Software Architecture: Structural Modeling

Presenter: Dr. Ha Viet Uyen Synh.

PART A

PACKAGE DIAGRAM

Motivation

Typical systems grow complex, i.e., hundreds of classes

- How do I control the complexity?
- A class diagram should fit on a sheet of paper (A4).
- A developer may grasp 7(+/-2) classes at a glance.

- How do I restrict scope (name space) and control change propagation?

- How do I build nested hierarchies of classes?

- How do I build layered architectures of classes?

- How do I show dependencies between classes at a higher abstraction level?

- How do I slice development work within a team?

- How do I specify interfaces between groups of classes, i.e., distinguish public from implementation-dependent classes?

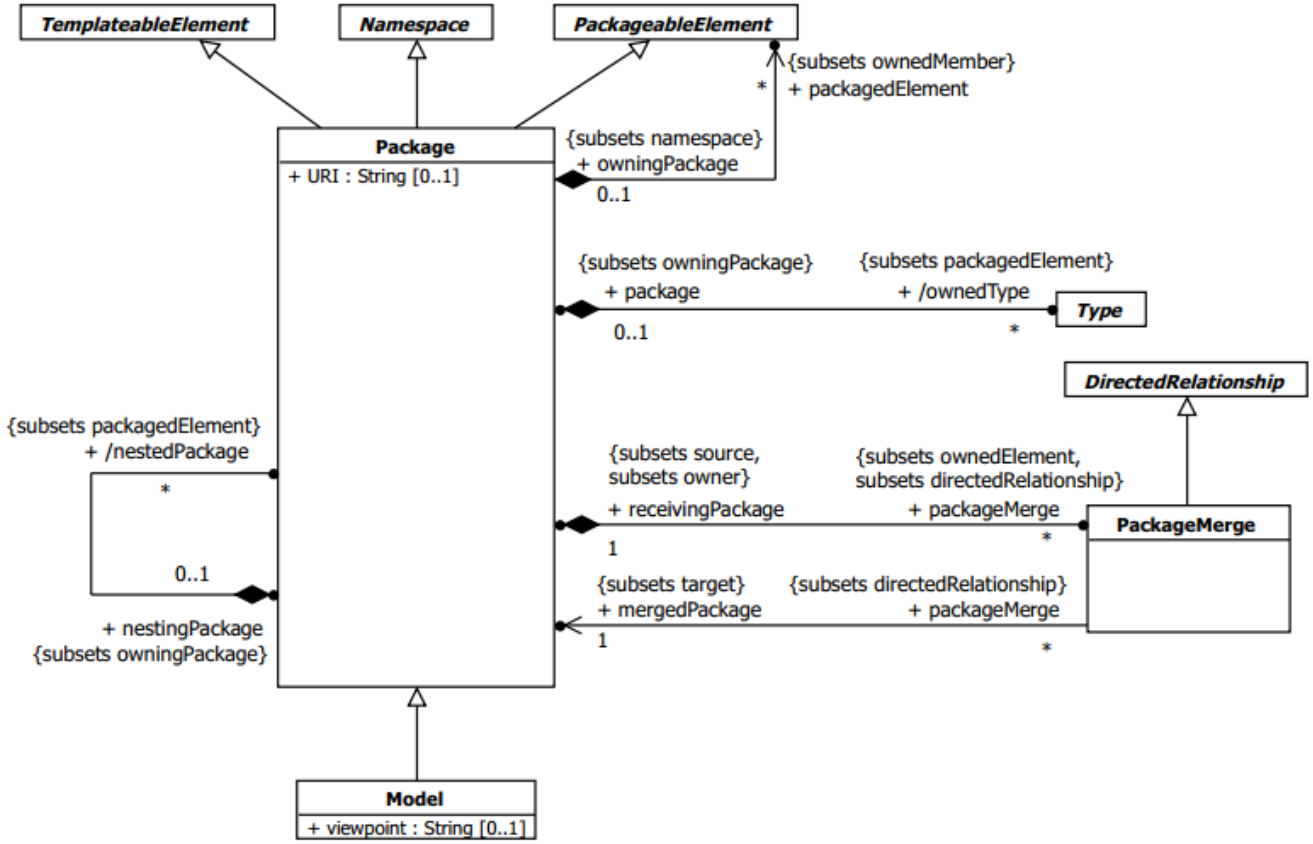
Packaging

One of the oldest questions arising in software development is:

How do you break down a **large** system into **smaller** systems?

Functional Decomposition	**OO-Packaging**
Map the overall system down to functions and sub-functions, starting from the use case.	Group classes into high-level units. Structure dependent classes and diagrams into logical sets.

Syntax

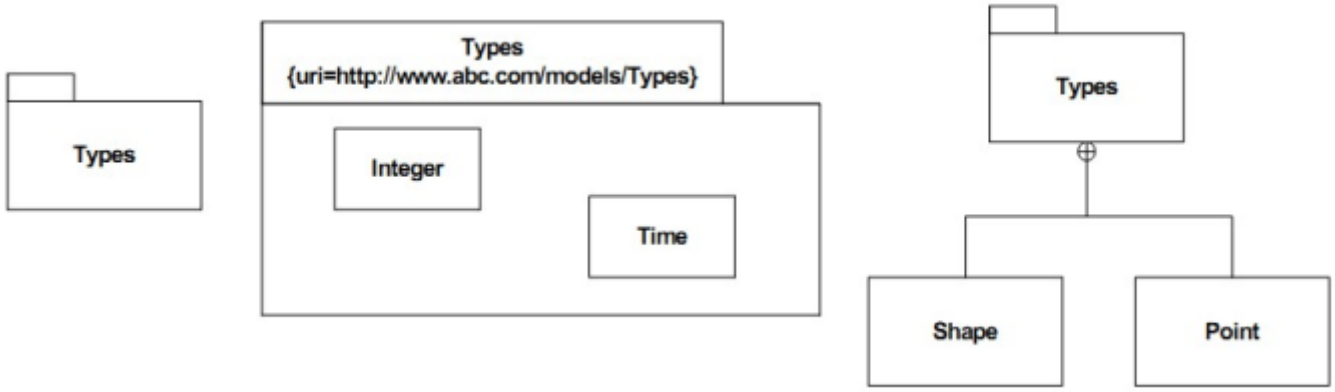

TemplateableElement
Namespace
PackageableElement
{subsets ownedMember}
+ packagedElement
*
Package
+ URI : String [0..1]
{subsets namespace}
+ owningPackage
0..1
{subsets owningPackage}
+ package
0..1
{subsets packagedElement}
+ /ownedType
*
Type
DirectedRelationship
{subsets packagedElement}
+ /nestedPackage
*
0..1
+ nestingPackage
{subsets owningPackage}
{subsets source,
subsets owner}
+ receivingPackage
1
{subsets ownedElement,
subsets directedRelationship}
+ packageMerge
*
PackageMerge
{subsets target}
+ mergedPackage
1
{subsets directedRelationship}
+ packageMerge
*
Model
+ viewpoint : String [0..1]

Package Diagrams

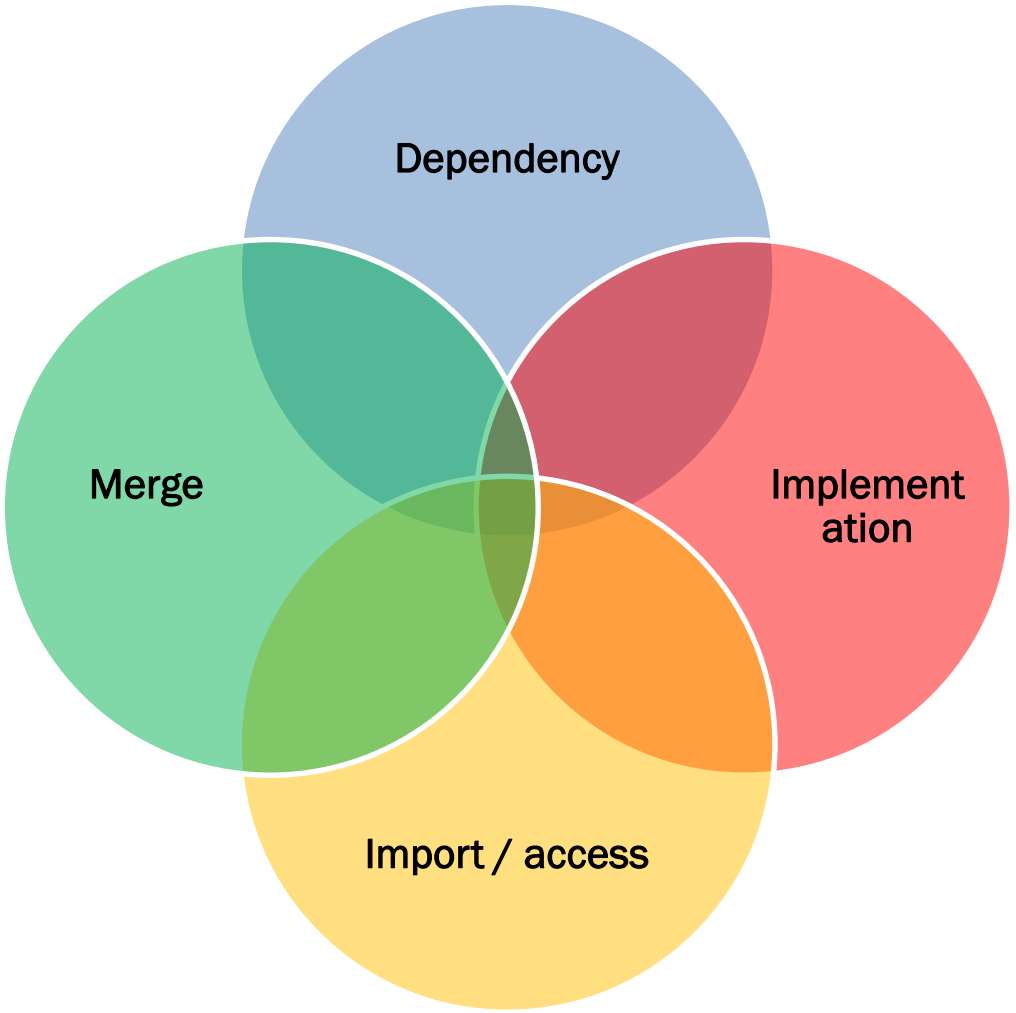
- Packages
  - groups of “basic elements”, e.g., classes or use cases
  - namespaces, i.e., all members should have unique names
  - represented as file folders
  - can contain other packages, creating hierarchy

- Relations
  - dependencies, implementations, …
  - *imports* and *merges*

Package


Types
Types
{uri=http://www.abc.com/models/Types}
Integer
Time
Types
Shape
Point

Relationships



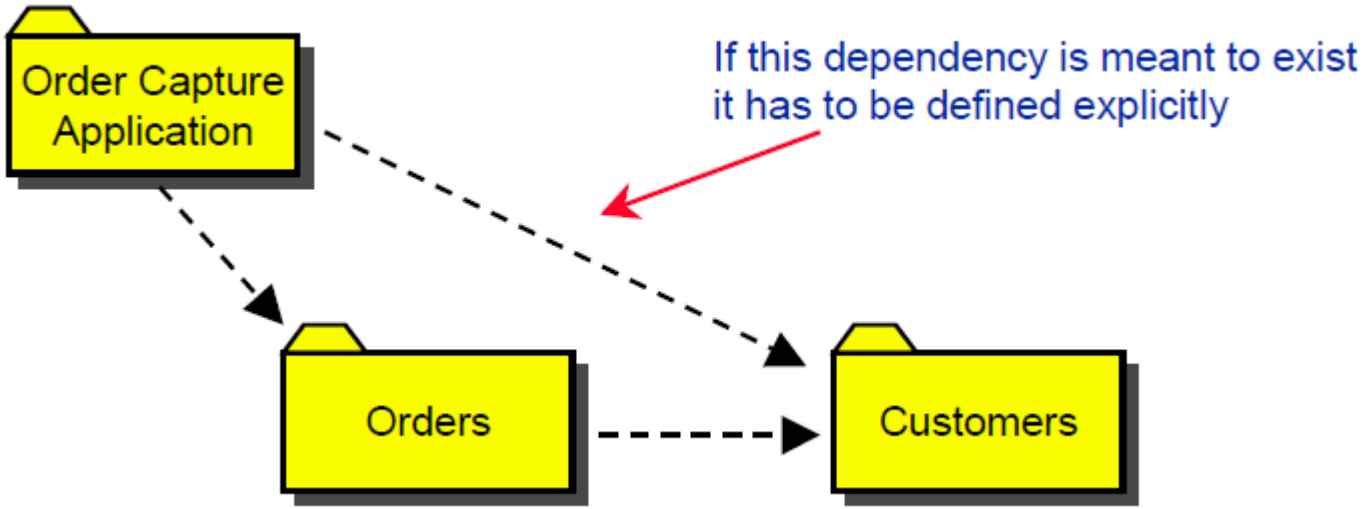
Dependencies

Package A depends on package B if A contains a class which depends on a class in B

Dependencies

There is a vital difference between package dependencies and compilation dependencies:

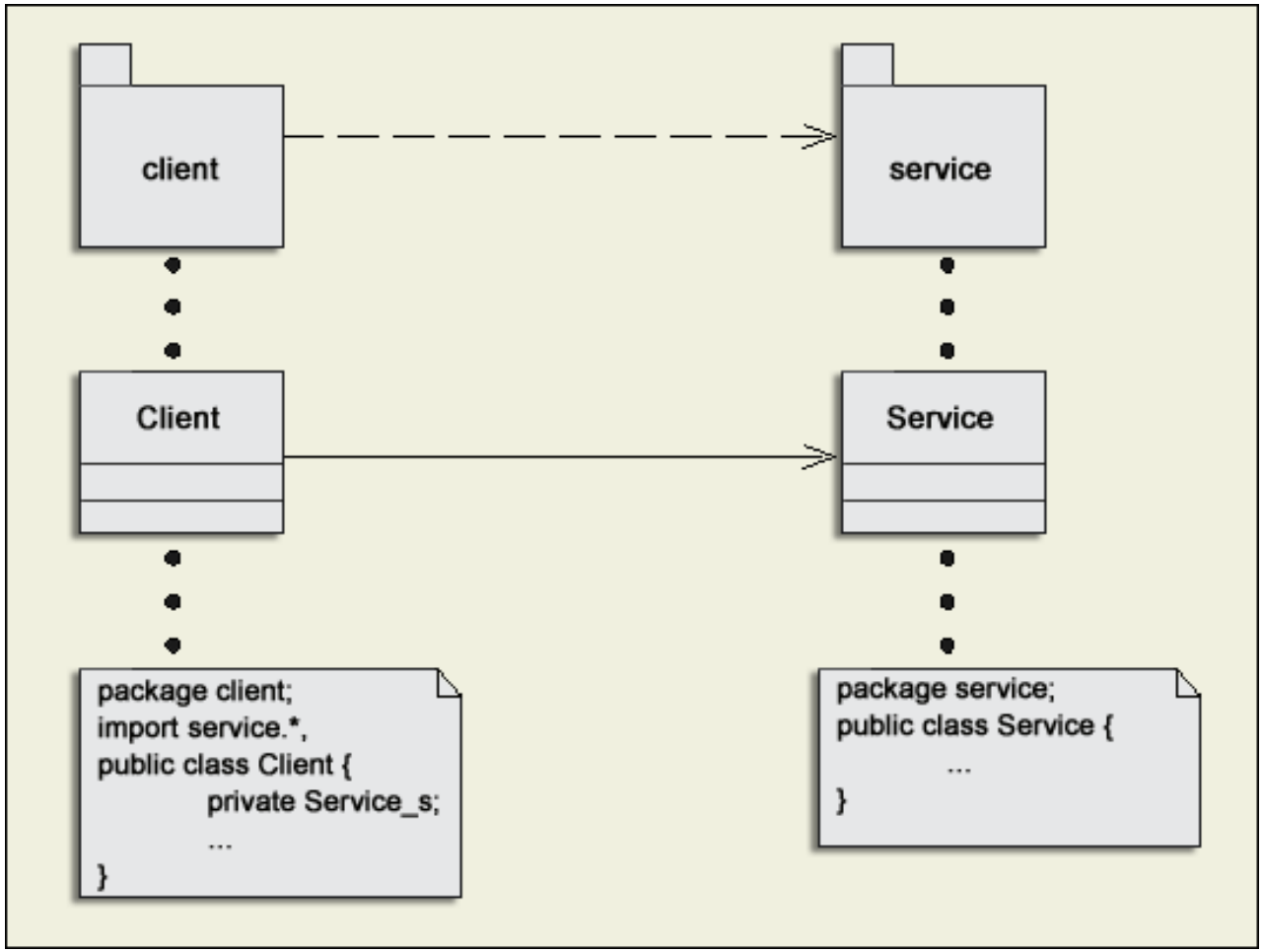
Dependencies between packages are not transitive



