Static Modeling (Structural Modelling)

Objectives

Discuss Structure Diagrams

- *Class Diagram (Already covered)*
- Component and Subsystem Diagrams
- Package Diagrams
- Composite Structure Diagrams
- Deployment Diagram

Large-scale Static Modeling (No direct code Translation)

- "Big things" in UML come in two flavors:
  1. "Big things" used to **represent** large system structures
     ⇒ UML *composite structures*, *component diagrams, subsystem diagrams*

  2. "Big things" used to **organize** large system structures
     - Keeping track of hundreds or thousands of classes
     - Effectively sharing work among many developers

     ⇒ UML *packages*

Outline

- Objectives
  - Discuss Structure Diagrams
  - *Class Diagram (Already covered)*
  - Component and Subsystem Diagrams
  - Package Diagrams
  - Composite Structure Diagrams
  - Deployment Diagram

What is a component?

- A component is a structured class with “aspirations”

UML 2.0 specification: "*A component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment*"

  - **May be *substituted* for by other components provided they all support the same protocol**

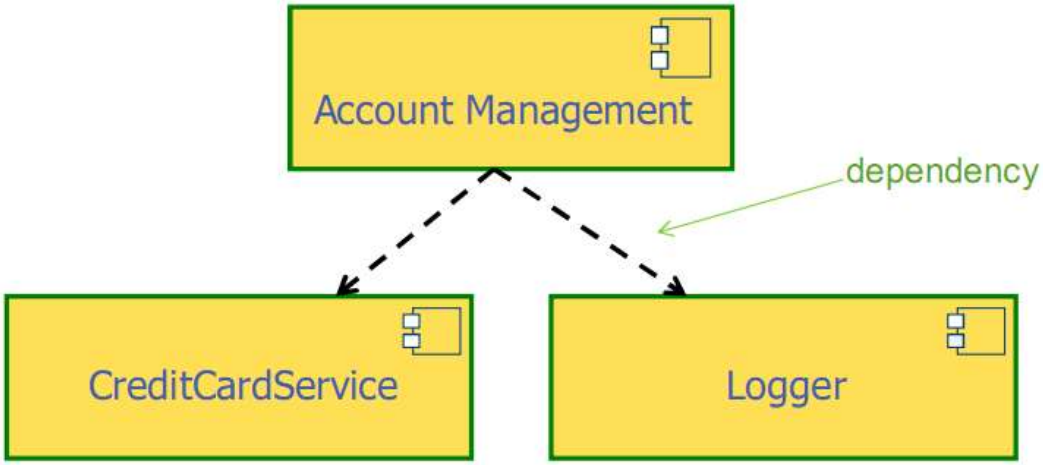
Component Diagram: Syntax

Components can be:

- Physical - can be directly instantiated at run-time
- Logical - a purely logical construct e.g., a subsystem

Component Diagram: Dependencies

Components may need other components to implement their functionality

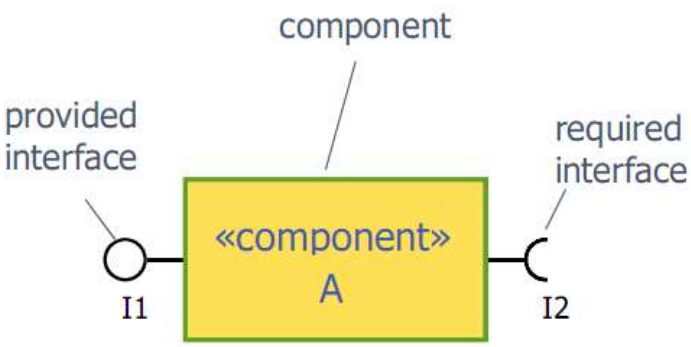


This is a relatively high-level view of a system. To refine the diagram we have to use interfaces, ports, internal structure

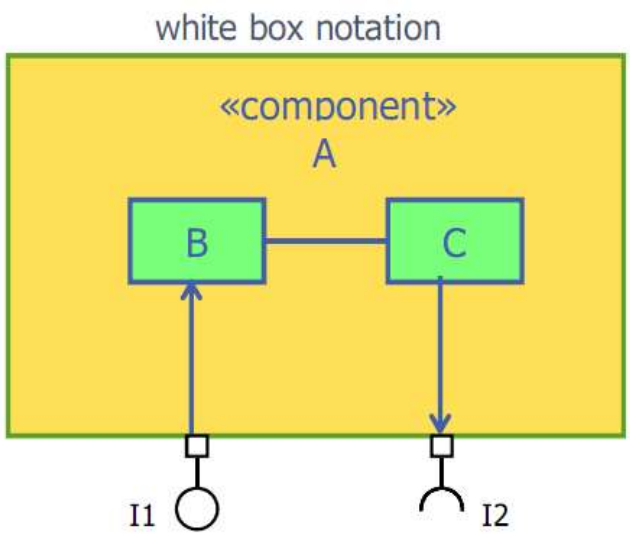
Component Diagram: Views

- UML uses two views of components: *black-box* and *white-box*

Black-box view shows the interfaces the component provides/requires



White-box view shows how a component realizes the interface it provides

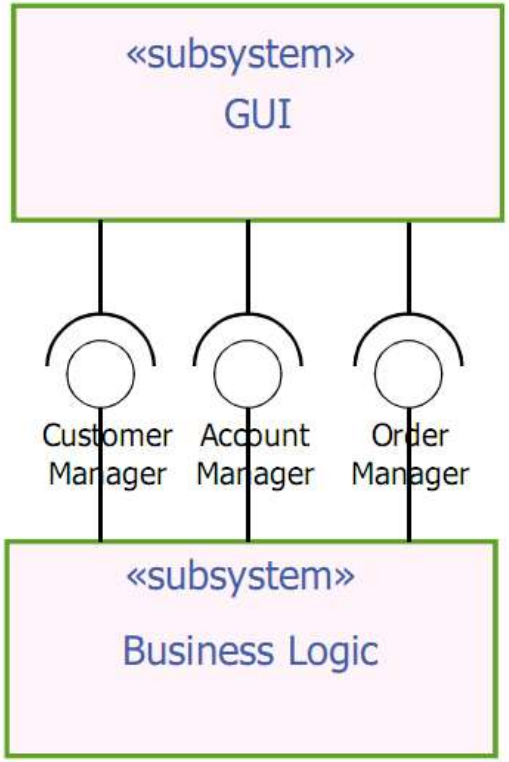


Component Diagram: Standard Stereotypes

Stereotype	Semantics
«entity»	A persistent information component representing a business concept.
«implementation»	A component definition that is not intended to have a specification itself. Rather, it is an implementation for a separate «specification» to which it has a dependency.
«specification»	A classifier that specifies a domain of objects without defining the physical implementation of those objects. For example, a Component stereotyped by «specification» only has provided and required interfaces - no realizing classifiers.
«process»	A transaction based component.
«service»	A stateless, functional component (computes a value).
«subsystem»	A unit of hierarchical decomposition for large systems.

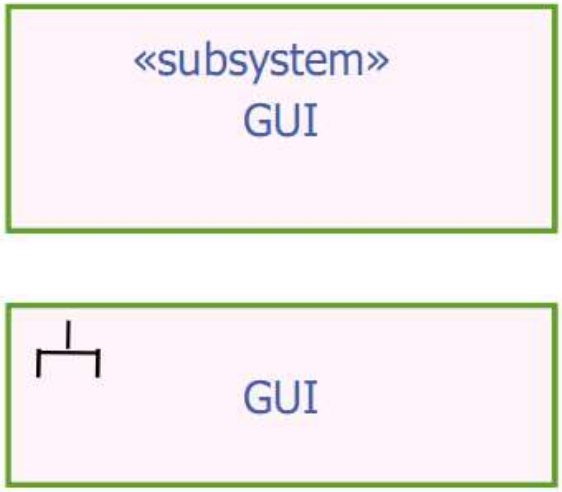
Subsystems

- A *subsystem* is a *component* that acts as a unit of decomposition for a larger system at the highest level of abstraction
- It is an architectural-level structured class
- It is a logical construct used to decompose a larger system into manageable chunks
- Subsystems *cannot* be instantiated at run-time, but their contents can
- Ports and Interfaces connect subsystems together to create a system architecture

