Software Design and Analysis
CS-3004
Lecture# 12

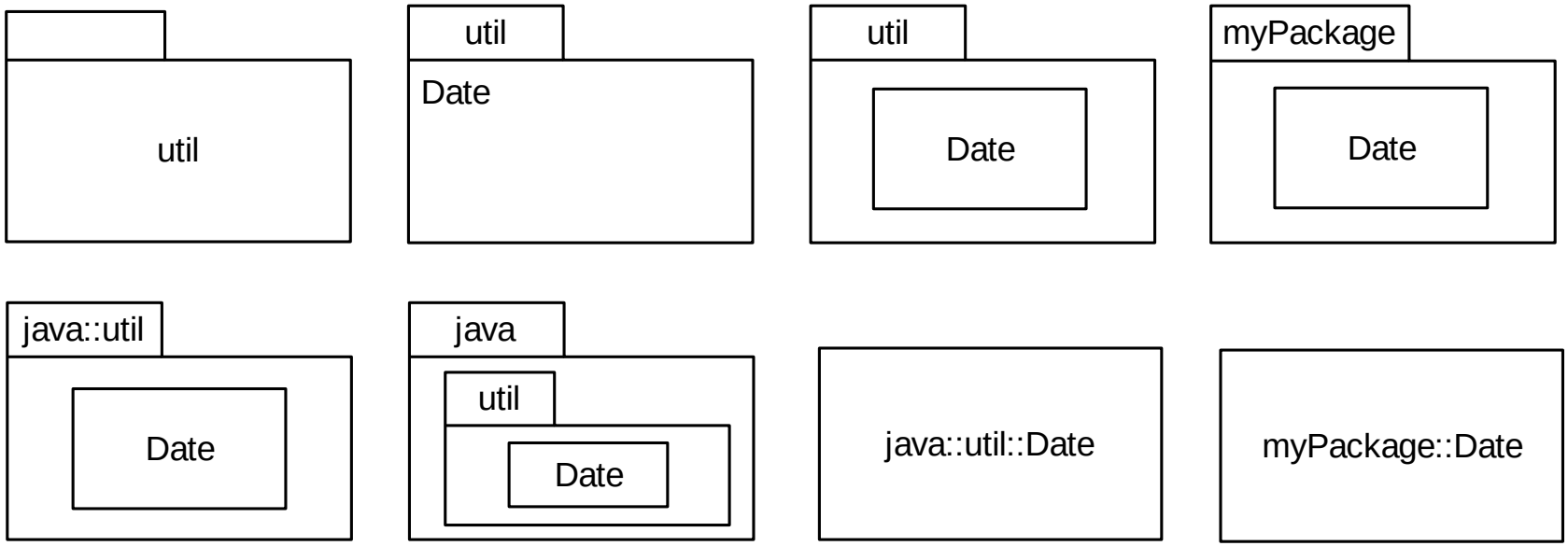
Dr. Javaria Imtiaz,
Mr. Basharat Hussain,
Mr. Majid Hussain

quest lab

Outline

- Package Diagram
- Component Diagram
- Deployment Diagram
- Activity Diagram

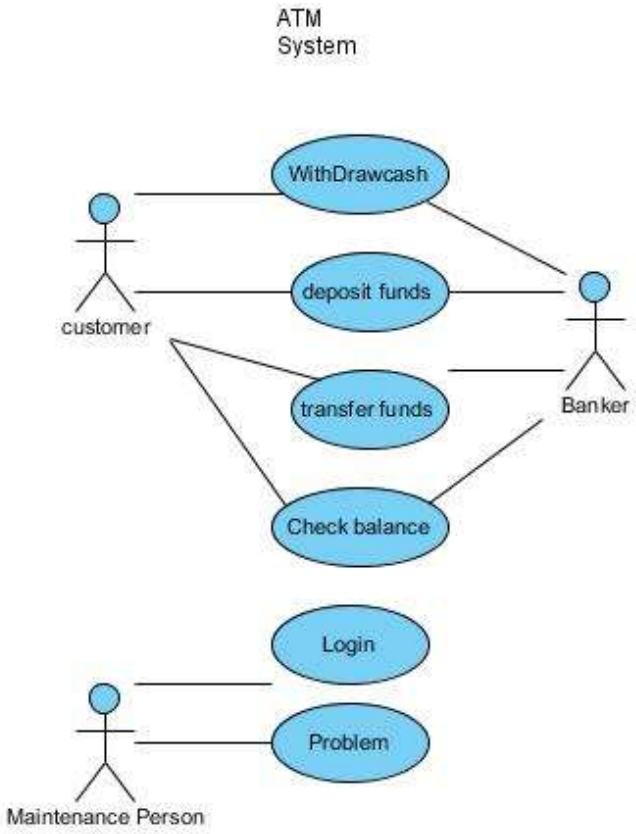
Packages

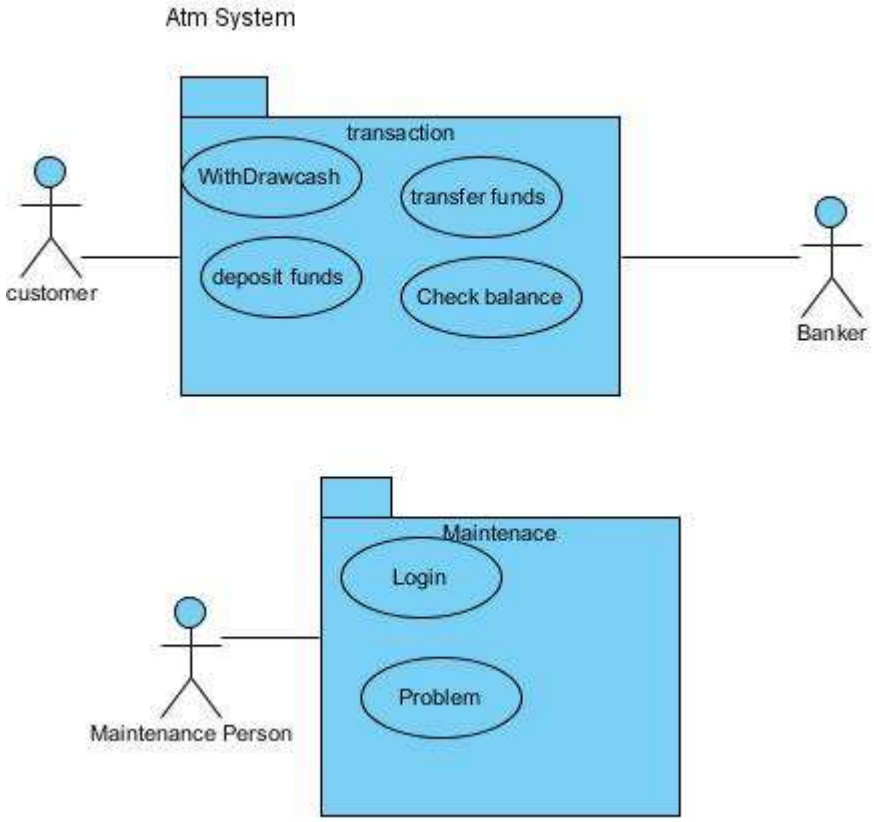

util
util
Date
util
Date
myPackage
Date
java::util
Date
java
util
Date
java::util::Date
myPackage::Date

Package

- Classes are basic structural units in an OO system
- In large systems with hundreds of classes, need to group the (related) classes together into *packages*
- A *package* in UML can be a collection of any packageable UML elements but it is most commonly a collection of classes
- A package may contain other packages (subpackages)
- A package in UML corresponds to a package in Java or a namespace in C++
- Each package is a namespace
  - There must never be more than one class within a package with any given name
  - Classes in different packages can have the same name
- If several teams working on project, each could work on a different package
  - Would mean they don't have to worry about name clashes
- To distinguish between classes with the same name in different packages, use *fully qualified name*
  - e.g., java::util::Date, myPackage::Date
- UML package icon is a tabbed folder
  - Can show just name or contents too
    - If just name shown, then can be written in the middle of the icon, otherwise name written on tab
  - Can show all details of class or even class diagram within package
  - At other extreme, can just list names of classes within the package icon