If there are cycles in dependencies, these cycles should be localized, and, in particular, should not cross the tiers

Package Relationship

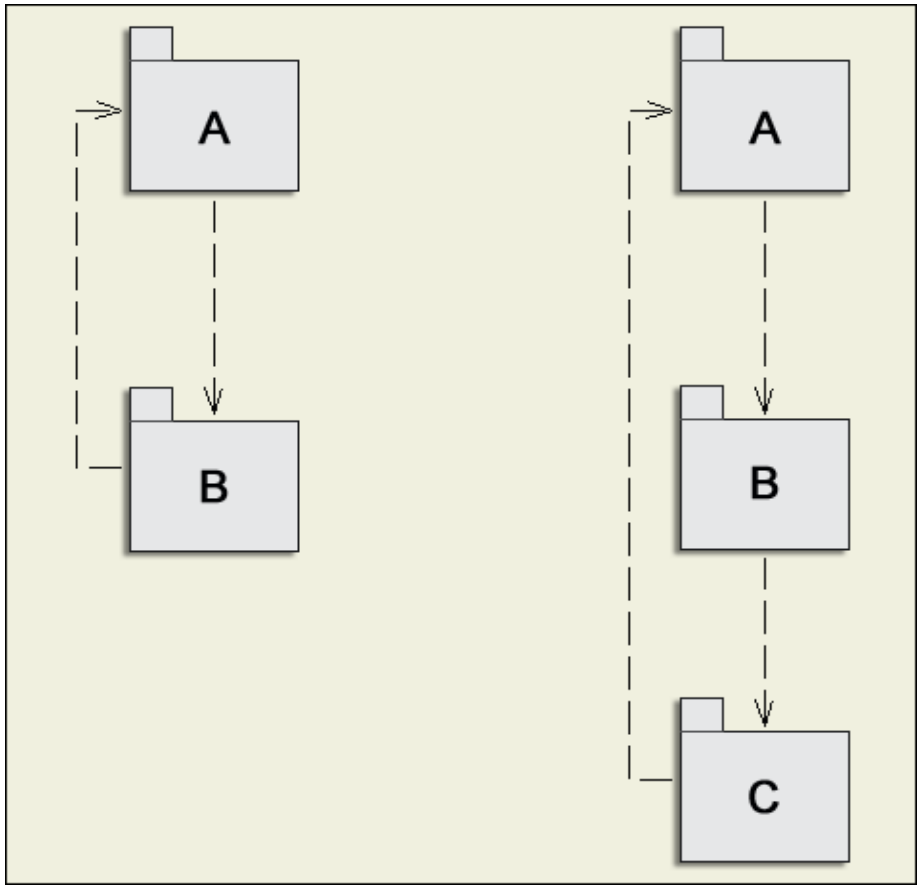
Package dependency diagram :



Package Relationship

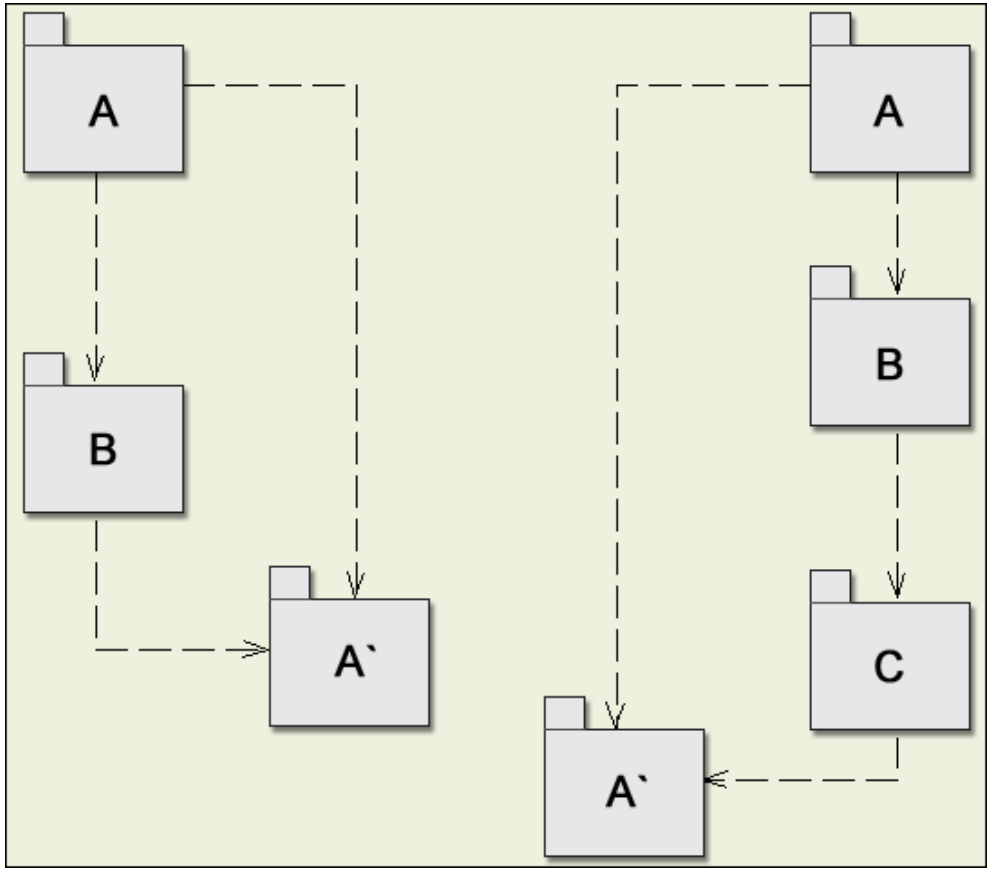
Two types of relationship: Unidirectional and Bidirectional

Unidirectional Diagram :



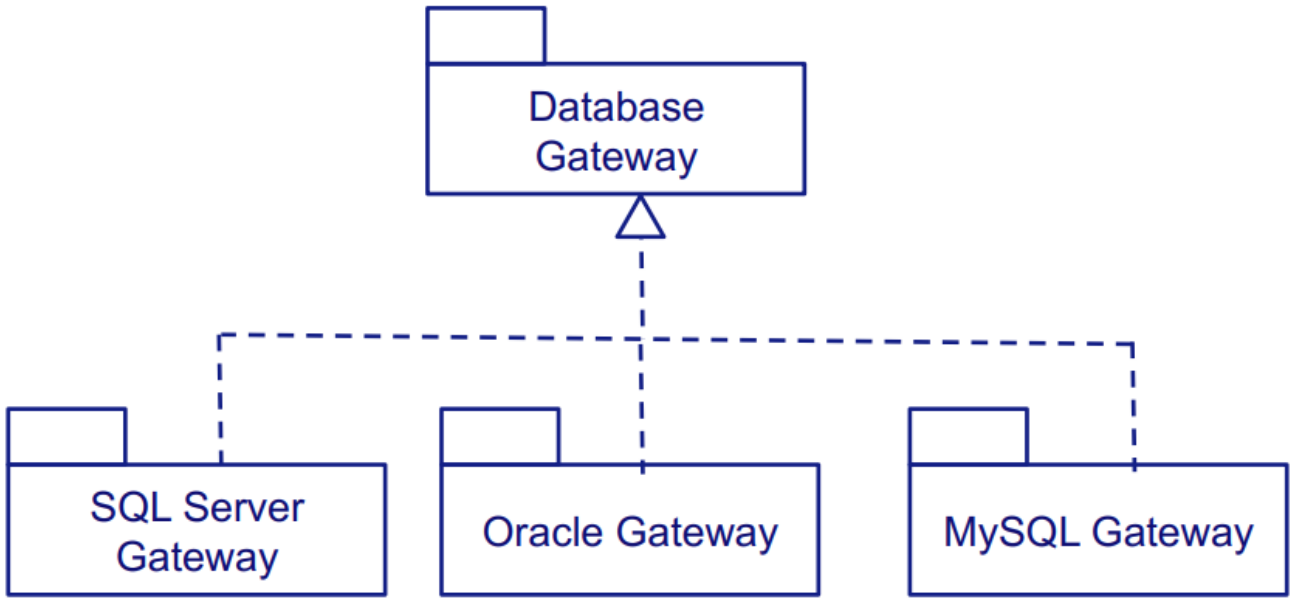
Package Relationship

Bidirectional Diagram :



Implementations

Meaningful if multiple variants are present



Import / access

To understand the **import / access** relation between packages

- We need to know how **elements can reference each other**
- What does an **element import / access** mean
- How this notion can be generalized to **packages**

Import / access

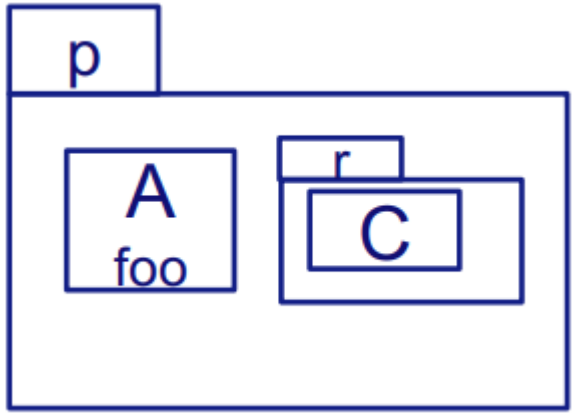
Fully qualified name: a globally unique identifier of a package, class, attribute, method.

Fully qualified name is composed of

- **qualifier**: all names in the hierarchic sequence above the given element
- the **name** of the given element itself

Notation

- UML, C++, Perl, Ruby **p::A::foo**, **p::r::C**
- Java, C# **p.A.foo**, **p.r.C**



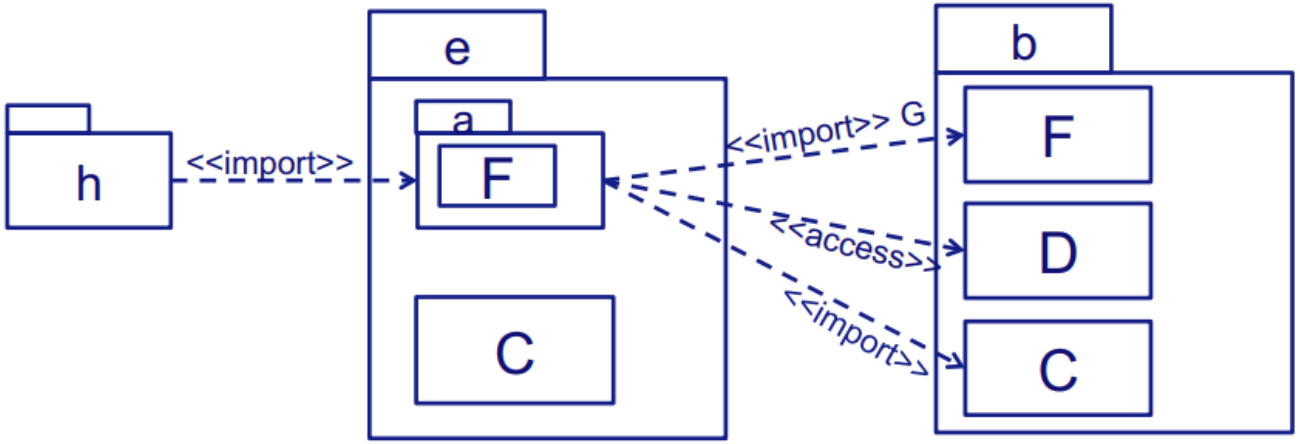
Import / access

Element import allows an element in another package to be referenced using its name without a qualifier

- **<<import>>** imported element within importing package is public

- **<<access>>** imported element within importing package is private

Exercise #1



Determine scopes of sub-packages in **b**, **e** and **h**

Package import

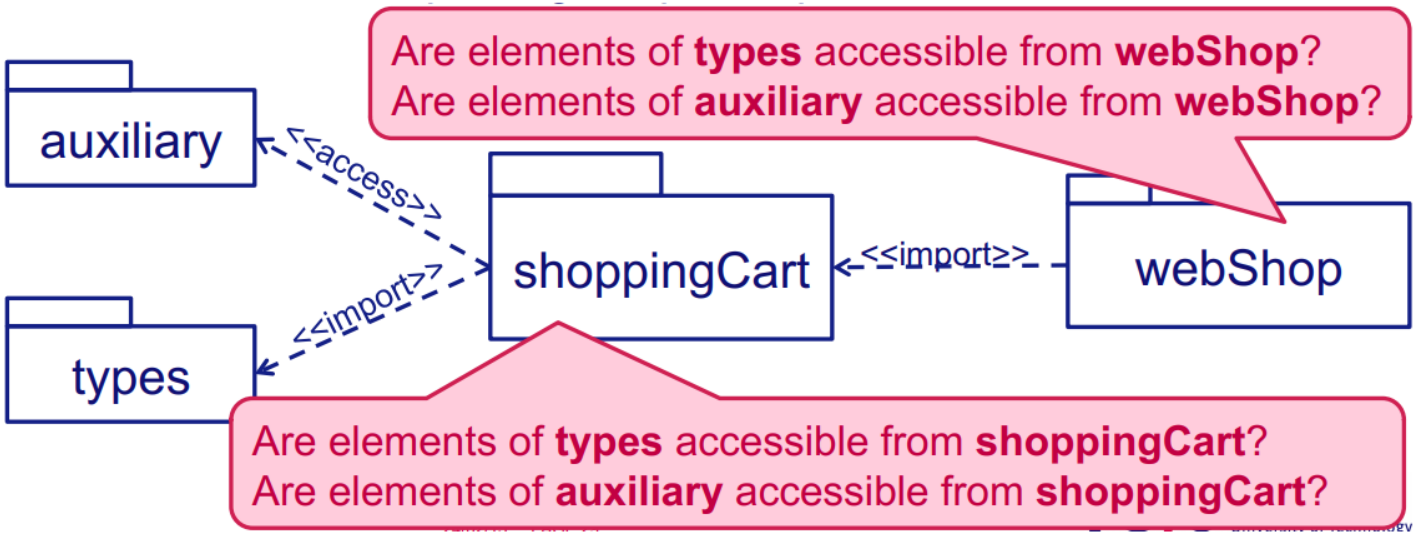
A **package import** is a directed relationship that identifies a package whose members are to be imported

- Conceptually equivalent to having an element import to each individual member of the imported package

