CMPS411 Spring 2018

Lecture 9

# Component and Deployment Diagrams

# Review: Diagrams in UML

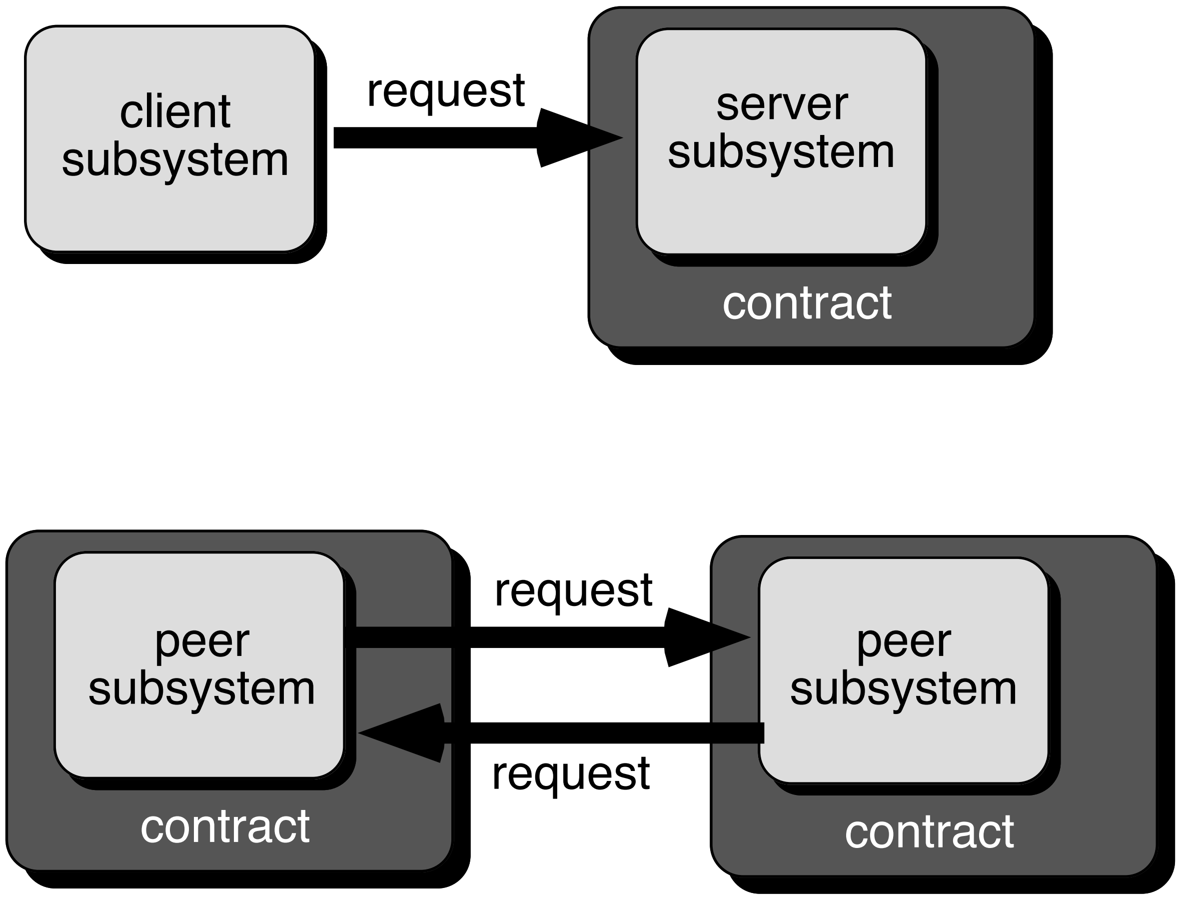
1. Class Diagram
2. Object Diagram
3. Component Diagram
4. Deployment Diagram

**Structural Diagrams**

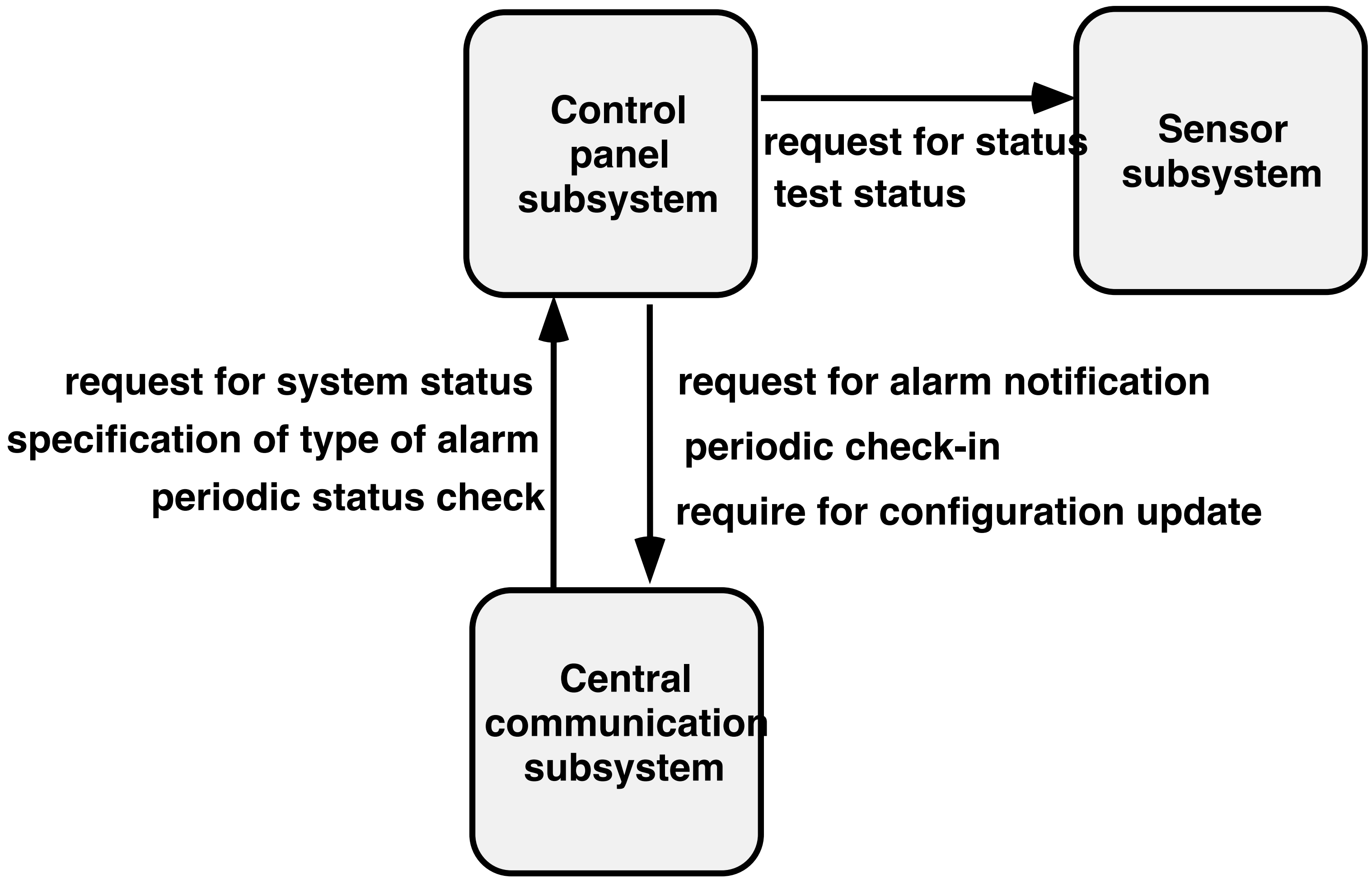
5. Use Case Diagram
6. Sequence Diagram
7. Collaboration Diagram
8. State Diagram
9. Activity Diagram

