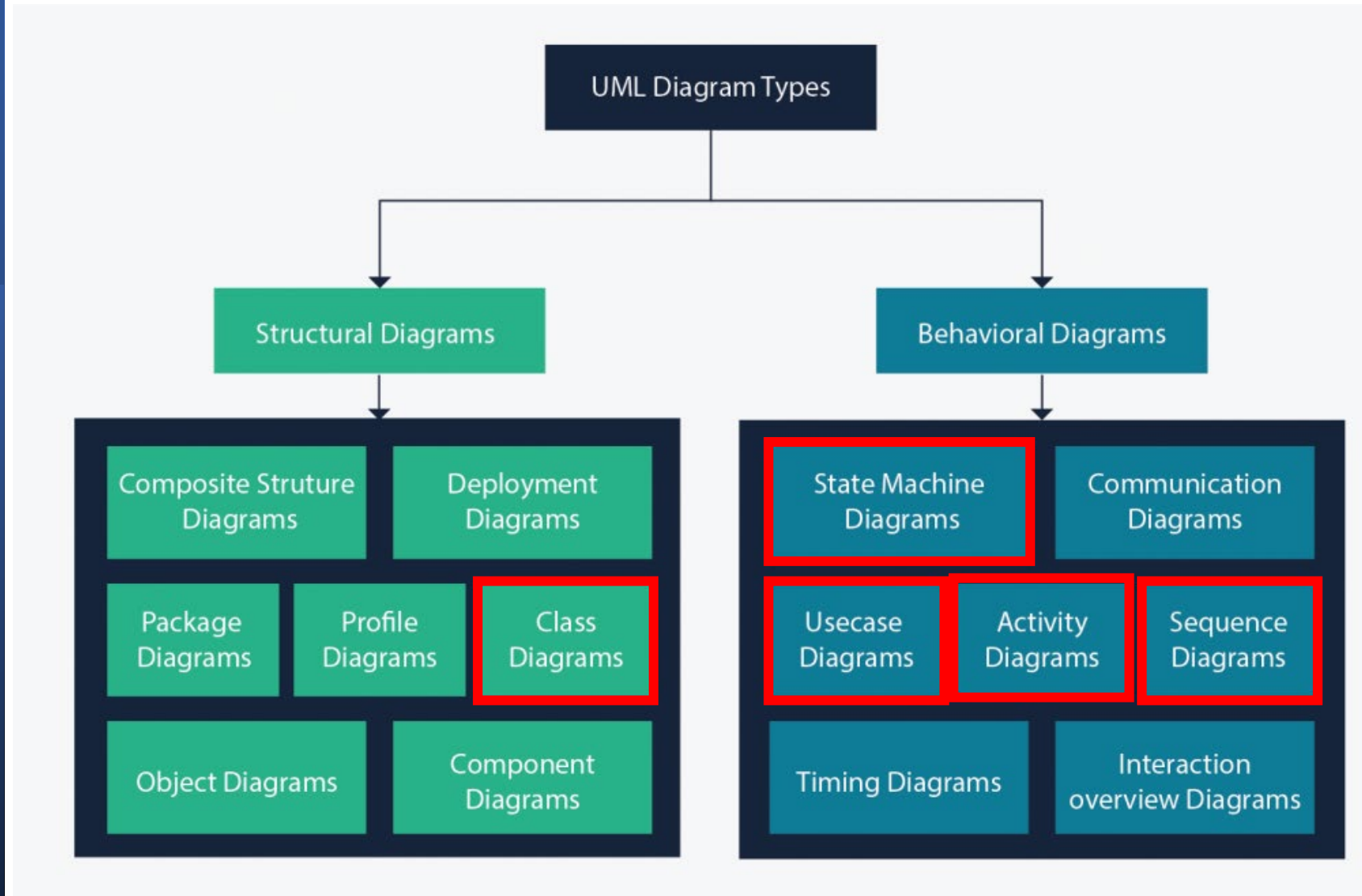
Unified
Modeling
Language: You
may already
know...

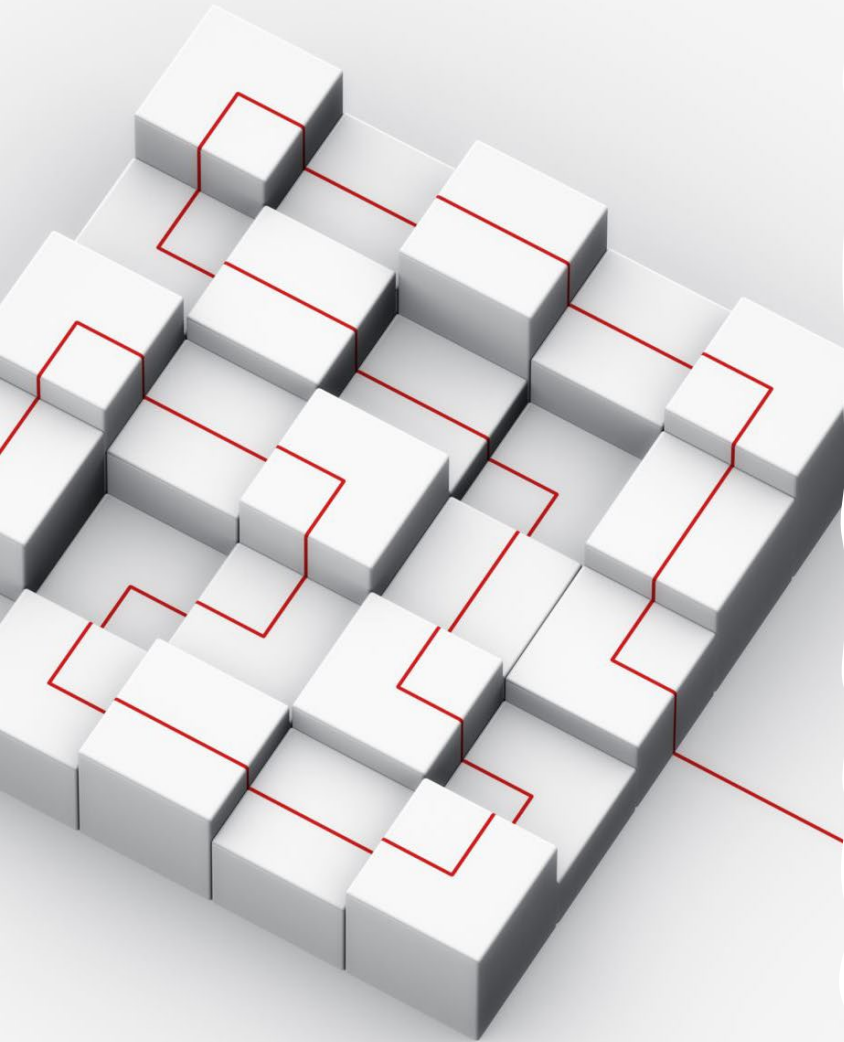


Unified Modeling Language: Architectural View



Unified Modeling Language: Low Level Design View





Architectural View – Structural Diagram: Component

Component Diagram

Static view of components and their relationships

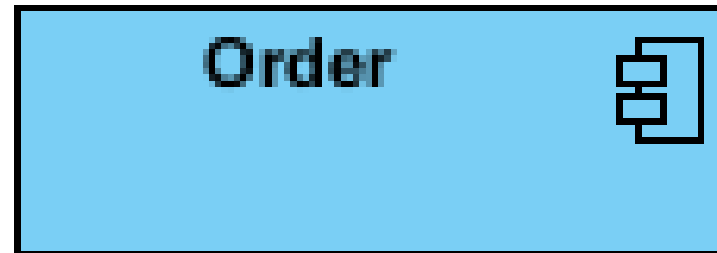
Shows how software components are interconnected and how they will interact with one another, emphasizing the organization and dependencies of different components.

Can be used to represent a software architecture

Component Diagram: Component

Components represented as

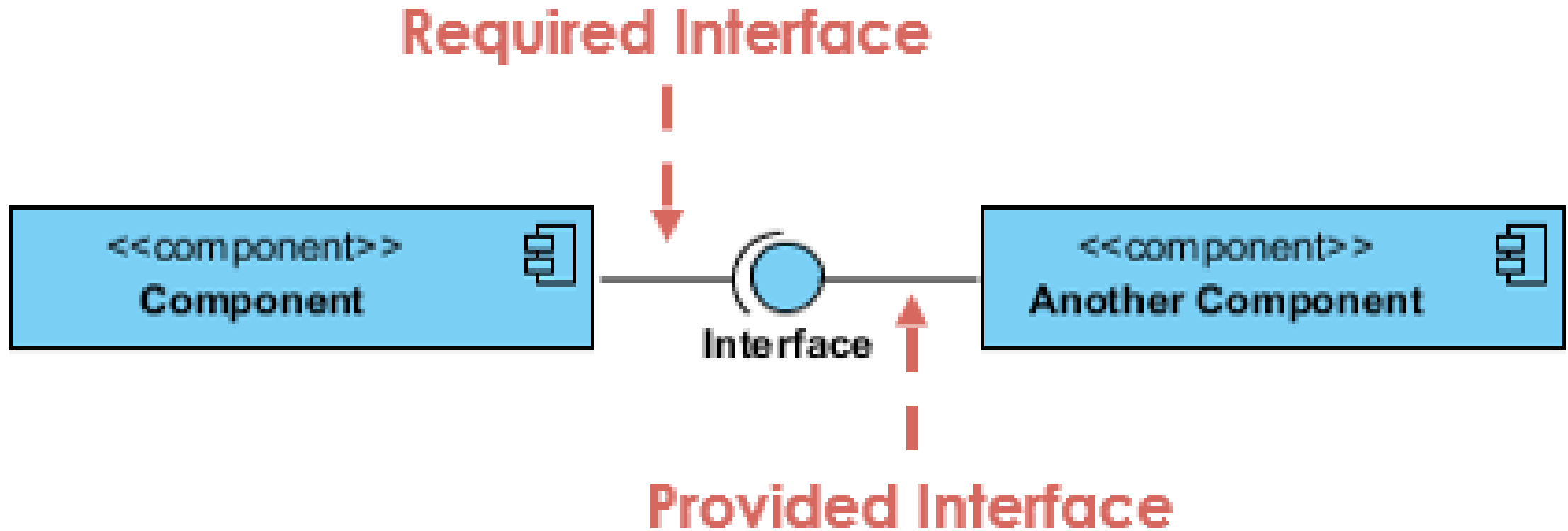
1. A rectangle with the component's name
2. A rectangle with the component icon
3. A rectangle with the stereotype text and/or icon