- **<<import>>** if package import is public

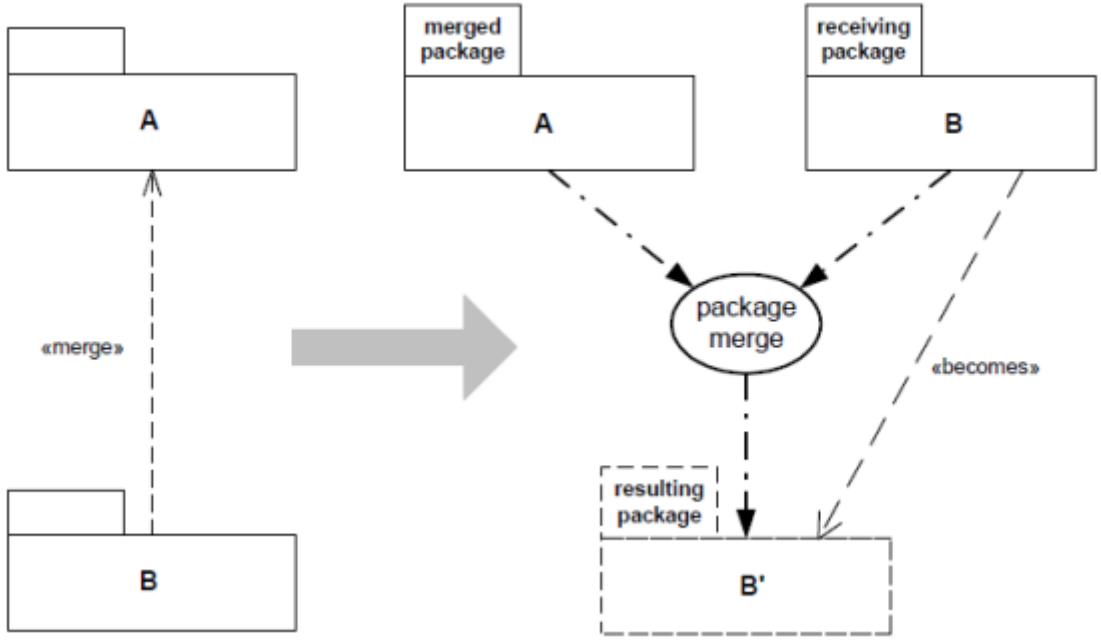
- **<<access>>** if package import is private

Exercise #2


Are elements of **types** accessible from **webShop**?
Are elements of **auxiliary** accessible from **webShop**?
auxiliary
<<access>>
shoppingCart
<<import>>
webShop
<<import>>
types
Are elements of **types** accessible from **shoppingCart**?
Are elements of **auxiliary** accessible from **shoppingCart**?

Merge

A *PackageMerge* is a directed relationship between two Packages that indicates that the contents of the target *mergedPackage* are combined into the source *receivingPackage* according to a set of rules

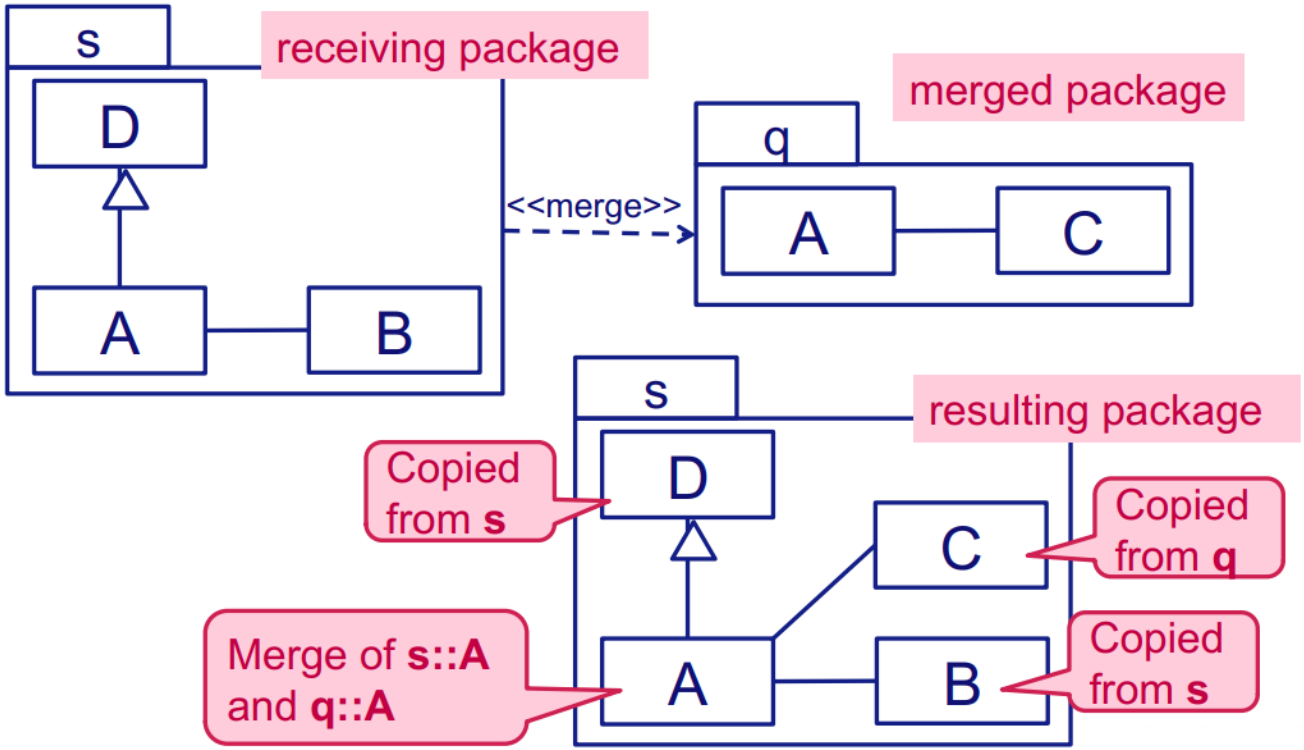


Merge

Merge is **possible** only if

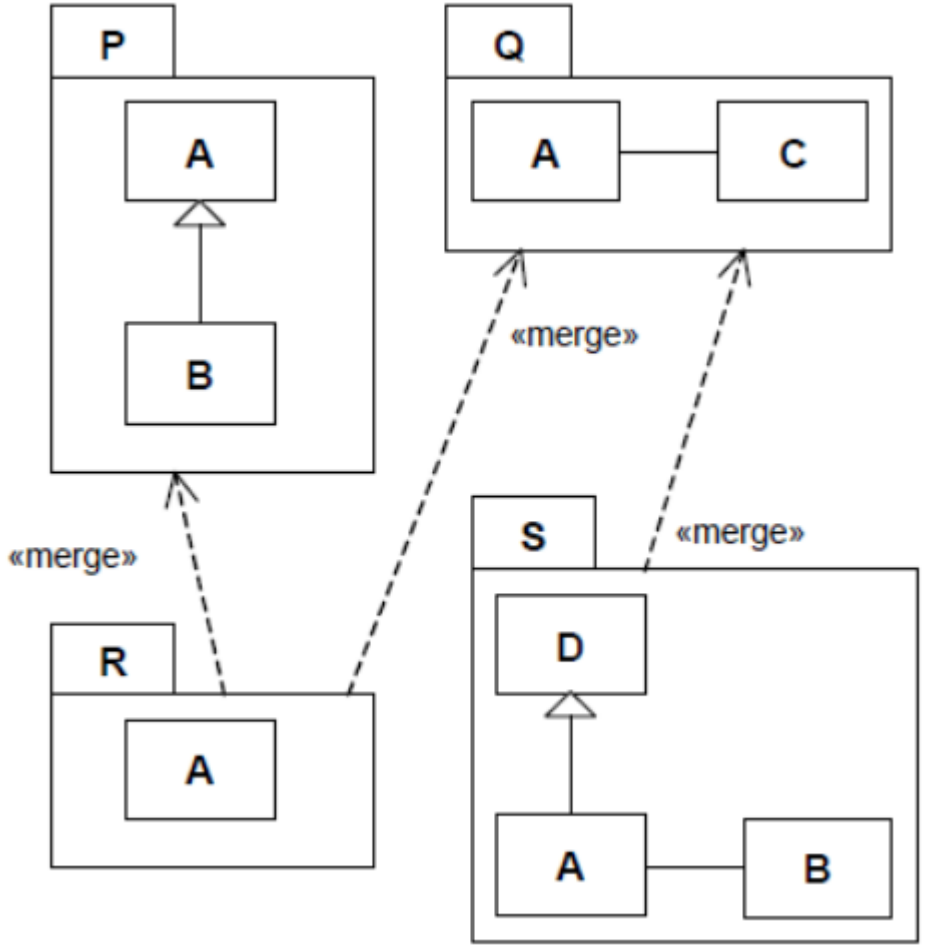
- There is *no cycle* on “merge” dependencies
- Receiving package does *not contain* the merged package
- Receiving package is *not contained* in the merged package
- Receiving element *cannot have references* to the merged element
- Matching typed elements should have the *same type* (class) or a common supertype (superclass)

Example

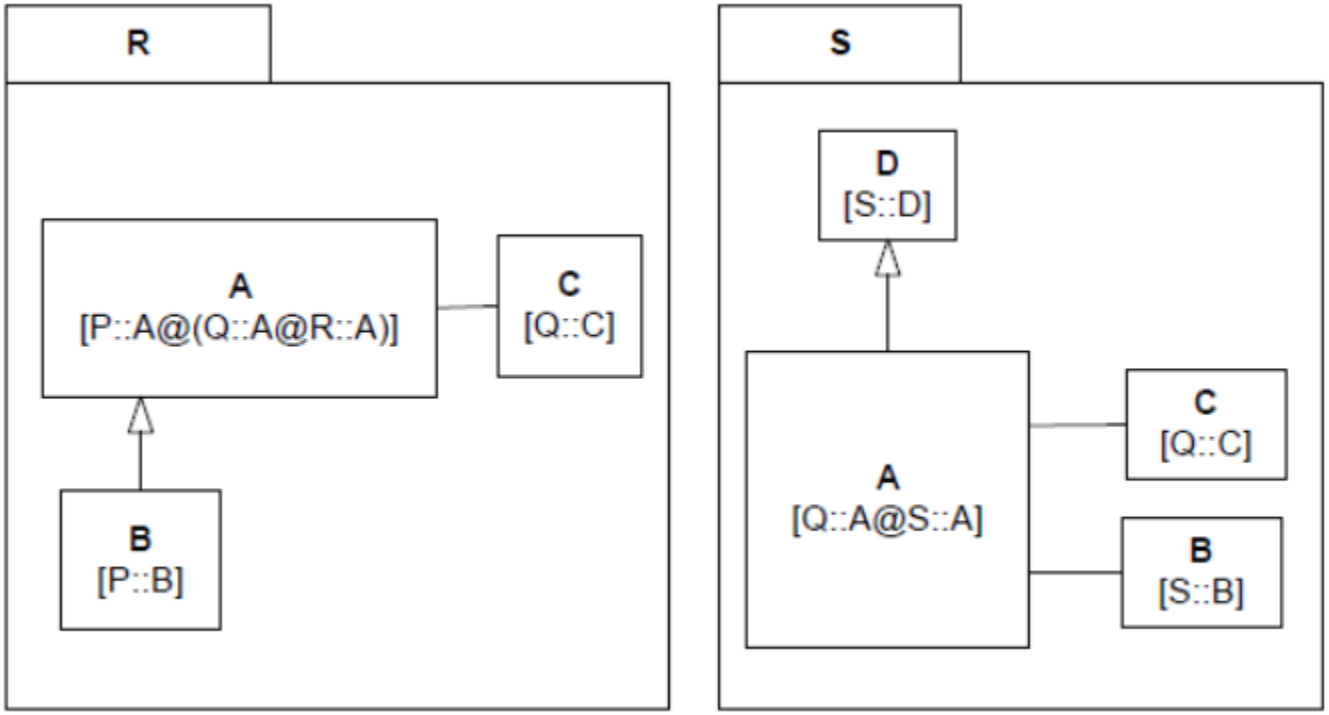

s
receiving package
D
A
B
<<merge>>
merged package
q
A
C
s
resulting package
Copied from s
D
C
Copied from q
Merge of s::A and q::A
A
B
Copied from s

Exercise #3

Estimate the concept of packages R and S

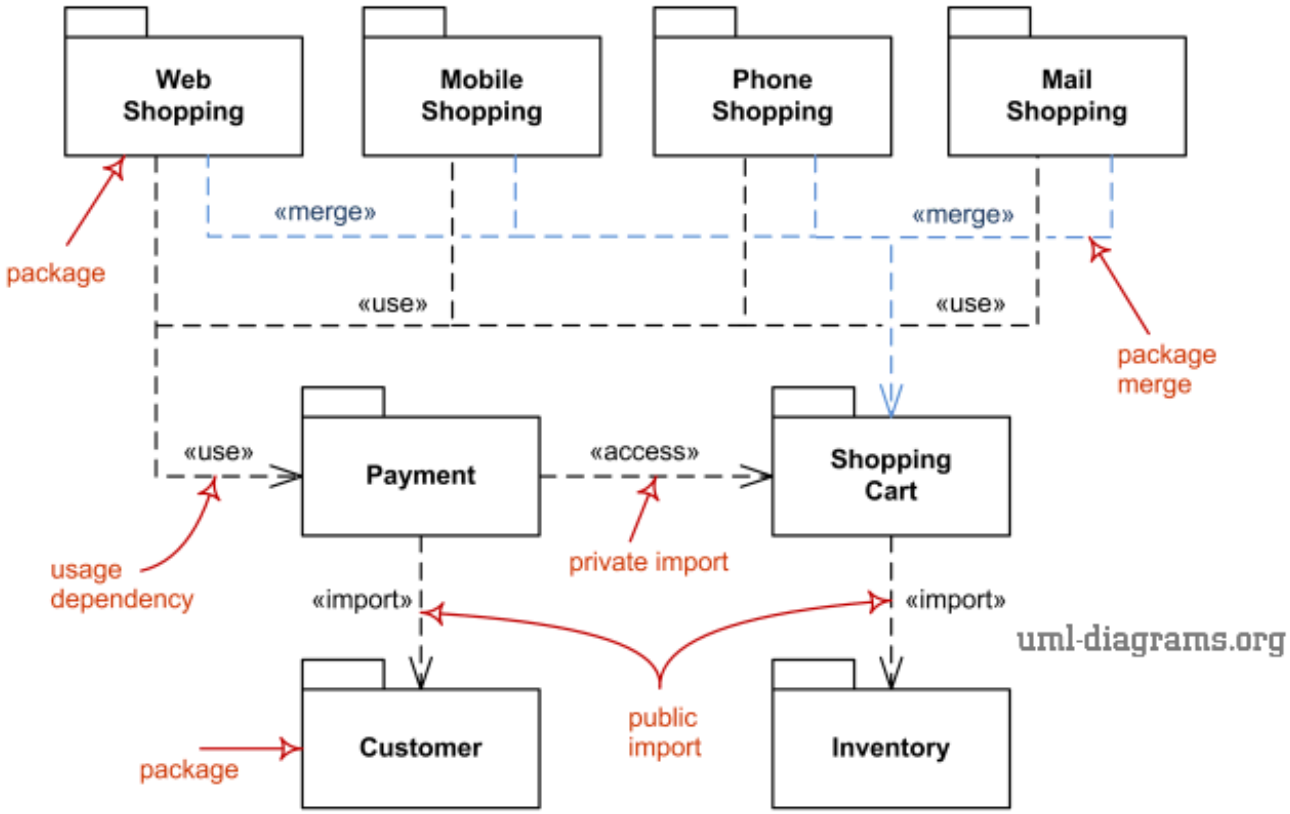


Solution



where X@Y signifies the resulting element from the merge transformation applied to matching receiving element X and merged element Y.

