OOAD: Structural Modeling

Presenter: Dr. Ha Viet Uyen Synh.

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.

Package Diagrams

- Package diagrams and package dependencies result from class diagrams.

- Class A depends on class B, A → B, if changes to the definition of class B may cause an effect on class A.

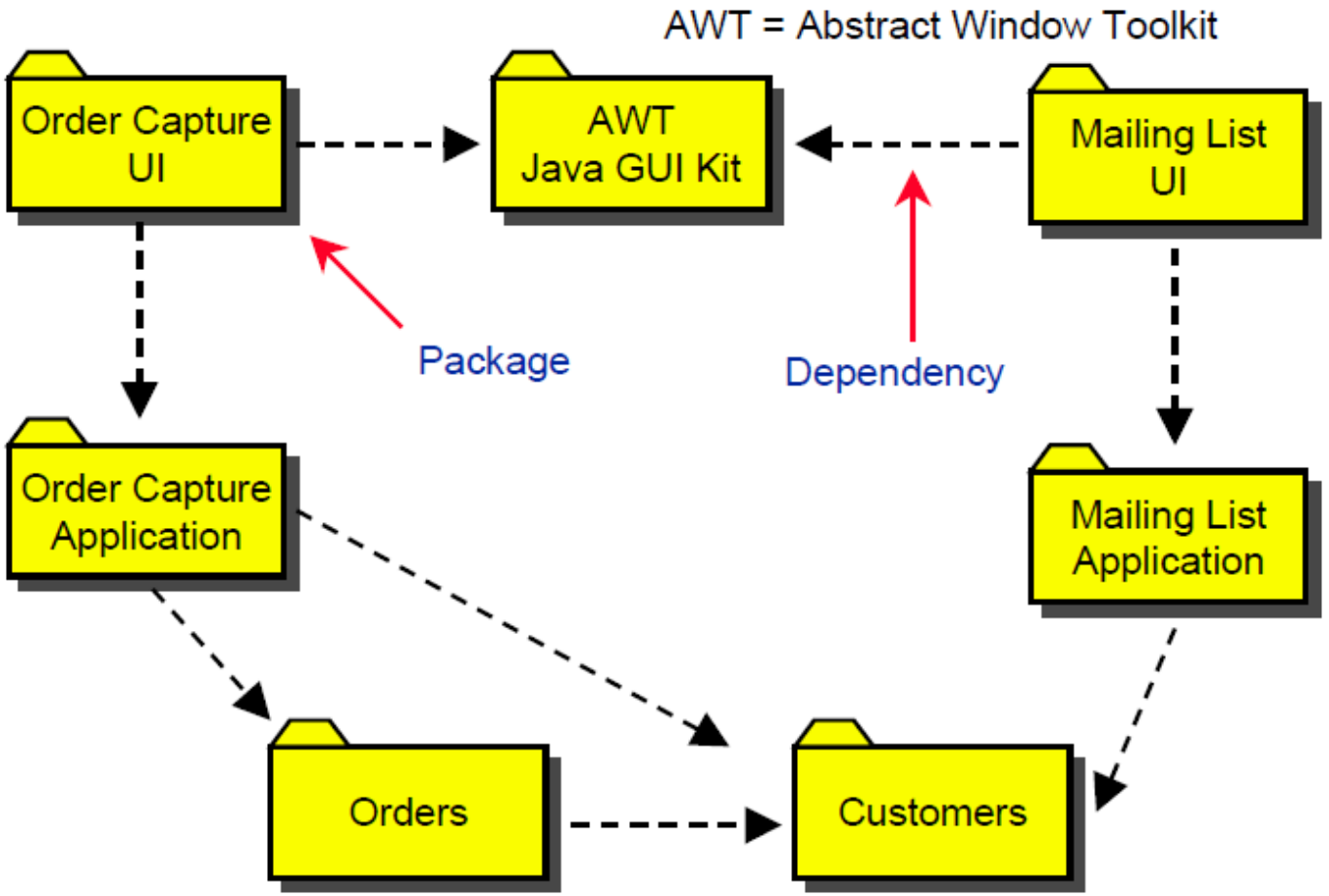
Examples:

- class A sends a message to class B;
- class A has class B as part of its data;
- class A mentions class B as parameter to an operation.