Component Diagram: Interface

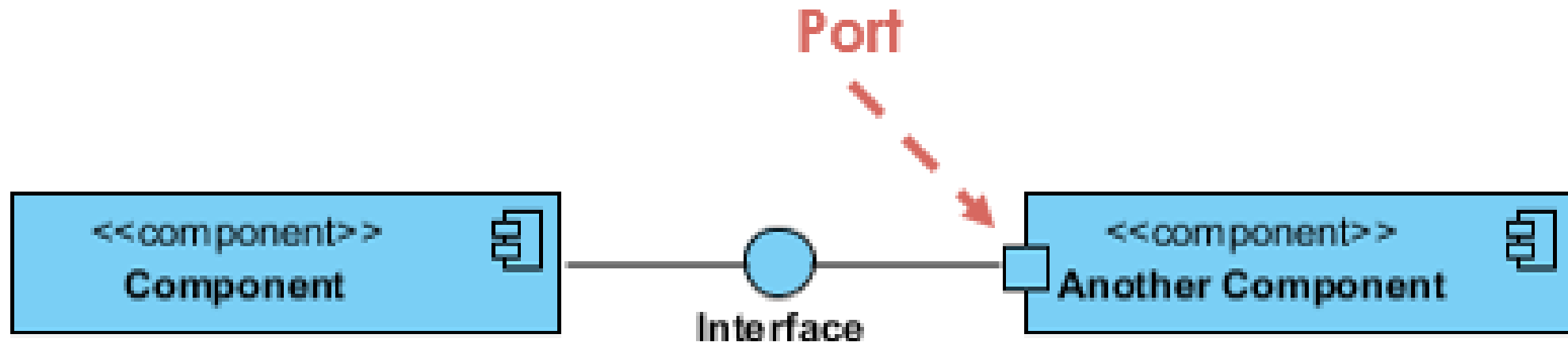
Provided interface symbols with a complete circle at their end represent an interface that the component provides - "lollipop" symbol

Required Interface symbols with only a half circle at their end (a.k.a. sockets) represent an interface that the component requires.



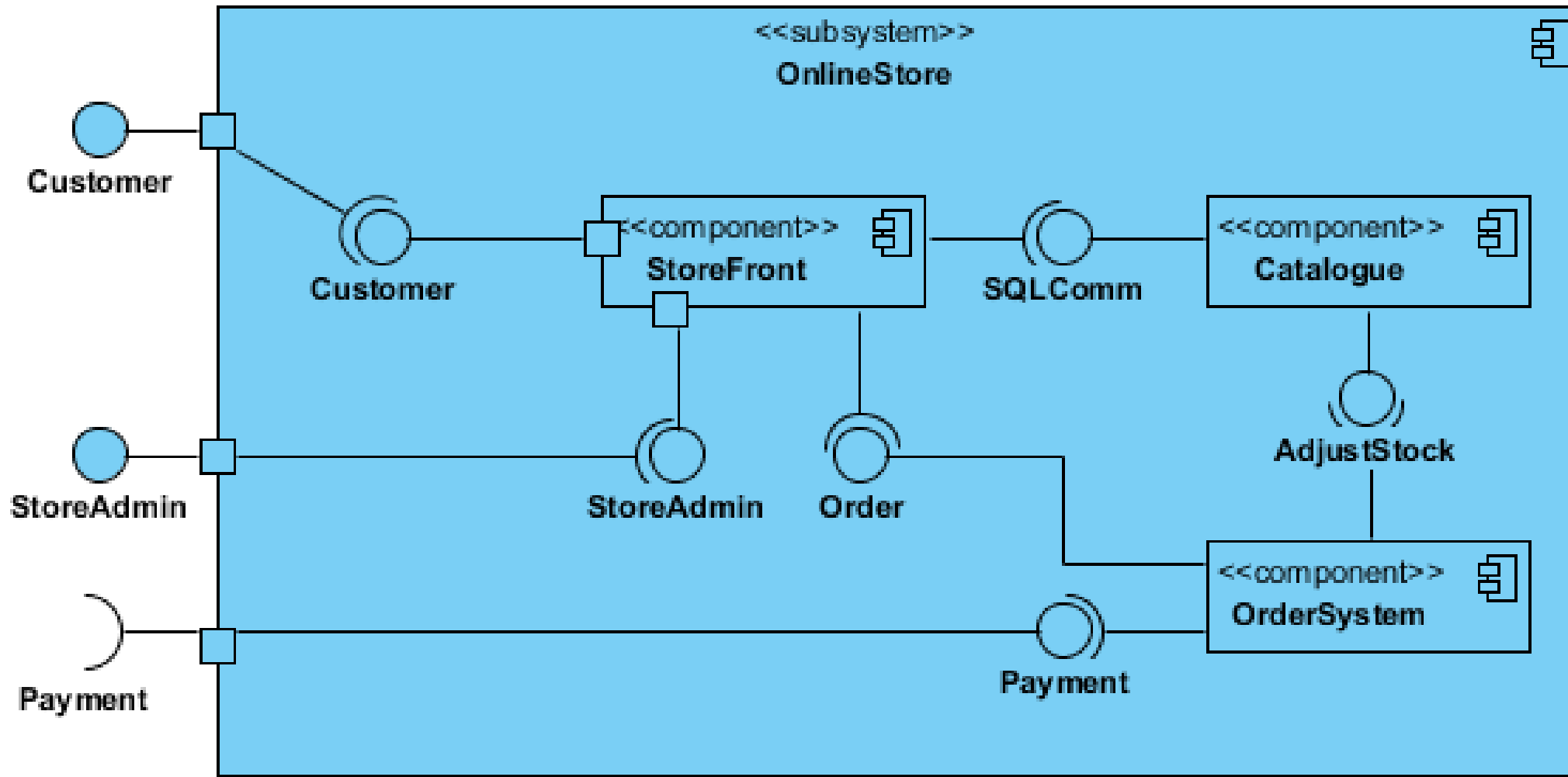
Component Diagram: Port

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.