**Behavioral Diagrams**

# Systems and subsystems: Basic Concepts

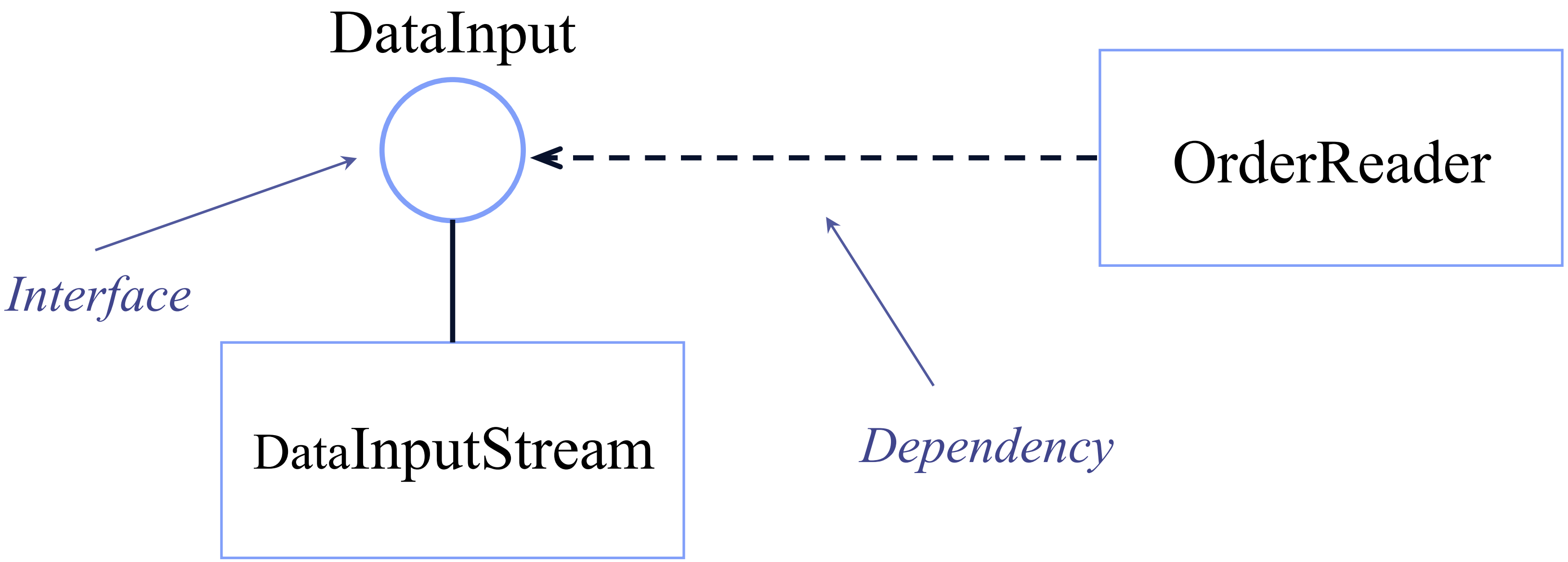


# Systems and Sub-Systems: Basic



# UML Basic: Interfaces

- Lollipops ("short-hand notation")



# Breaking a System into Subsystems?

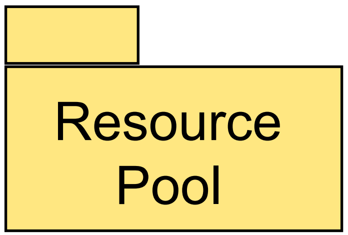
- Roman principle: Divide & conquer
  - Split up a large system into manageable parts
- In structured methods: functional decomposition
- In OO: Group classes into higher level units:

**Packages**
(conceptual; at development time)

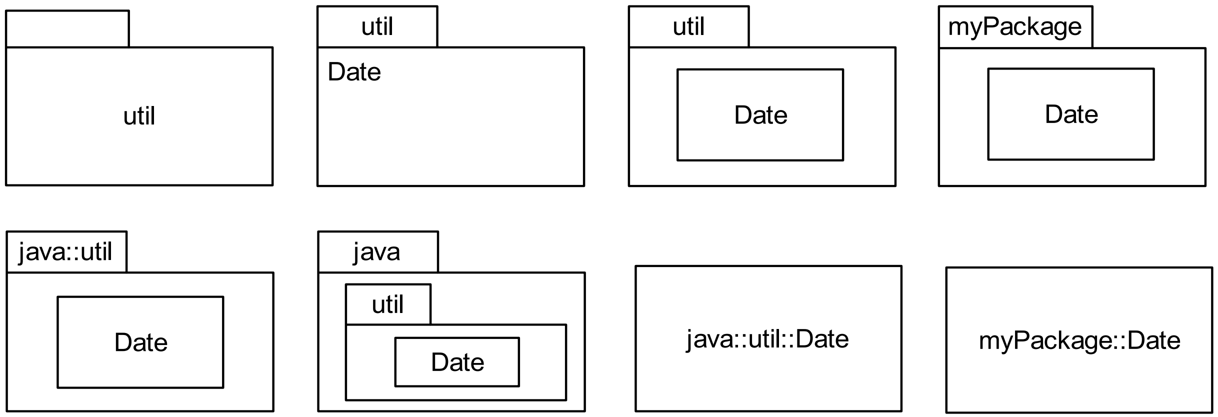
**Components**
(physical; at run time)

# Packages

- General purpose mechanism for organizing elements into groups
- Package forms a namespace
  - Names are unique within ONE package
  - UML assumes an anonymous root package

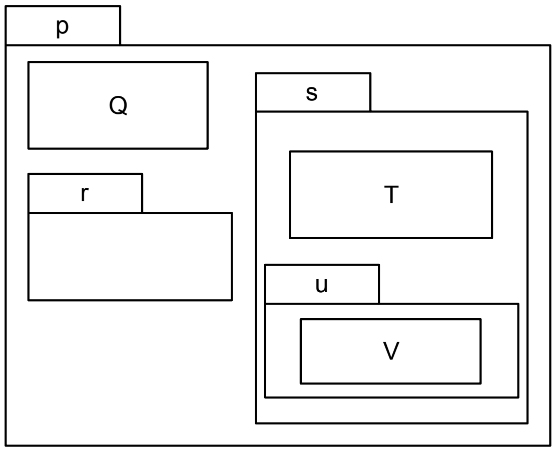


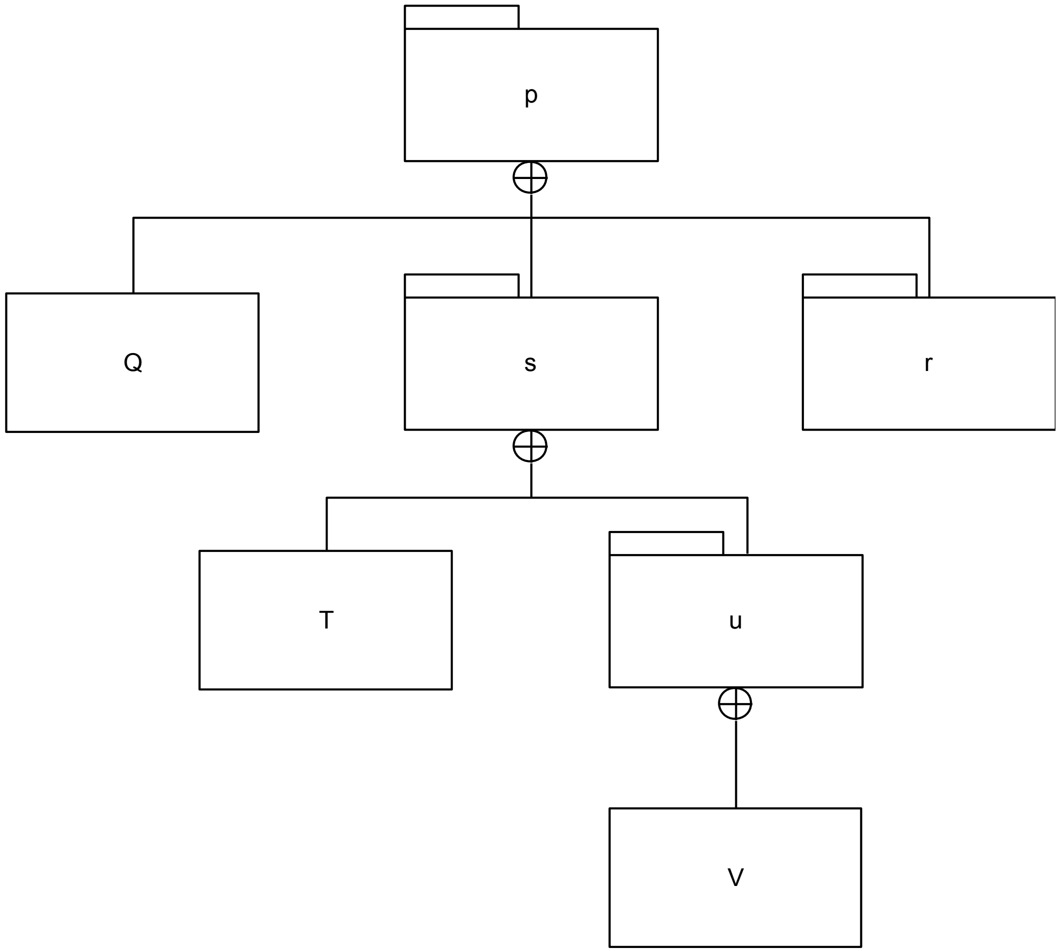
# Packages




- Classes are basic structural units in an OO system
- In large systems with hundreds of classes, **need to group the 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 a 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