- In an ideal OO-world, modifications which don't change a class interface should not affect any other classes.

The art of large scale design involves minimizing dependencies

Example: Package Diagram

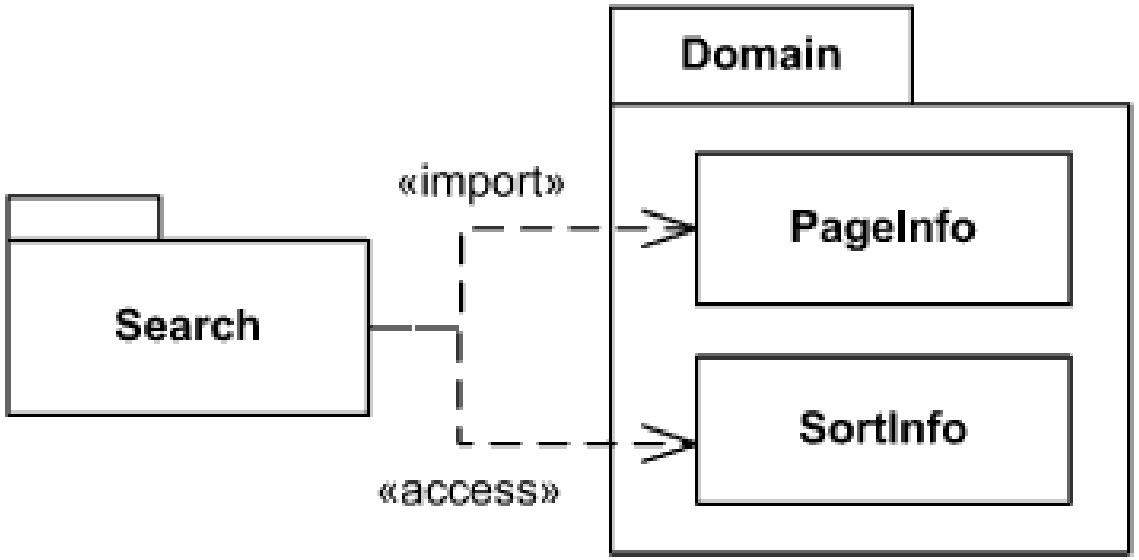


Element Import:

The keyword **«import»** is shown near the dashed arrow if the visibility is **public**

The keyword **«access»** is shown to indicate **private** visibility

Public import of PageInfo element and private import of SortInfo element from Domain package.



Package Import:

Package Import (PackageImport) is a directed relationship between an importing **namespace** and imported **package**

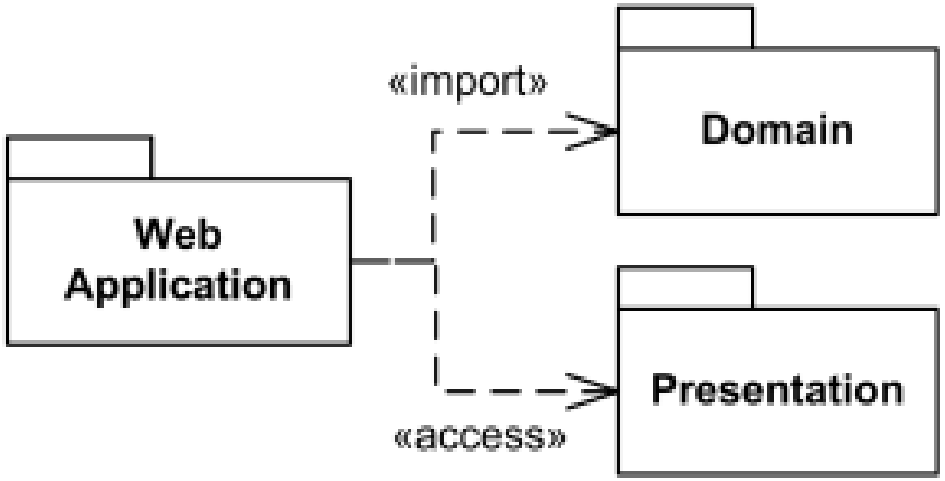
A package import is shown using a dashed arrow with an open arrowhead from the importing namespace to the imported package.