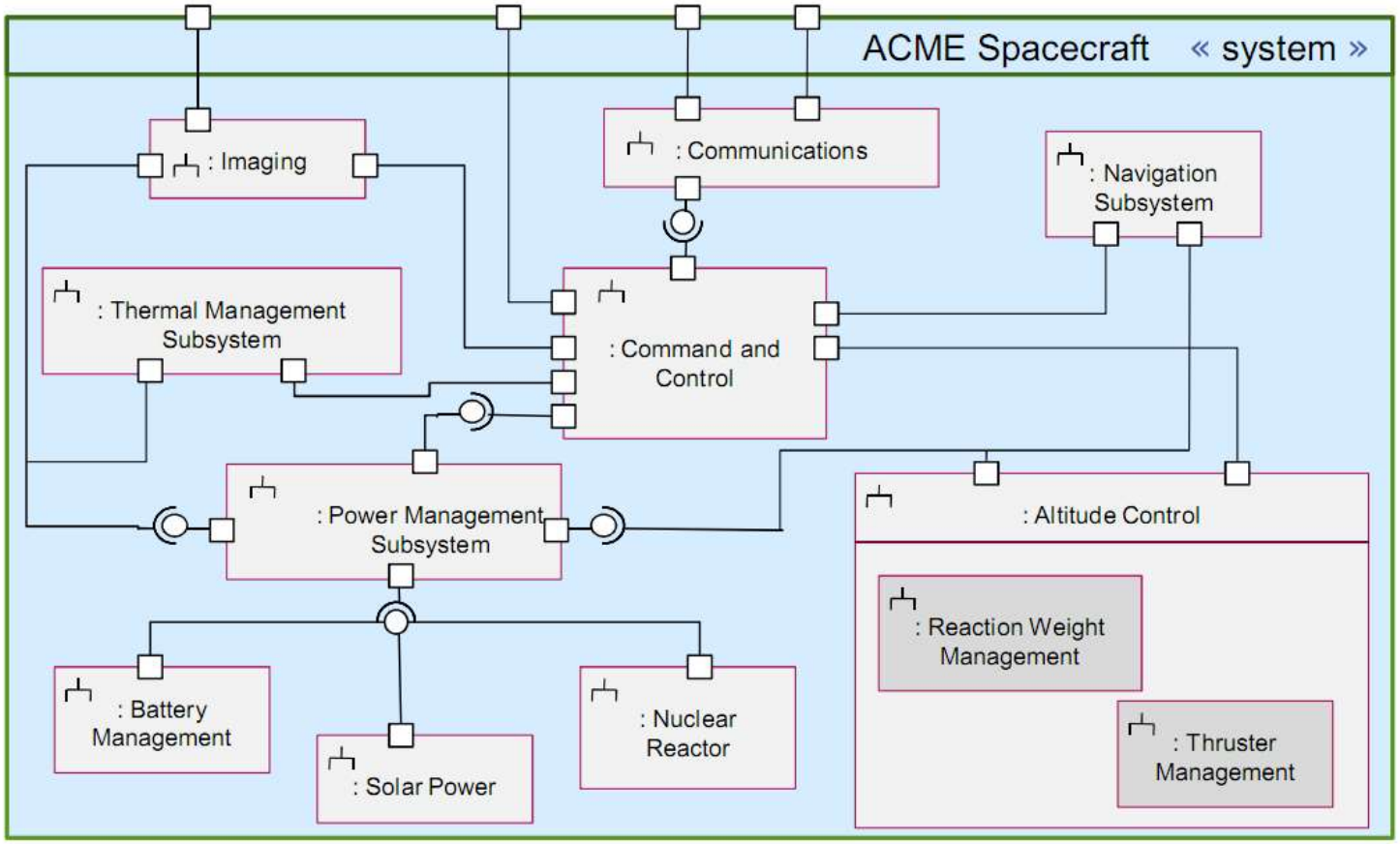


Subsystem Diagrams

- Syntax



Subsystem Diagrams



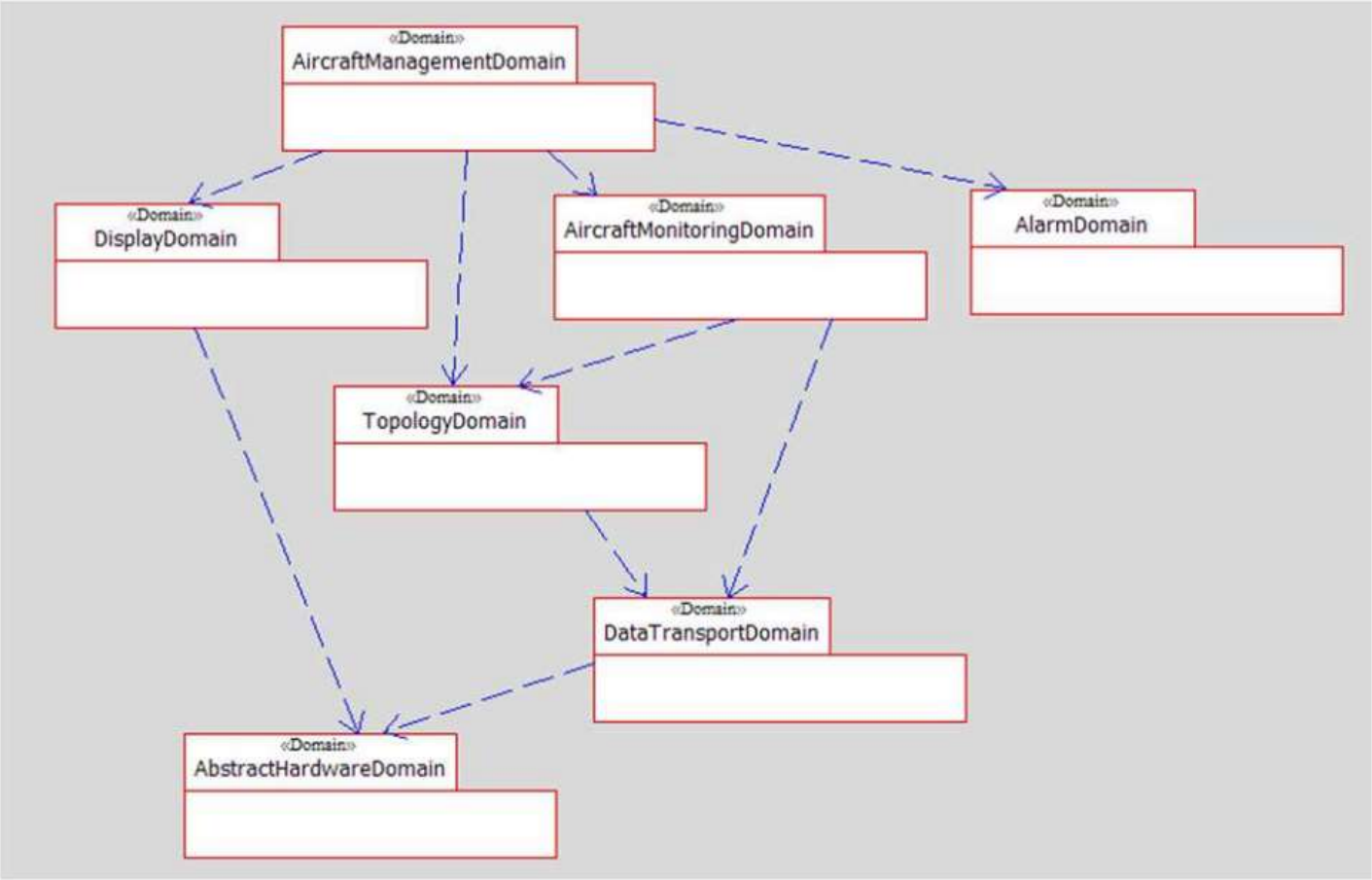
Outline

- Objectives
  - Discuss Structure Diagrams
  - *Class Diagram (Already covered)*
  - Component and Subsystem Diagrams
  - Package Diagrams
  - Composite Structure Diagrams
  - Deployment Diagram