Use Case Package Diagram:

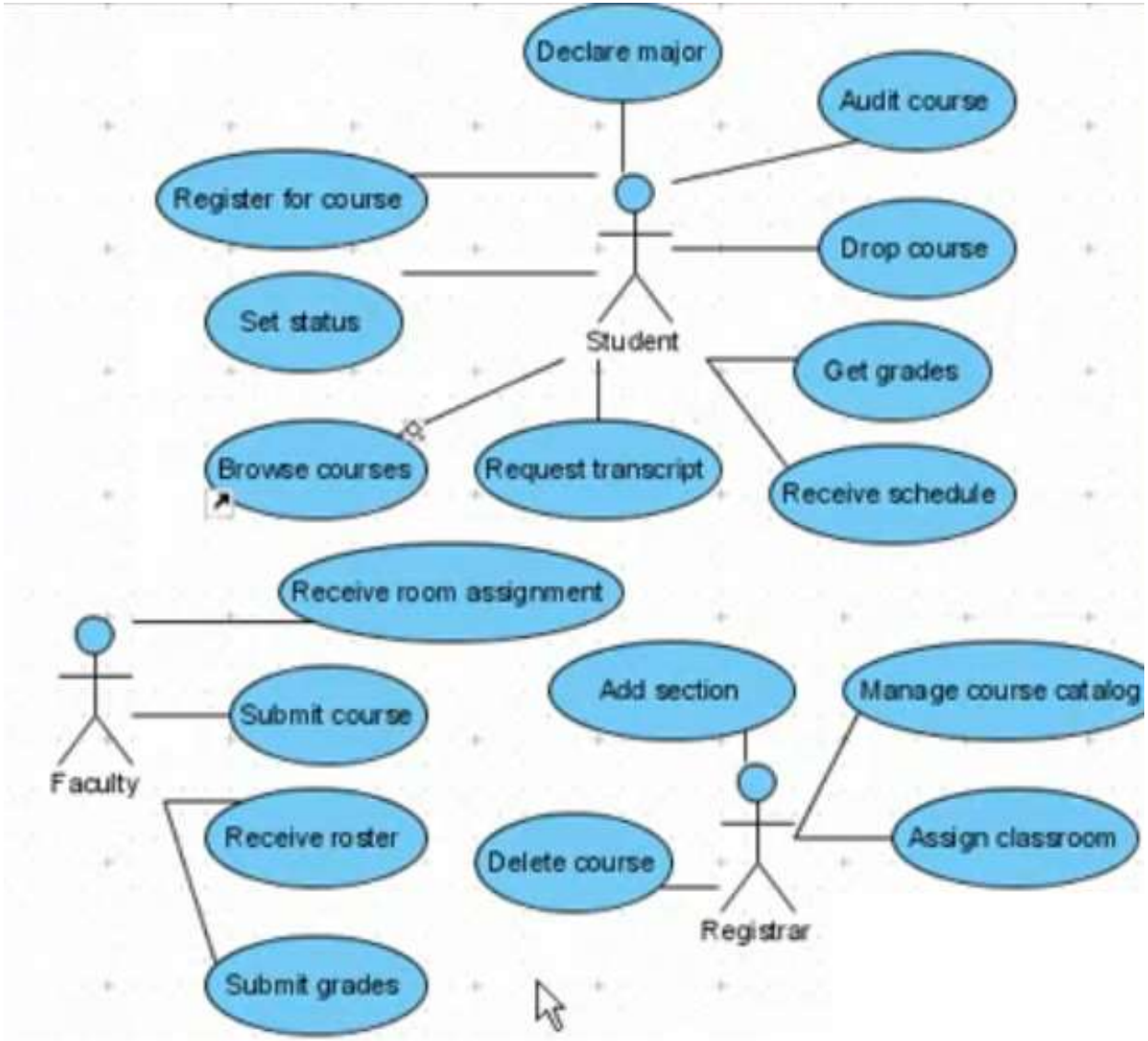




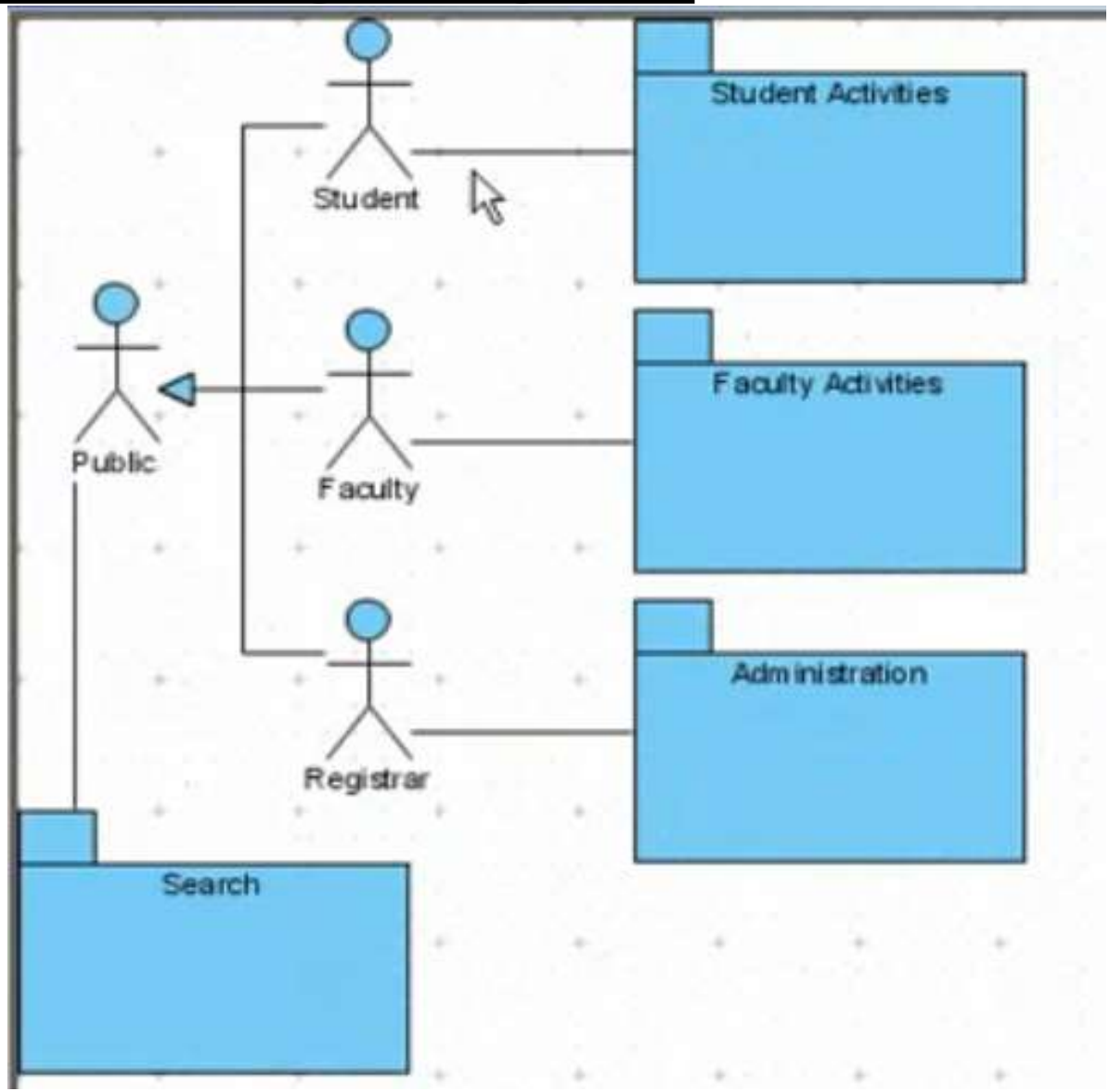
Use case Diagram ➔➔

Use case Package Diagram