Summary


Web Shopping
Mobile Shopping
Phone Shopping
Mail Shopping
package
«merge»
«merge»
«use»
«use»
package merge
«use»
Payment
«access»
Shopping Cart
usage dependency
private import
«import»
«import»
uml-diagrams.org
public import
package
Customer
Inventory

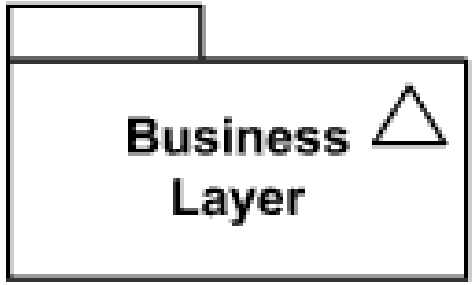
Model

Model is a package which captures a view of a system.

View of the system defined by its purpose and abstraction level.

Model is notated using the ordinary package symbol (a folder icon) with a small triangle in the upper right corner of the large rectangle.

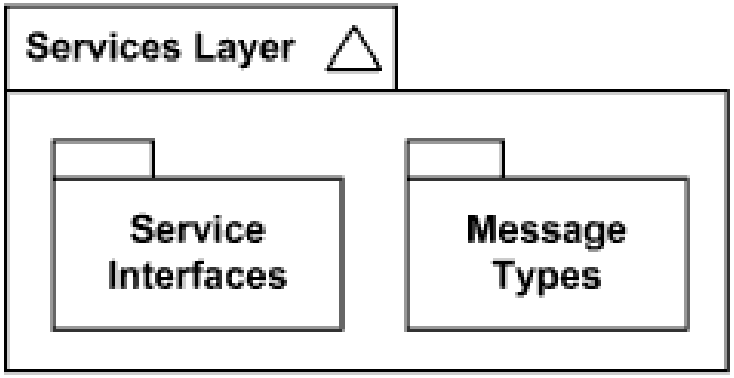
Business layer model :



Model

If contents of the model are shown within the large rectangle, the triangle may be drawn to the right of the model name in the tab.

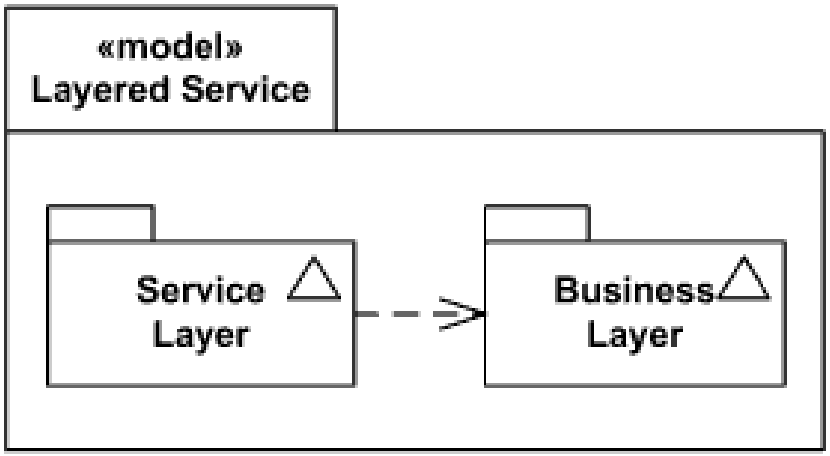
Service Layer model contains service interfaces and message types.



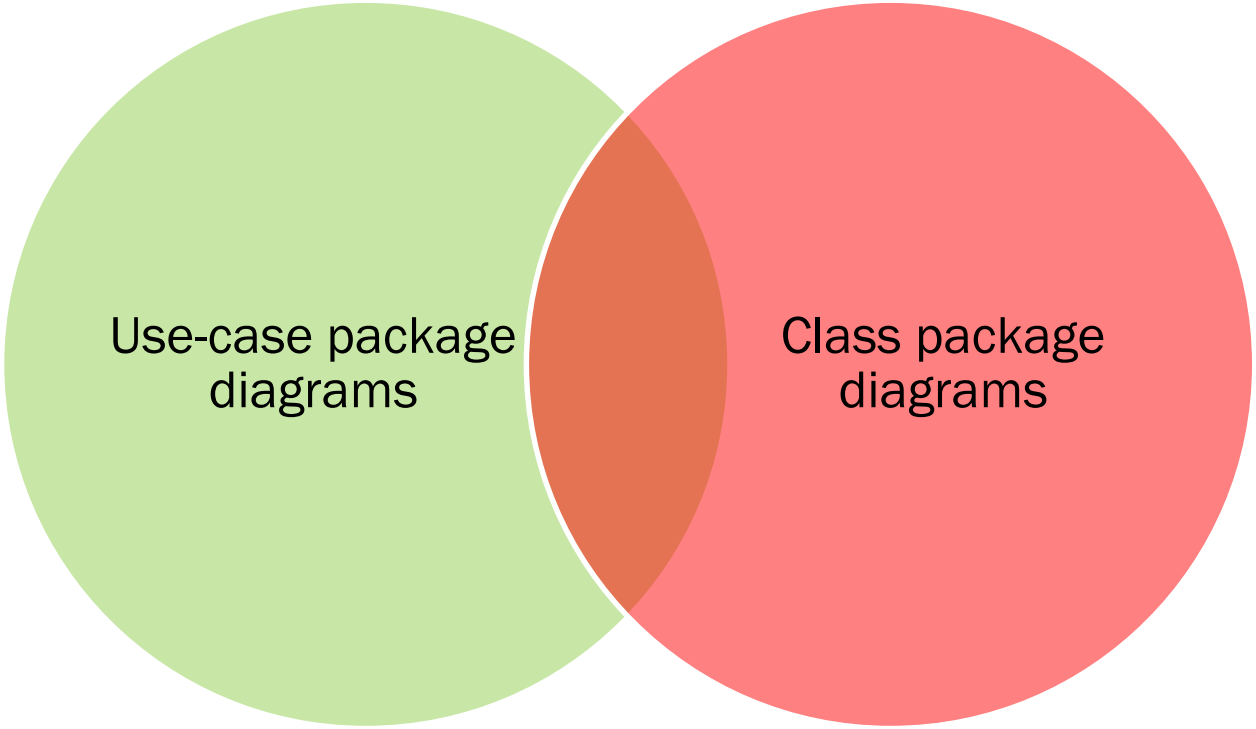
Model

Model could be notated as a package with the keyword «model» placed above the name of the model.

Stereotyped model Layered Service :



Types of Package Diagrams



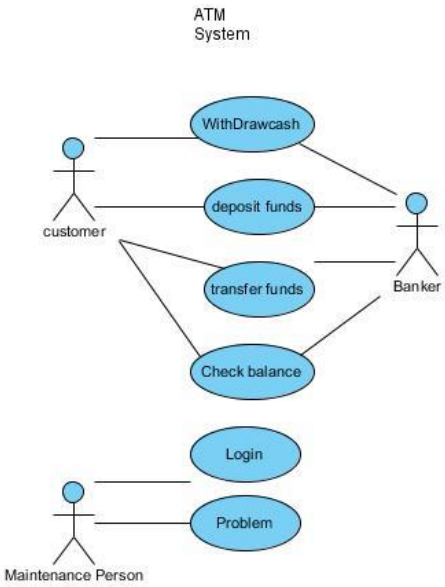
Use-Case Package Diagrams

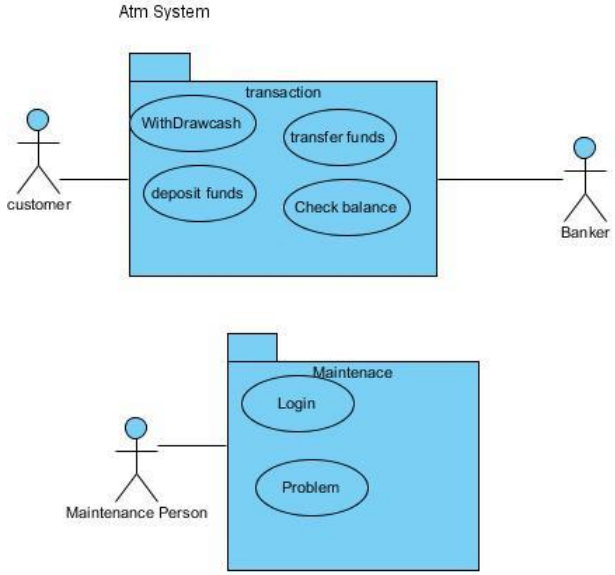
Heuristics to organize use cases into packages:

- Keep *associated use cases* together: included, extending and inheriting use cases belong in the same package.

- Group use cases on *the basis of the needs* of the main actors.