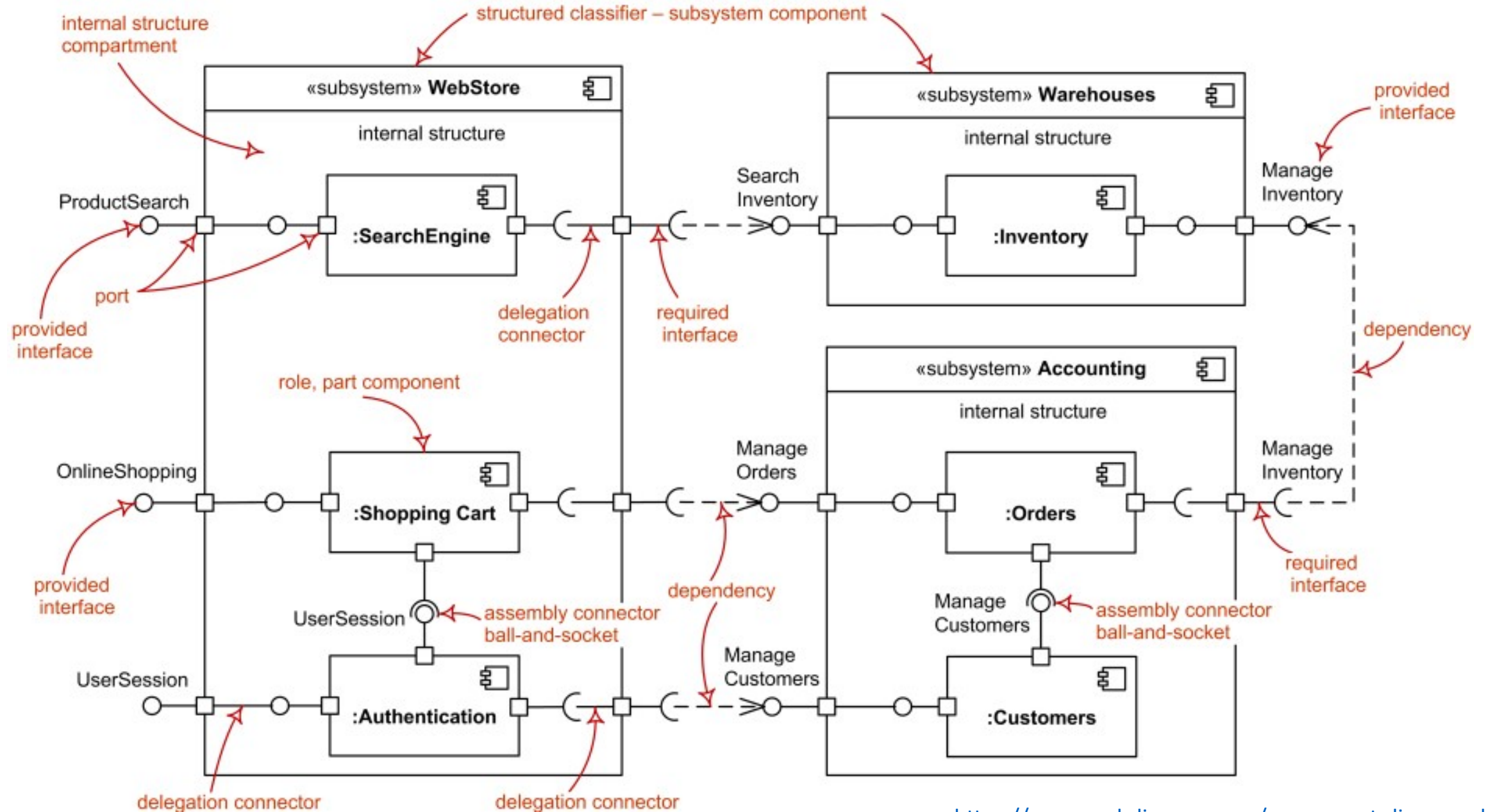


Component Diagram: Subsystem

Collection of components that work together to execute a set of related functionalities and is often used to represent a larger segment of your system.
subsystem notation element has the keyword of <<subsystem>>



Component Diagram Example: ecommerce



Example Quiz




Consider a simplified social media application that consists of the following software components:

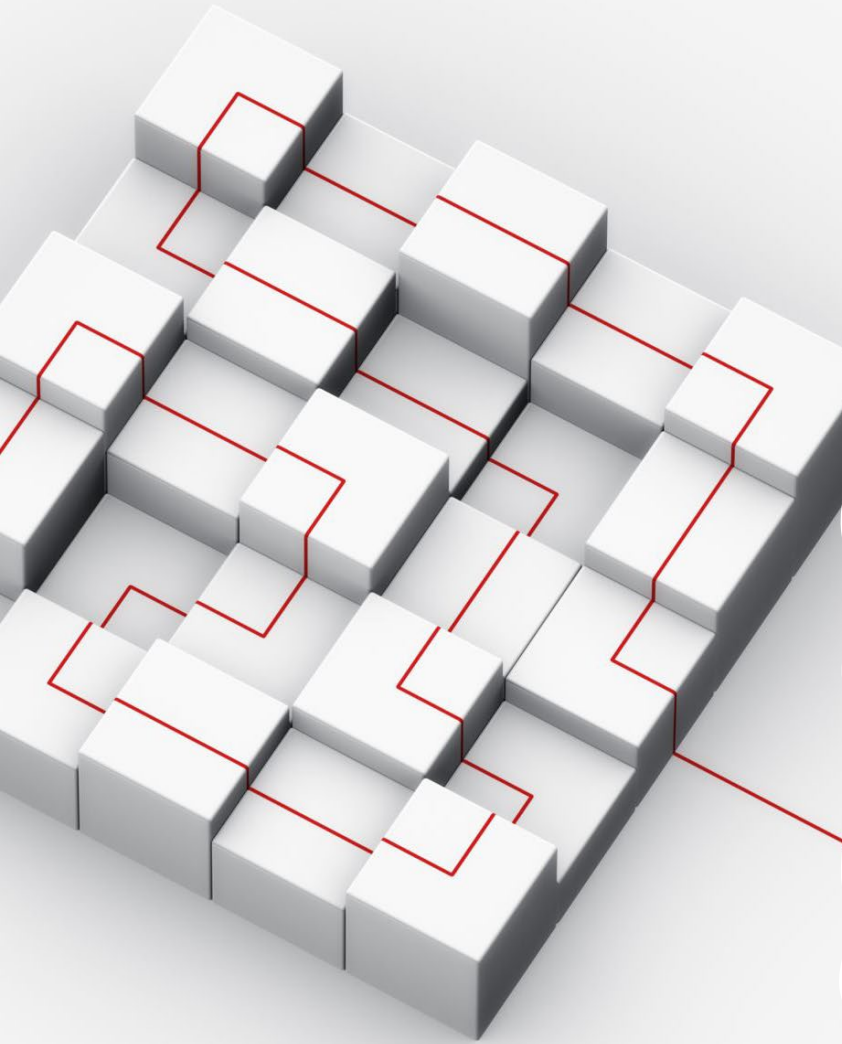
- **User Profile Component:** Manages user information, profile creation, and updates.
- **Post Management Component:** Handles creating, deleting, and editing posts and content.
- **Friendship Management Component:** Manages friend requests, friend lists, and related operations.
- **Messaging Component:** Handles direct messages between users.
- **Notification Component:** Manages notifications related to posts, friend requests, and messages.

Example Quiz



Which of the following statements is likely accurate regarding the dependencies between these components?

- A. The Messaging Component depends on the Post Management Component to send messages.
-  B. The Notification Component depends on the Friendship Management Component to notify users of new friend requests.
- C. The User Profile Component depends on the Messaging Component to update user profiles.
- D. The Post Management Component depends on the User Profile Component to create posts.



Architectural View – Structural Diagram: Deployment

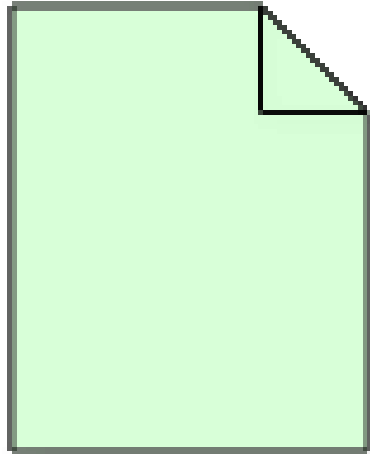
Deployment Diagram

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

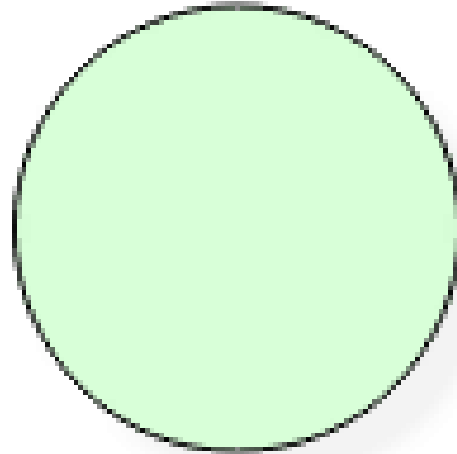
Maps the software architecture created in design to the physical system architecture that executes it. In distributed systems, it models the distribution of the software across the physical nodes.