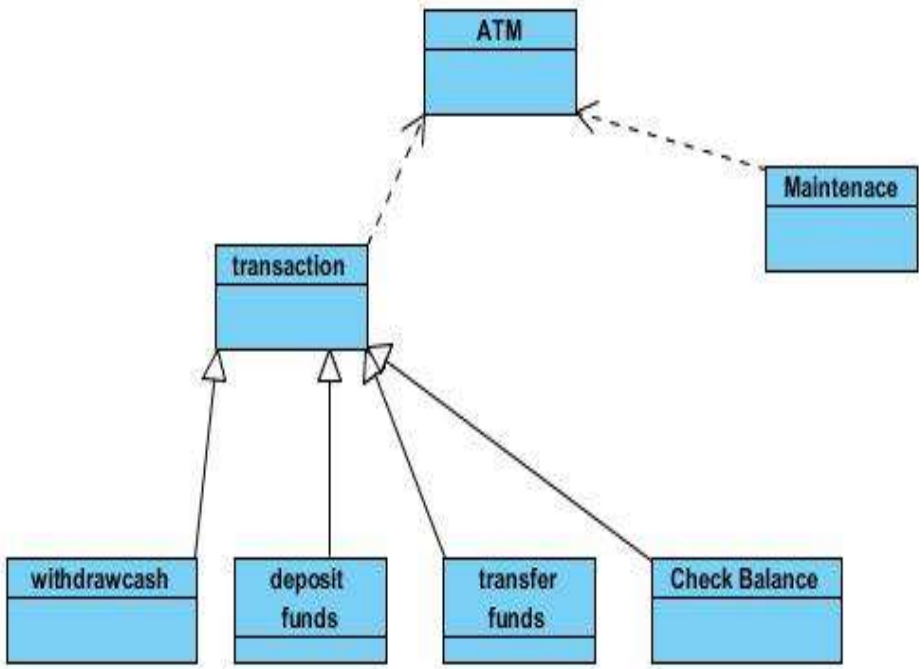
Use Case Package Diagram:

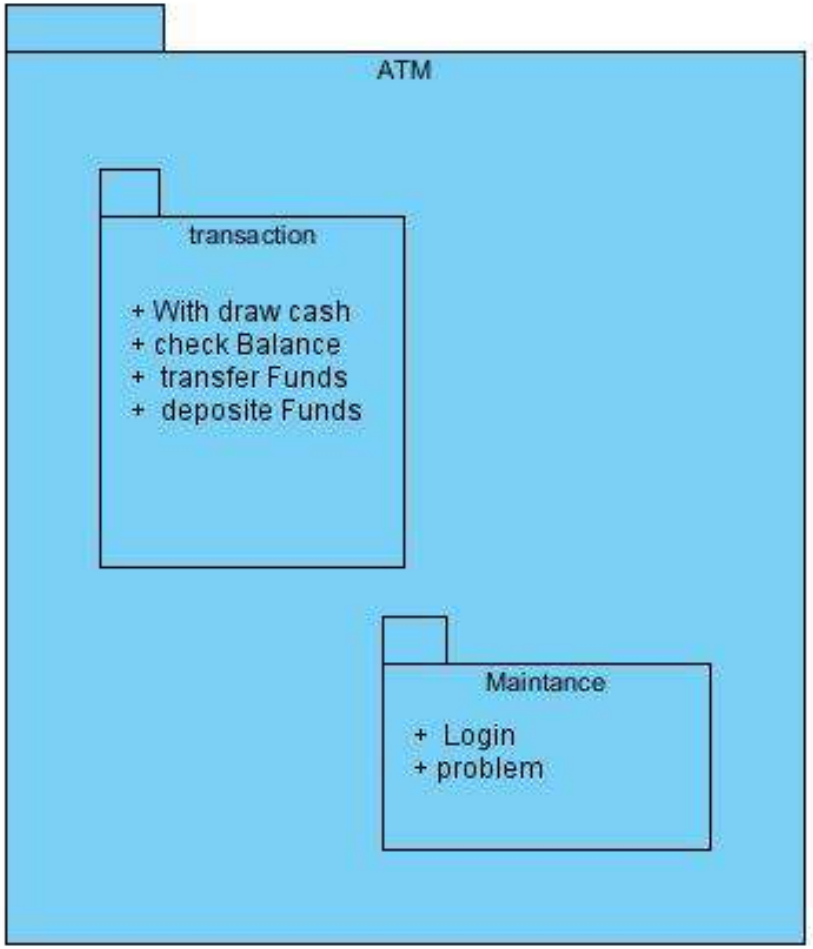


Use Case Package Diagram:



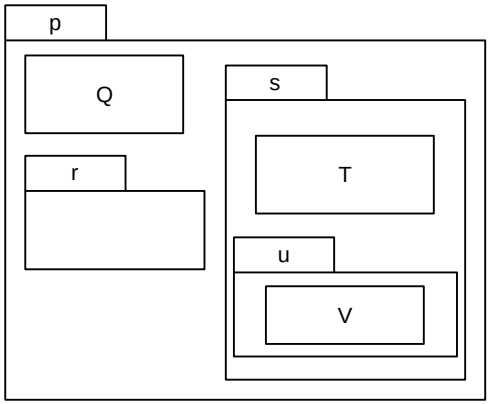
Class Package Diagram

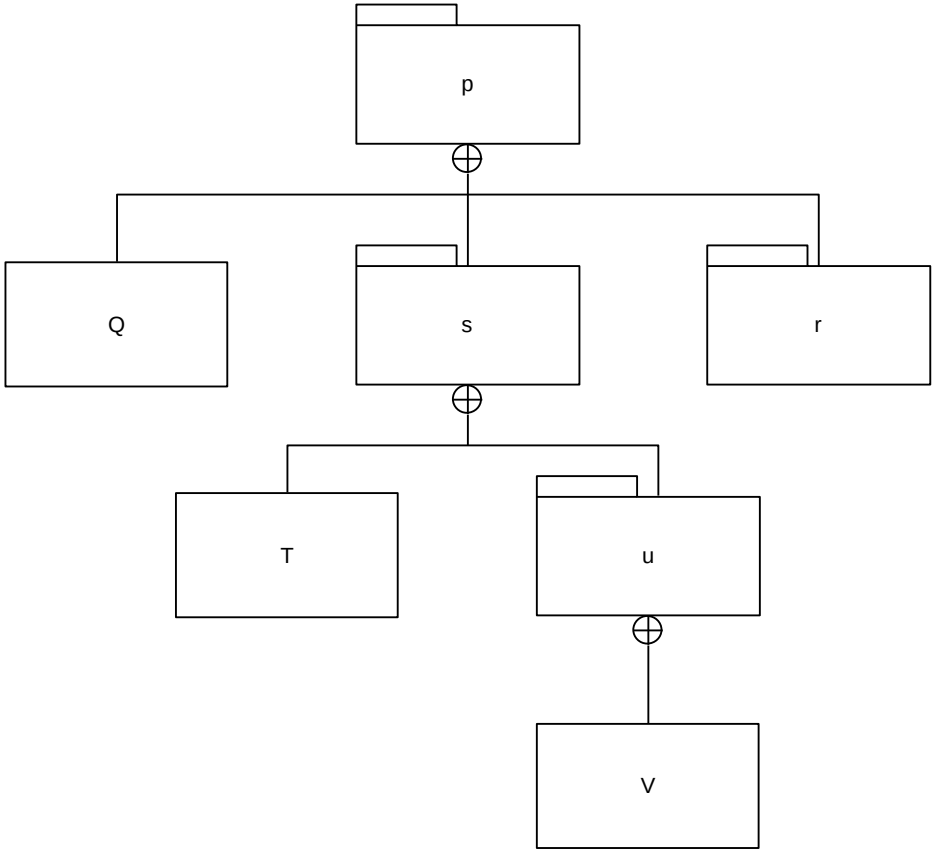




class Diagram ➔➔ class Package Diagram

Package membership



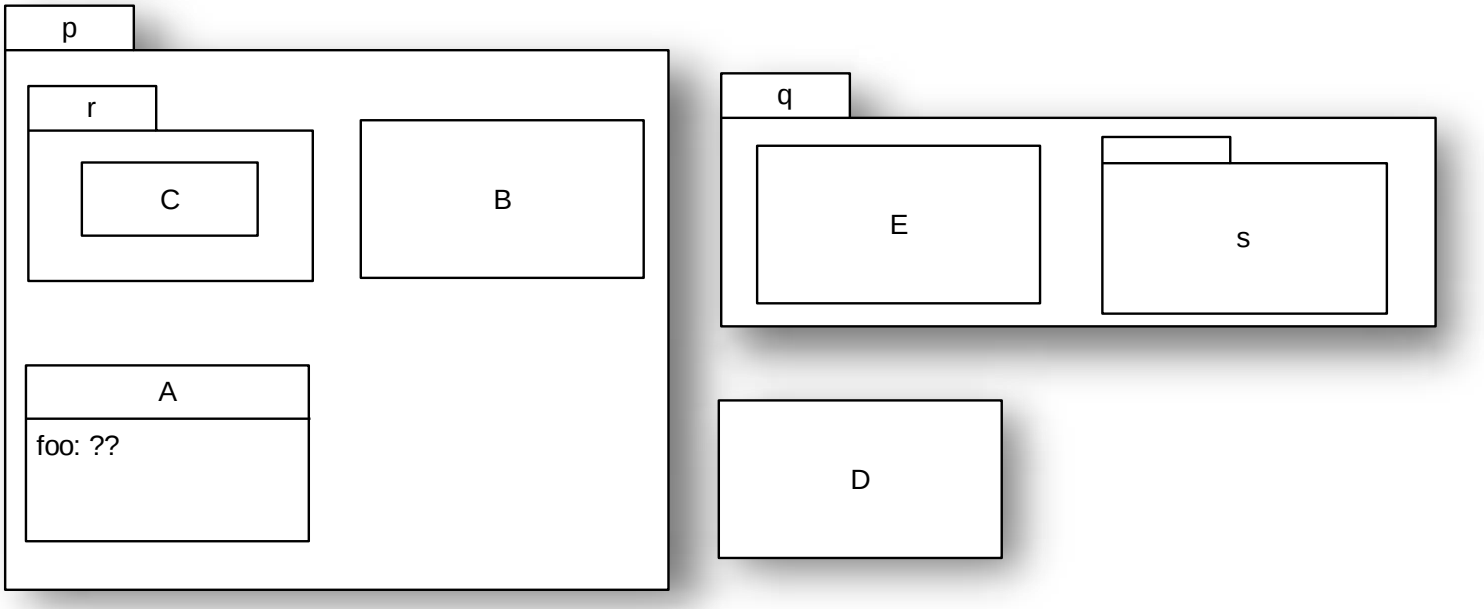


- These diagrams convey the same information

Class visibility and facades

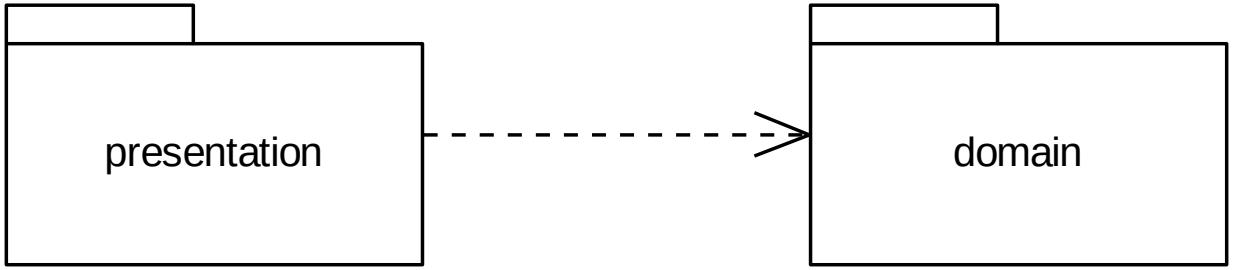
- Classes inside a package can be public or private
  - Indicate by preceding class name with + or -, respectively
- Public class is part of package's public interface
  - Can be used by classes in other packages
- Private class is hidden within package
  - Can only be used by other classes within the package
- Can control access to a package by making all classes **private** and then adding special, **public *façade* classes** that delegate public operations to the private classes within the package

Relationships between packages



- Element can refer to other elements that are in its own package and in enclosing packages without using fully qualified names.
- Element x must use fully qualified name to access element in package that does not contain x.
- foo can be of class B or D?
- foo can be of class q::E or r::C

Packages and dependencies



- Package diagram shows packages and their dependencies
- Package A depends on package B if A contains a class which depends on a class in B.
  - Inter-package dependencies summarise dependencies between classes

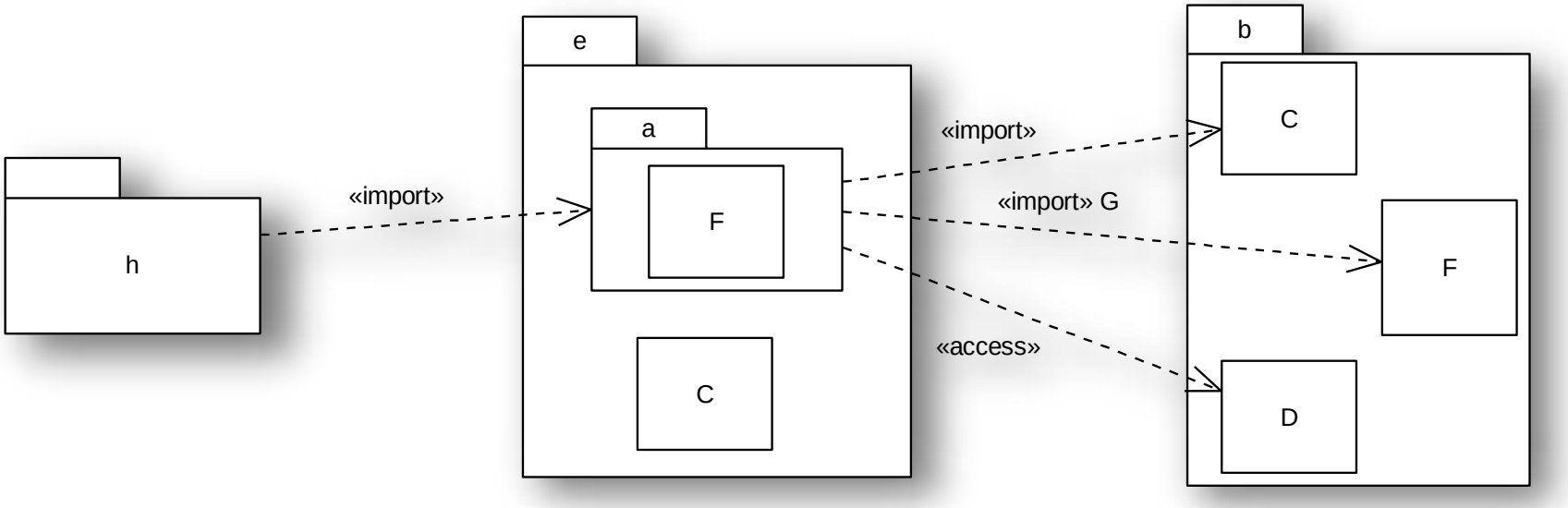
Package import

- Element import identifies an element in another package and allows the element to be referenced using its name **without a qualifier.**
- Element import indicated by dashed arrow with open arrowhead from importing package to imported element and labelling arrow with
  - Keyword <<import>>, if visibility of imported element within importing package is public and
  - Keyword <<access>>, if visibility of imported element within importing package is private

Import: Indicates that functionality has been imported from one package to another.

Access: Indicates that one package requires assistance from the functions of another package.