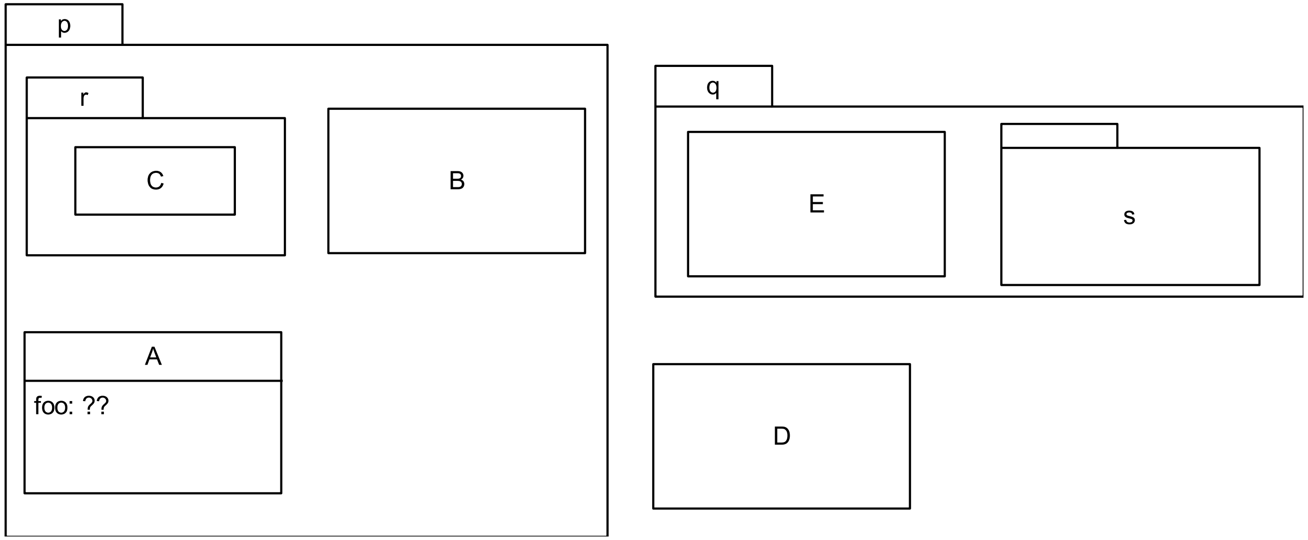
# Package Membership



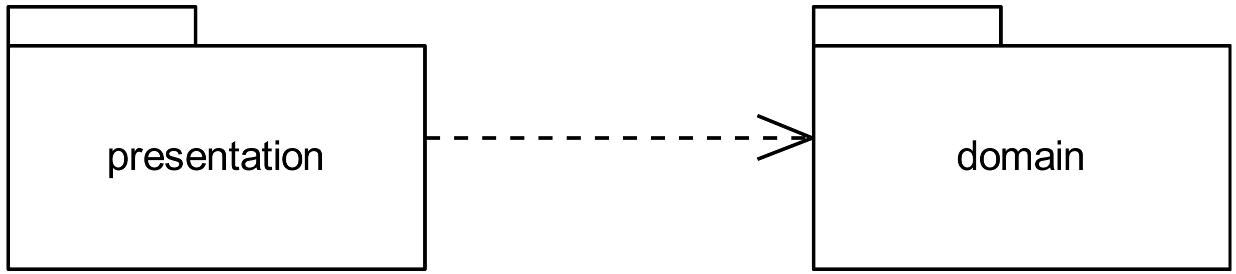

- These diagrams convey the same information

# Relationships between Packages



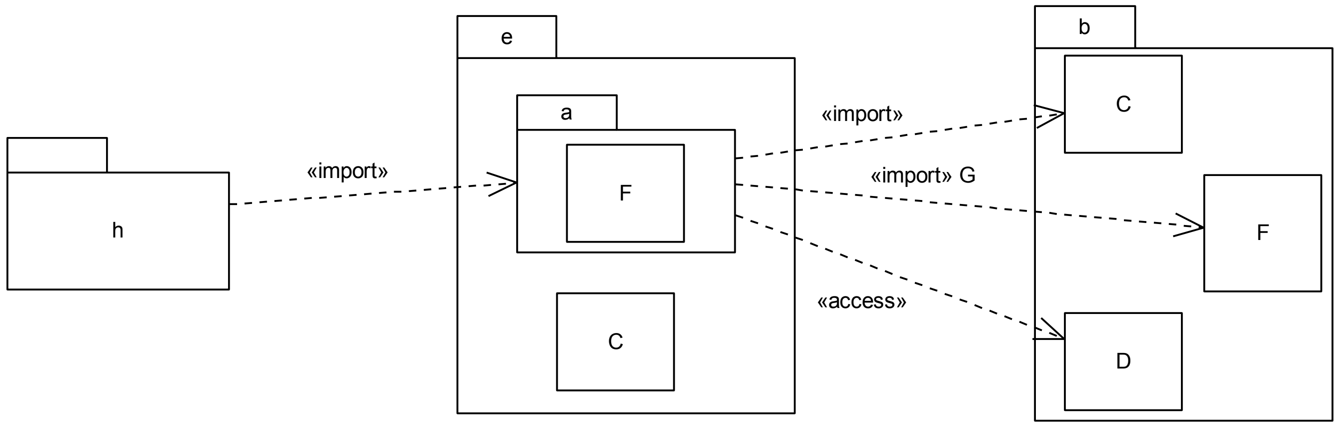

- Element/component can refer to other elements/components that are in its own package and in enclosing packages without using fully qualified names
- Element x must use fully qualified name to access element in package that does not contain x
- foo can be of class B or D
- foo can be of class q::E or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

# Packages and Dependencies



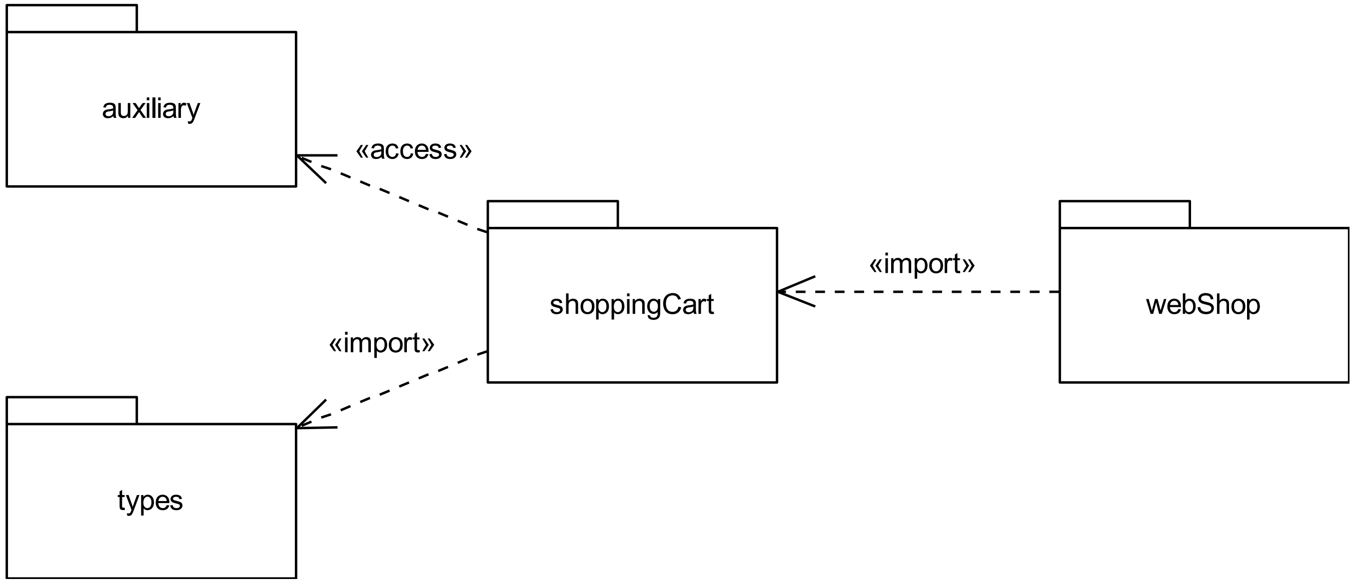

- Package diagram shows packages and their dependencies
- Package A depends on package B if A contains a class which depends on a class in B
  - Inter-package dependencies summarise dependencies between classes

# Element Import




- Element/component 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
- 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**