It looks exactly like dependency and usage relationships.

The **visibility** of a PackageImport could be either public or private.

Package Import :

Private import of Presentation package and **public import** of Domain package.



Package Merge:

A package merge is a directed relationship between two packages.

It indicates that content of one package is extended by the contents of another package.

Package merge used when elements defined in different packages have the same name and are intended to represent the same concept.

Package merge is shown using a dashed line with an open arrowhead pointing from the receiving package to the merged package.

Package Merge :

Rules for Merging Packages:

Private elements within the package do not merge with the receiving package.

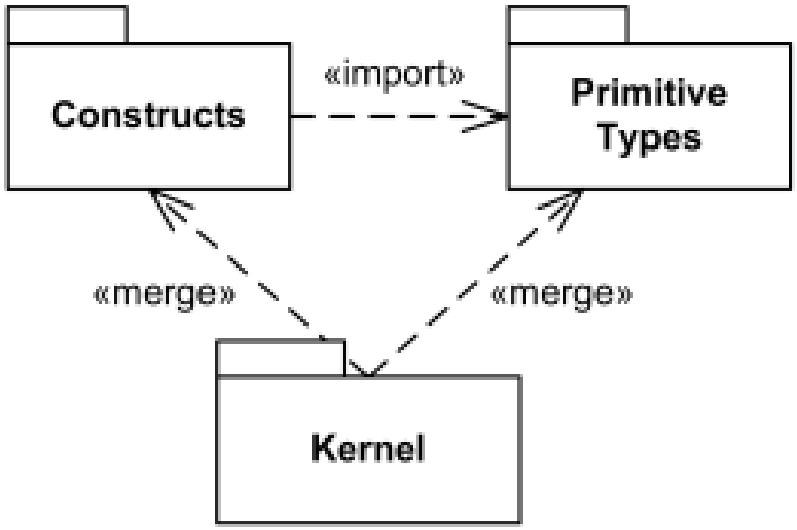
UML allows multiple inheritance in package merge.

Any sub packages within the package are added to the receiving package.

If both packages have different packages of the same name, a merge takes place between those packages.

Package Merge :

UML packages Constructs and Primitive Types are **merged** by UML Kernel package.



Relationships:

Dependency

Generalization

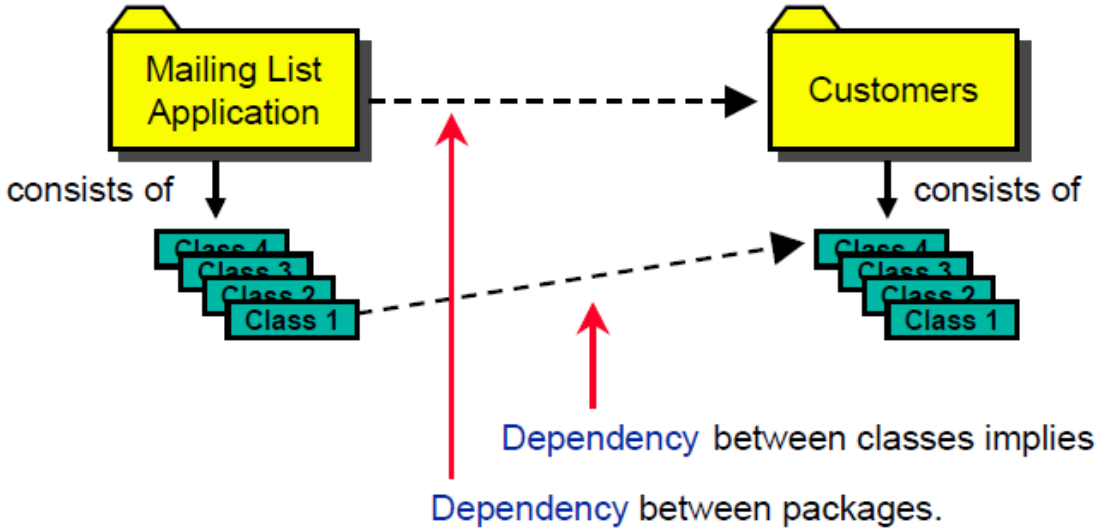
Refinement

Dependencies

A relationship between two packages is called a package dependency.