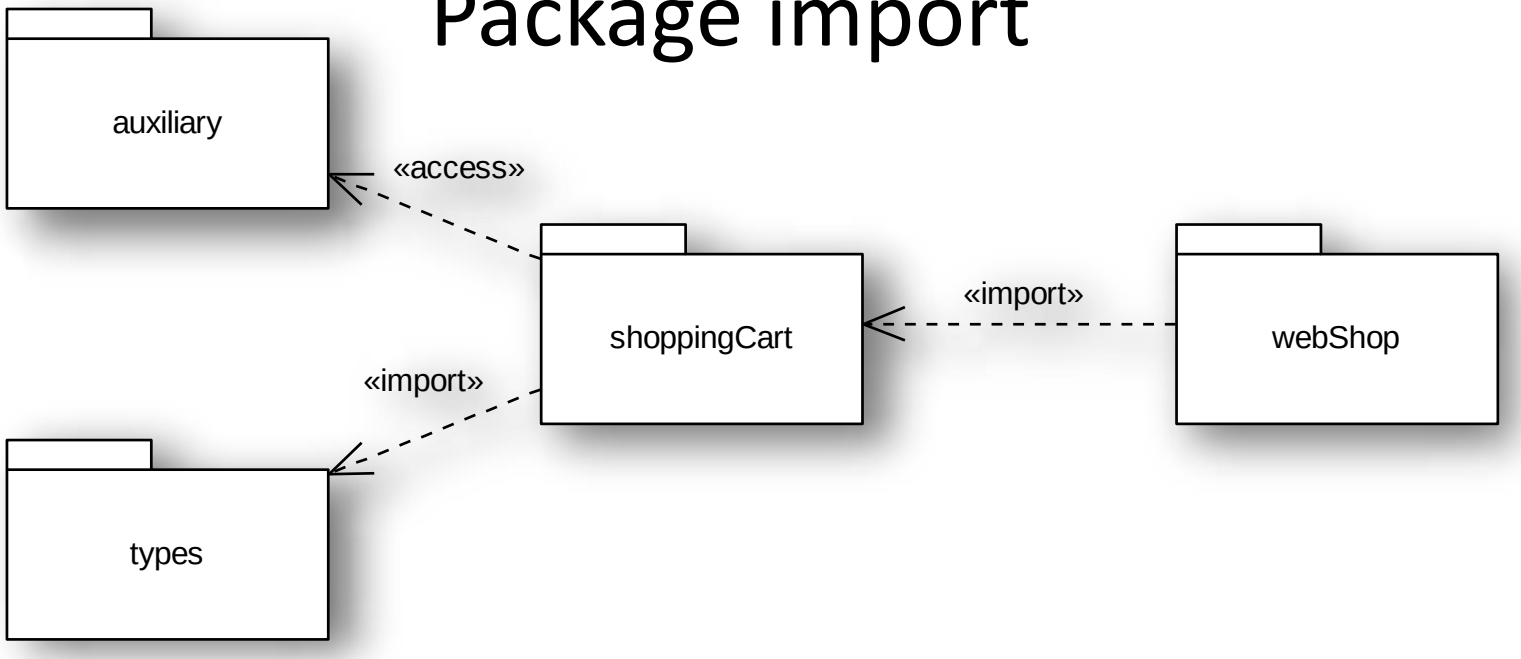
Element import



In above example

- Class b::C is referred to as just C in package a, and has public visibility within package a
- Imported class C hides outer class e::C which must be referred to by its fully qualified name (before import, C referred to e::C in package a)
- Class b::F is imported into package a, but there is already a class called F in a, therefore cannot import b::F without aliasing it
  - Class b::F is referred to as G in package a
- Imported class b::D can be referred to as D in package a, and has private visibility within package a
- Package h imports package a which means that, in h, b::C is referred to as C and b::F is referred to as G
  - b::D is not accessible from h because its visibility in a is private

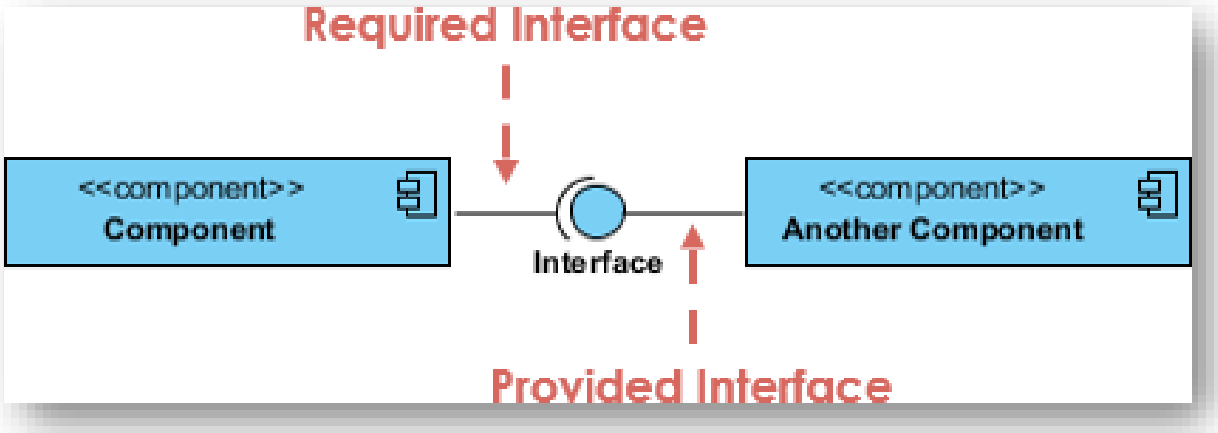
Package import



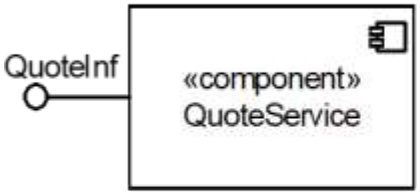
- A package import is a directed relationship that identifies a package whose members are to be imported by a namespace (package)
- Importing namespace adds names of members of imported package to its namespace
- Conceptually equivalent to having an element import to each individual member of the imported namespace
- Notated using dashed line with open arrowhead from importing namespace to imported package, labelled with keyword
  - <<import>> if package import is public and
  - <<access>> if package import is private
- If package import is public, then imported elements will be visible outside of importing package, while if it is private, they will not be visible
- In example above, elements in **types** are imported to **shoppingCart** and then further imported to **webShop**
- But elements of auxiliary only accessible from shoppingCart, not webShop

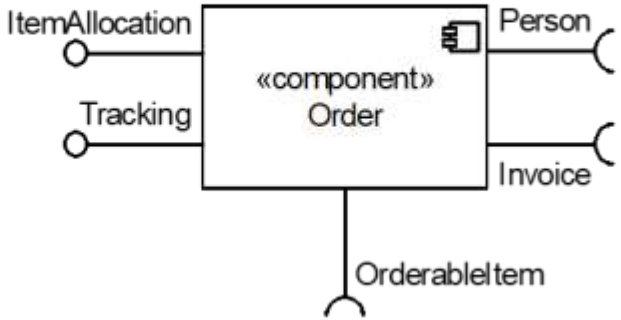
Components

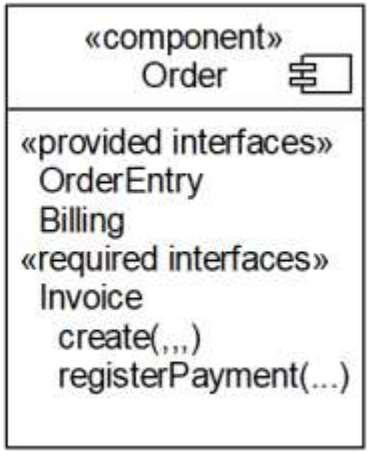
- UML defines a *component* to be
  - "a modular unit with well-defined interfaces that is replaceable within its environment" (UML Superstructure Specification, v.2.0, Chapter 8)
- Component-based design emphasises reuse
  - Component is an autonomous unit within a system
- Component defines its behaviour in terms of provided and required interfaces
  - A component may be replaced with another if they both provide and require the same interfaces
- In UML, a component is a special type of class
  - However, a component will often be a collection of collaborating classes
- A component can therefore have attributes and operations and may participate in associations and generalizations
- A component *provides* interfaces that it realizes and exposes to its environment
- A component may require interfaces from other components in its environment in order to be able to provide all its functionality
- *External* or "Black box" view on a component considers its publicly visible properties and operations
- *Internal* or "White box" view on a component considers its private properties and *realizing classifiers* (i.e., the classes and other elements inside the component) and shows how the behaviour of the component is realized internally