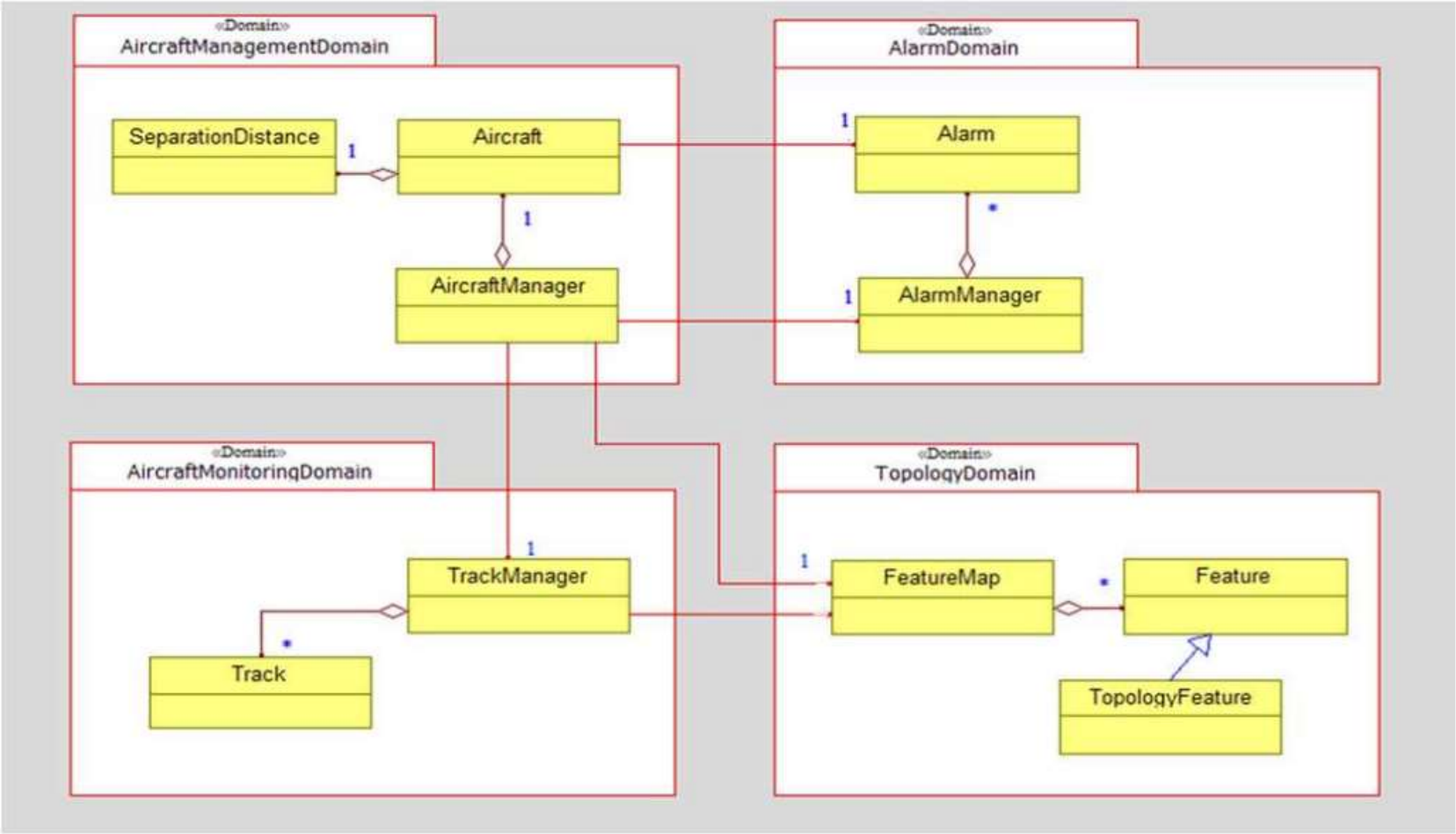
Packages

- A package is a *general purpose grouping mechanism*
  - Group semantically related elements
  - Define a “semantic boundary” in the model
  - Each package defines an *encapsulated namespace* i.e., all names must be unique within the package
- Think of it as a folder that is used to organize UML model elements at *design time*

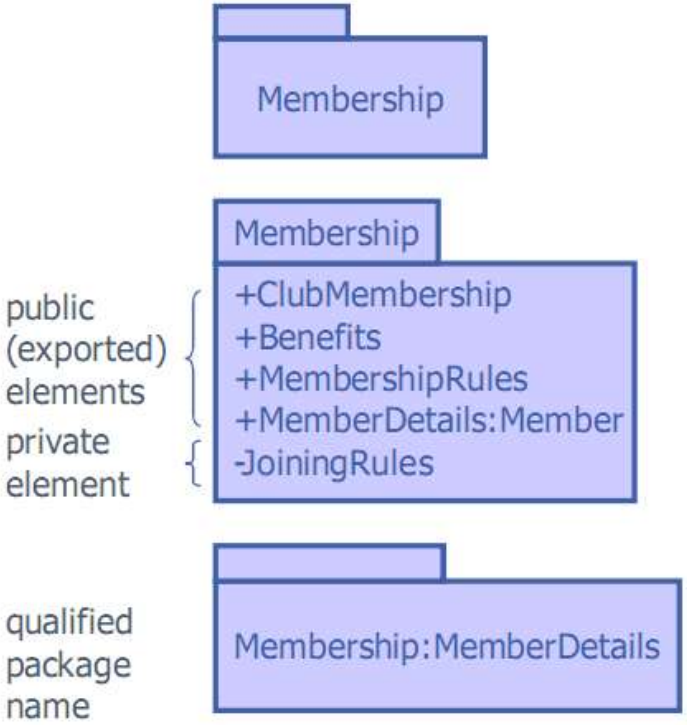
Packages

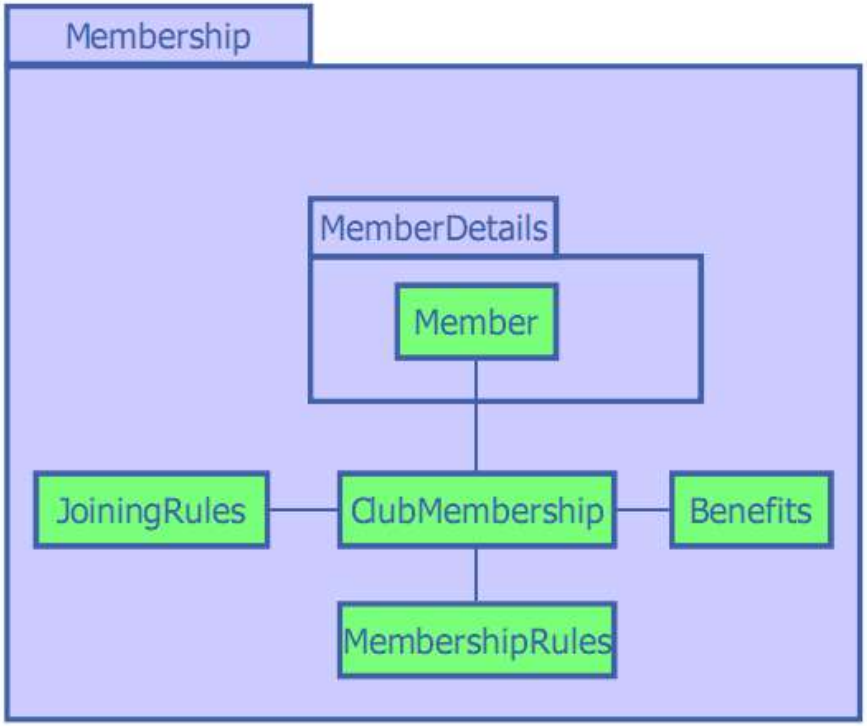

«Domain»
AircraftManagementDomain
«Domain»
DisplayDomain
«Domain»
AircraftMonitoringDomain
«Domain»
AlarmDomain
«Domain»
TopologyDomain
«Domain»
DataTransportDomain
«Domain»
AbstractHardwareDomain

Packages


«Domain»
AircraftManagementDomain
SeparationDistance
1
Aircraft
1
AircraftManager
«Domain»
AlarmDomain
1
Alarm
*
1
AlarmManager
«Domain»
AircraftMonitoringDomain
1
TrackManager
*
Track
«Domain»
TopologyDomain
1
FeatureMap
*
Feature
TopologyFeature

Package Syntax



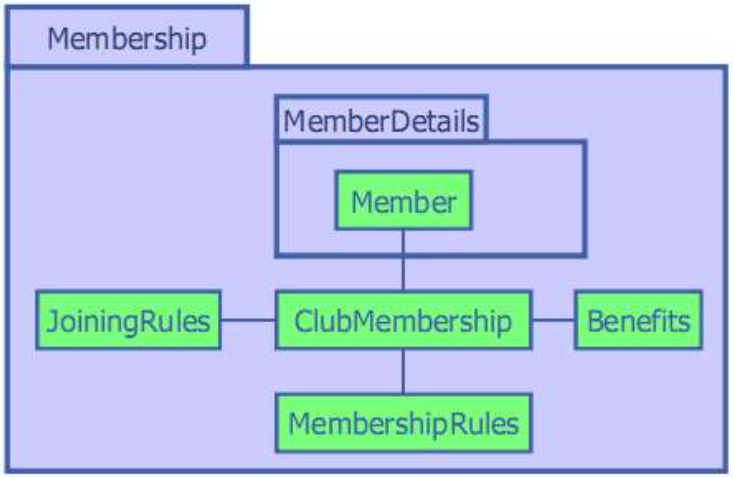


Package Visibility

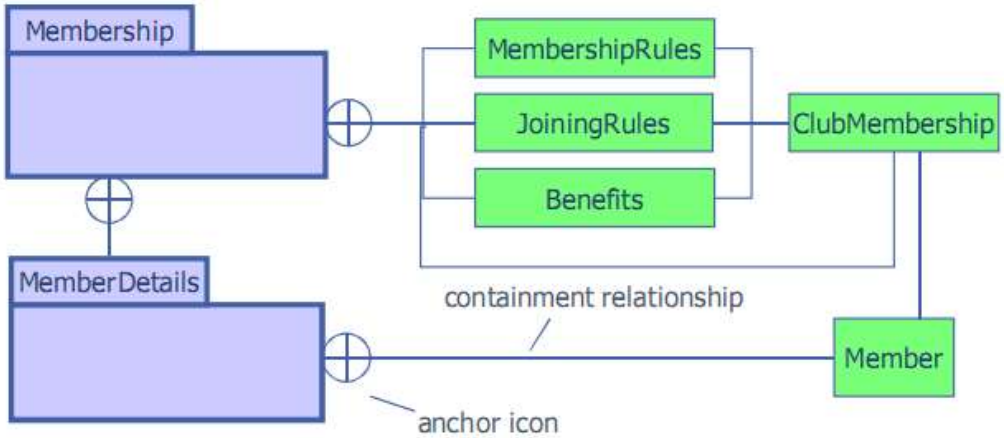
- Package elements may have one of two levels of visibility: public or private
  - + public: elements may be used outside of the package
  - - private: elements may be used only by other elements of the same package