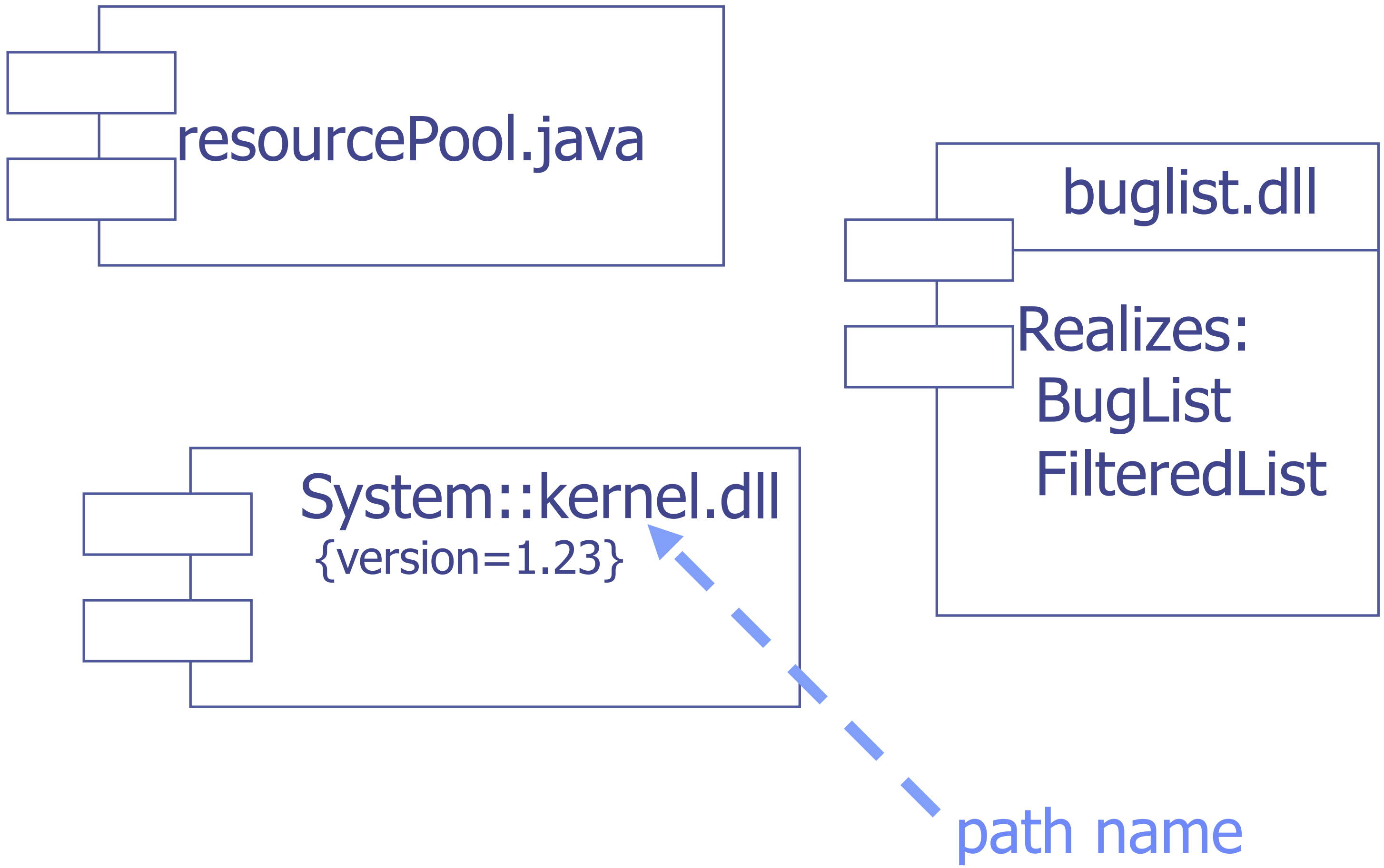
# Component Diagram

- Component Diagram: High-Level Interaction and Dependencies Among Software Components
- Captures the Physical Structure of the Implementation
- Built as Part of Architectural Specification
- Purposes:
  - Organize Source Code
  - Construct an Executable Release
  - Specify a Physical Database
- Main Concepts: Component, Interface, Dependency, Realization
- Developed by Architects and Programmers

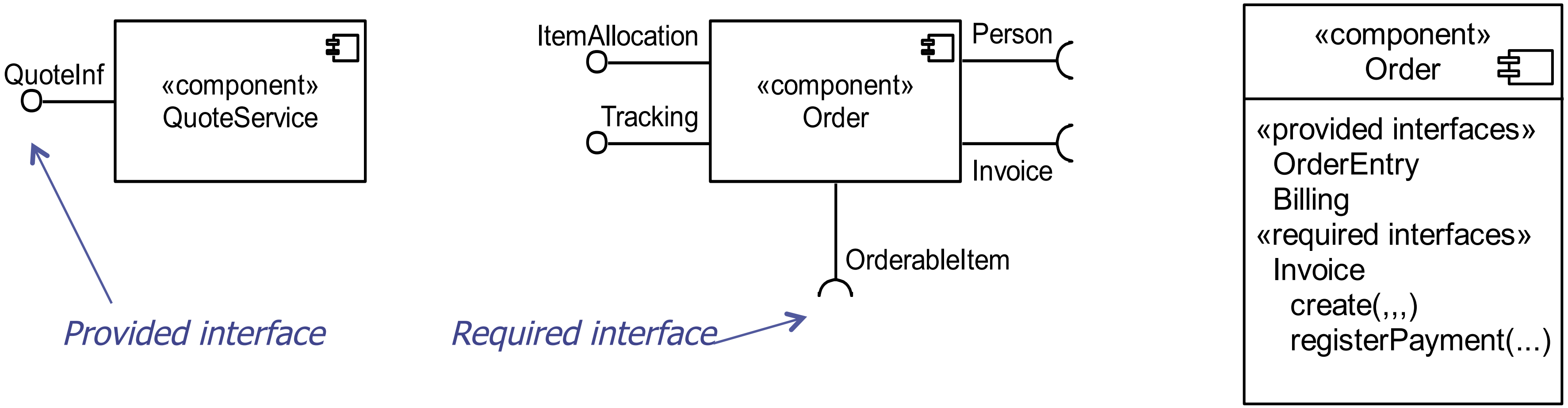
# Components

- 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 component**s 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