Required Interface
<<component>>
Component
Interface
<<component>>
Another Component
Provided Interface

Component notation



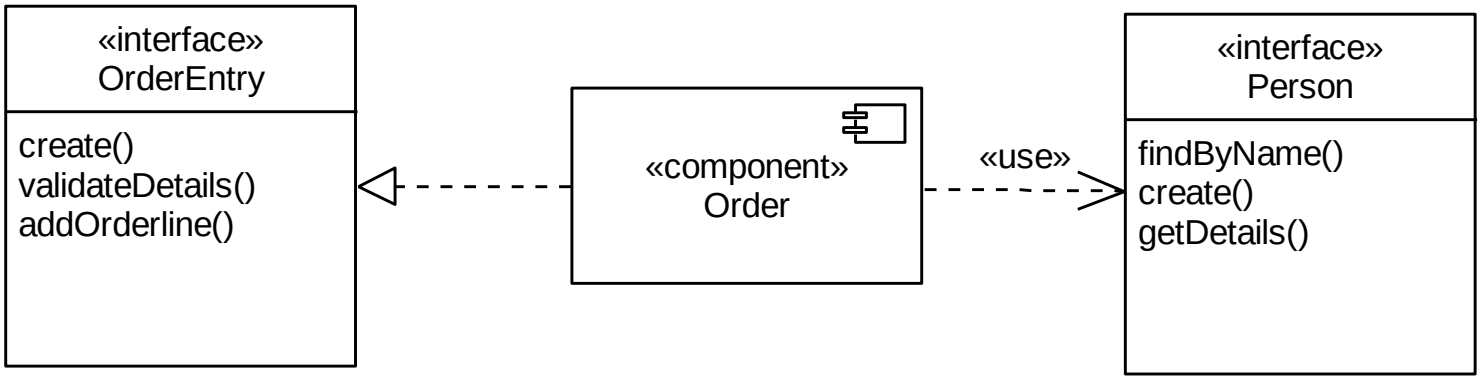




- Component icon is a class icon with the keyword <<component>>
  - Optionally can include a component icon in the top, right-hand corner

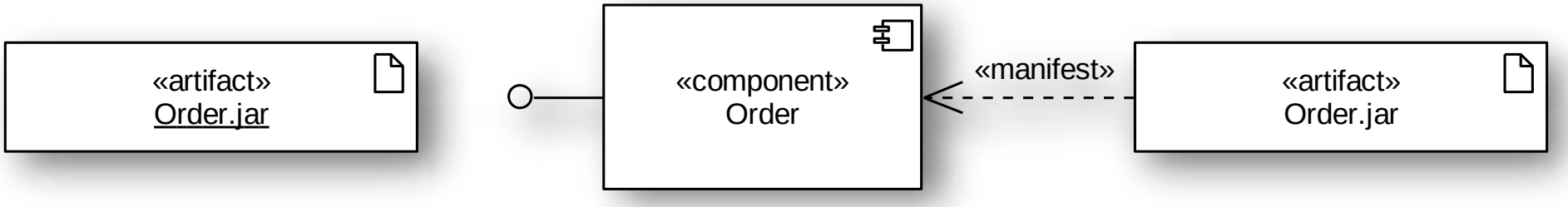
- Black-box view shows only interfaces provided and required either using "ball-and-socket" notation or listed in a compartment of the component rectangle

Internal, white-box view on components



- Interfaces can also be displayed in full using class icons, use
  - Dashed arrow with white triangle head for provided interfaces and
  - Dashed arrow with open arrowhead with keyword <<use>> for required interfaces

Artifacts



- An artifact is a concrete element in the physical world
- An *artifact* is the specification of a physical piece of information such as a binary executable, a database table or an implemented component such as a DLL or a Java class file
- Each artifact has a filename
- An artifact is represented by a normal class rectangle with the keyword <<artifact>> or an artifact icon in the top right corner
- Name of artifact may (optionally) be underlined
- Artifact is said to *manifest* model elements that are used to construct the artifact
  - Manifestation indicated by dependency arrow with keyword <<manifest>>

Component Diagram – another example

(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)

Component Diagram – another example

What is a Deployment Diagram?

- Deployment Diagram – a diagram that shows the physical relationships among software and hardware components in a system
  - Components – physical modules of code
  - Connections – show communication paths
  - Dependencies – show how components communicate with other components
  - Nodes – computational units, usually a pieces of hardware

A deployment diagram is a UML diagram type that shows the execution architecture of a system, including nodes such as hardware or software execution environments, and the middleware connecting them.

- A deployment diagram is a UML diagram type that shows the execution architecture of a system, including nodes such as hardware or software execution environments, and the middleware connecting them.

- Deployment diagrams are typically used to visualize the physical hardware and software of a system.

- The term Deployment itself describes the purpose of the diagram. Deployment diagrams are used for describing the hardware components, where software components are deployed. Component diagrams and deployment diagrams are closely related.

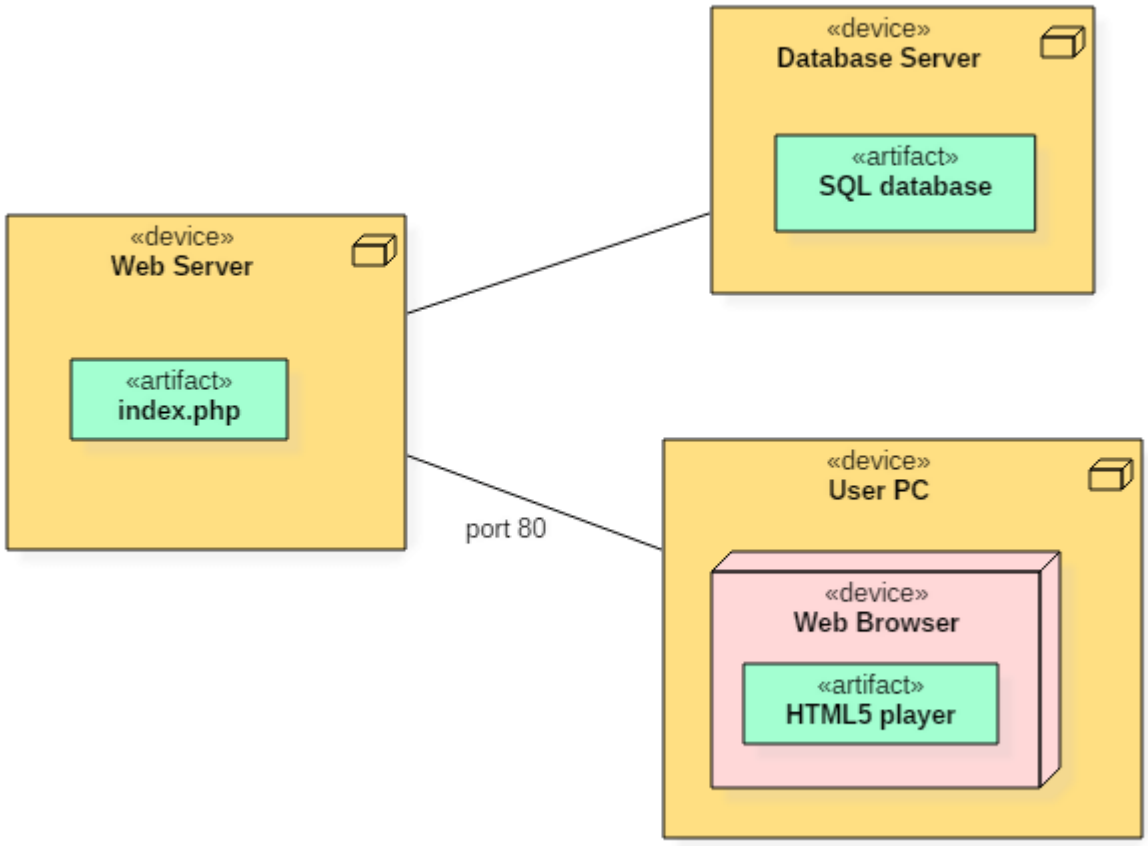
Purpose

- Component diagrams are used to describe the components and deployment diagrams shows how they are deployed in hardware.