Deployment Diagram: Notations

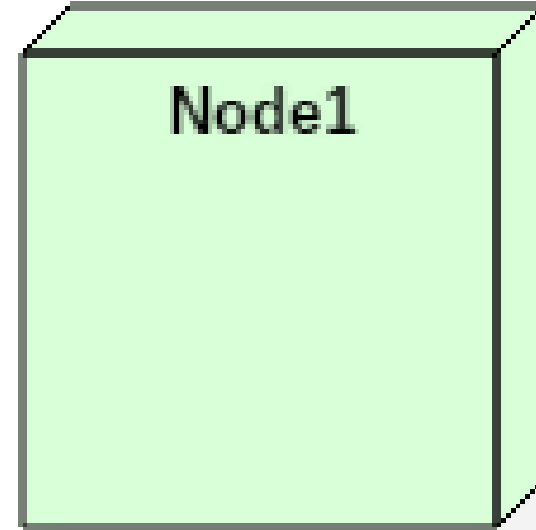


Artifact1

source files, executables, scripts,
database tables, or documentation

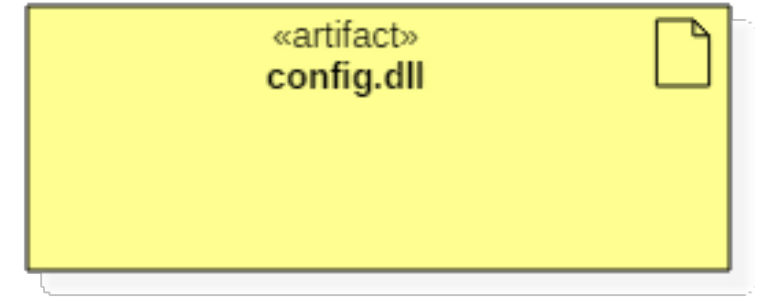


Interface1



Servers, computers, mobile
devices, cloud environment

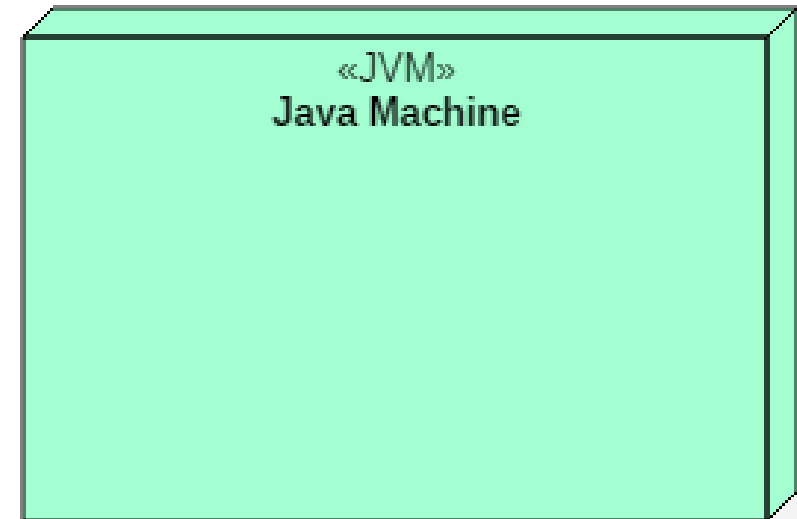
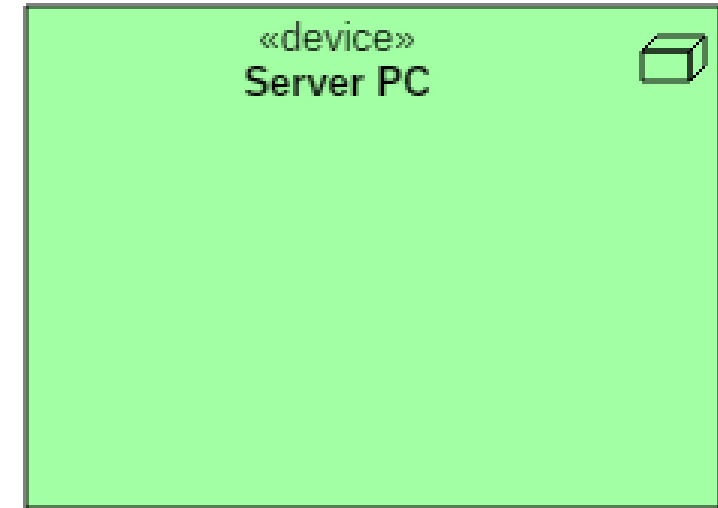
Deployment Diagram: Artifact



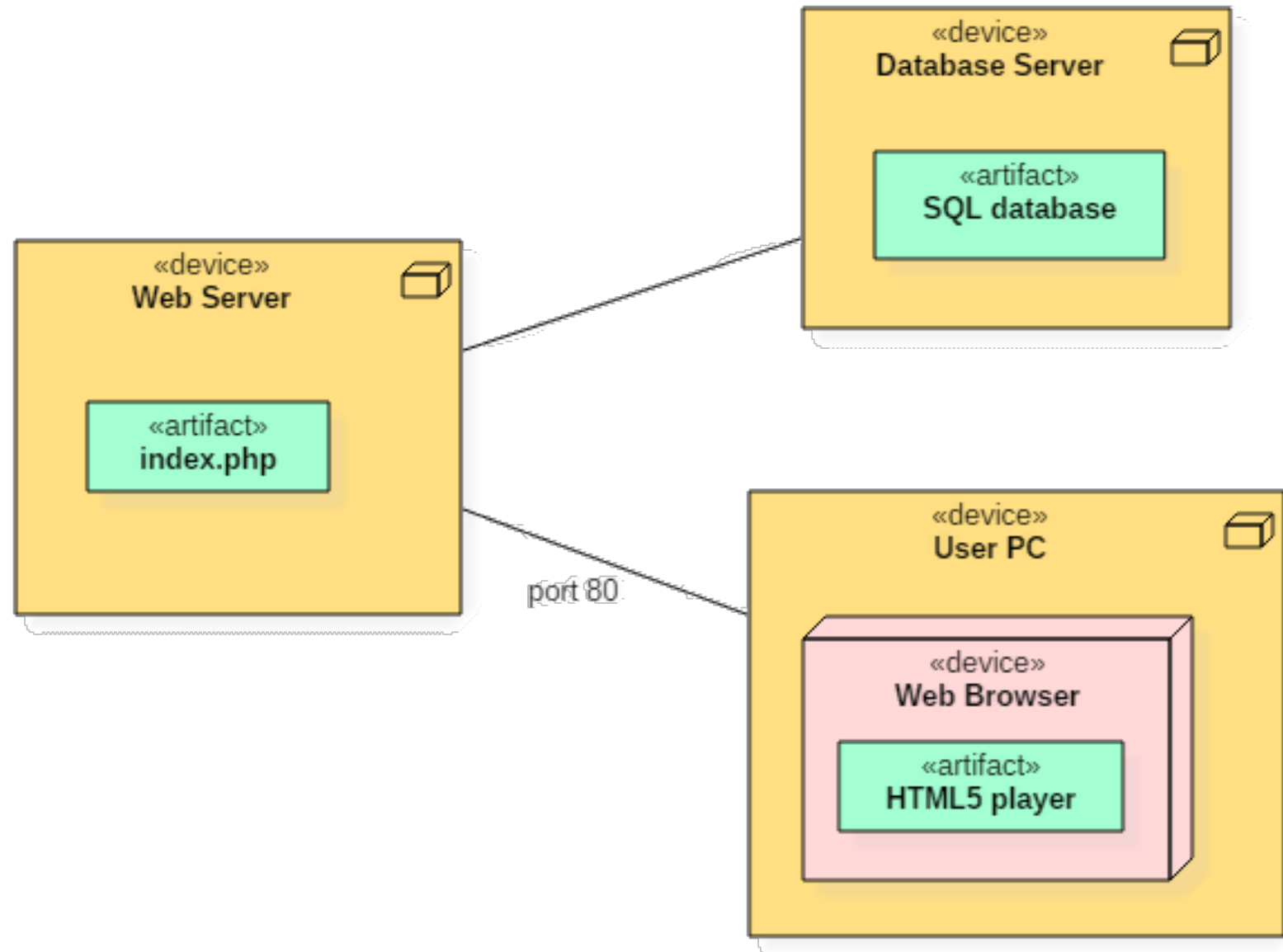
- Represent physical entities that are used or produced in a software development process
- Artifacts are deployed on the nodes. The most common artifacts are as follows:
 - Source files
 - Executable files
 - Database tables
 - Scripts
 - DLL files
 - User manuals or documentation
 - Output files
- 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