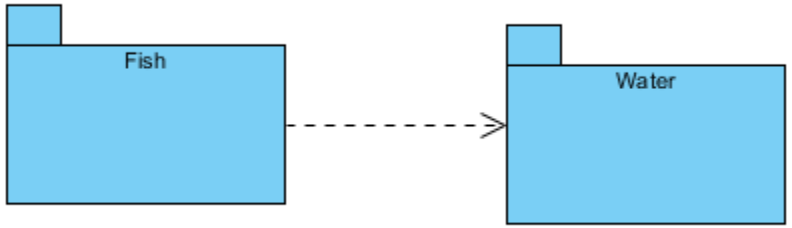
The dependency relationship between packages is consistent with the relationship between classes.

Ex. If changing the contents of a package, P2, affects the contents of another package, P1, we can say that P1 has a Package Dependency on P2.



Dependencies

One Package depends on another package.

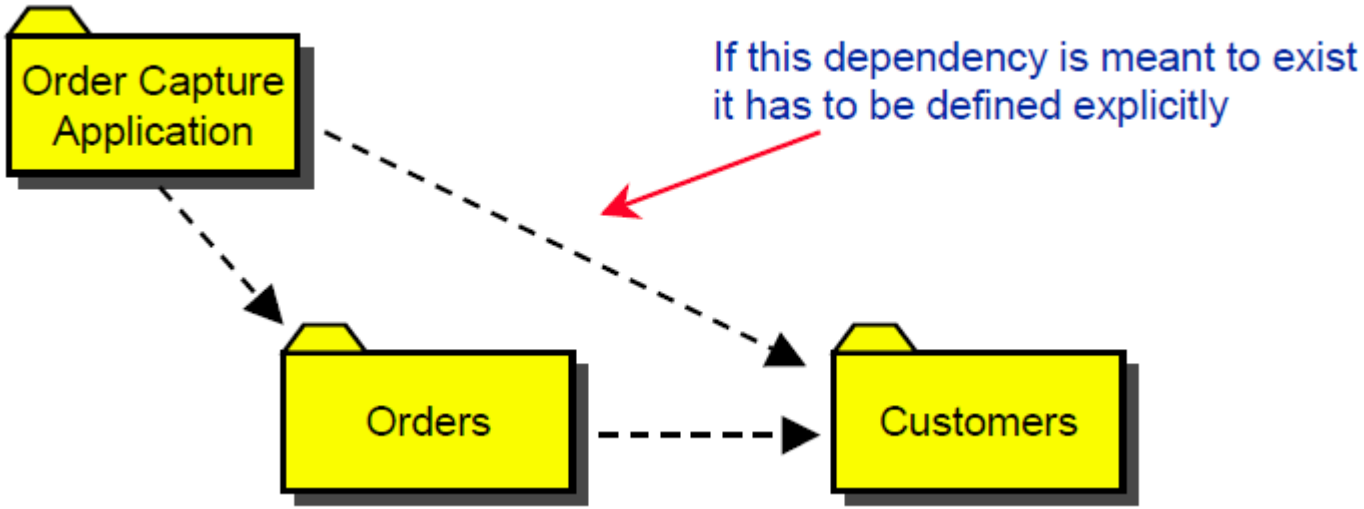


Fish depends on water.