# Component Symbol



# Component Notation

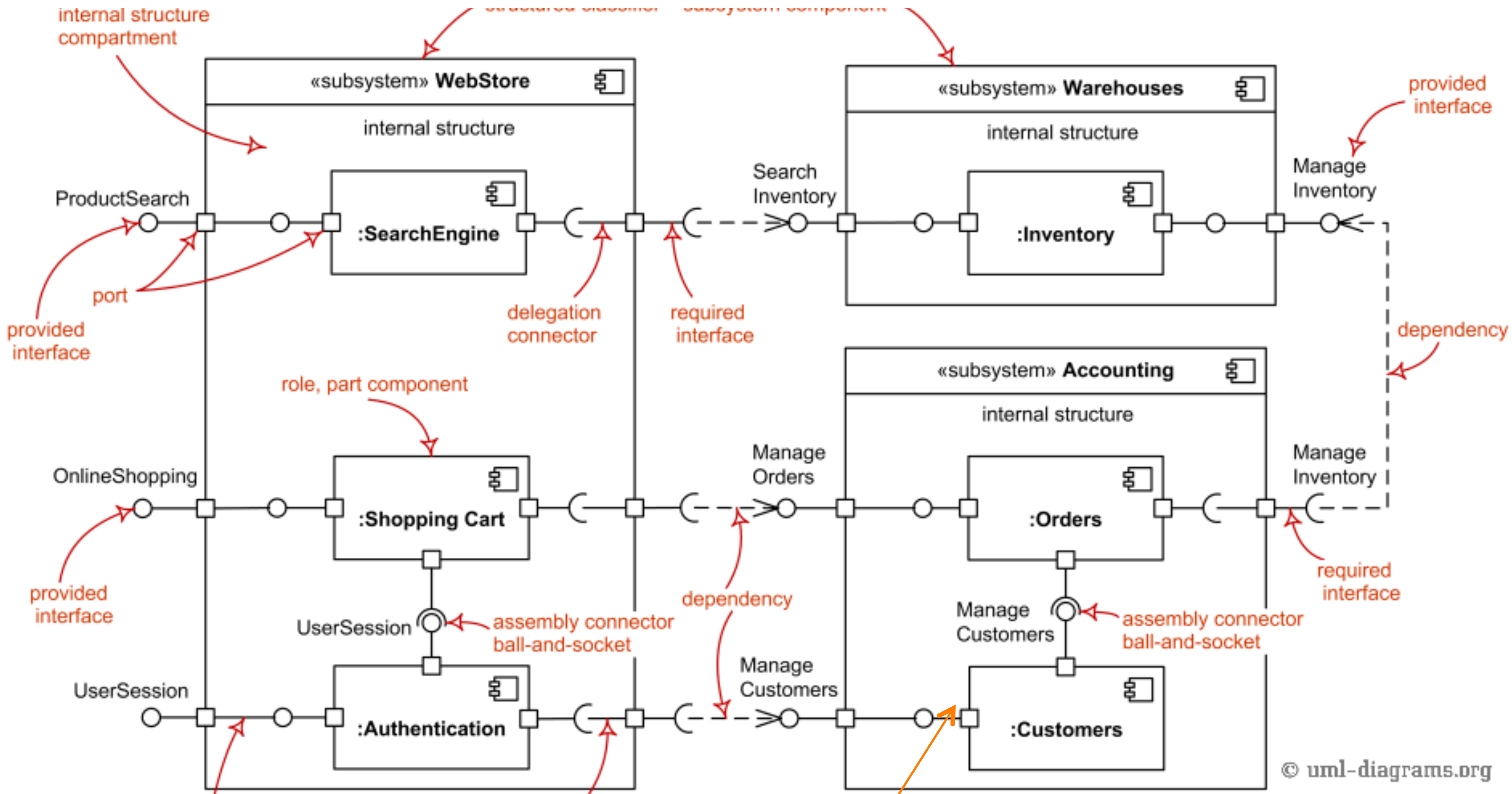



- Component icon is a class icon with the keyword <<component>>
  - Optionally can include a component icon in the top, right-hand corner
- A component has its behavior defined in terms of
  - *provided interfaces,* and
  - *required interfaces*

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

# Component Diagram Notations

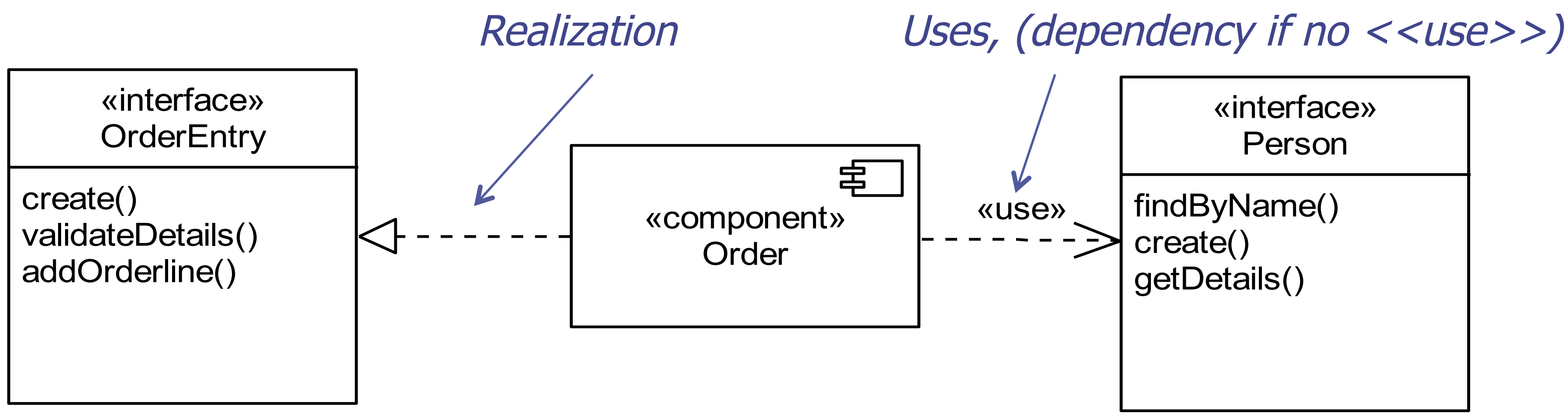
*Subsystem components*




- A **delegation port** is a connector that links the external contract of a component (as specified by its ports) to the realization of that behavior.

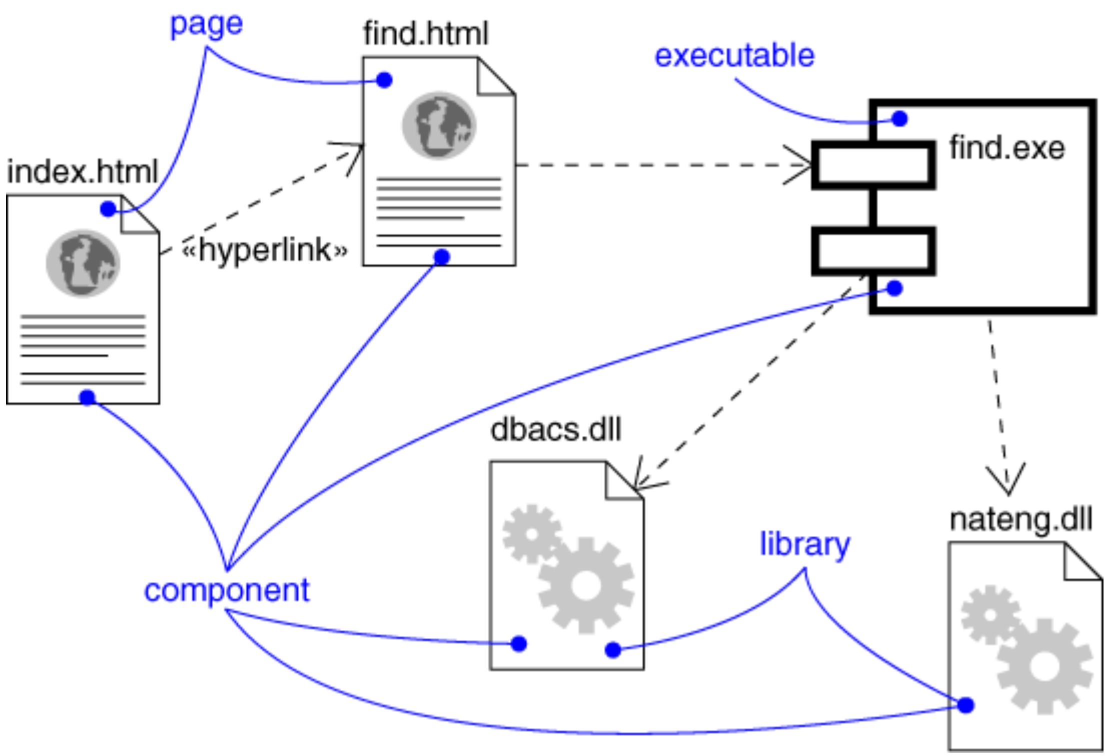
- A **simple port**

# 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 use.
  - The same arrow without <<use>> is dependency

# Component Diagram and Implementation




◆ Captures the Physical Structure of the Implementation

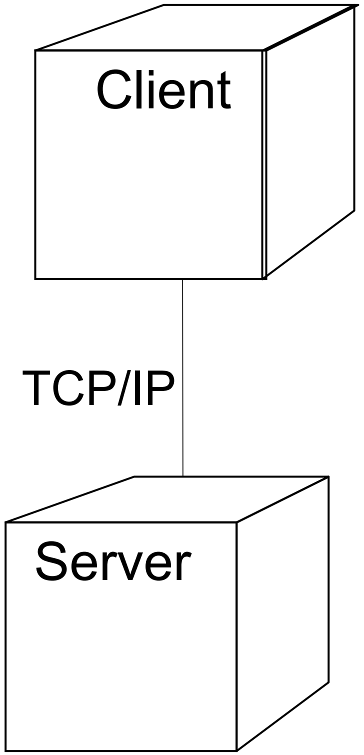
# Components versus Classes

- Both have names and realize interfaces
- Class
  - Logical abstraction
  - Attributes and operations
- Component
  - Physical thing that exist on machines
  - Physical packaging of logical things (classes, interfaces, …)
  - Only has operations (only reachable thru its Interface)