Nested Packages

- If an element is visible within a package then it is visible within all nested packages
  - e.g., *Benefits* is visible within *MemberDetails*



- Show containment using nesting or the containment relationship



Package Dependencies

dependency	semantics
Supplier ←«use»- - - - Client	An element in the client uses an element in the supplier in some way. The client depends on the supplier. Transitive.
Supplier ←«import»- - - - Client	Public elements of the supplier namespace are added as public elements to the client namespace. Transitive.
Supplier ←«access»- - - - Client not transitive	Public elements of the supplier namespace are added as private elements to the client namespace. Not transitive.
Analysis Model ←«trace»- - - - Design Model	«trace» usually represents an historical development of one element into another more refined version. It is an extra-model relationship. Transitive.
Supplier ←«merge»- - - - Client	The client package merges the public contents of its supplier packages. This is a complex relationship only used for metamodeling - you can ignore it.

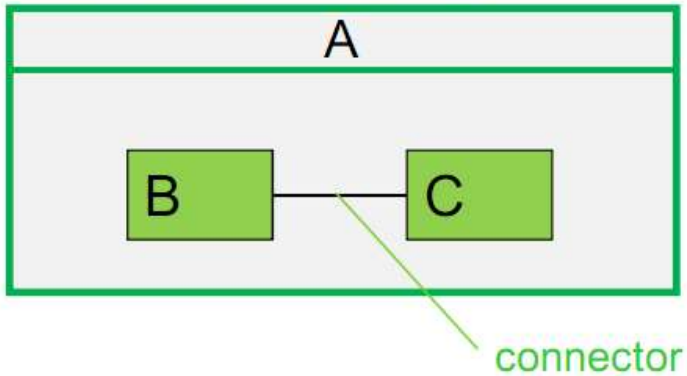
Outline

- Objectives
  - Discuss Structure Diagrams
  - *Class Diagram (Already covered)*
  - Component and Subsystem Diagrams
  - Package Diagrams
  - Composite Structure Diagrams
  - Deployment Diagram

Composite Structure: Definition

- A *Structured Class (or classifier)* represents a class whose behavior can be completely or partially described through interactions between parts.

- Parts are instance roles linked together with connectors



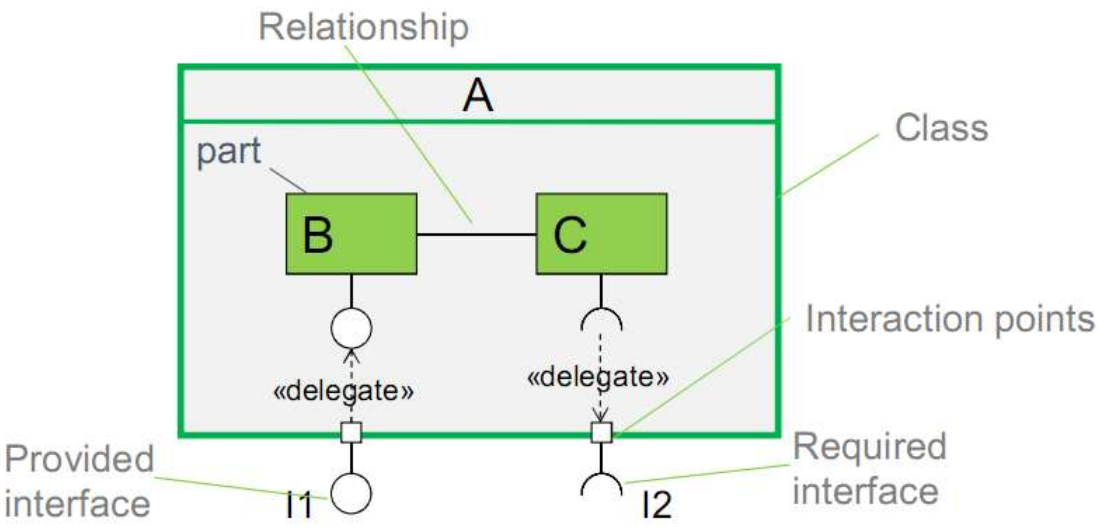
- The concept of structured class is based on decomposition and encapsulation

Composite Structure: Definition

- A structured class is *not* a simple runtime container of parts
  - It has the responsibility to
    - create and destroy its part instances
    - Coordinate the activities and collaboration of the part instances
- The connectors are created by the structured class
  - They link together the parts so that they can collaborate in the context of the structured class

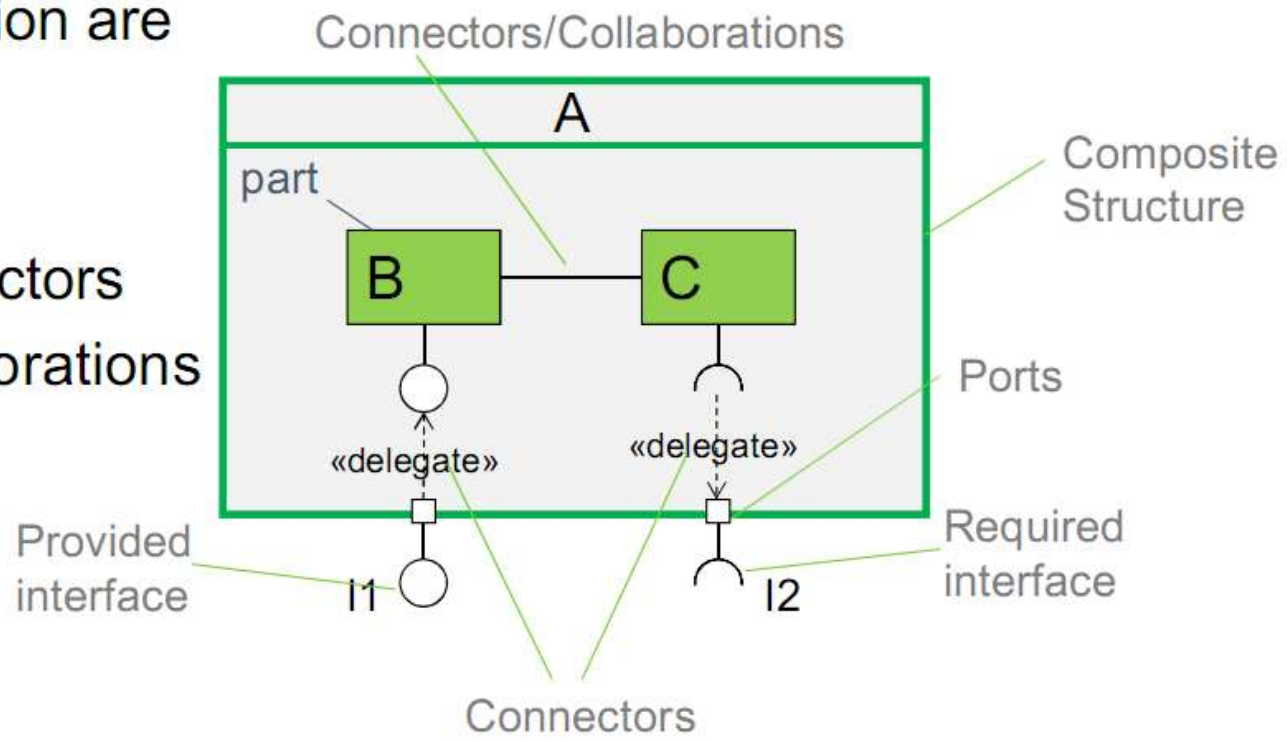
Composite Structure Diagram

A *composite structure diagram* is a diagram that shows the internal structure of a *structured class*, including its interaction points to other parts of the system. It shows the configuration and relationship of parts, that together, perform the behavior of the class at run time.