Dependencies

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

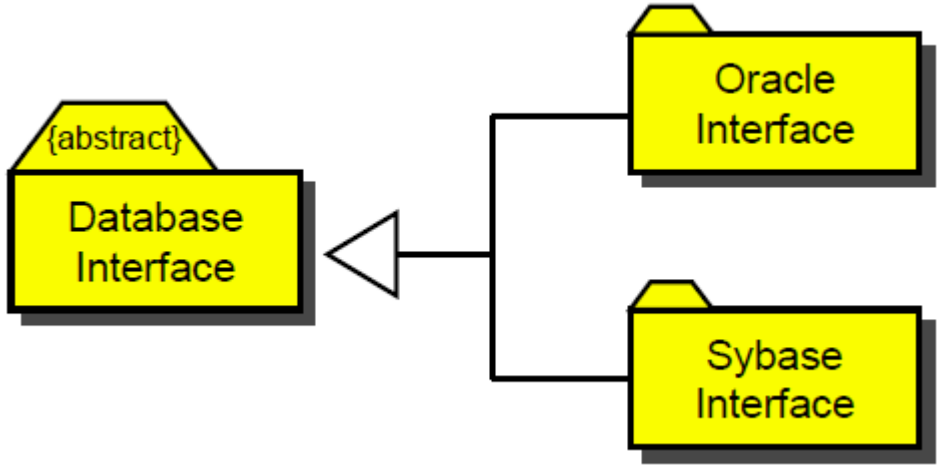
Dependencies between packages are not transitive



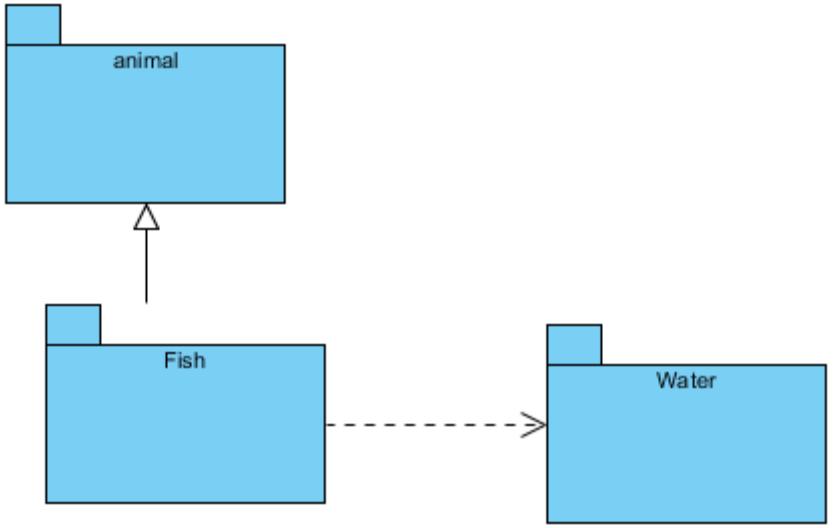
Generalizations

- Generalization between packages means that the specific package must conform to the interface of the general package.
- To emphasize the role of a general interface, the package can be marked as {abstract}.

Example: An abstract database interface consisting of several classes is implemented either for Oracle or for Sybase.



Generalizations

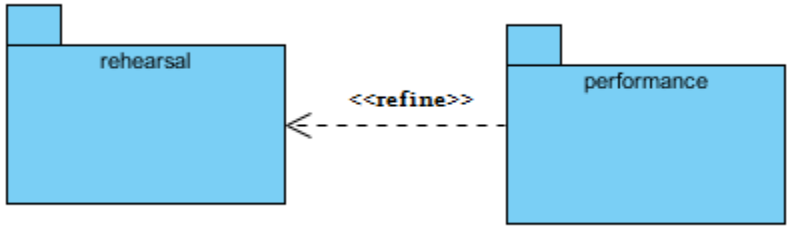


Fish is a kind of Animal.

Refinements

Refinement shows different kind of relationship between packages.