Example Quiz



Match the file to the corresponding Deployment Diagram parts – Artifact or Node

BackendAPI

Artifact

Google App Engine


Node

SmartPhone

Node

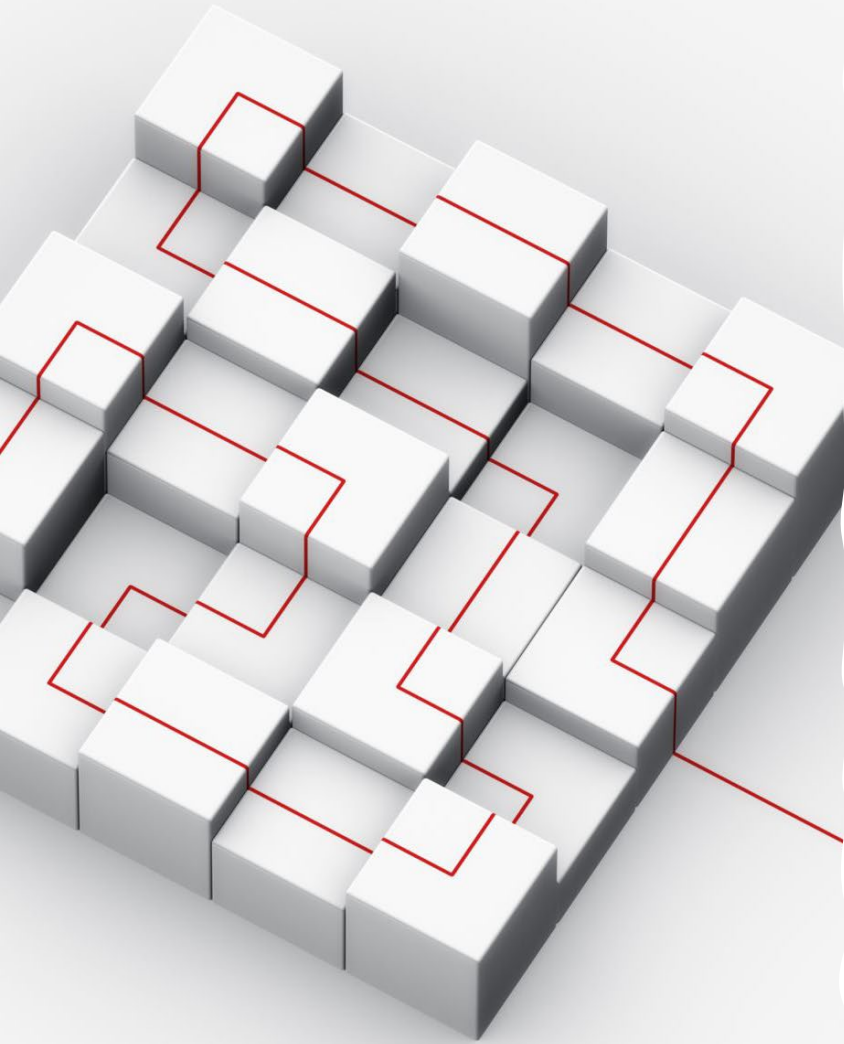
WebApp

Artifact



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





Low Level Design View – Structural Diagram: DCD

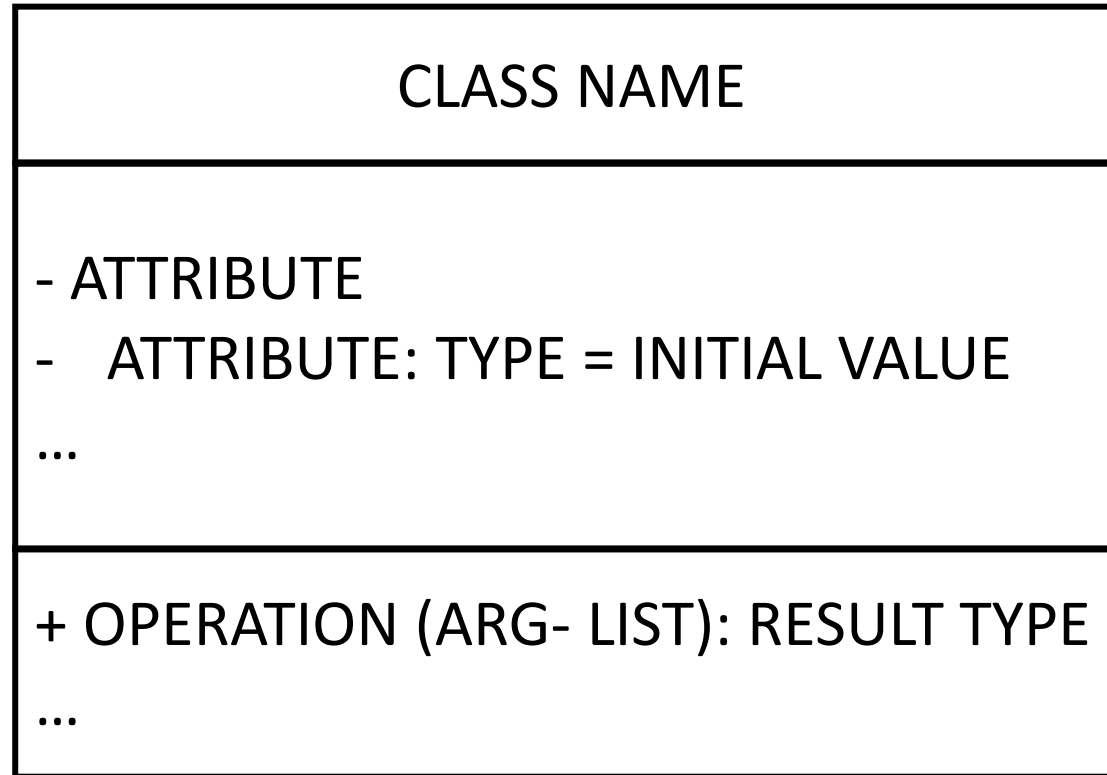
Design Class Diagram

Static, Structural View of the System

Describes

Classes and their Structure
Relationships among classes

Class Diagram: Class



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



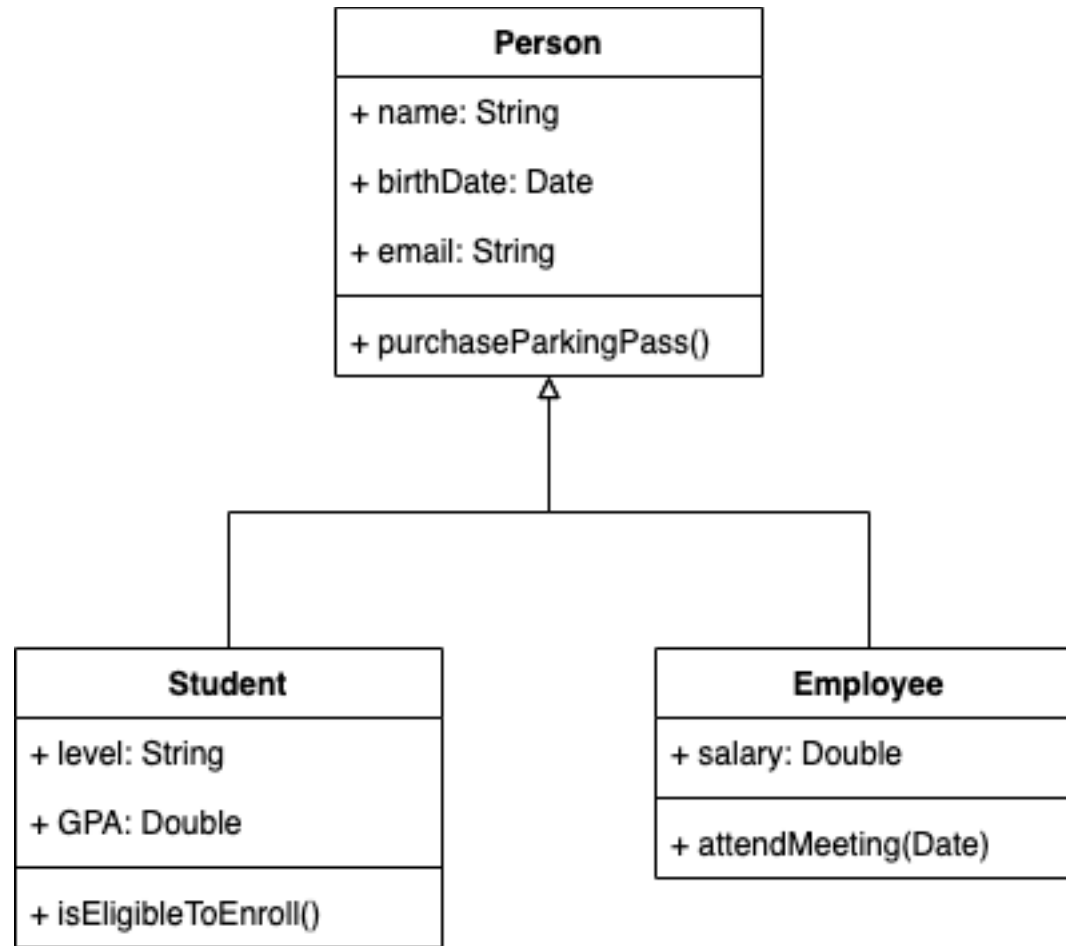
The relationship between the whole and the
part (parent and child), but the part can exist
independent of the whole