# Deployment Diagrams

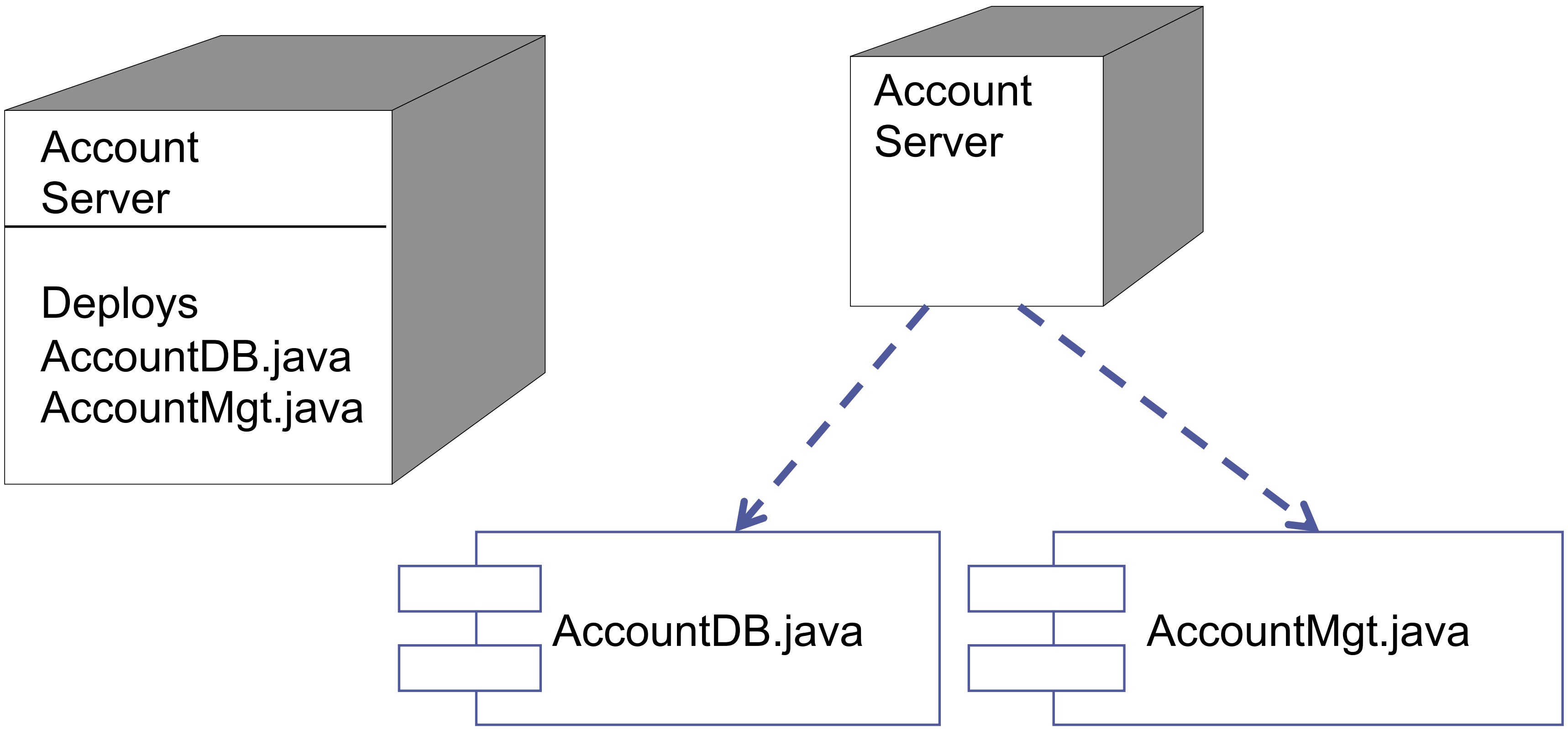
- Show physical relationship among software & hardware components
- Show where components of a distributed system are located

# Deployment Diagrams and Nodes



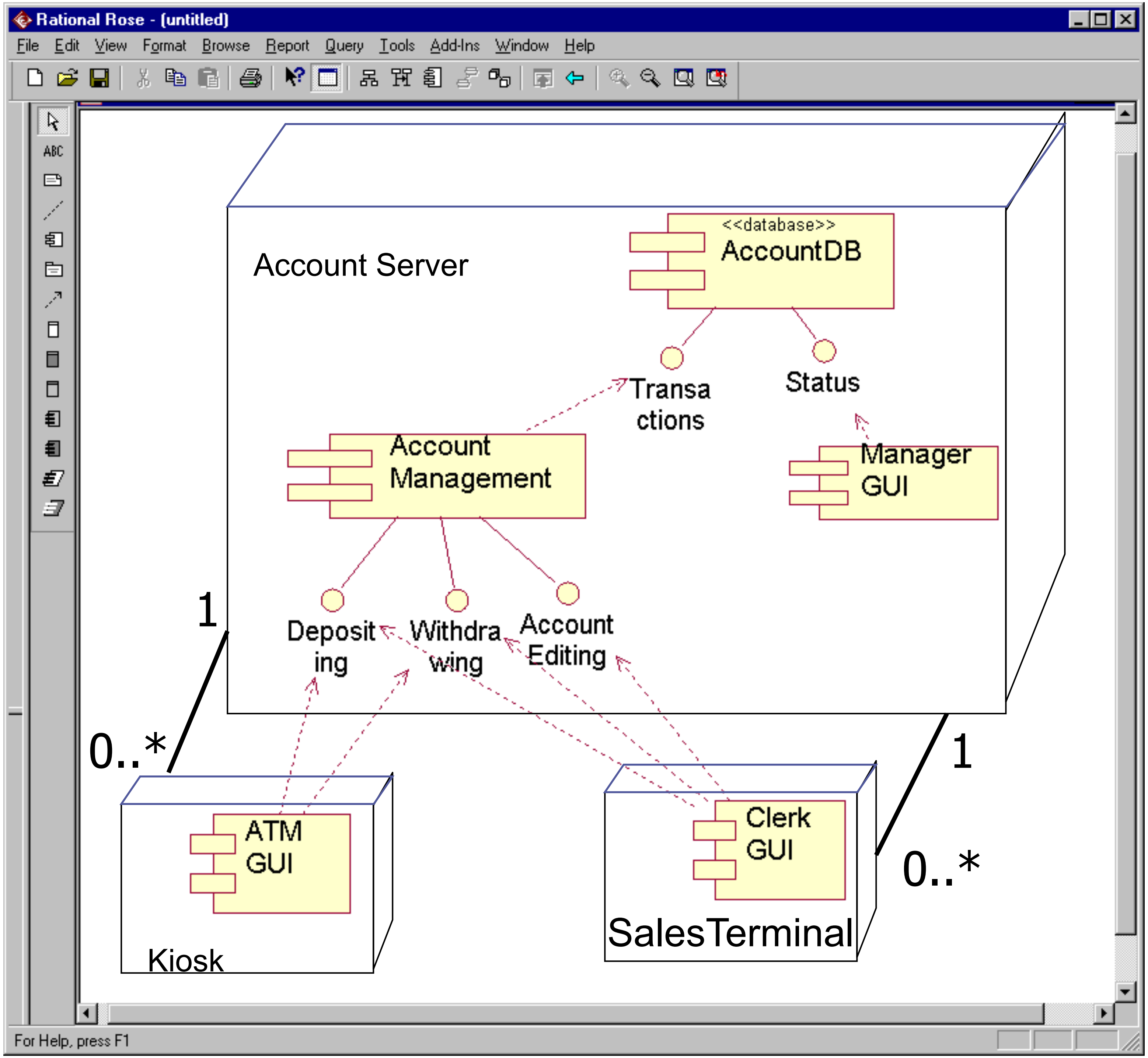

- Nodes: Computational units (most often: Hardware)
- Connections: Communication paths over which the system will interact