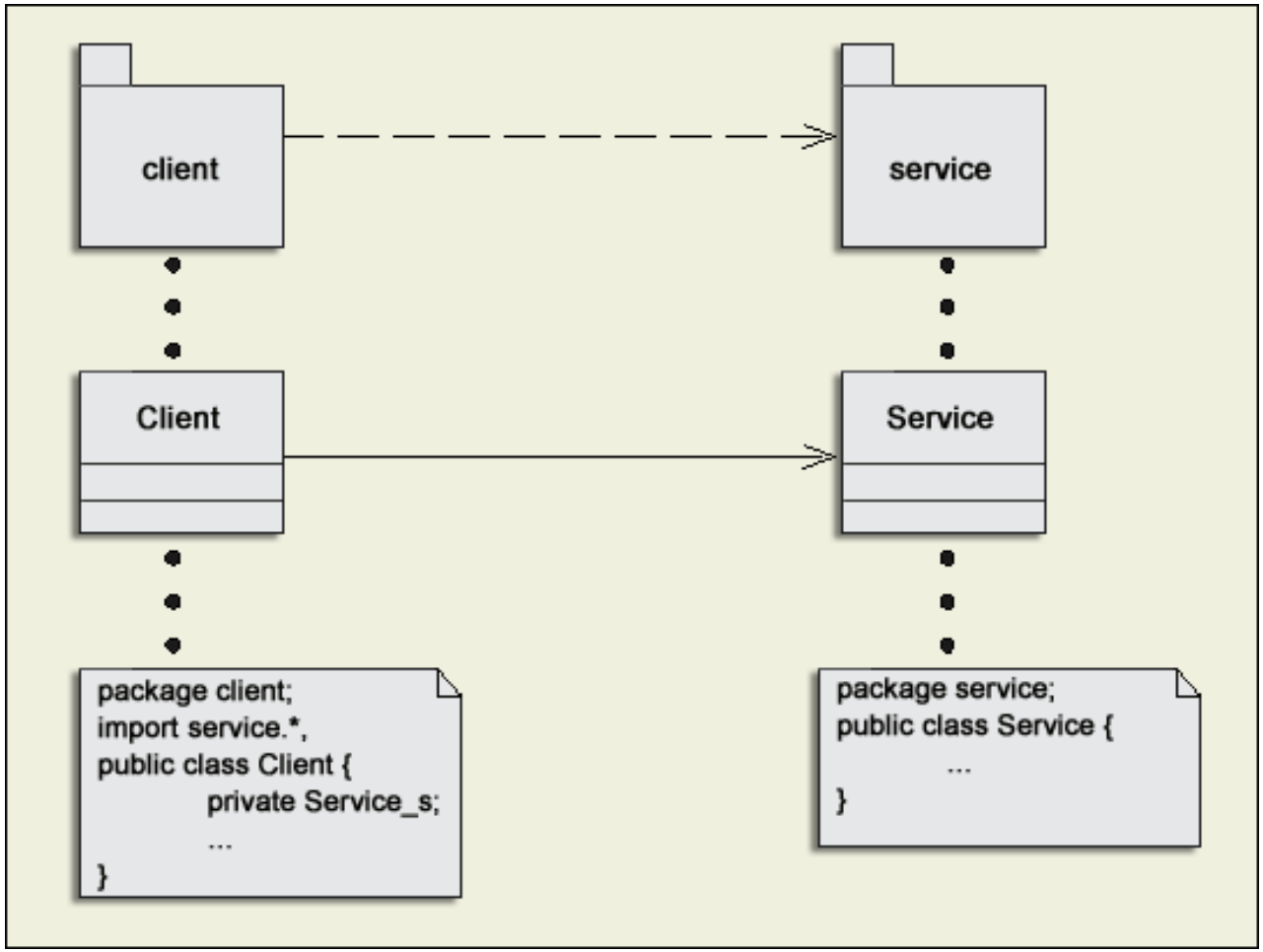
One Package refines another package, if it contains same elements but offers more details about those elements.



Performance refines rehersals.