Use Case Package Diagram

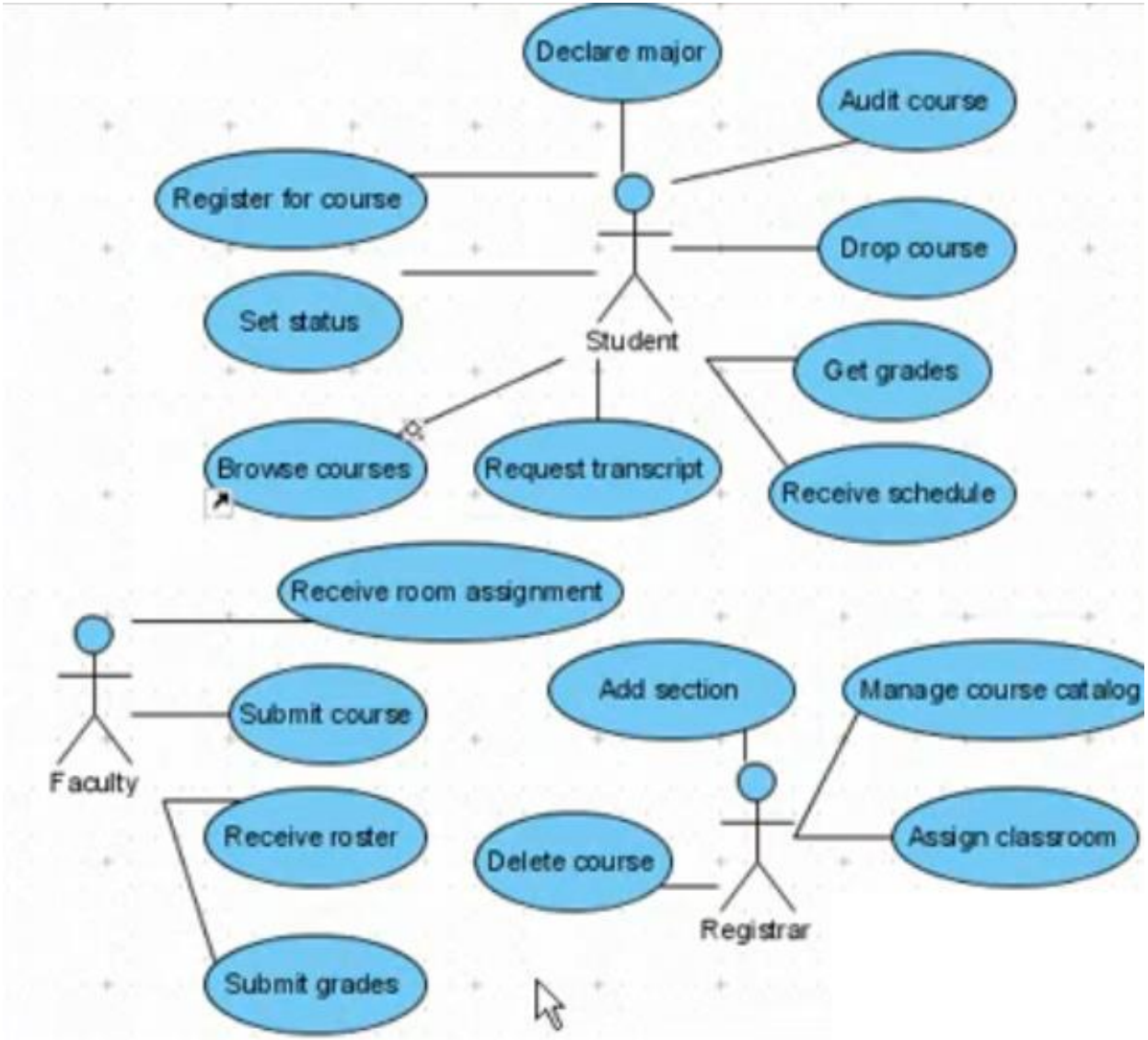




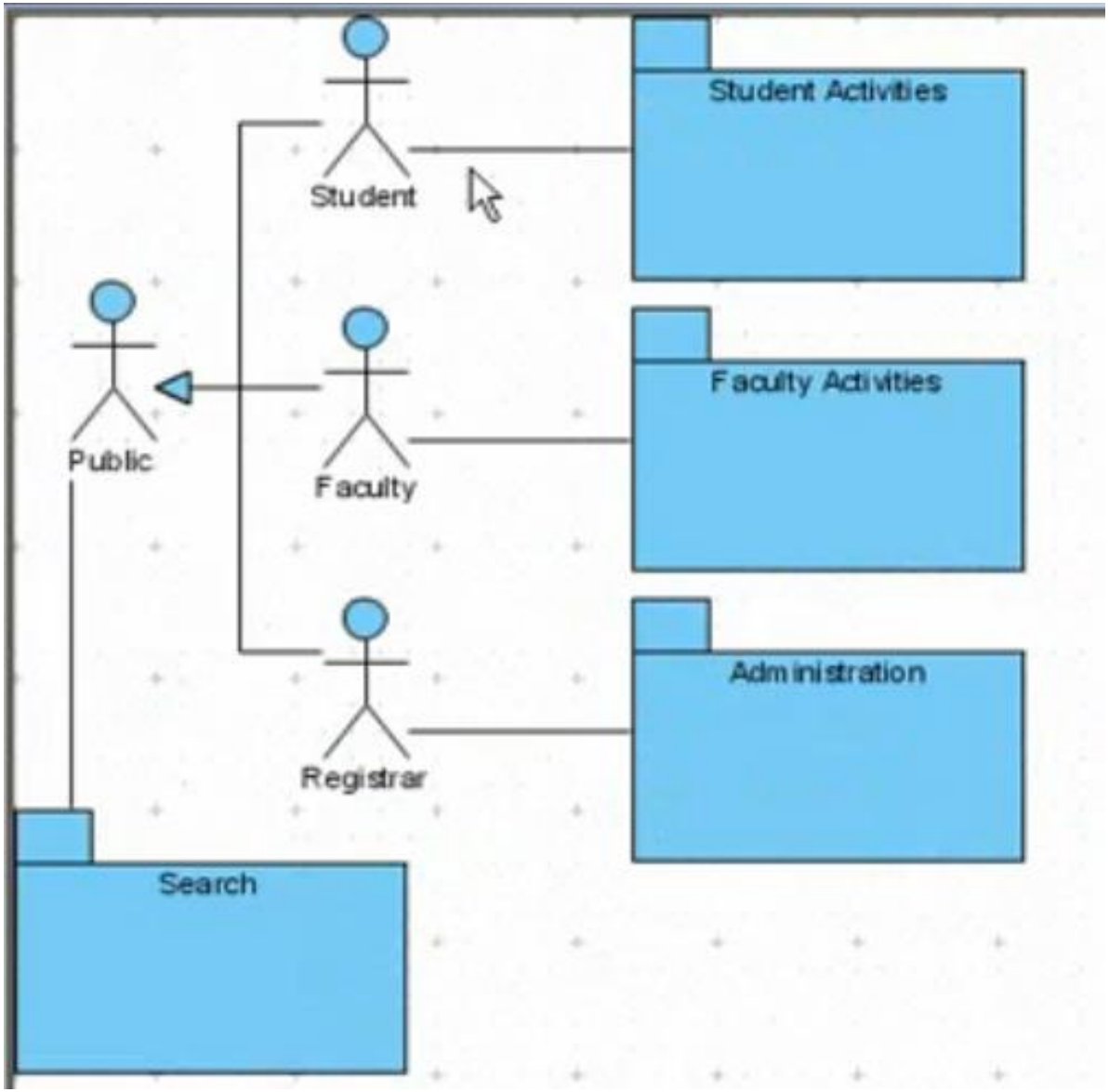
Use case Diagram ➔➔

Use case Package Diagram

Use Case Package Diagram



Use Case Package Diagram

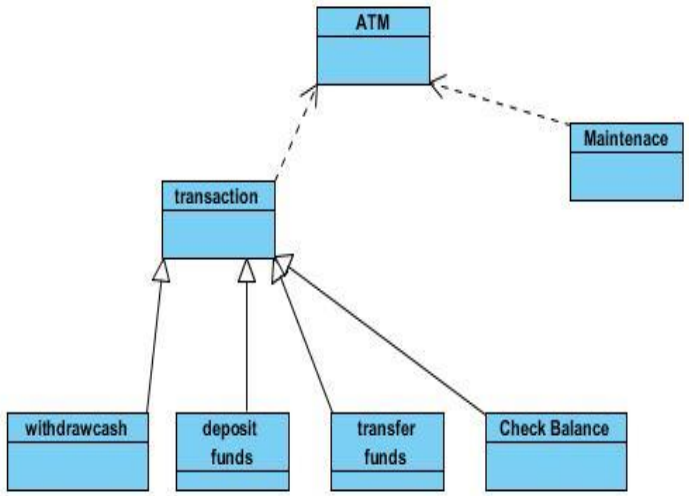


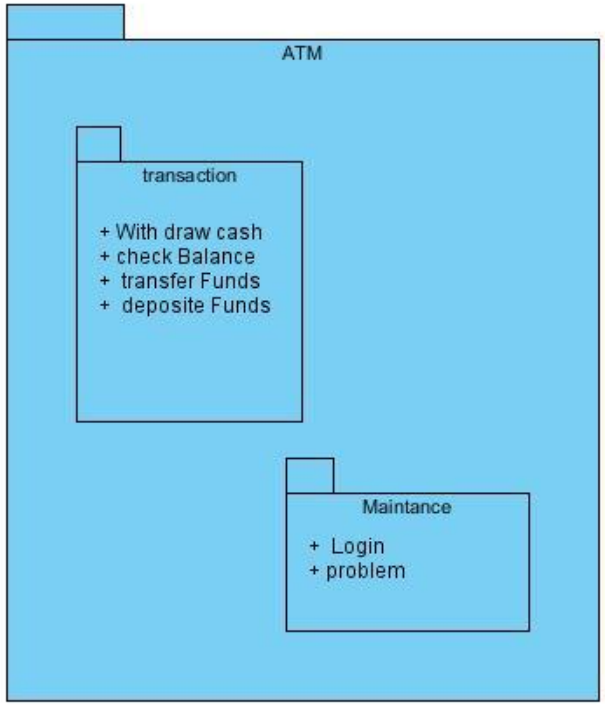
Class Package Diagrams

Heuristics to organize classes into packages:

- Classes of a *framework* belong in the same package.
- Classes in the same *inheritance* hierarchy typically belong in the same package.
- Classes related to one another via *aggregation* or *composition* often belong in the same package.
- Classes that *collaborate* with each other a lot often belong in the same package.

Class Package Diagram





class Diagram ➔➔ class Package Diagram

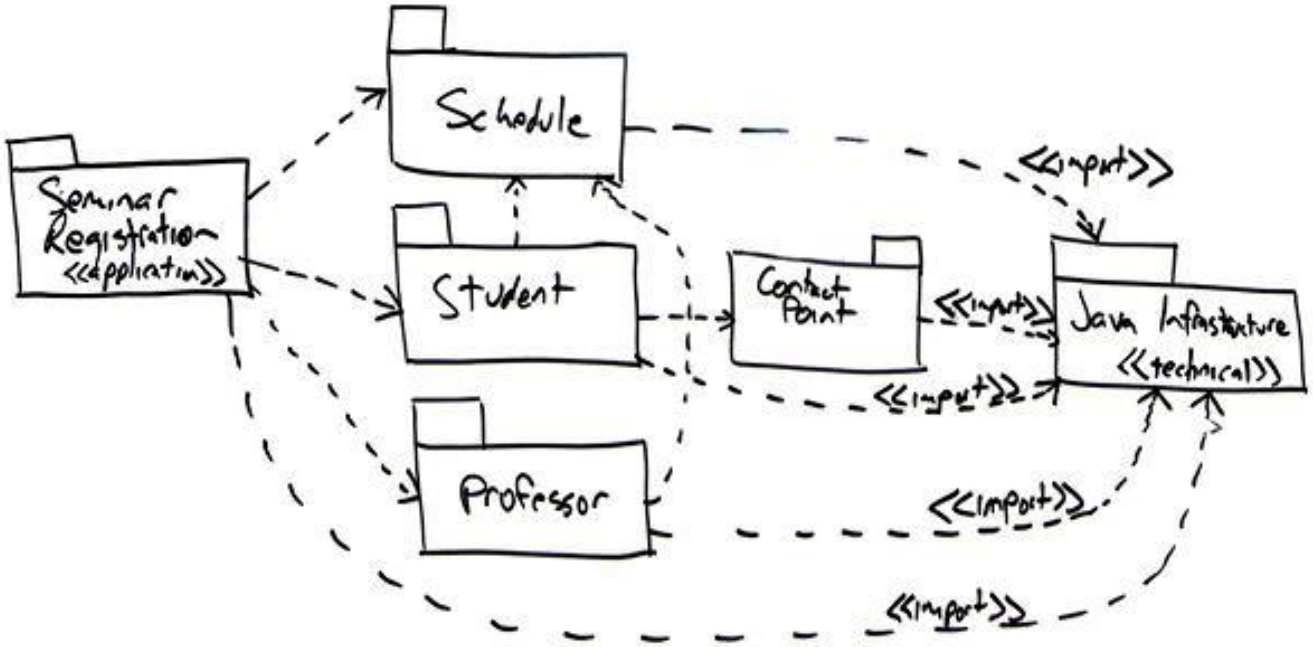
When to Use Package Diagrams

- Use package diagrams for distributing and balancing work between development groups.

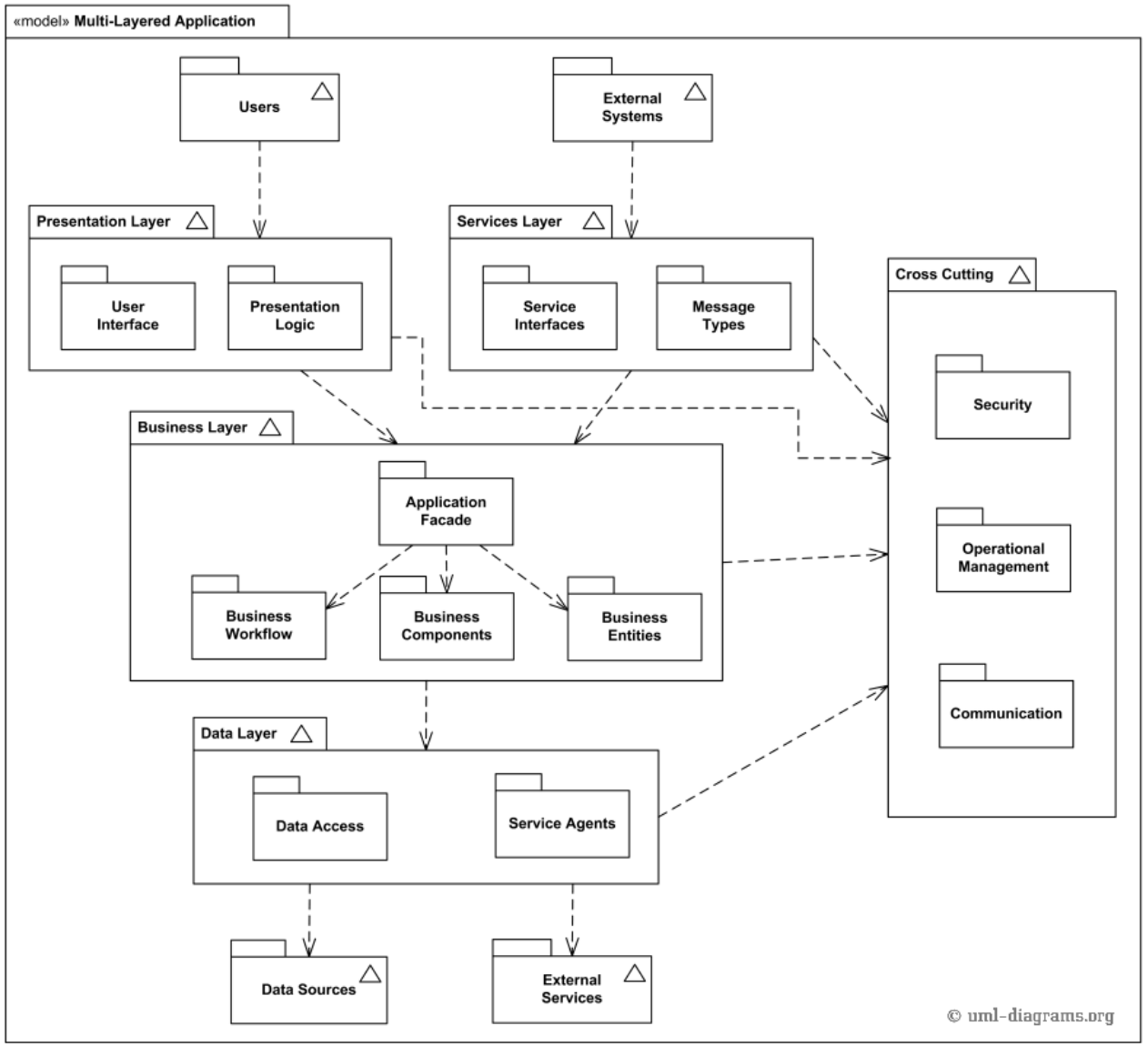
- Package diagrams are helpful to explore the possibilities of partitioning tasks in the development process.

- Package diagrams are extremely useful for testing purposes: rather apply tests on packages (i.e., several interdependent classes) than on single routines.

Exercise #4


Schedule
Seminar Registration
«application»
Student
Professor
Contact Point
Java Infrastructure
«technical»
«import»
«import»
«import»
«import»
«import»

Exercise #5


«model» Multi-Layered Application
Users
External Systems
Presentation Layer
User Interface
Presentation Logic
Services Layer
Service Interfaces
Message Types
Cross Cutting
Security
Operational Management
Communication
Business Layer
Application Facade
Business Workflow
Business Components
Business Entities
Data Layer
Data Access
Service Agents
Data Sources
External Services
© uml-diagrams.org

PART B

COMPONENT DIAGRAM

Drawing Subsystems in UML

System design must model static and dynamic structures:

- Component Diagrams for static structures
  - show the structure at design time or compilation time
- Deployment Diagram for dynamic structures
  - show the structure of the run-time system

Note the lifetime of components

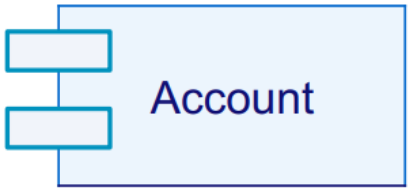
- Some exist only at design time
- Others exist only until compile time
- Some exist at link or runtime

Components

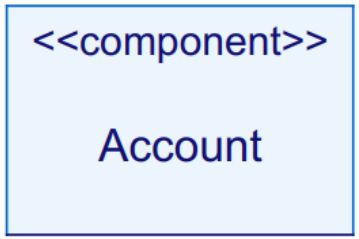
Component is a modular unit with well-defined interfaces that is replaceable within its environment