# Nodes and Components



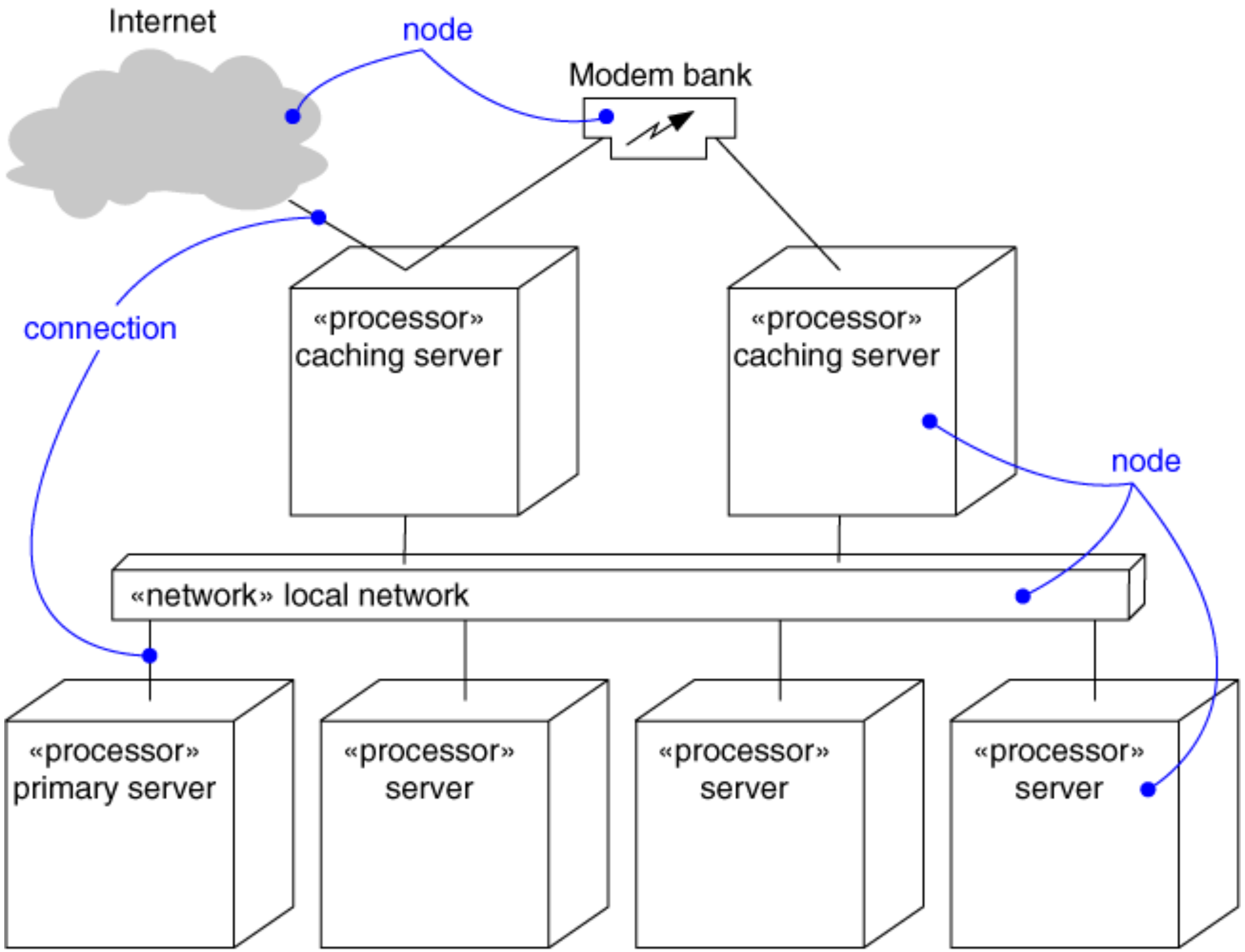
# Deployment Diagram

- Deployment Diagram: Focus on the placement and configuration of components at runtime
- Captures the topology of a system’s hardware
- Built as part of architectural specification
- Purposes:
  - Specify the distribution of components
  - Identify performance bottlenecks
- Main concepts: Node, Component, Dependency, Location
- Developed by architects, networking engineers, and system engineers


Rational Rose - (untitled)
File Edit View Format Browse Report Query Tools Add-Ins Window Help
ABC
<<database>>
AccountDB
Account Server
Transa ctions
Status
Account Management
Manager GUI
Deposit ing
Withdra wing
Account Editing
1
0..*
ATM GUI
Kiosk
Clerk GUI
SalesTerminal
1
0..*
For Help, press F1