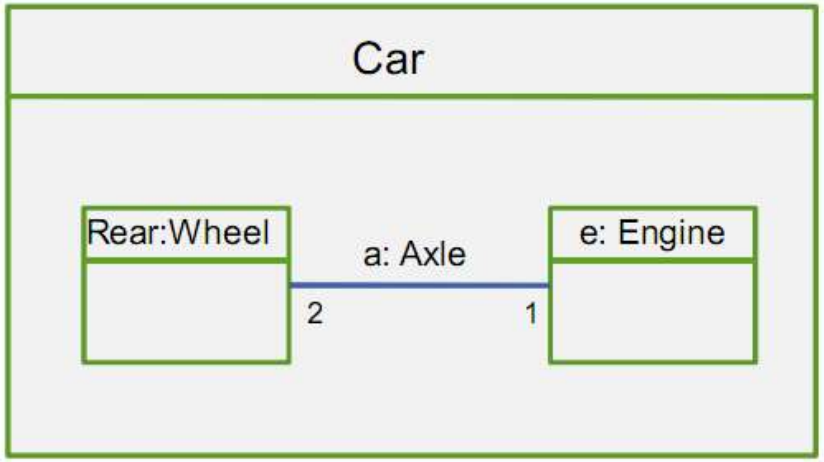
Composite Structure Diagram

- The key composite structure entities identified in UML 2.0 specification are
  - Parts
  - Ports
  - Connectors
  - Collaborations

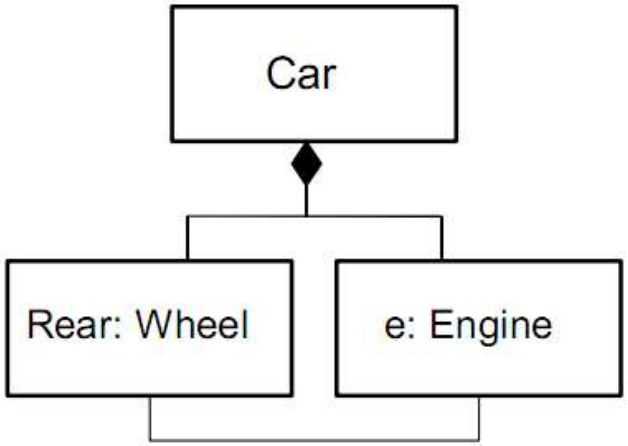


Composite Structure Diagram - Parts

- A part is an element that represents a set of one or more *instances* which are owned by a containing class instance.



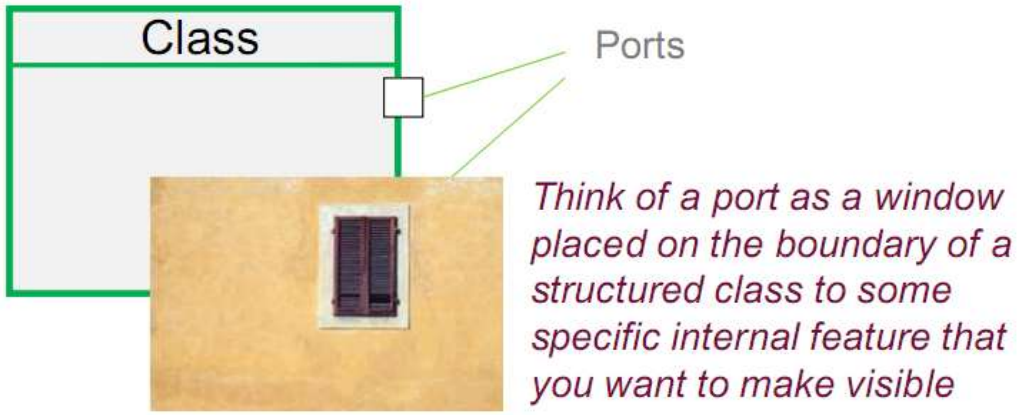
Nested Notation



Class Diagram Notation

Composite Structure Diagram - Ports

- A port is a typed element that represents an externally visible part of a containing class instance.
- Ports define the interaction between a classifier and its environment.
- A port can appear on the boundary of a contained part or a composite structure.



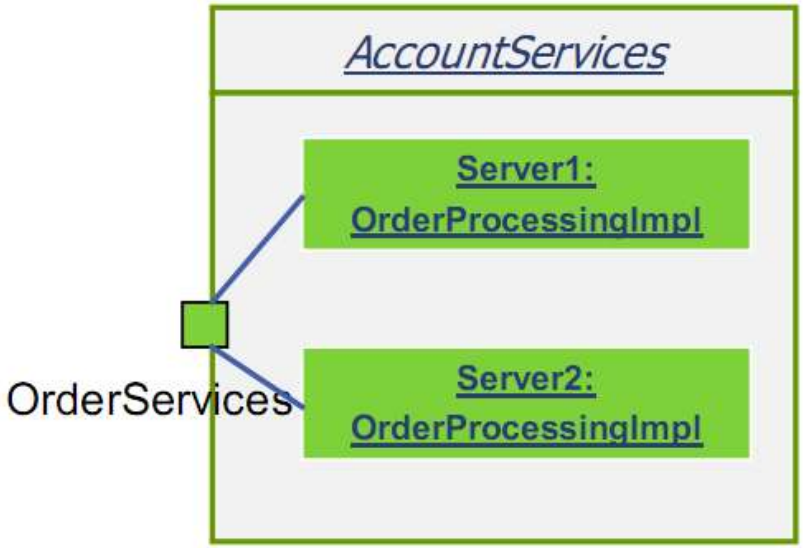
Composite Structures: Ports

- A port is a way to offer functionality from a composite structure without exposing the internal details of how that functionality is realized
- Exposing the functionality through a port allows the class to be used by other classes that conforms to the port's specifications.

Composite Structures: Realizing Port Implementations

A delegate connector is used for defining the internal workings of a component's external ports and interfaces.

Composite Structures: Multiple Connectors

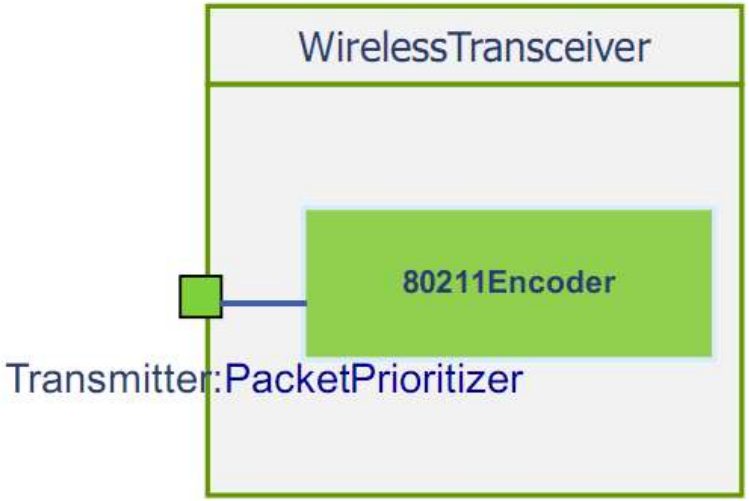


There are two instances of **OrderProcssingImpl**: Server1 and Server2.

When data is received at the **OrderServices** port it is forwarded to one of the two servers for processing

Composite Structures: Port Typing

- UML 2.0 allows to specify the type of a port using classes
- When a port is instantiated, the corresponding class manipulates the communication it receives before passing it to the realizing classifiers



- *Transmitter* is explicitly typed as *PacketPrioritizer*
- The *PacketPrioritizer* class could e.g., order high-priority messages before normal traffic

Composite Structures: Ports

- Ports are *optional!*
  - There is NOTHING you can do with ports that you cannot do without them
  - An operation of an internal part may be used directly by an external object (this structured class is called *transparent*)
    - ⇒Tightly couples the structured class internal structure with its environment
- Ports enforce encapsulation of internal parts

Composite Structures: Ports and Interfaces

- Ports are different than interfaces
  - Ports are instantiable while interfaces are not

- A port instance is a connection slot into an instance of a class that either
  - relays a message to an internal part (called *relay port*)
  - or accepts the message and hands it off to the object for handling (called *behavioral port*)
- Ports have identity
- Ports may have behavior