- fosters reuse
- stresses interfaces

Graphical representation: special kind of class

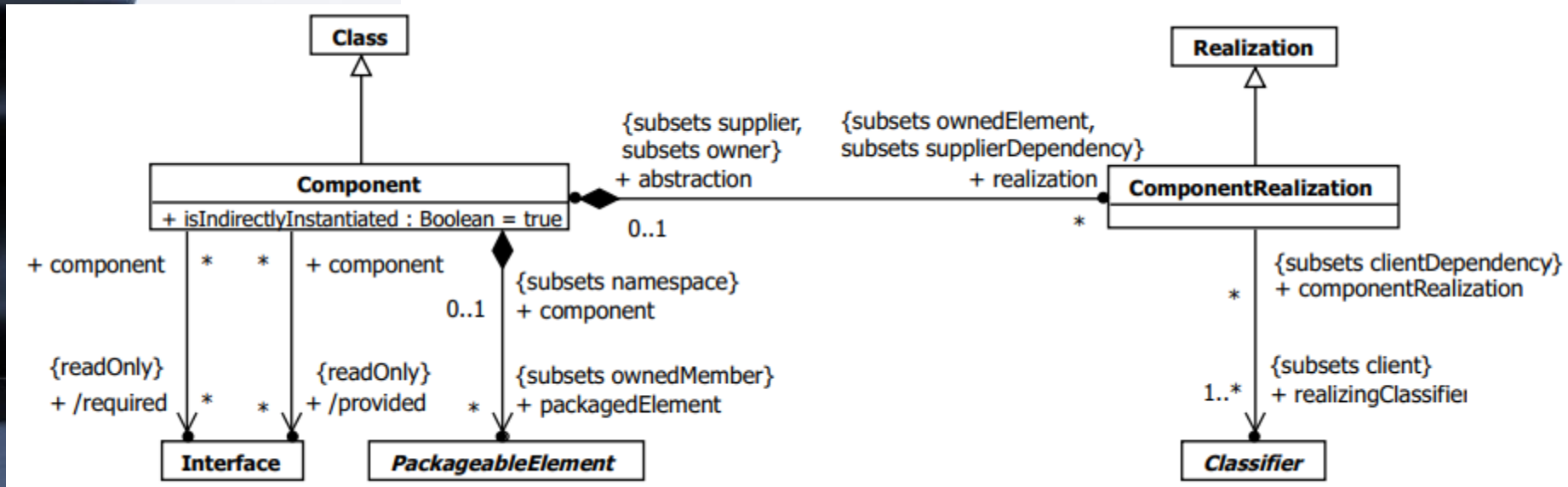


UML 1

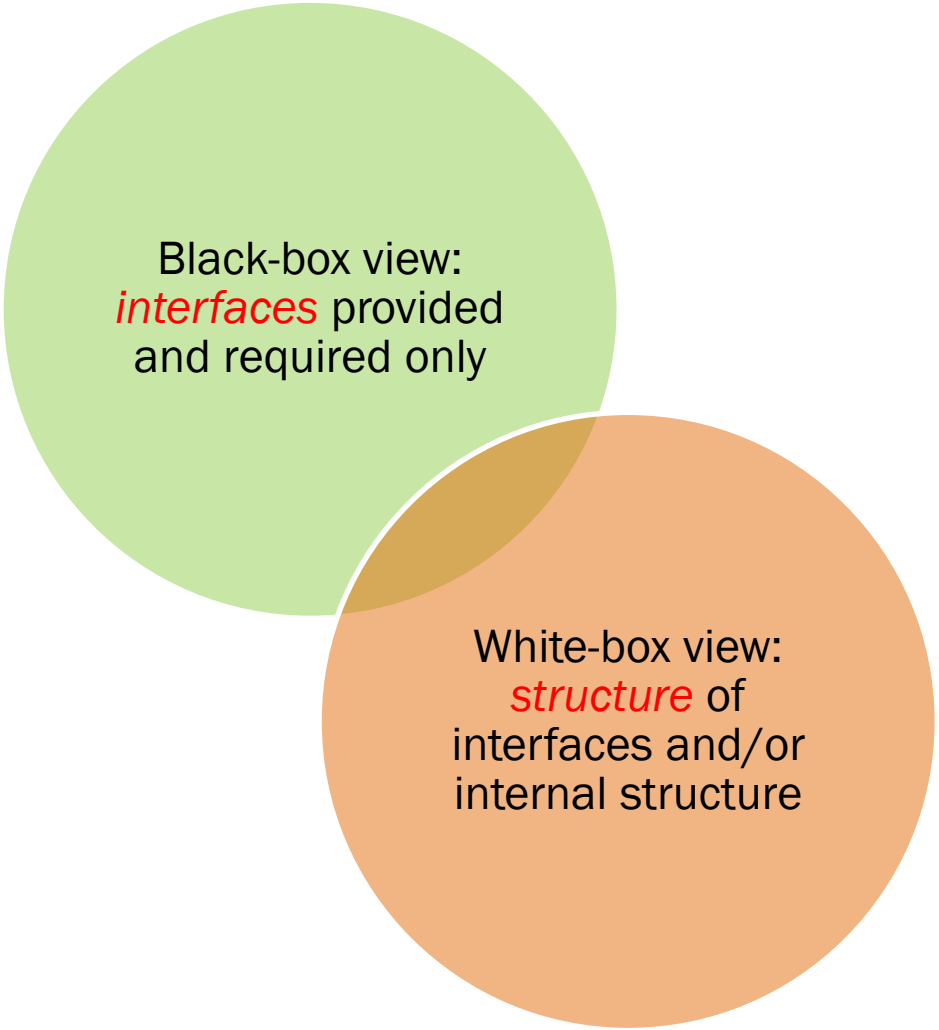


UML 2

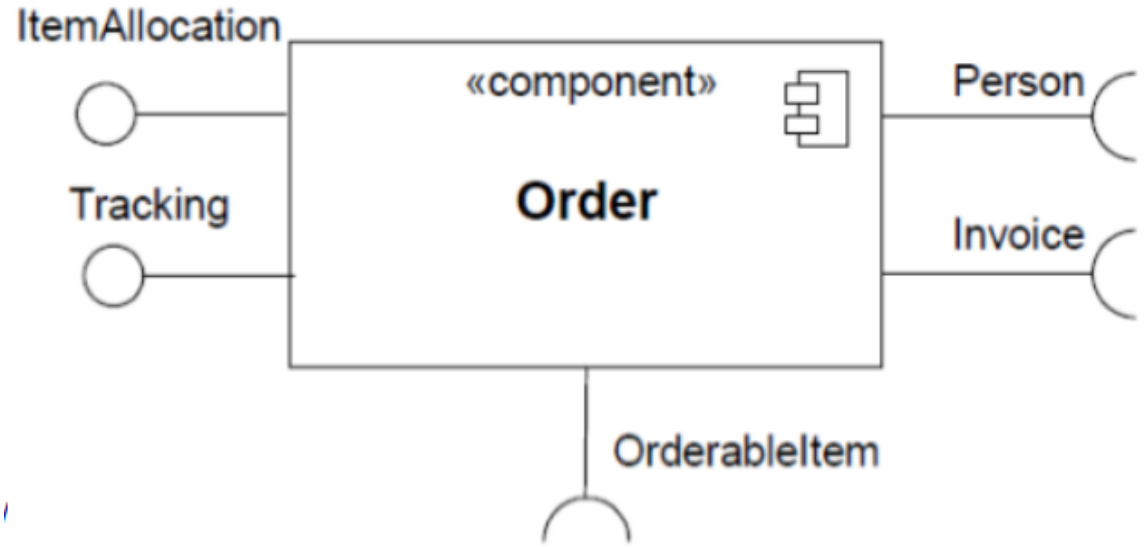
Syntax

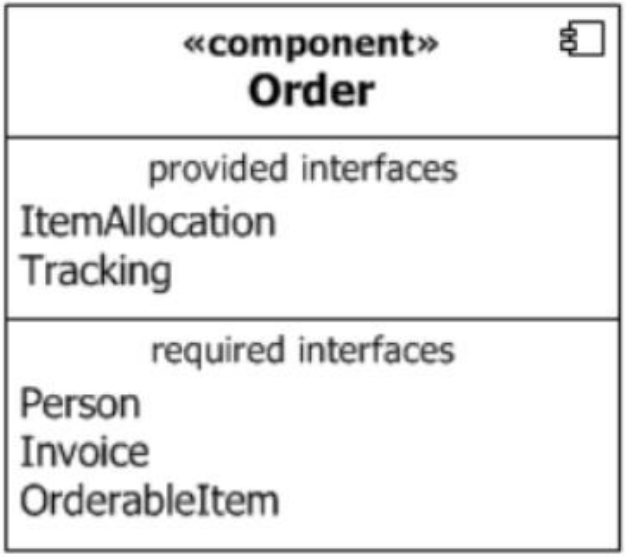

Class
Realization
{subsets supplier,
subsets owner}
+ abstraction
{subsets ownedElement,
subsets supplierDependency}
+ realization
Component
+ isIndirectlyInstantiated : Boolean = true
ComponentRealization
0..1
*
+ component
* *
+ component
{subsets namespace}
+ component
0..1
{subsets clientDependency}
+ componentRealization
*
{readOnly}
+ /required
* *
{readOnly}
+ /provided
* {subsets ownedMember}
+ packagedElement
1..*
{subsets client}
+ realizingClassifier
Interface
PackageableElement
Classifier