Composition: X **has** a Y

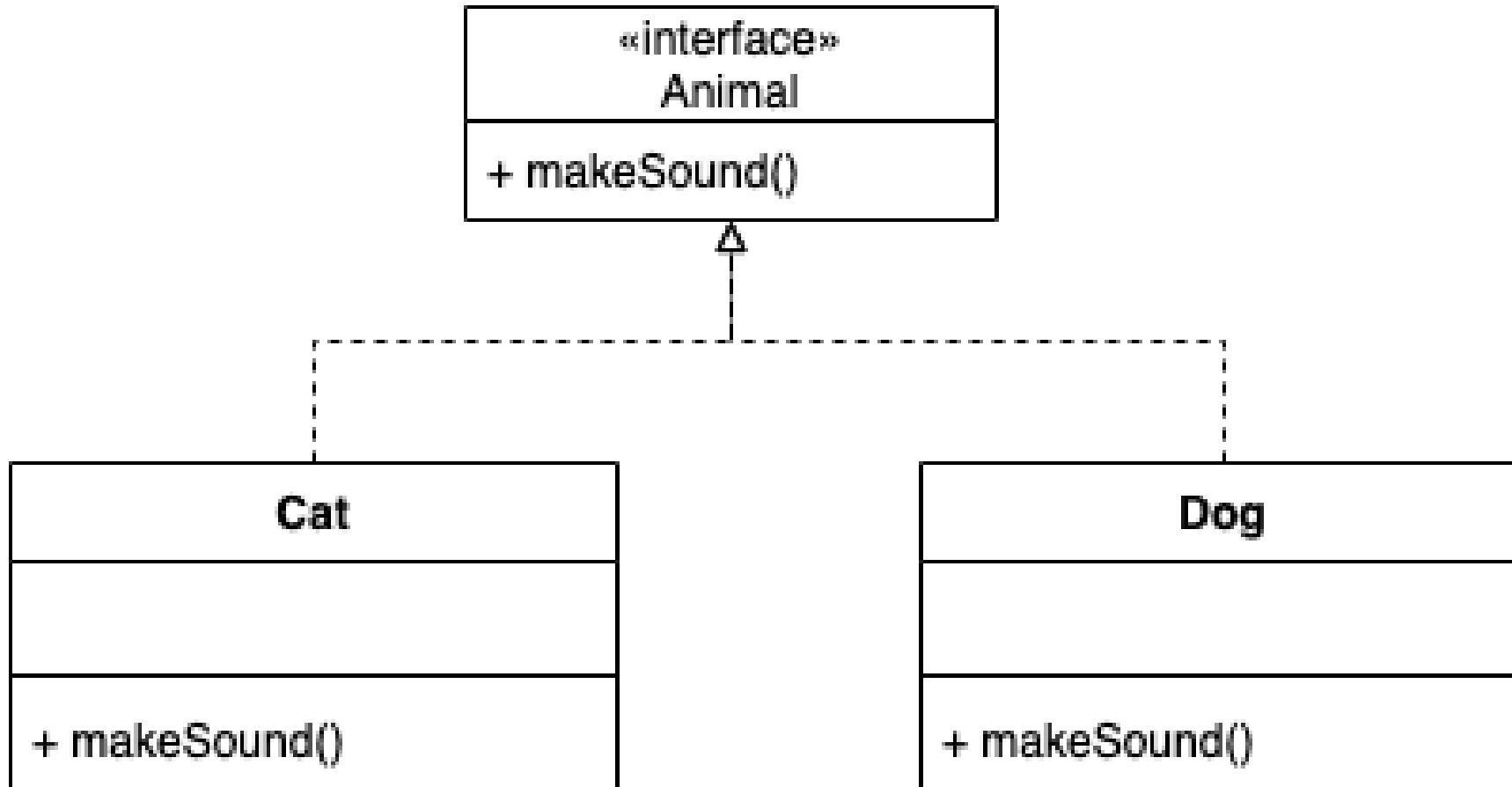


The relationship between the whole and
the part, but the part cannot exist
independent of the whole.

Generalization Example

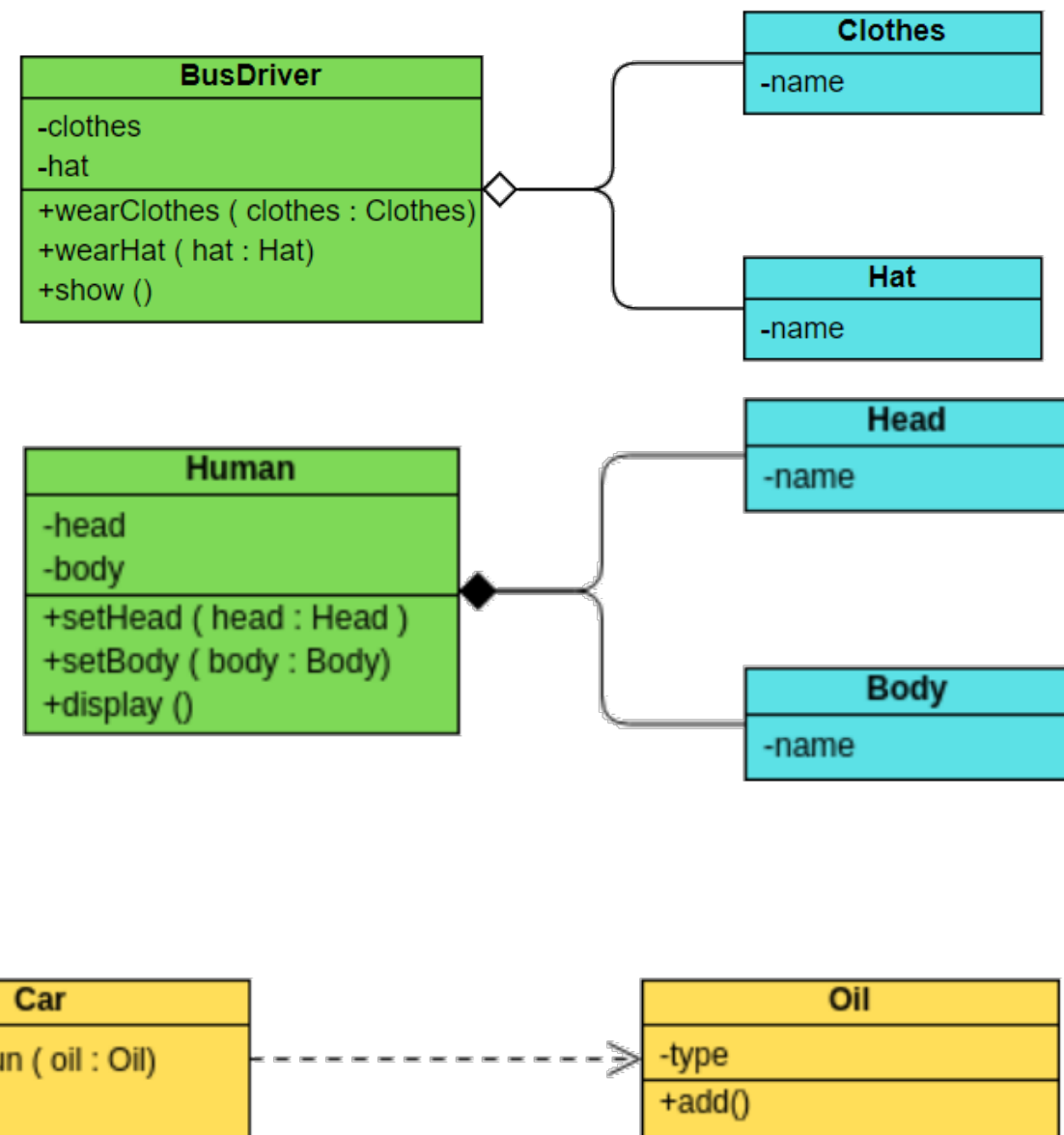


Generalization (Realization) Examples: Interface



Association Types

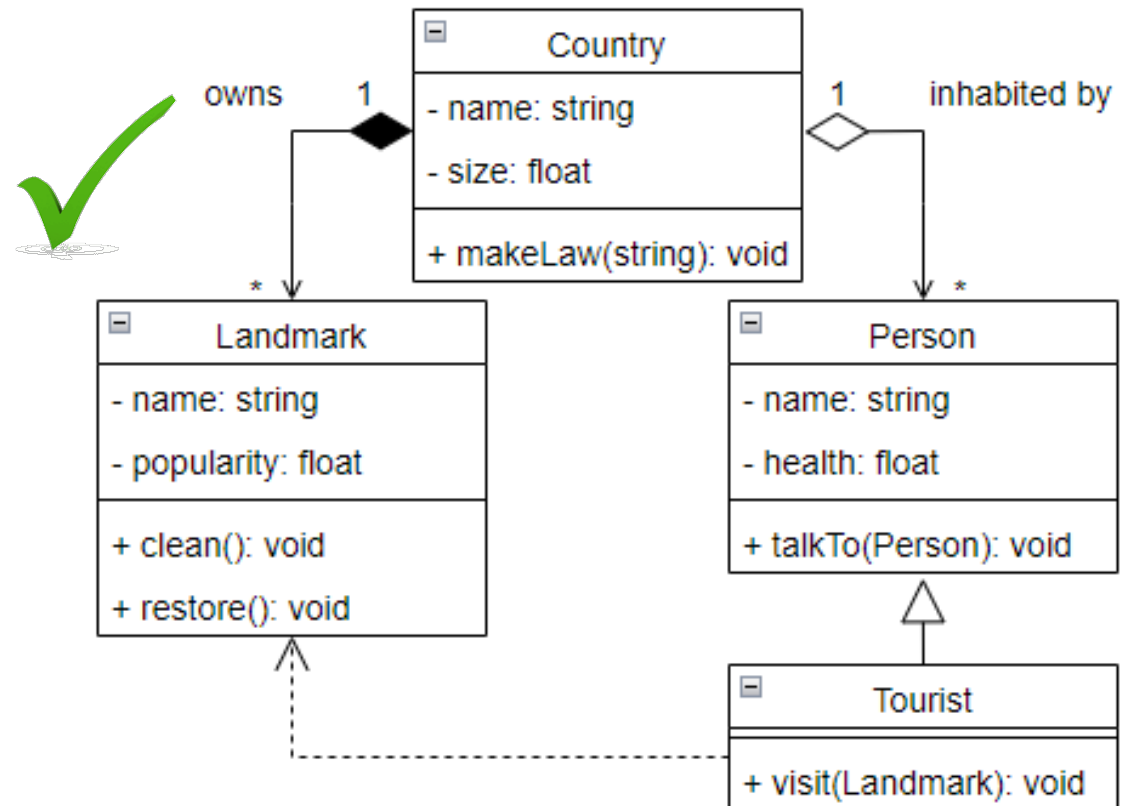
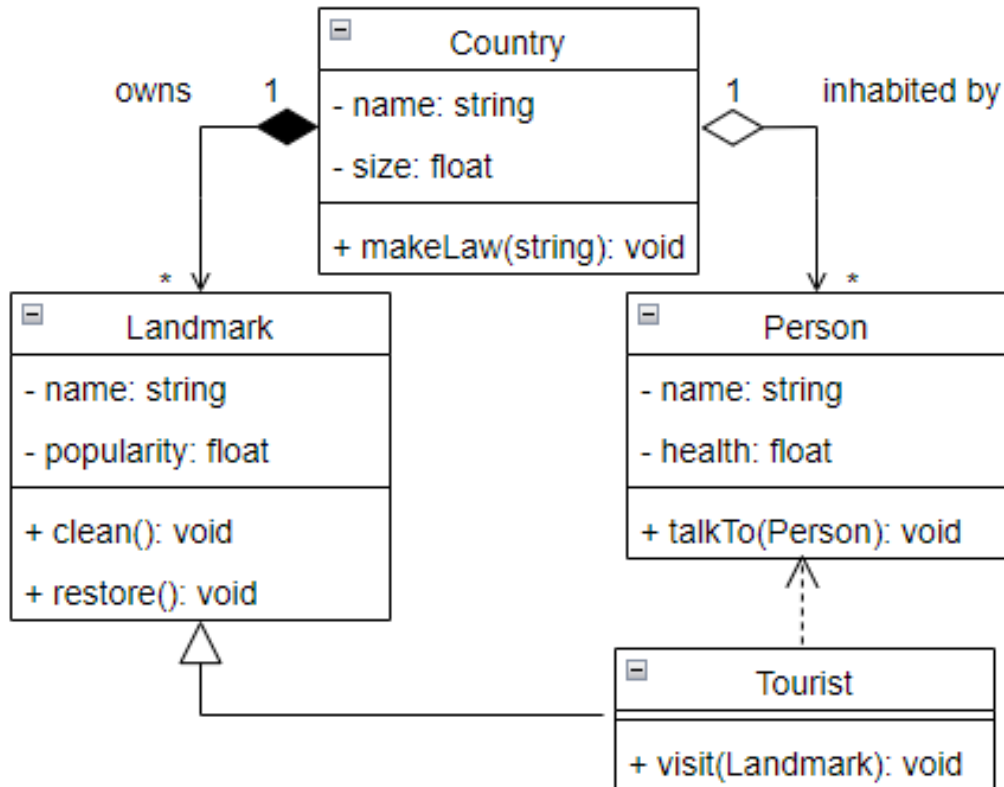
- aggregation: “is part of”
 - child can exist independently of the parent
- composition: “is entirely made of”
 - stronger version of aggregation
 - the parts live and die with the whole
 - child cannot exist independent of the parent
 - symbolized by a black diamond
- dependency: “uses temporarily”
 - symbolized by dotted line
 - often is an implementation detail, not an intrinsic part of that object's state