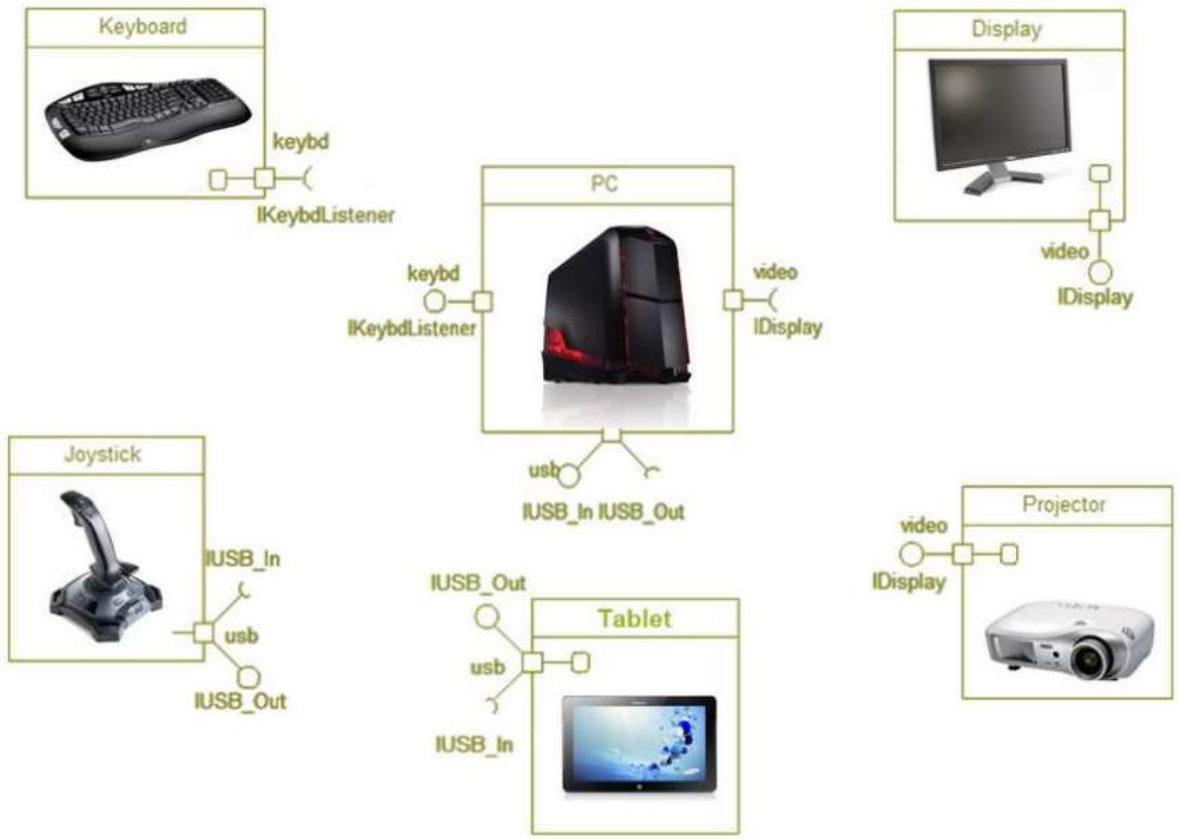
- An interface is a named collection of operations, but those operations are provided elsewhere
- They have no behavior
- They just allow a collection of services to be given a name
- The required and offered aspects form a contract to which the client and the server agree to adhere

Exercise 1: How can we model the following in UML?

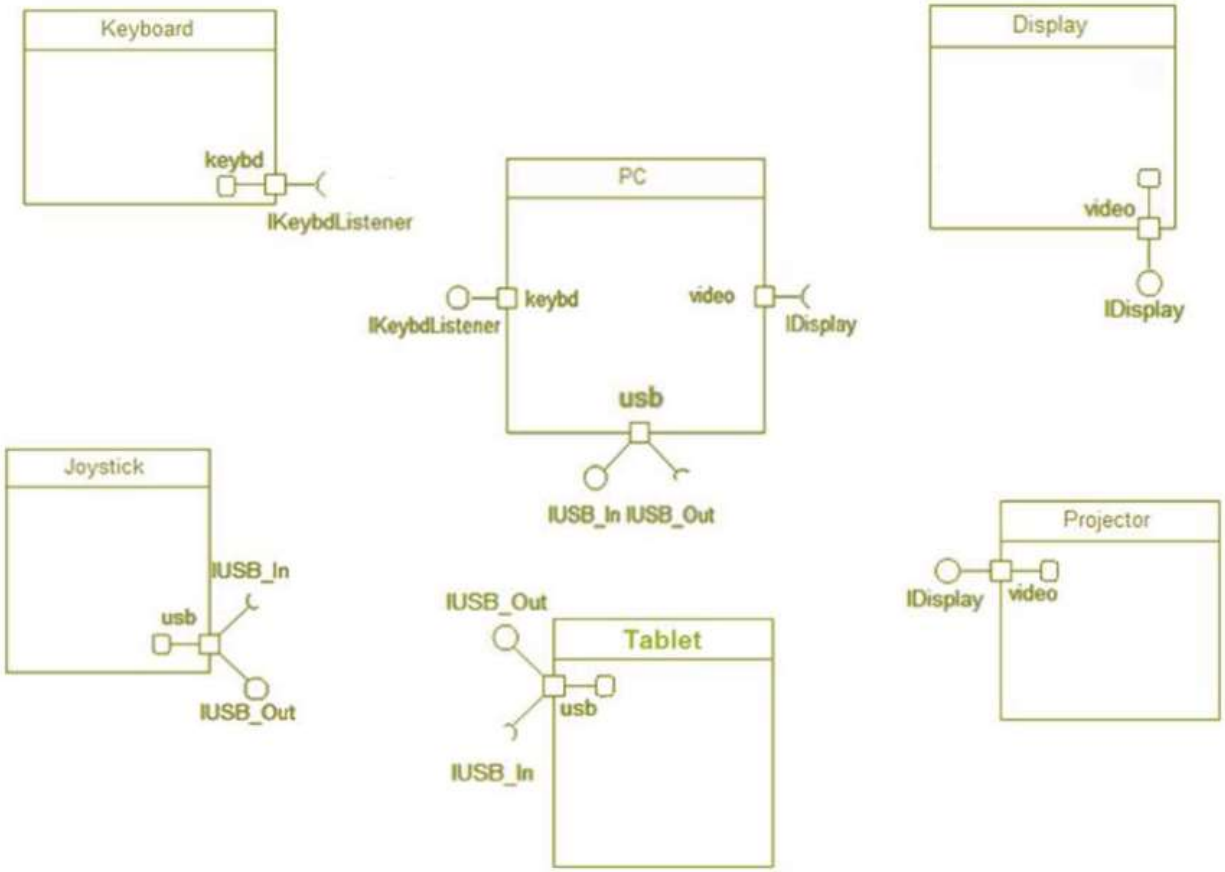
Exercise 1

- Define ports for each device

Exercise 1: Devices



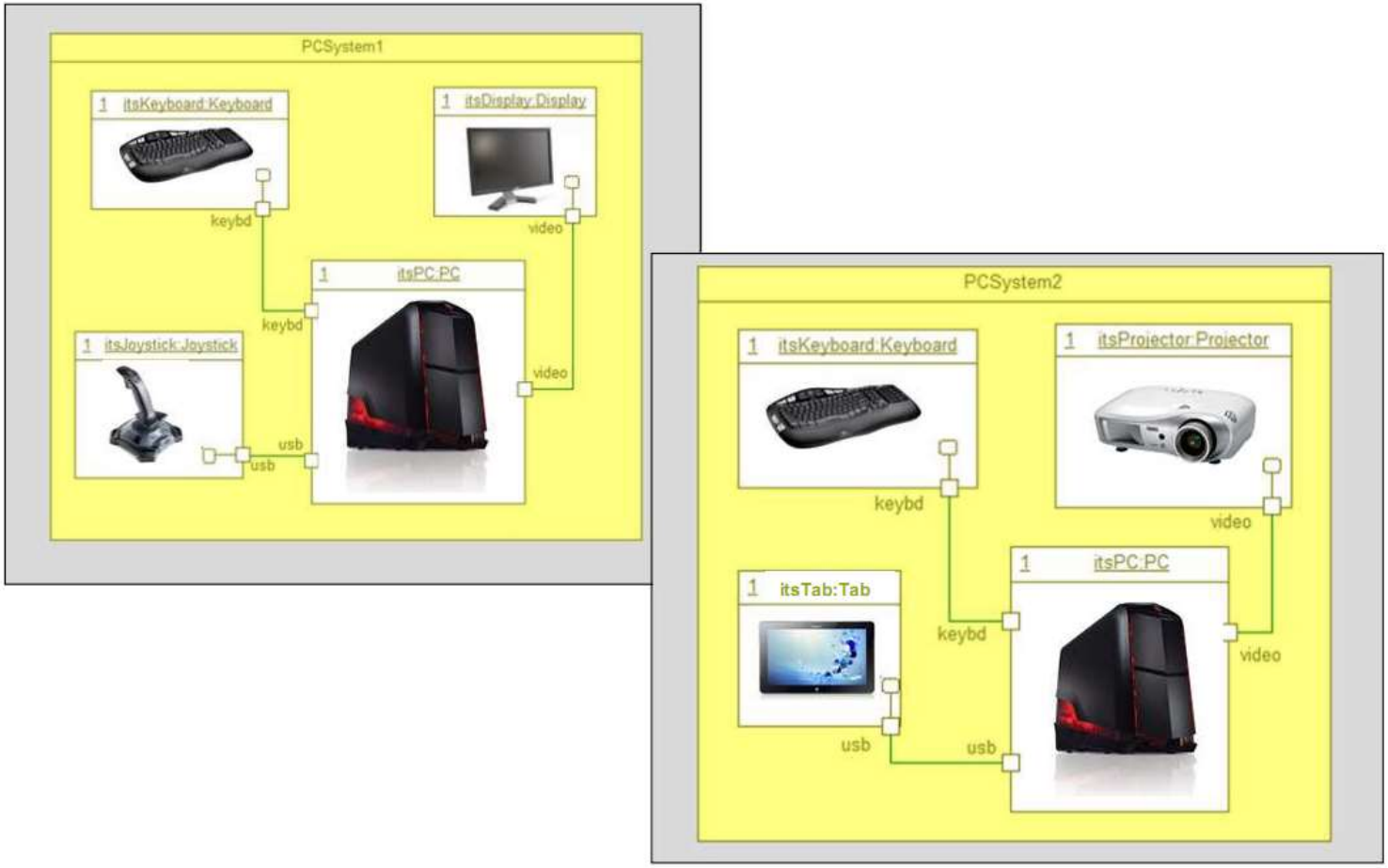
Exercise 1: UML 2.0 Ports



Exercise 1

2. We are interested in building two PC systems. *PCsystem1* consists of a display, keyboard, joystick and *PCsystem2* consists of a tablet, PC, keyboard and projector. Draw a composite structure diagram for the two systems.