Package Relationship

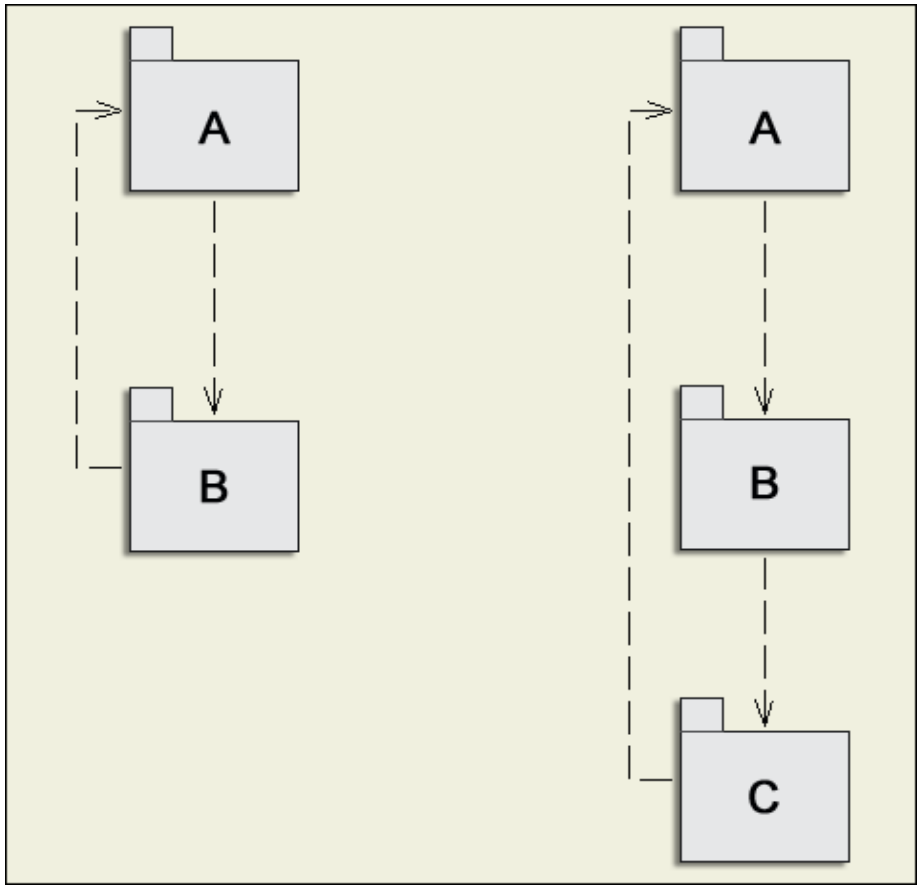
Package dependency diagram :



Package Relationship

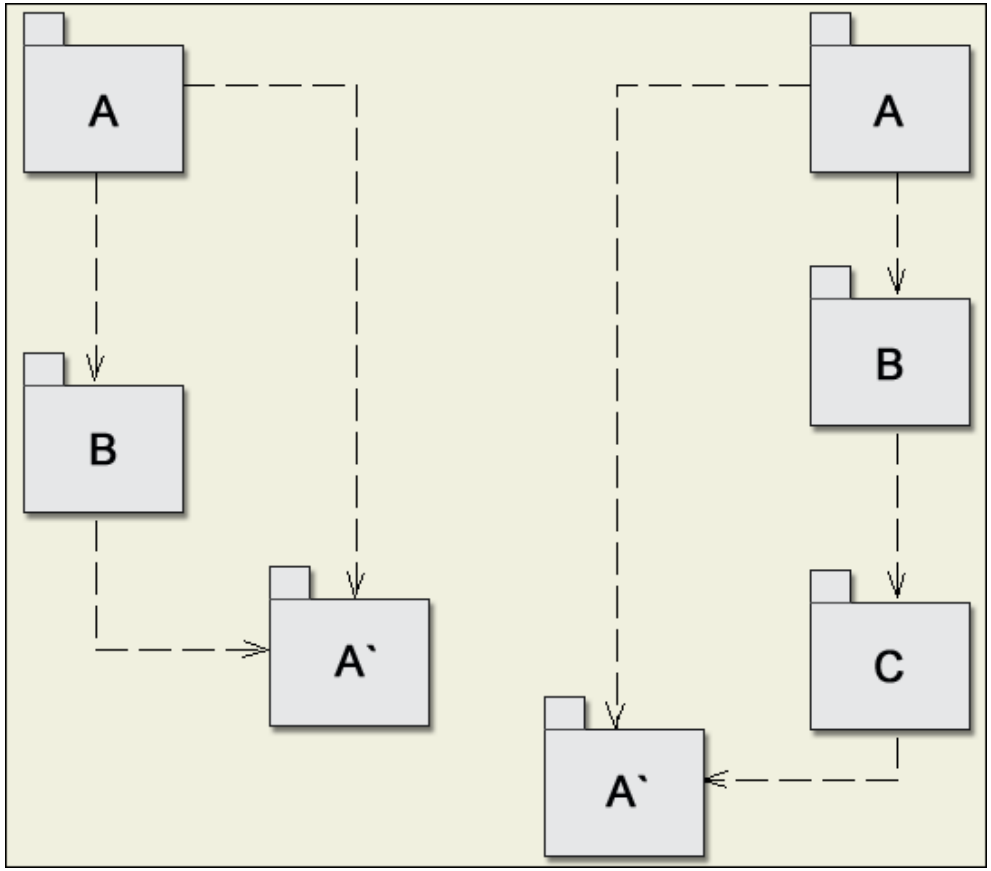
Two types of relationship: Unidirectional and Bidirectional