Views of Components

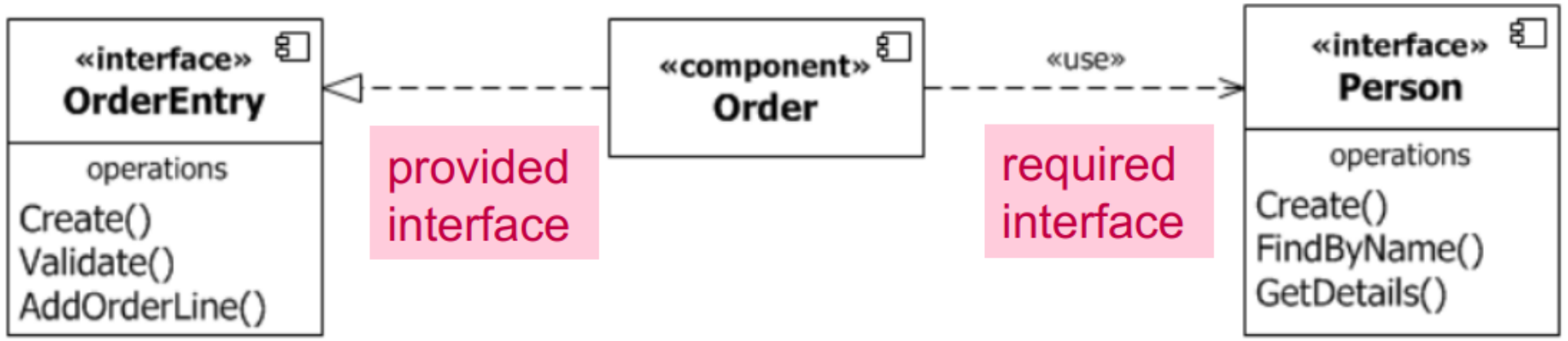


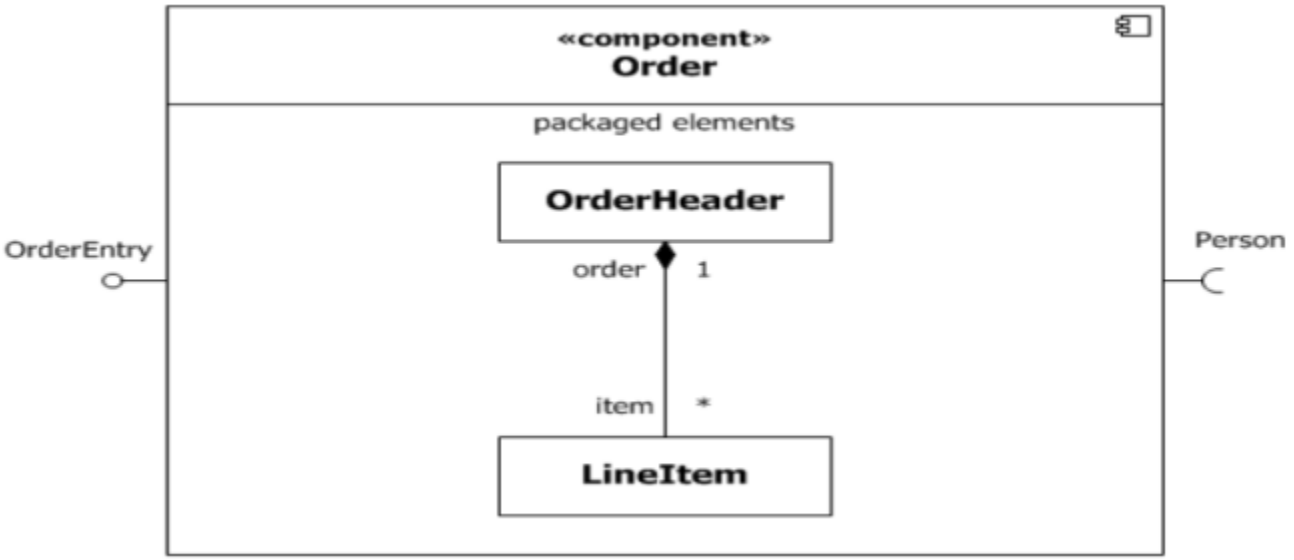
Back-box view



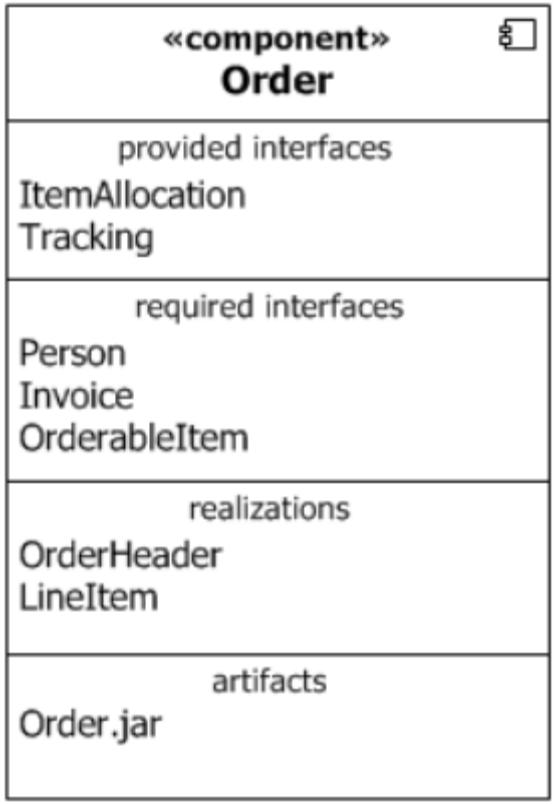


White-box view





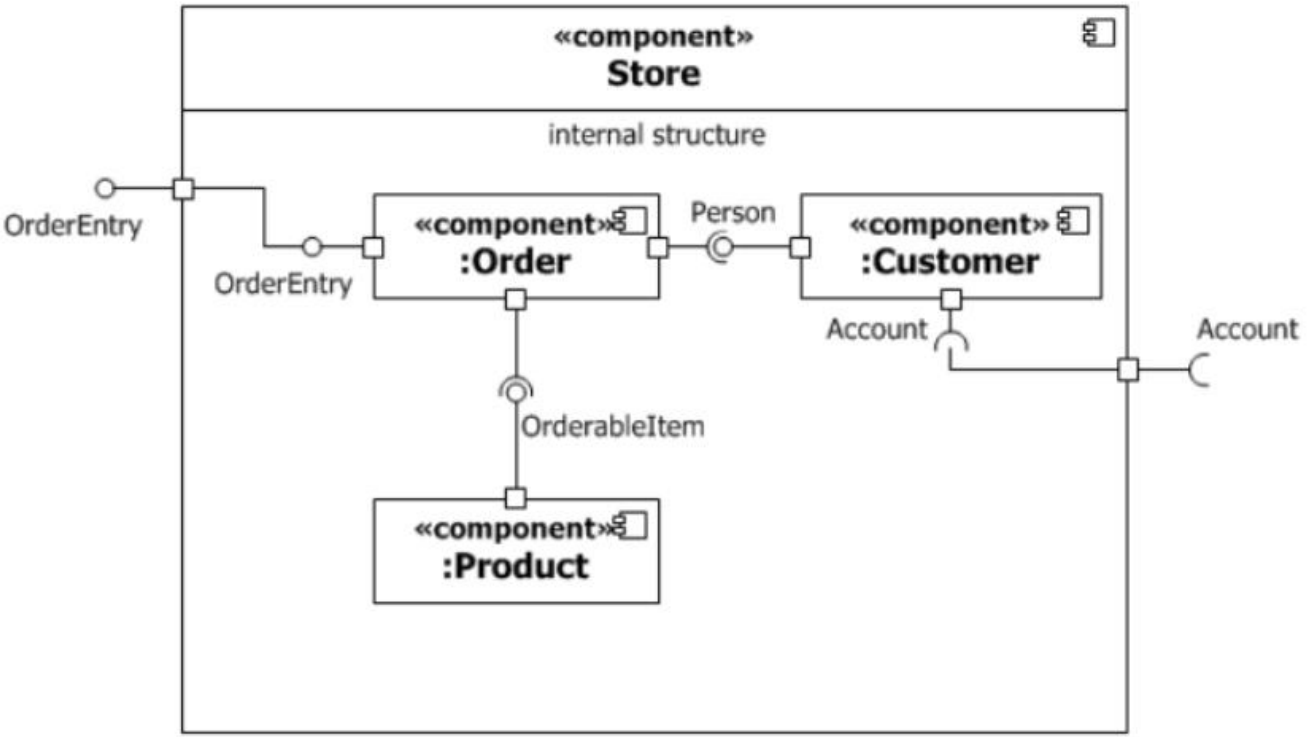
White-box view



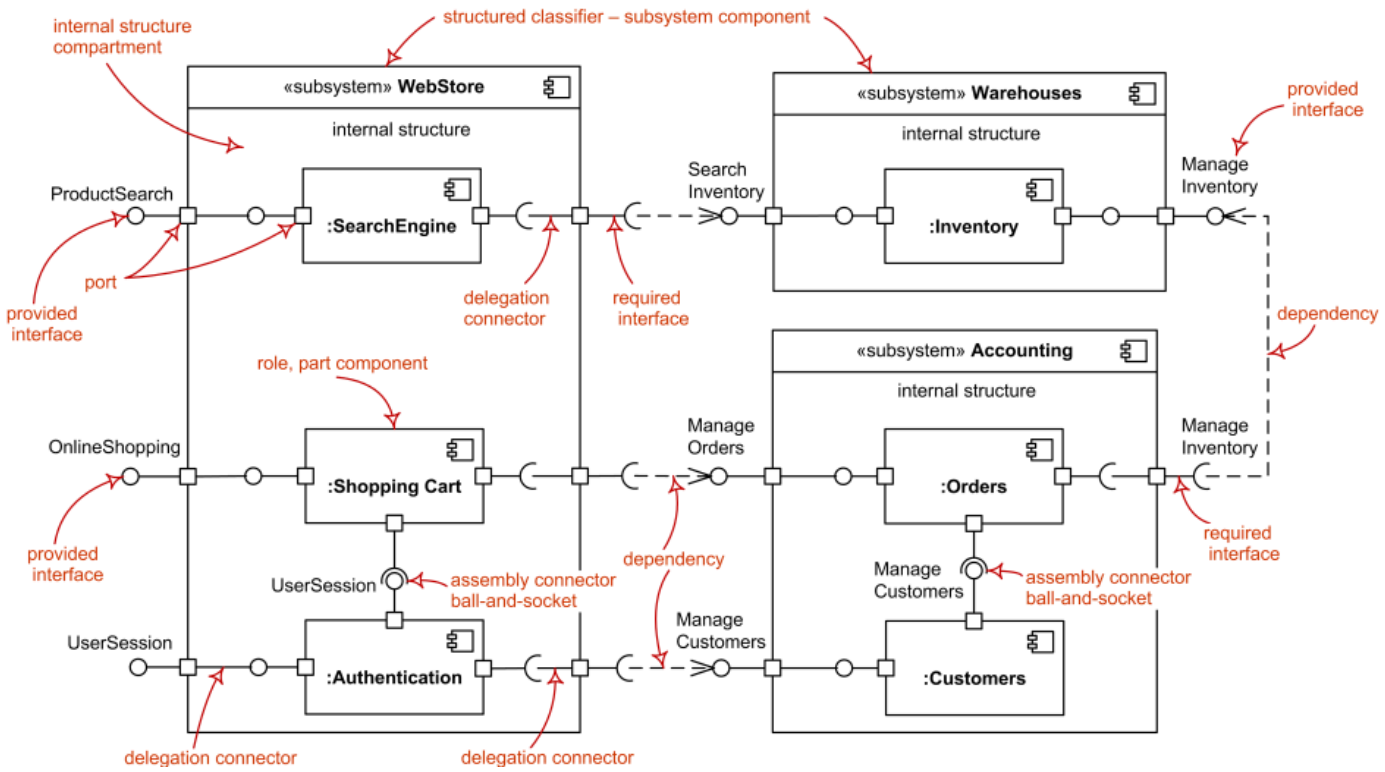
Nested components

Components can be *contained* in other components

Interfaces can then be *delegated* through *ports*

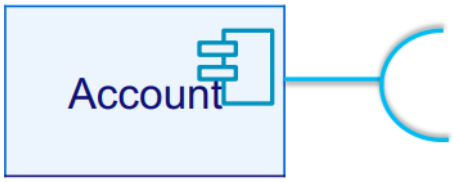


Summary

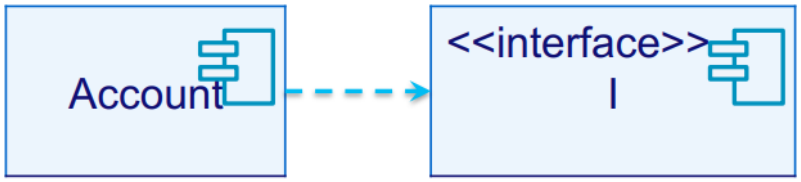

internal structure compartment
structured classifier – subsystem component
«subsystem» WebStore
internal structure
:SearchEngine
ProductSearch
port
provided interface
delegation connector
required interface
Search Inventory
«subsystem» Warehouses
internal structure
:Inventory
Manage Inventory
provided interface
dependency
role, part component
«subsystem» Accounting
internal structure
:Shopping Cart
OnlineShopping
provided interface
Manage Orders
:Orders
Manage Inventory
required interface
UserSession
assembly connector ball-and-socket
dependency
Manage Customers
assembly connector ball-and-socket
UserSession
:Authentication
Manage Customers
:Customers
delegation connector
delegation connector

Exercise #6

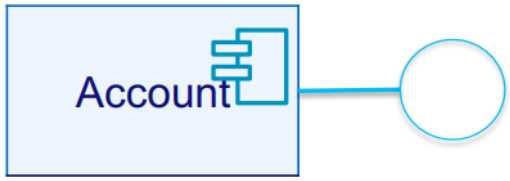
Which notation indicates that I is provided by Account?



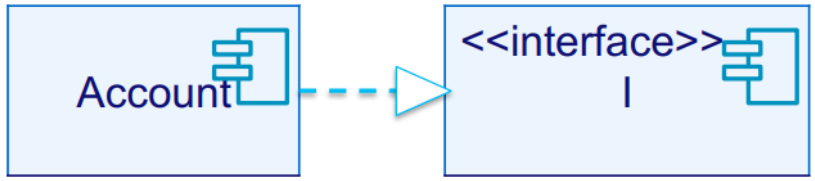
a)



b)



c)



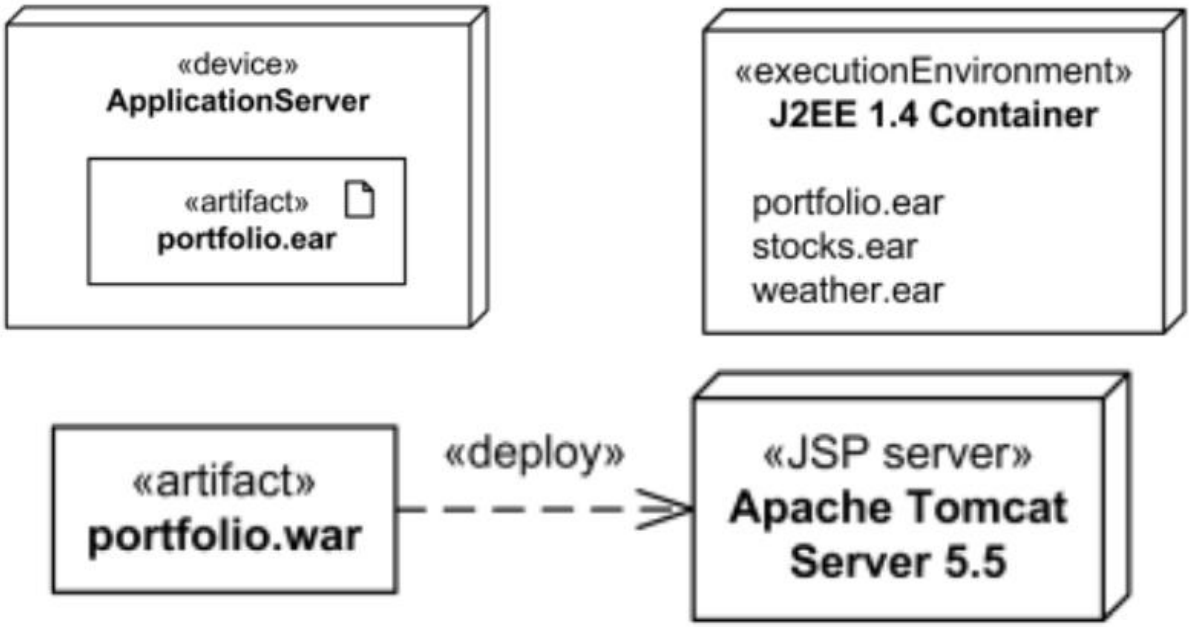
d)

PART C

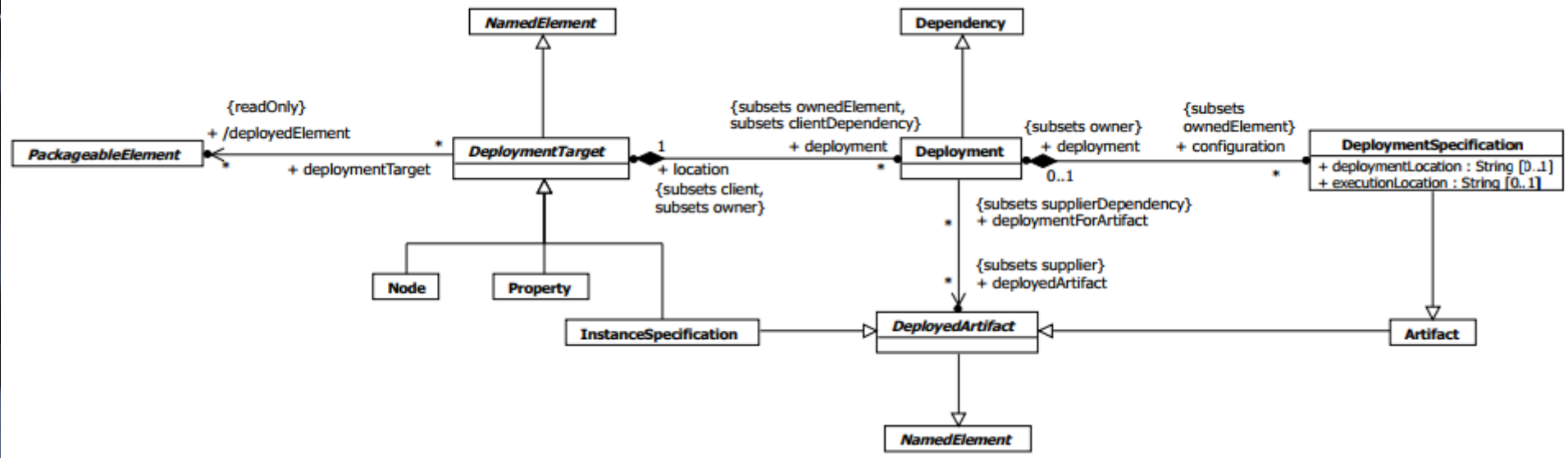
DEPLOYMENT DIAGRAM

Deployment

Deployment is relationship between logical and/or physical elements of systems (*Nodes*) and information technology assets assigned to them (*Artefacts*).



Syntax


NamedElement
Dependency
{readOnly}
+ /deployedElement
PackageableElement
*
*
+ deploymentTarget
DeploymentTarget
1
+ location
{subsets client,
subsets owner}
{subsets ownedElement,
subsets clientDependency}
+ deployment
*
Deployment
{subsets owner}
+ deployment
0..1
{subsets
ownedElement}
+ configuration
*
DeploymentSpecification
+ deploymentLocation : String [0..1]
+ executionLocation : String [0..1]
{subsets supplierDependency}
+ deploymentForArtifact
*
{subsets supplier}
+ deployedArtifact
*
Node
Property
InstanceSpecification
DeployedArtifact
Artifact
NamedElement

Nodes

Nodes

- **devices**: application server, client workstation, ...
- **execution environments**: DB system, J2EE container
- Graphical representation: **box**

Nodes can be **physically connected** (e.g., via cables or wireless)

- UML-parlance: CommunicationPath
- Graphical representation: as an association

Artefacts

Artefacts are information items produced during software development or when operating the system

- model files, source files, scripts, executable files, database tables, word-processing documents, mail messages, ...

- Graphical representation: “class-like”

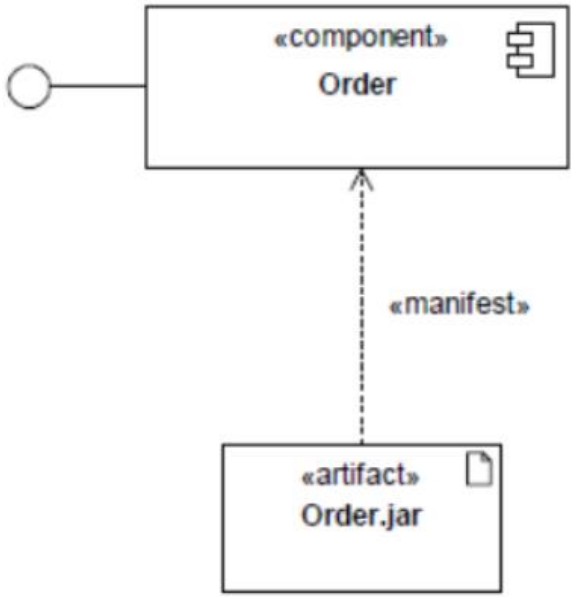
- Relations: dependencies

Manifestation

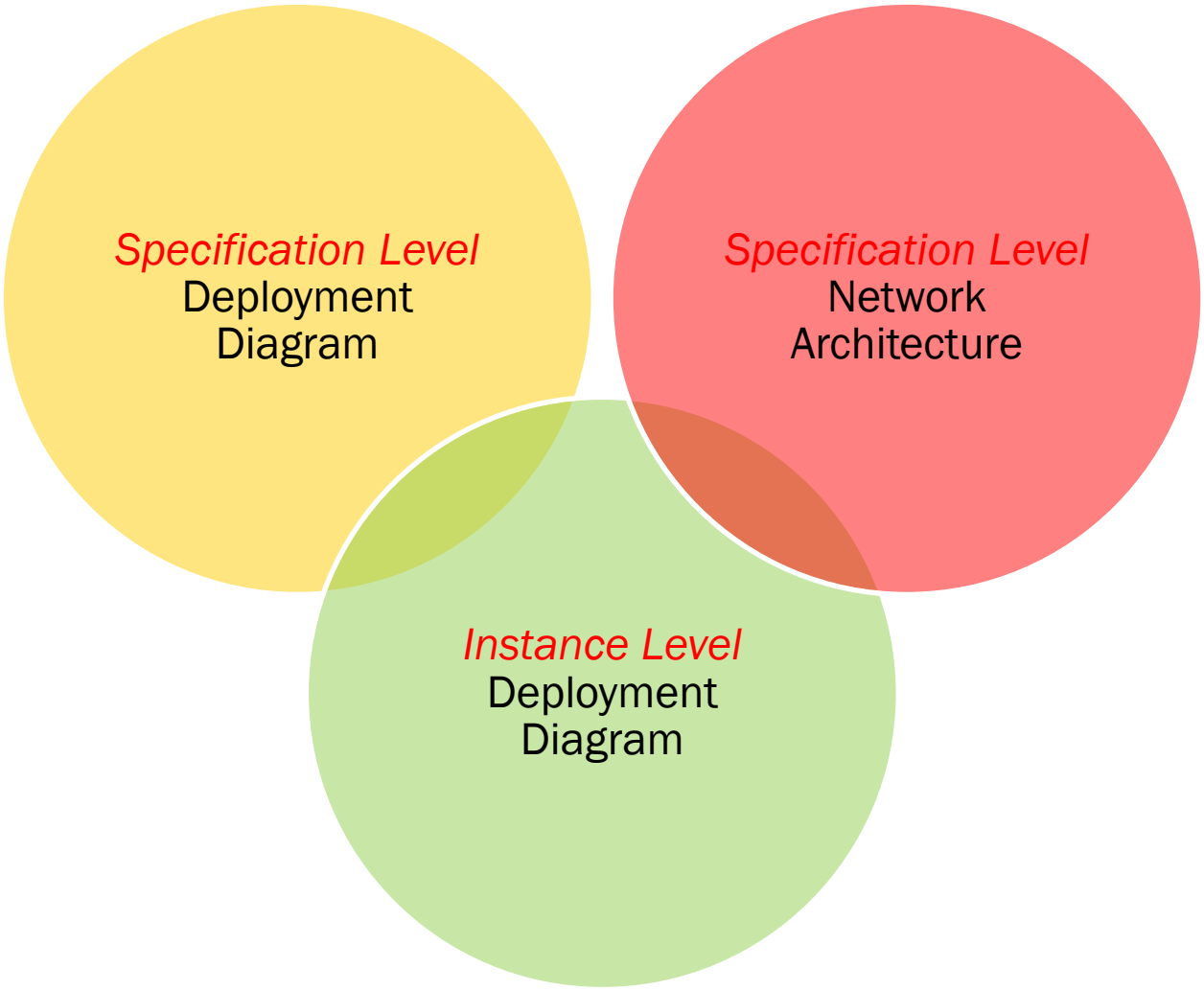
How do we know where a given use case, class, component, or package is deployed?