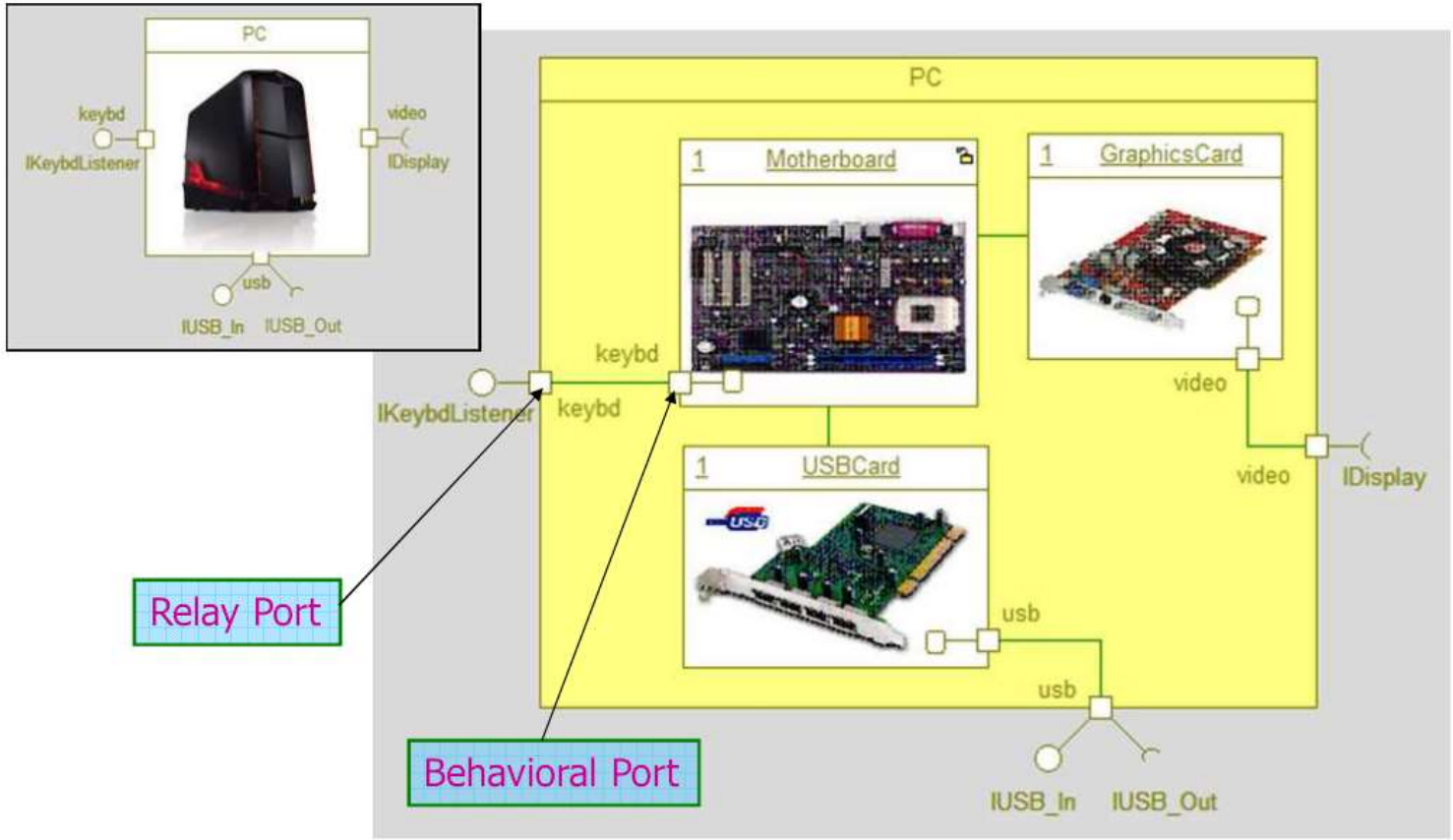
Exercise 1: PC System : Plug & Play



Exercise 1

3. Draw the internals of a PC in terms of the motherboard, graphics card and USBC card

PC Internals

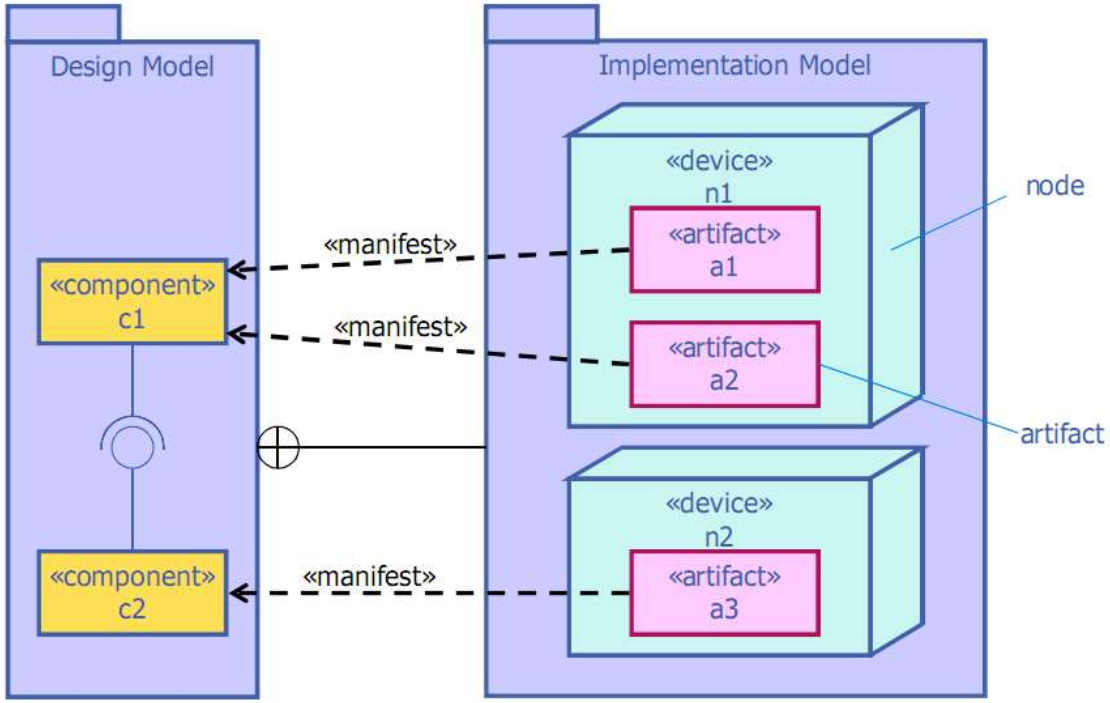


Outline

- Objectives
  - Discuss Structure Diagrams
  - *Class Diagram (Already covered)*
  - Component and Subsystem Diagrams
  - Package Diagrams
  - Composite Structure Diagrams
  - Deployment Diagram

Deployment Diagrams

- Deployment diagrams model the mapping of software pieces of a system to the hardware that is going to execute it



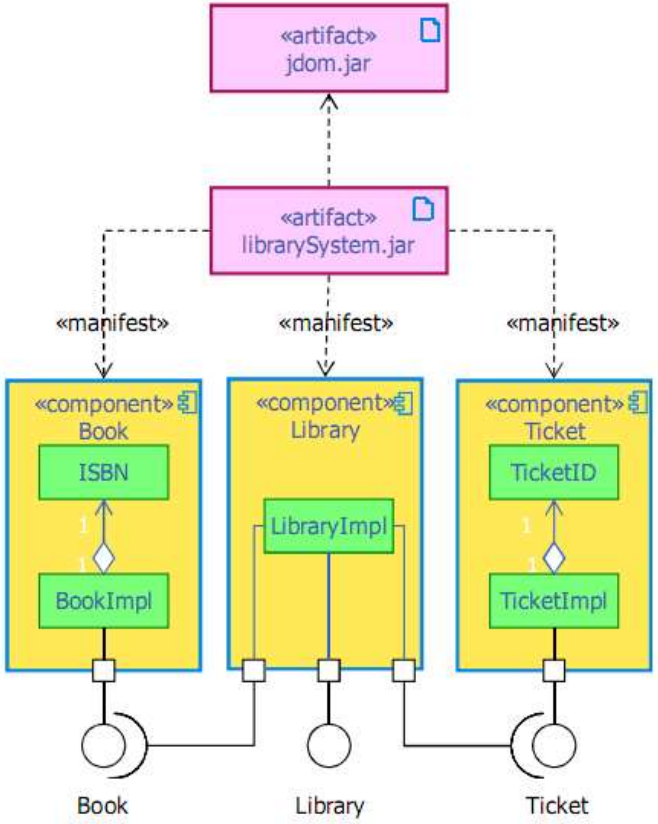
Deployment Diagrams: Artifacts

- An artifact represents a physical piece of information, e.g., user's manual, executable, etc.