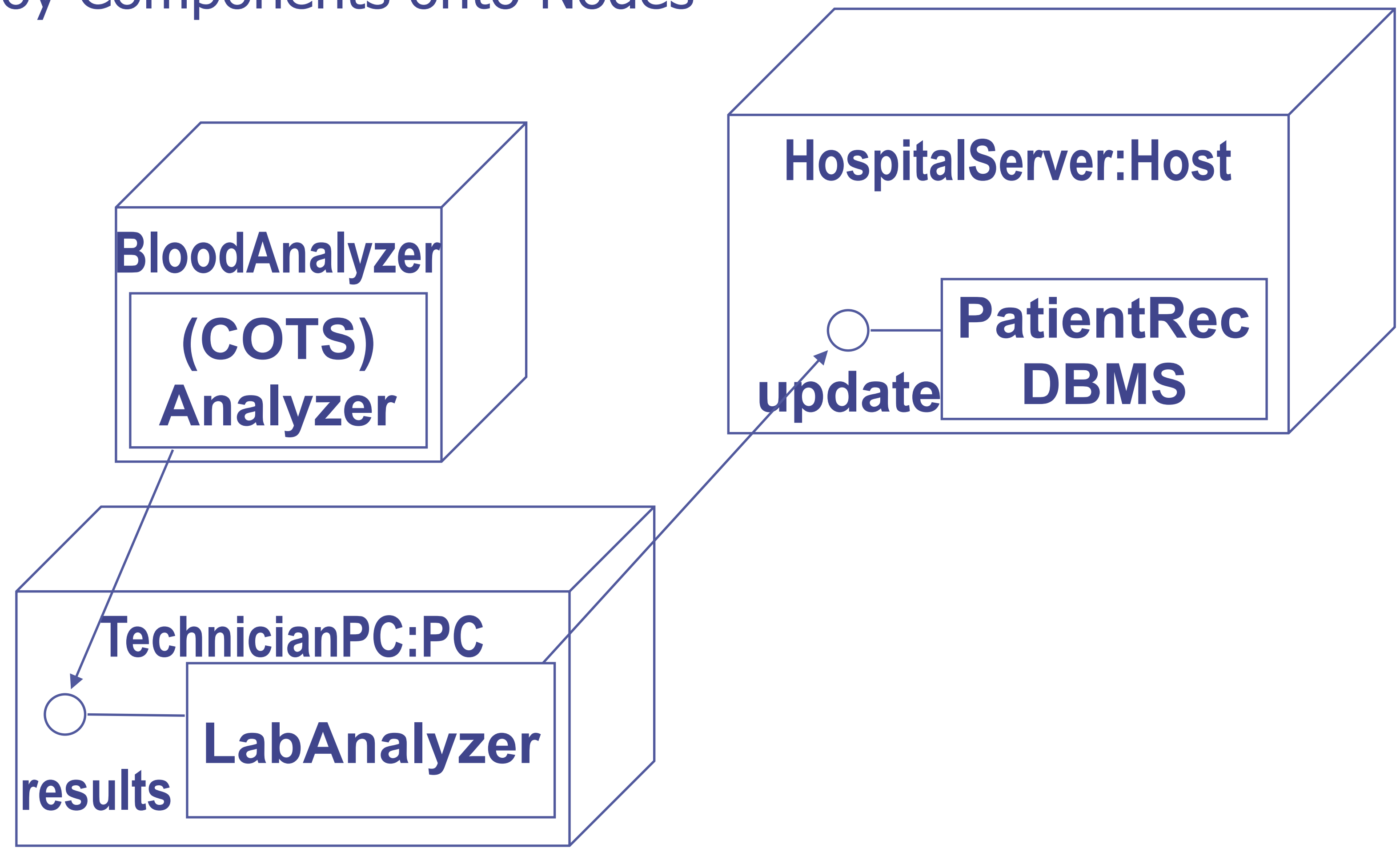
# Topology of Deployment Diagram

- Captures the Topology of a system's hardware

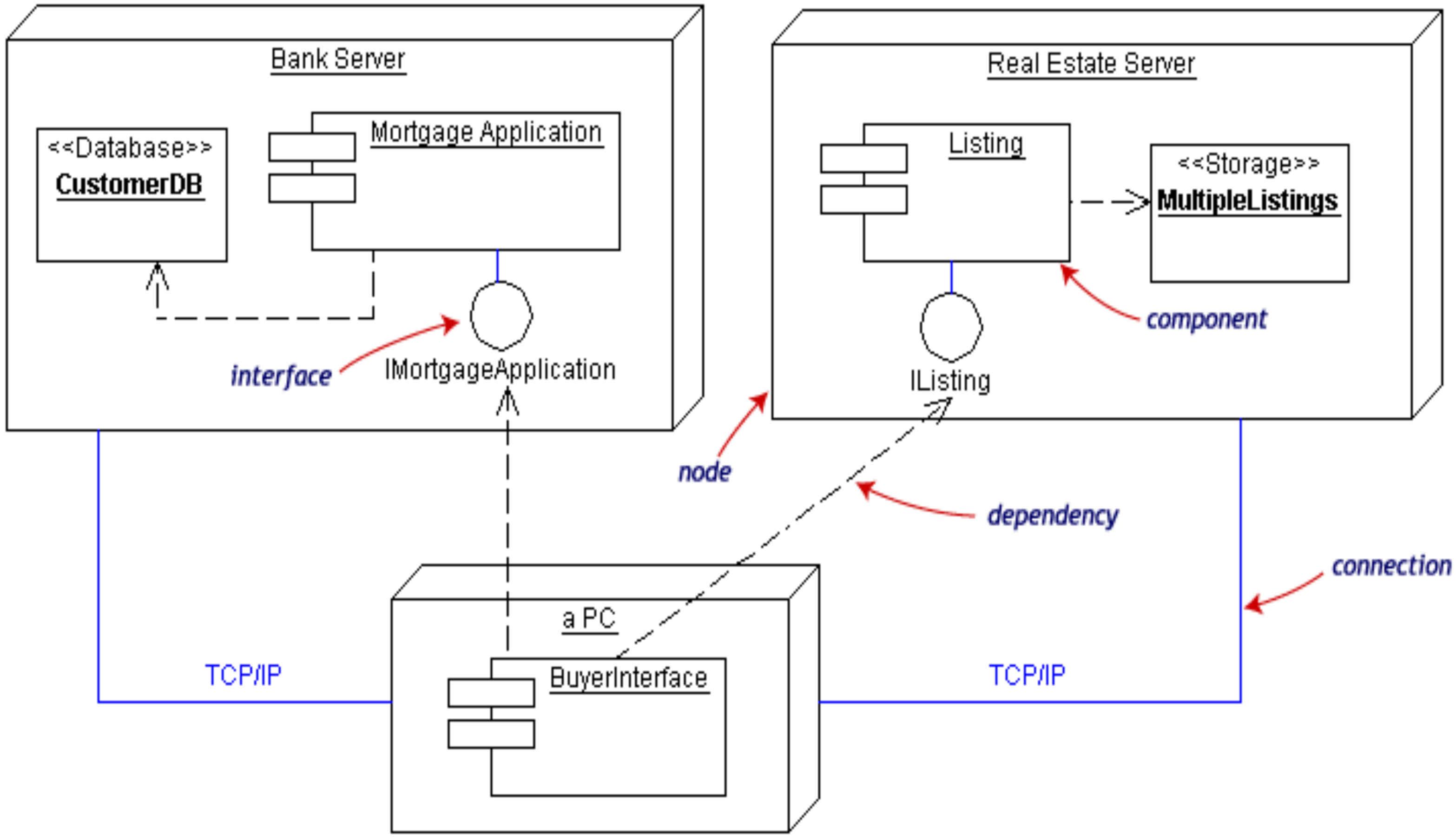


# Deployment Components

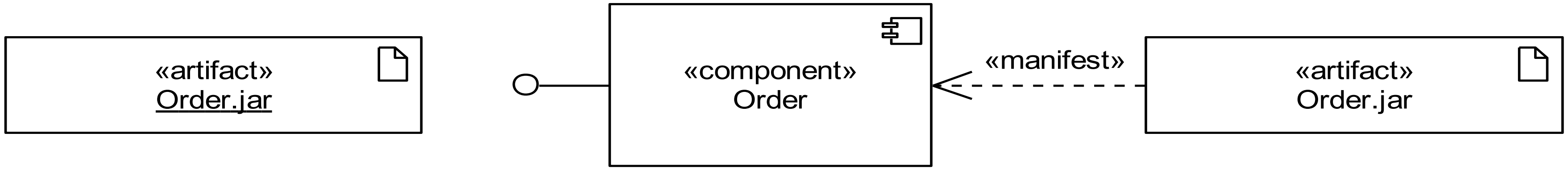
- Deploy Components onto Nodes



# Combining
# Component and Deployment Diagrams

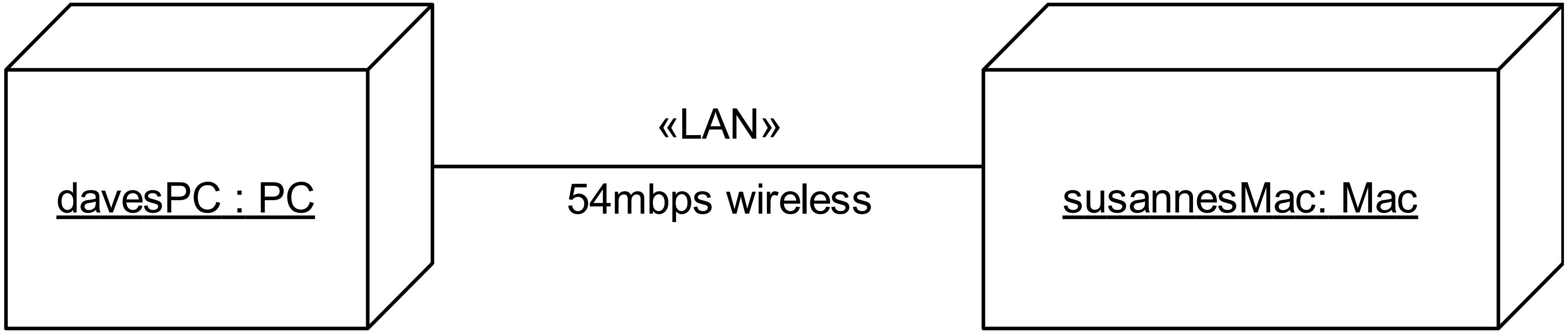


# Artifacts



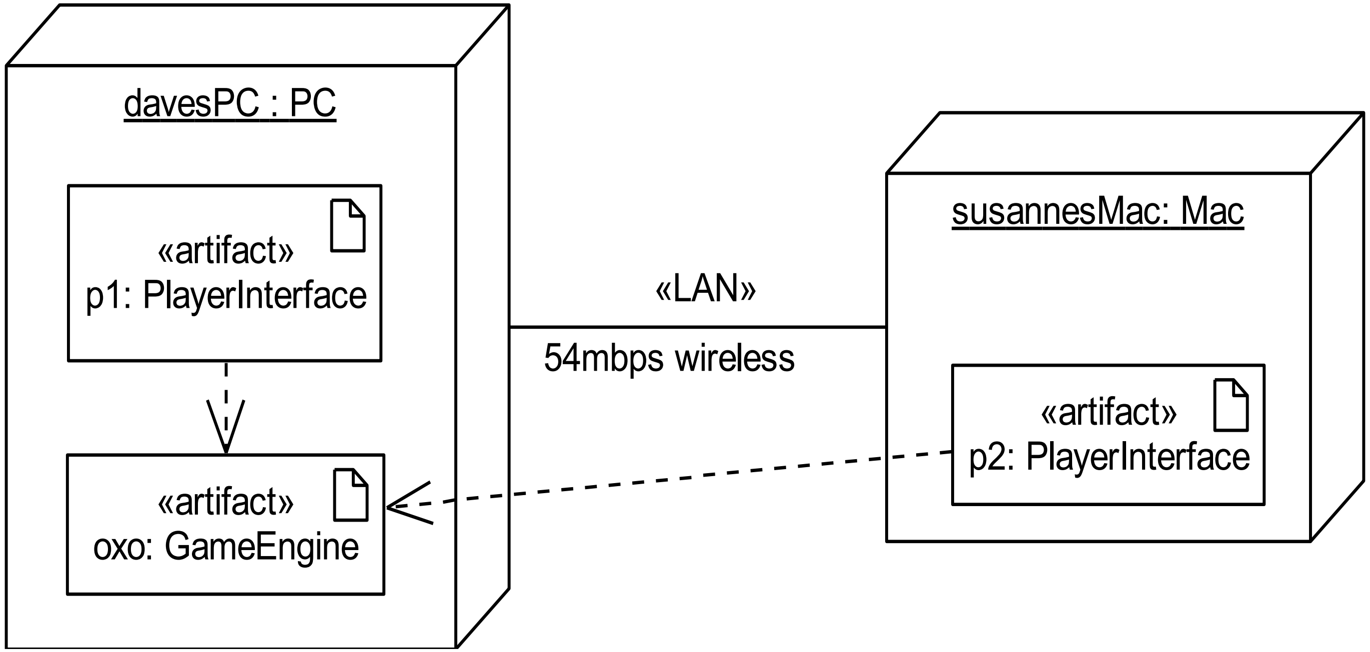

- An artifact is a concrete element in the physical world
- An *artifact* is the specification of a physical piece of information such as a binary executable, a database table or an implemented component such as a DLL or a Java class file
- Each artifact has a filename
- An artifact is represented by 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>>

# 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 Software Artifacts on Hardware Nodes




- Artifact shown inside a node shows 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

# Summary

- Packages
  - collections of related classes
  - defines a namespace
  - class visibility and facades
  - Element x must use fully qualified name to access element in package that does not contain x
  - Package dependencies
  - Importing elements and packages into a package: <<import>> and <<access>>
- Components
  - reusable modules
  - defined by interfaces required and provided
  - black box and white box views on components
- Artifacts, <<manifest>> key word
- Deployment diagrams