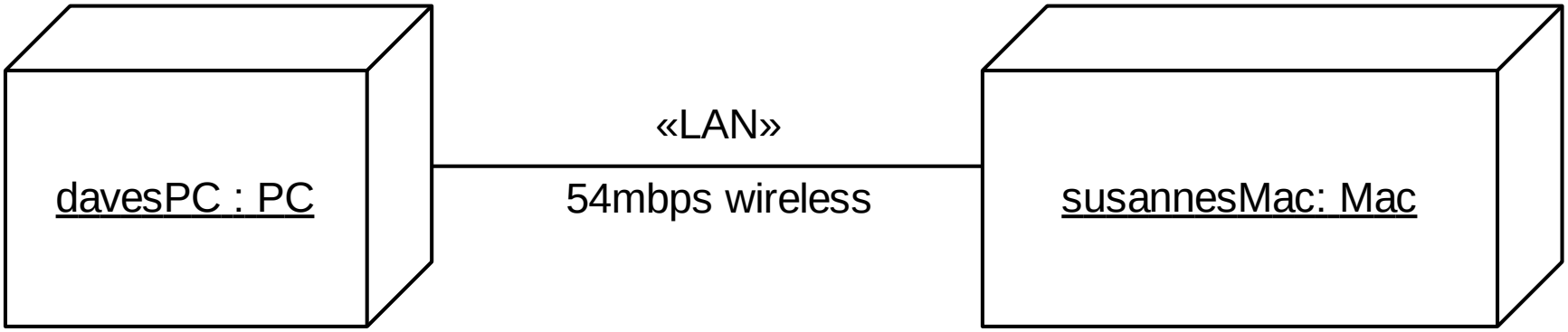
- UML is mainly designed to focus on the software artifacts of a system. However, these two diagrams are special diagrams used to focus on software and hardware components.

- Most of the UML diagrams are used to handle logical components but deployment diagrams are made to focus on the hardware topology of a system. Deployment diagrams are used by the system engineers.

- The purpose of deployment diagrams can be described as −
  - Visualize the hardware topology of a system.
  - Describe the hardware components used to deploy software components.
  - Describe the runtime processing nodes.

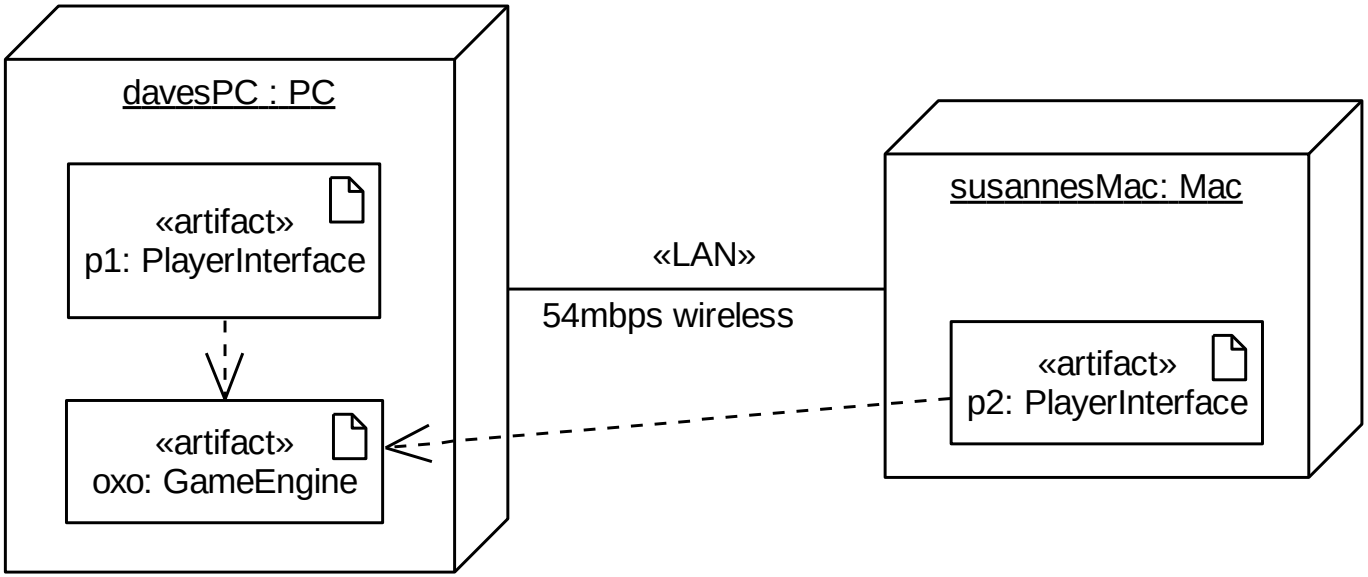

«device»
Database Server
«artifact»
SQL database
«device»
Web Server
«artifact»
index.php
port 80
«device»
User PC
«device»
Web Browser
«artifact»
HTML5 player

The physical layer



- Deployment diagram shows
  - The physical communication links between hardware items *(nodes)* (e.g., pcs, printers)
  - The relationships between physical devices *(nodes)* and processes *(artifacts)*
- Physical layer consists of the machines, represented by *nodes*, and the (physical) connections between them (e.g., cables), represented by *associations*
- Nodes have *node types*

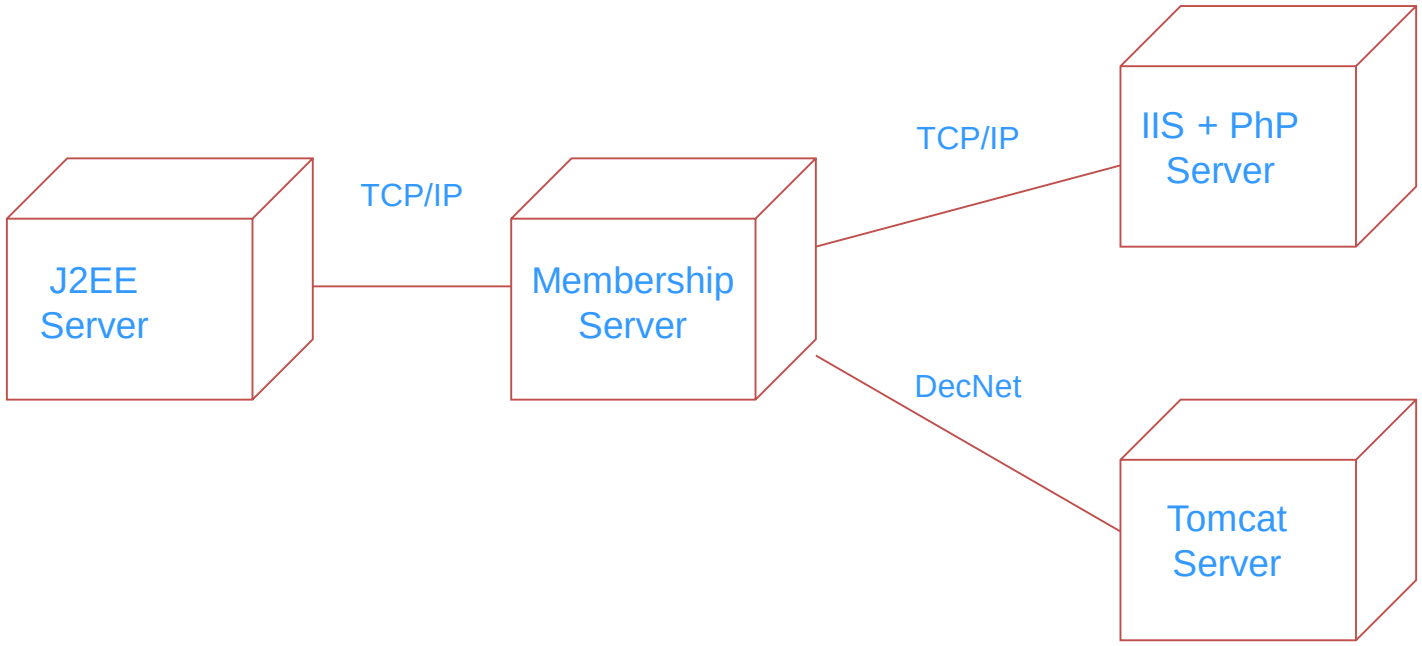
Deploying the software artifacts on the hardware nodes