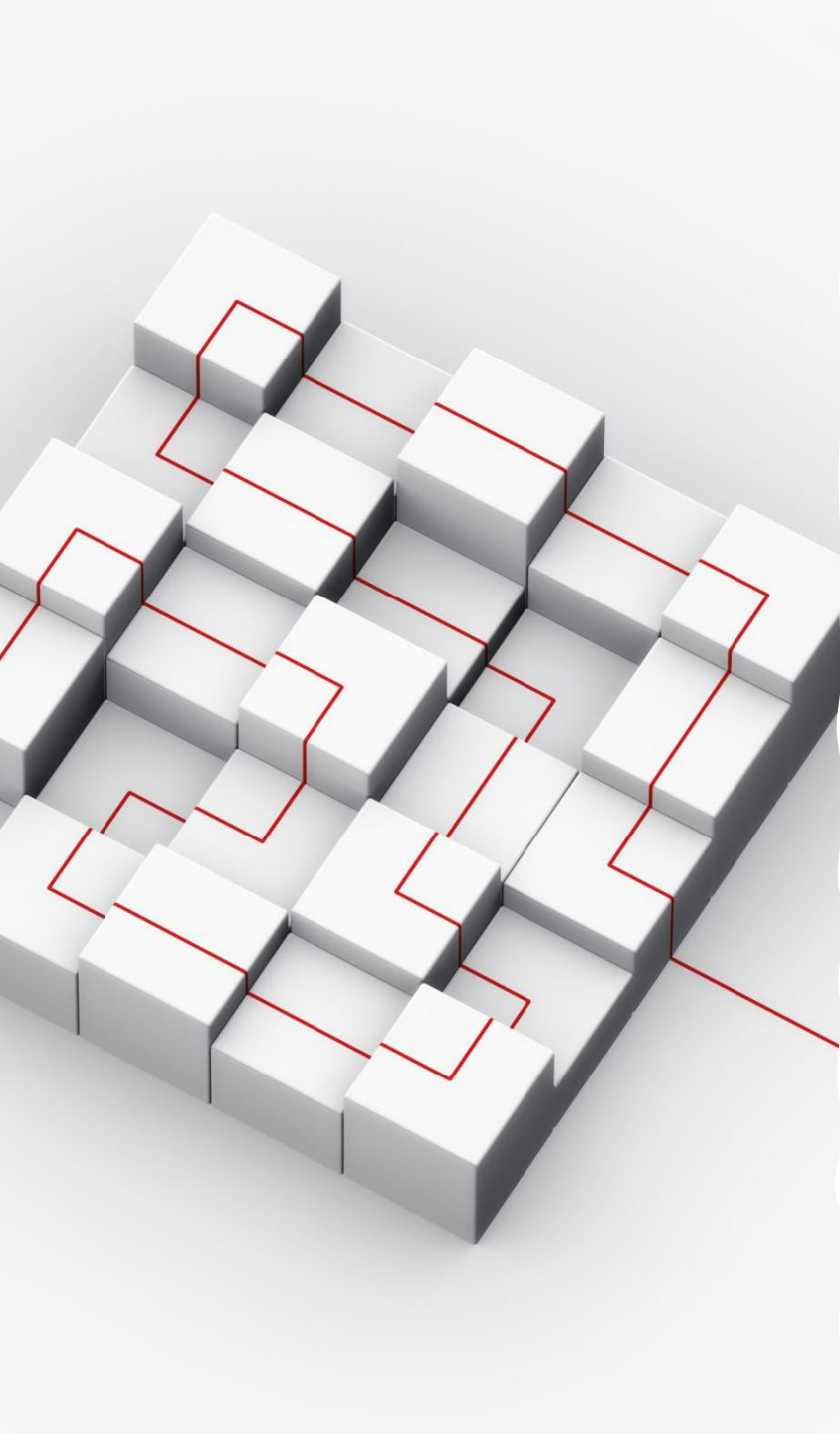




Example Quiz

Scenario: You are developing an application that simulates public travel trends. Each country in the system has a name, size, and can write laws. Each country also owns different landmarks, which have different names and popularity levels. The landmarks must be cleaned and restored over time, and **cannot move between countries**. Countries are also inhabited by people, who have a name and health level. They are able to talk to other people. Tourists are a type of person, with the additional ability to visit landmarks. Which is the correct DCD?





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

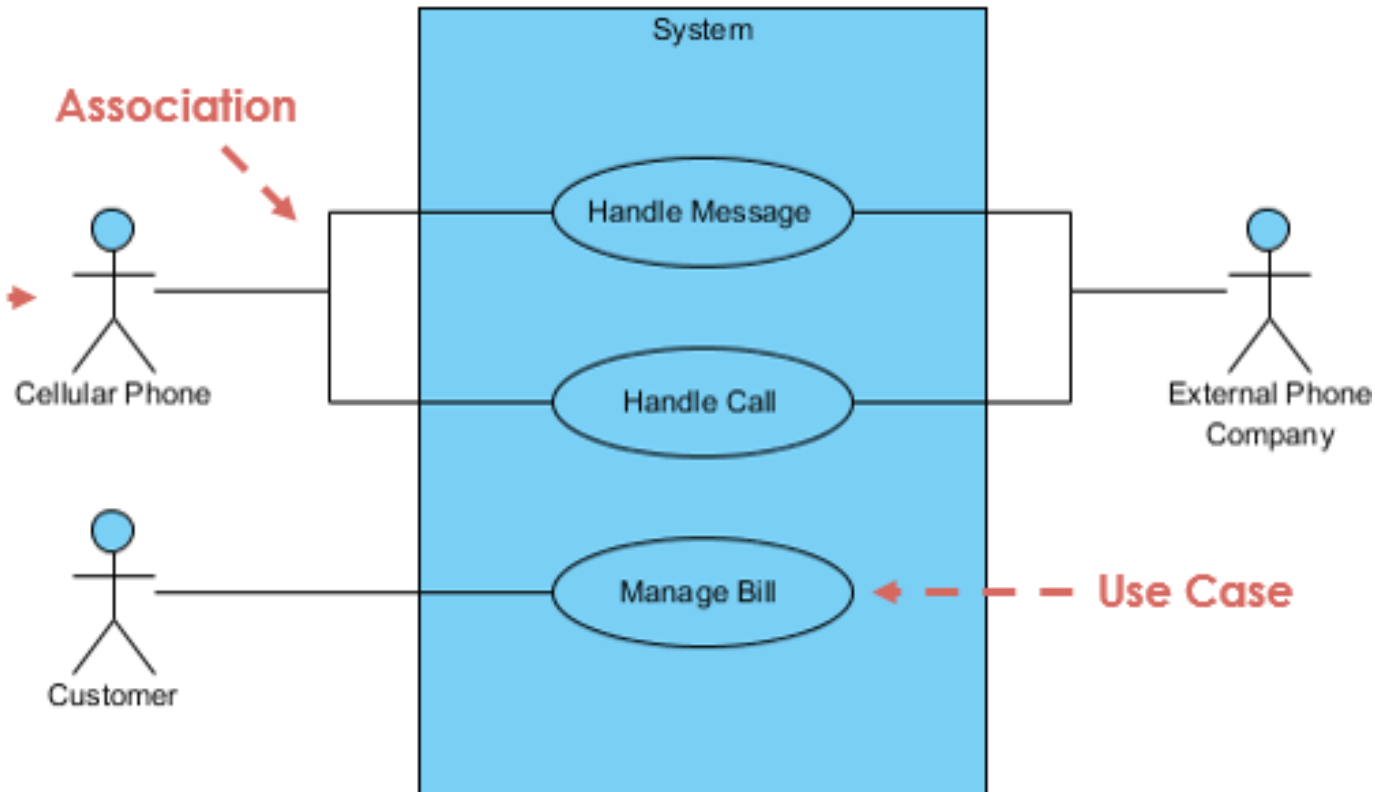
Primary actors to the left

System Boundary

Supporting actors to the right: they provide a service.