- Artifact instances represent particular *copies* of artifacts
  - You can have an artifact named loggin.jar that represents your loggin framework implementation. You may have several web applications installed on a server, each with their own copy of loggin.jar

Deployment Diagrams: Artifacts and Components

- Artifacts provide the physical manifestation for one or more components
- Artifacts can contain other artifacts
- Artifacts can depend on other artifacts



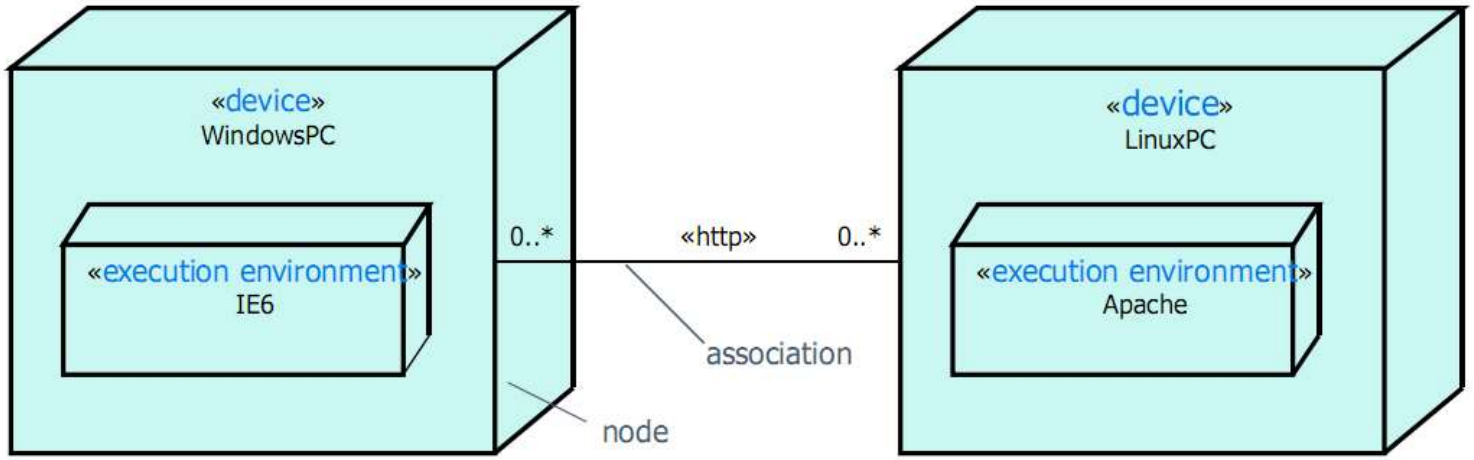
Deployment Diagrams: Artifacts Standard Stereotypes

- UML 2 provides a small number of standard stereotypes for artifacts

artifact stereotype	semantics
«file»	A physical file
«deployment spec»	A specification of deployment details (e.g. web.xml in J2EE)
«document»	A generic file that holds some information
«executable»	An executable program file
«library»	A static or dynamic library such as a dynamic link library (DLL) or Java Archive (JAR) file
«script»	A script that can be executed by an interpreter
«source»	A source file that can be compiled into an executable file

Deployment Diagrams: Nodes – descriptor form

- A **node** is a physical entity that can execute artifacts.

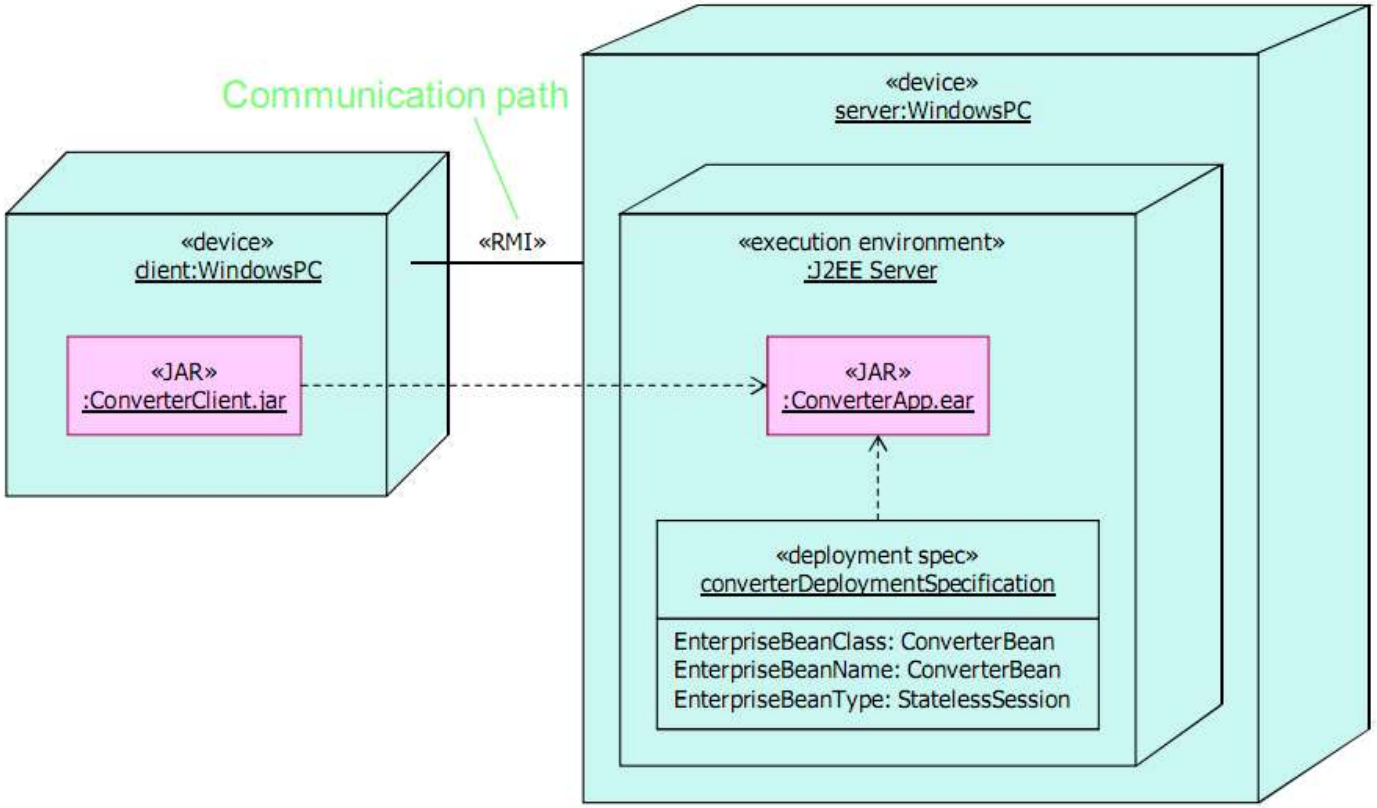


- A node represents a type of computational resource
  - e.g., *WindowsPC*
- Standard stereotypes are «***device***» and «***execution environment***»

Deployment Diagrams: Node Standard Stereotype

- A «***device***» is a node that represents a physical machine capable of performing calculations
- An «***execution environment***» is a node that represents a software configuration hosting specific types of artifacts
  - An execution environment is expected to provide specific services to hosted artifacts by means of interfaces
  - Example: a *Java 2 Enterprise Edition* (J2EE) application expects to run in an environment called *Application Server*

Deployment


Communication path
«device»
server:WindowsPC
«device»
dient:WindowsPC
«RMI»
«execution environment»
:J2EE Server
«JAR»
:ConverterClient.jar
«JAR»
:ConverterApp.ear
«deployment spec»
converterDeploymentSpecification
EnterpriseBeanClass: ConverterBean
EnterpriseBeanName: ConverterBean
EnterpriseBeanType: StatelessSession