- Artifact shown inside a node to show that it runs on the node
- If an artifact depends on another artifact then there must be a physical link between the nodes on which they are deployed

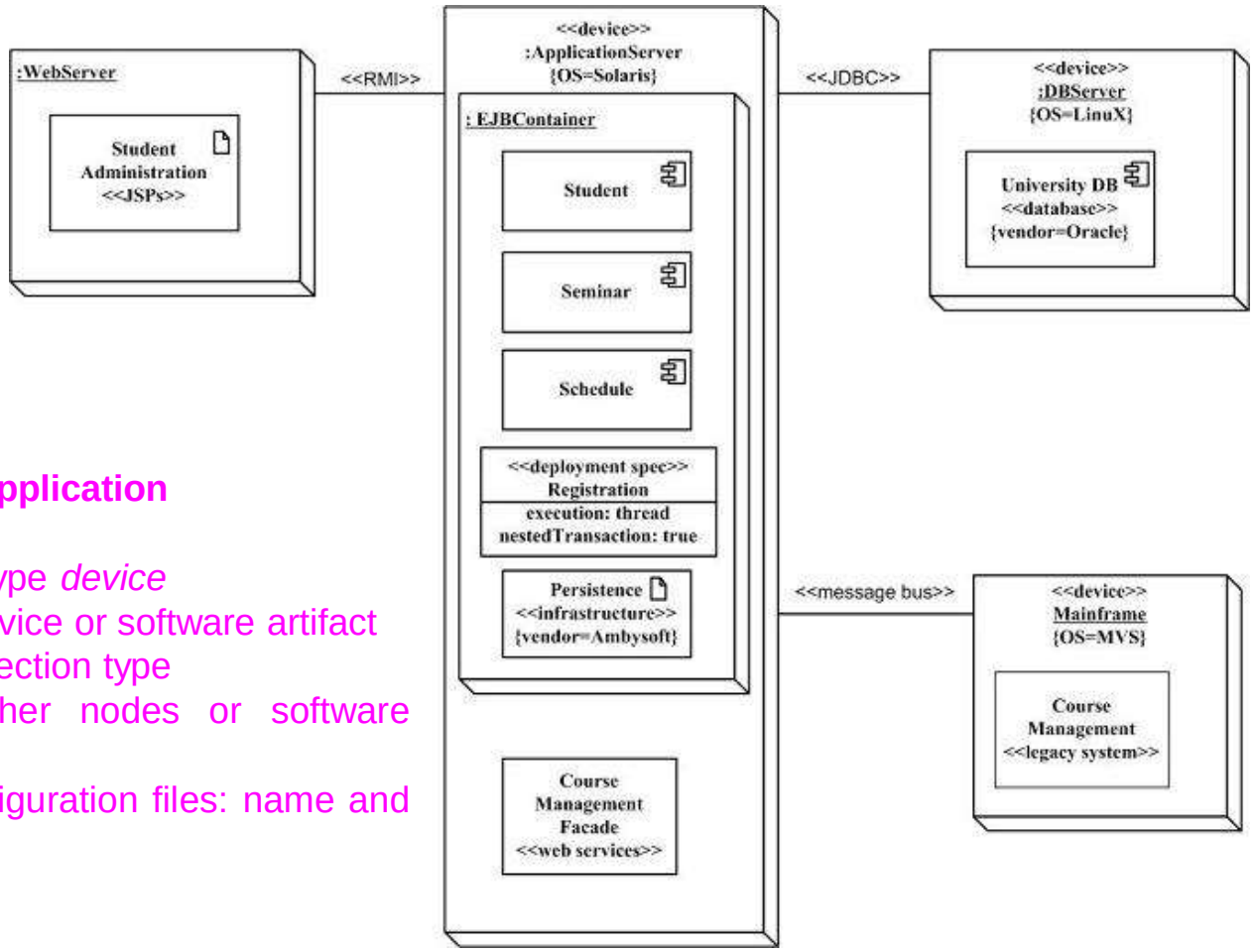
Deployment Diagram

- Shows a set of *processing* nodes and their relationships.
- Represents the static deployment view of an *architecture*.
- Nodes typically enclose one or more components.



Structural Diagrams - Deployment Diagram

http://www.agilemodeling.com/artifacts/deploymentDiagram.htm)



Student administration application

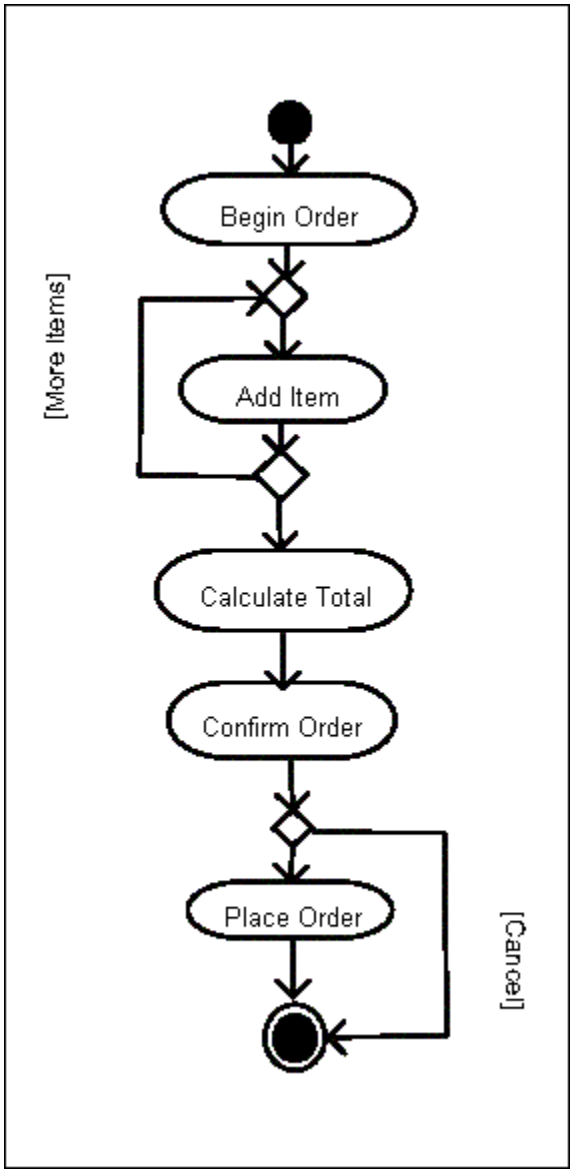
- Physical nodes - stereotype *device*
- *WebServer* - physical device or software artifact
- *RMI/message bus*: connection type
- Nodes can contain other nodes or software artifacts recursively
- Deployment specs: configuration files: name and properties

Activity Diagrams

- Fancy flowchart
  - Displays the flow of activities involved in a single process
  - States
    - Describe what is being processed
    - Indicated by boxes with rounded corners
  - Swim lanes
    - Indicates which object is responsible for what activity
  - Branch
    - Transition that branch
    - Indicated by a diamond
  - Fork
    - Transition forking into parallel activities
    - Indicated by solid bars
  - Start and End

Sample Activity Diagram

- Ordering System
- May need multiple diagrams from other points of view



Activity Diagram Example