- Use case / class / component / packages diagrams do not discuss deployment
- Deployment diagrams do not discuss use cases / classes / components / packages but only artifacts

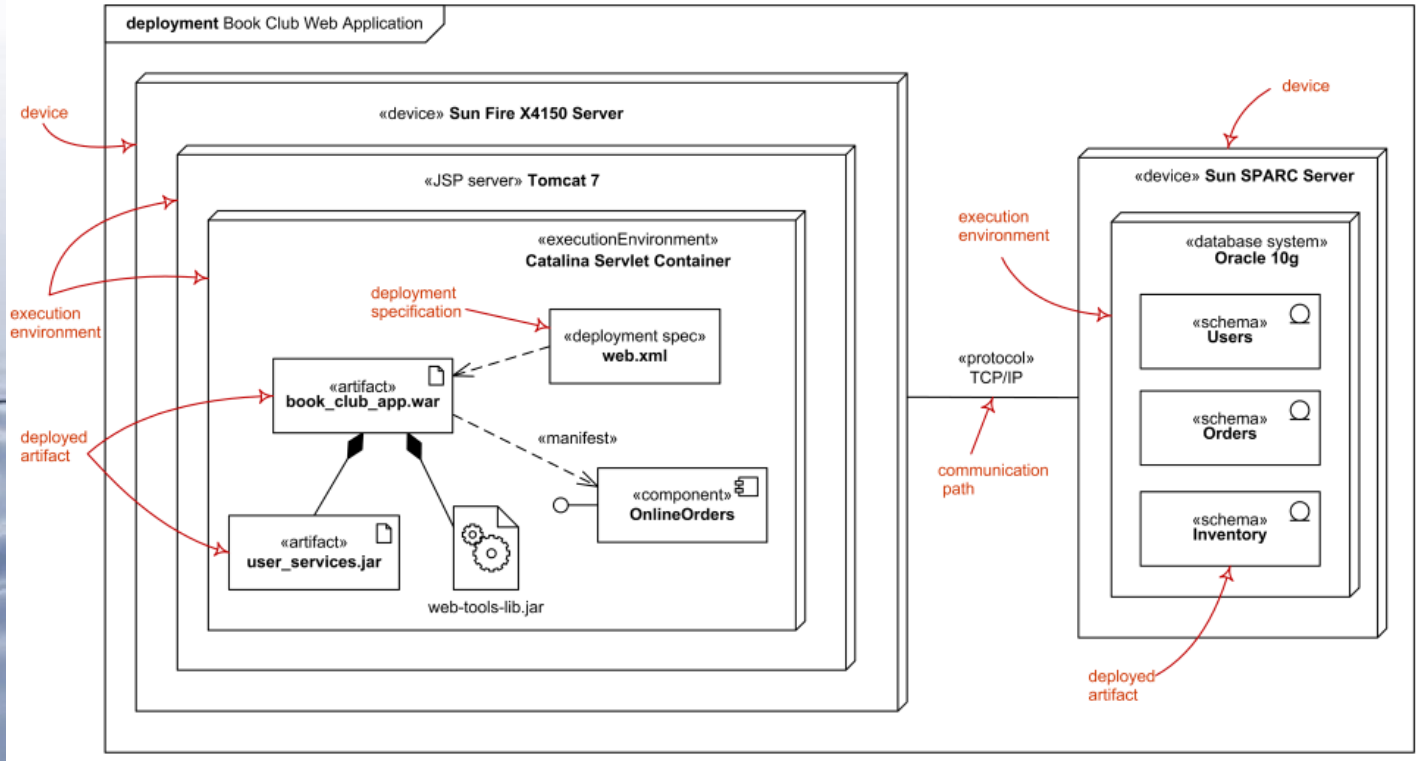
Manifestation maps artifacts to use cases / classes / components / packages



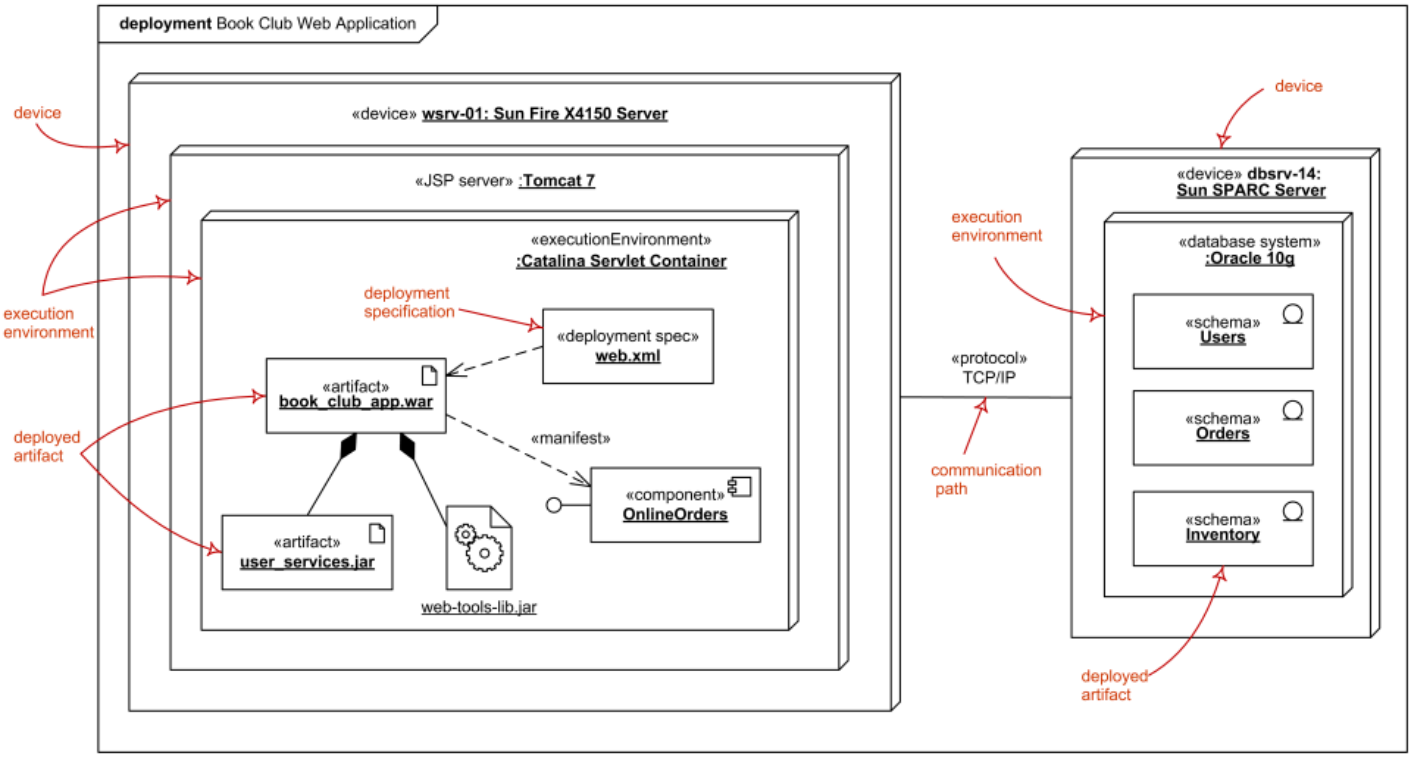
Levels of Deployment Diagrams



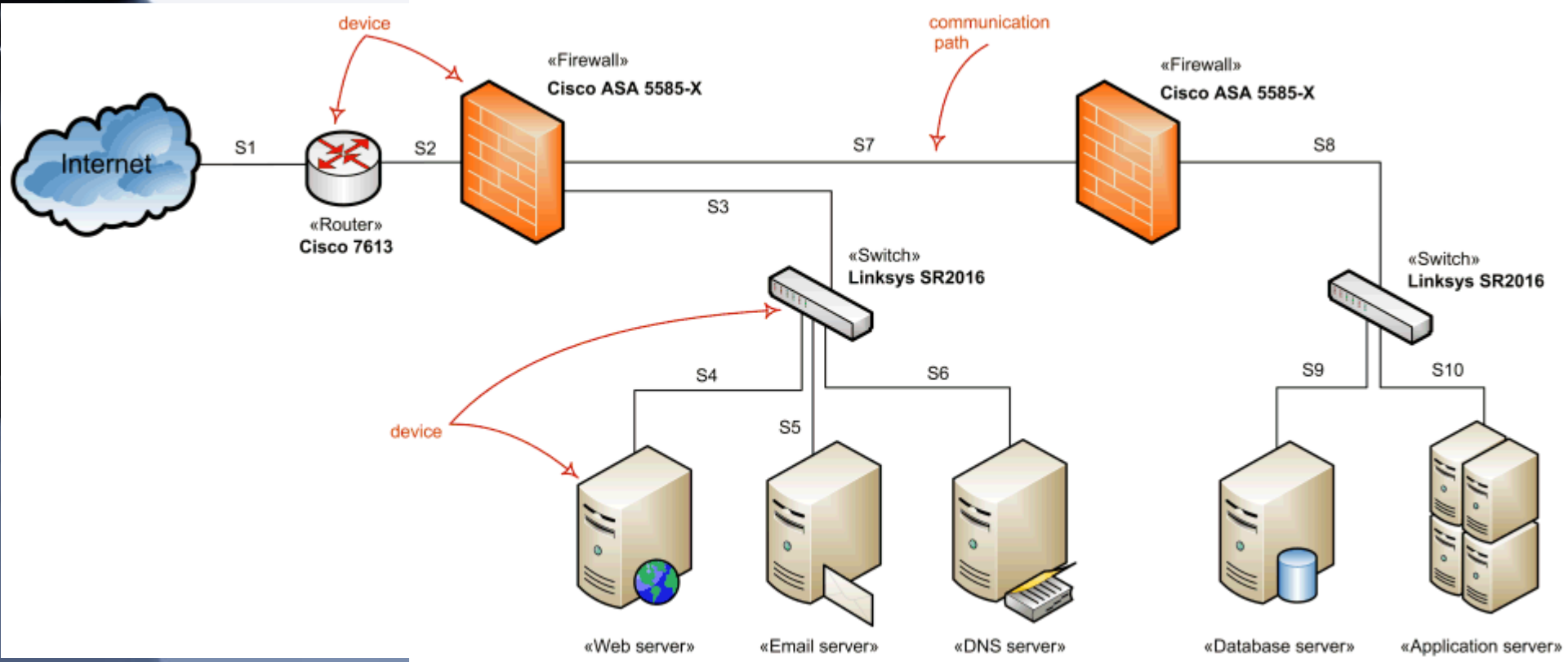
Specification Level Deployment Diagram



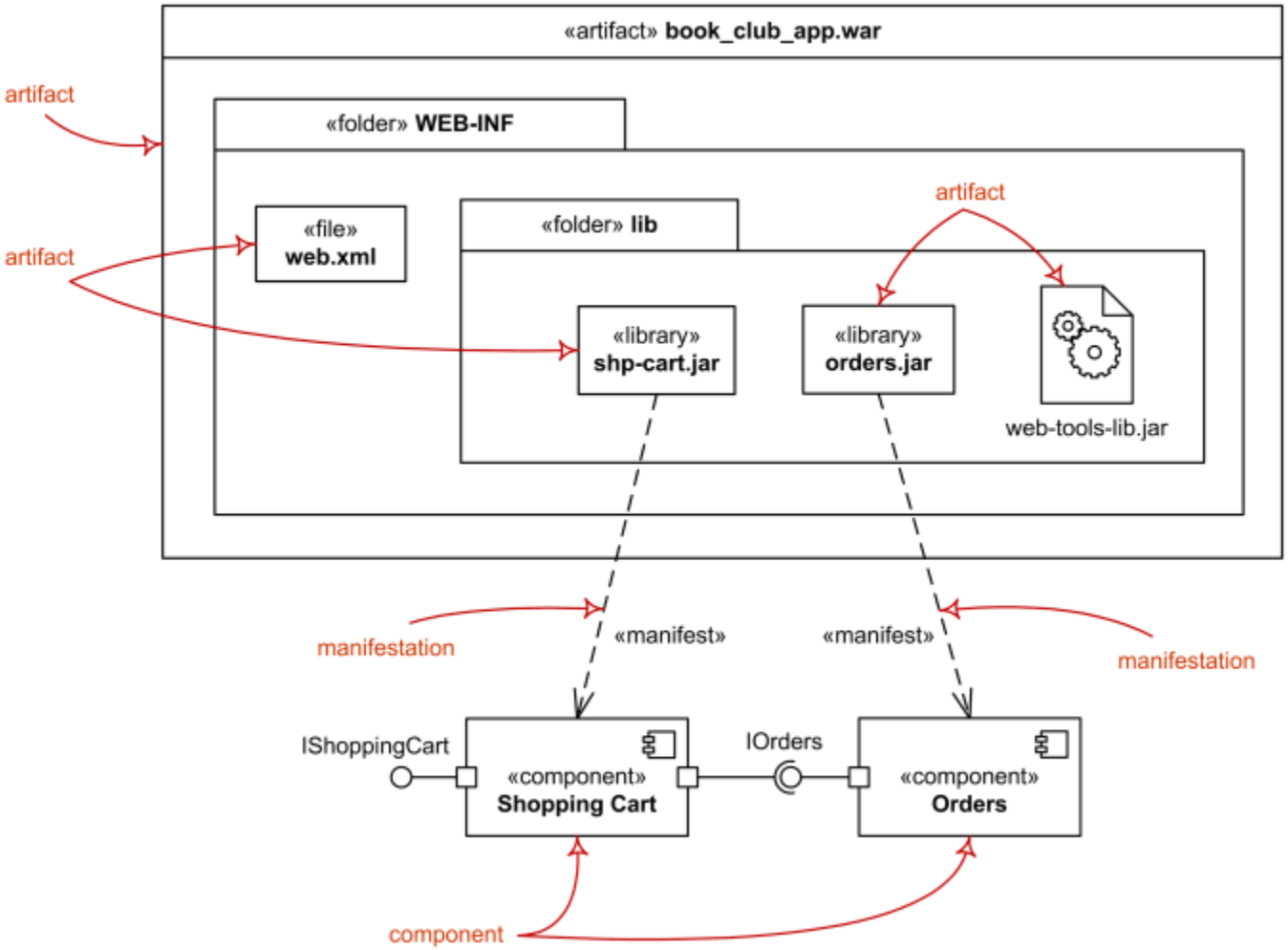
Instance Level Deployment Diagram



Specification Level Network Architecture



Summary


«artifact» book_club_app.war
artifact
«folder» WEB-INF
artifact
«file»
web.xml
«folder» lib
artifact
«library»
shp-cart.jar
«library»
orders.jar
web-tools-lib.jar
manifestation
«manifest»
«manifest»
manifestation
IShoppingCart
«component»
Shopping Cart
IOrders
«component»
Orders
component

Exercise #7

Identify correct statements pertaining to deployment diagrams:

a) Artefacts are physical elements of the system such as devices and execution environments.

b) Artefacts: information items produced during software development or when operating the system.

c) Manifestation maps artefacts to components, use cases, classes, components, packages.

d) Manifestation maps components, use cases, classes, components to artefacts.

Any Questions?

✉ hvusynh@hcmiu.edu.vn