Actor - - - - ->

Association



Use Case

Offstage actor towards the bottom

Use Case Relationships

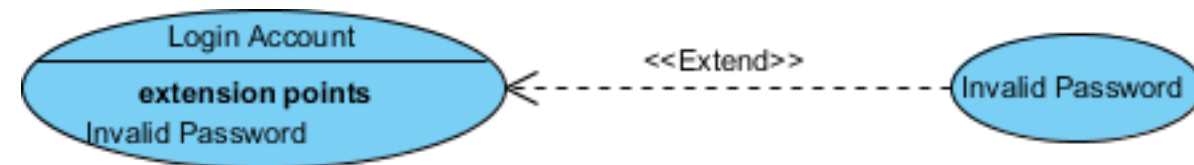
<<Include>>

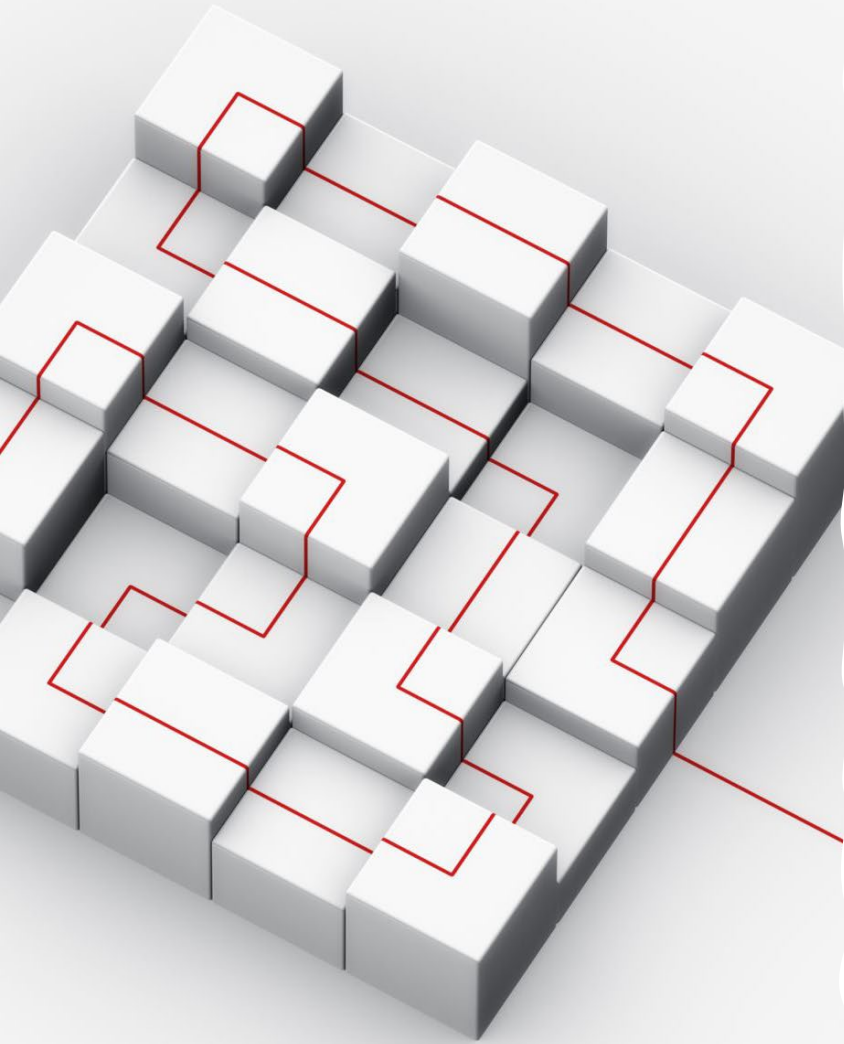
- Used when one use case is **used by** another use case
- Typically used if a sub-use case or series of steps is used by **several use cases**
- A **uses** relationship from base use case to child use case indicates that an instance of the base use case will **include** the behavior in the child use 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

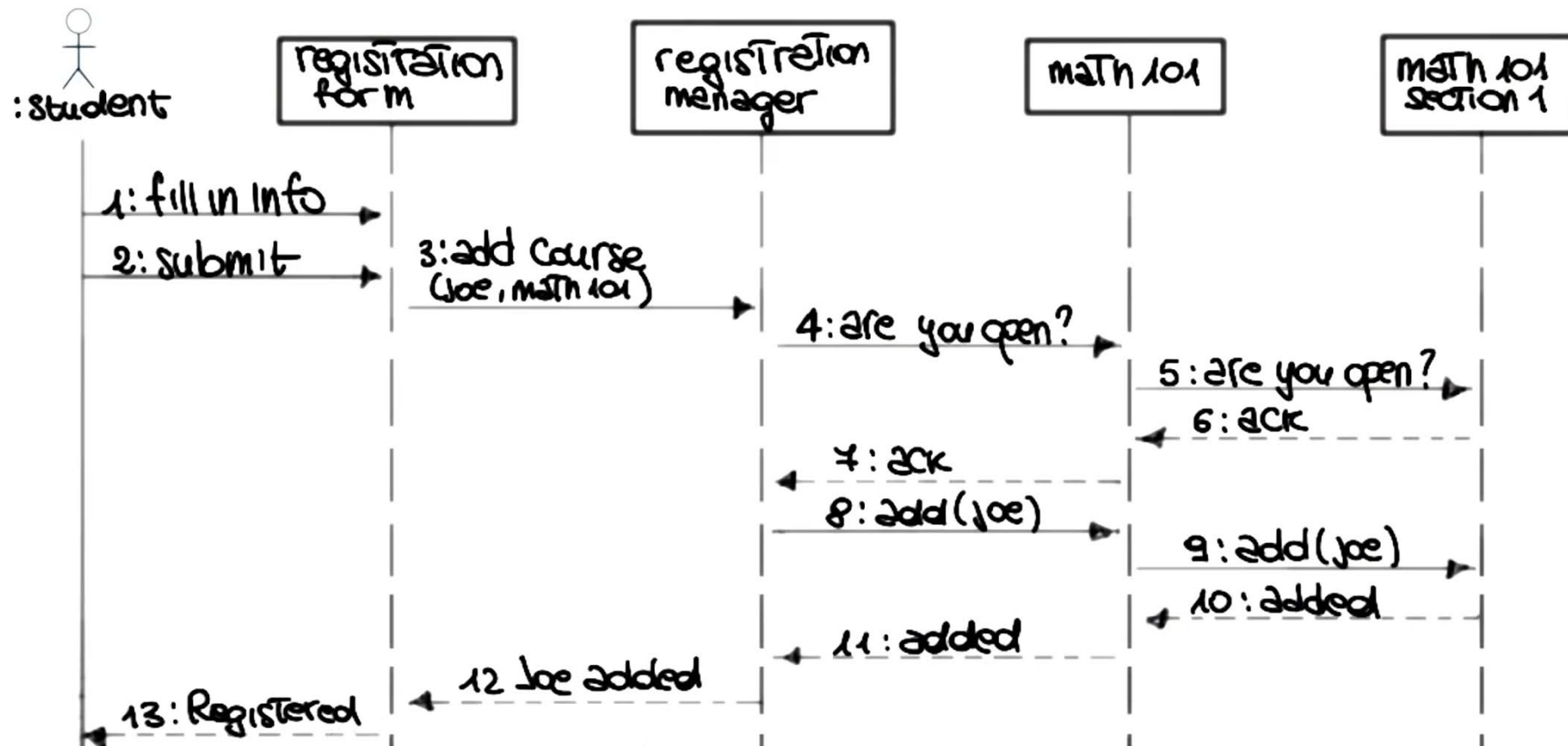





Low Level Design View – Behavioral Diagram: SD

Sequence Diagrams

Diagrams that emphasize time ordering of messages between classes/components





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



AI considerations

- guidance for using AI tools like Lucidchart to create UML diagrams on Canvas and website
- Progressive specific prompting may be needed
- Manual edits maybe needed
- For the assignment, in each of the 4 compulsory diagrams, you need to compare the manually generated output vs. the 2 AI tool-generated outputs using the following metrics - Usability/User-friendliness, Autocompletion Rate (how much of the generation was AI vs. manual tweaks to the AI-generated diagram), Setup Time/Learning Curve, Number of Prompt Iterations/Refinement, Total Time Taken.