Unidirectional Diagram :



Package Relationship

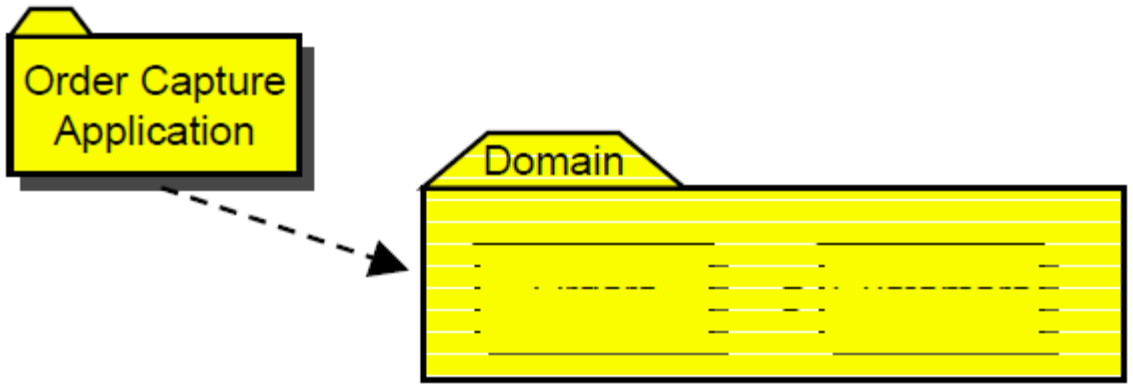
Bidirectional Diagram :



Nested Packages

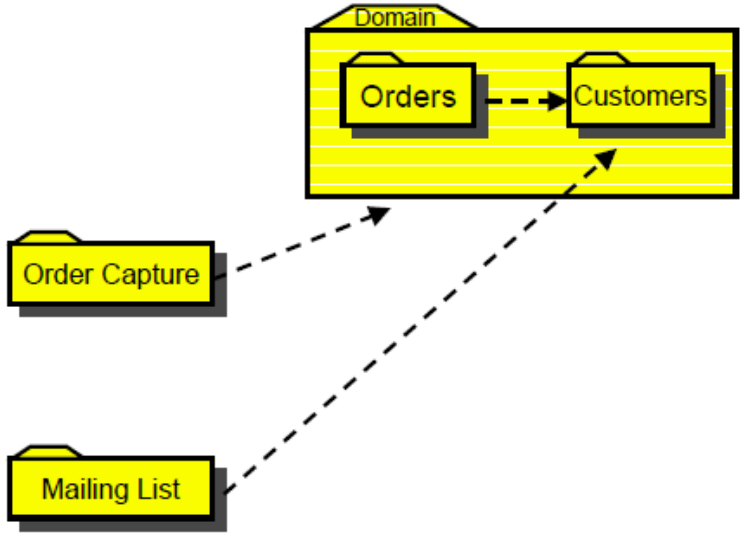
- Instead of drawing many separate dependencies, the technique of sub-packages reduces redundant dependency information.

- Draw dependencies to and from the overall package, instead of many separate dependencies.



Nested Packages

- An overall package can contain:
  - classes
  - class diagrams
  - package diagrams

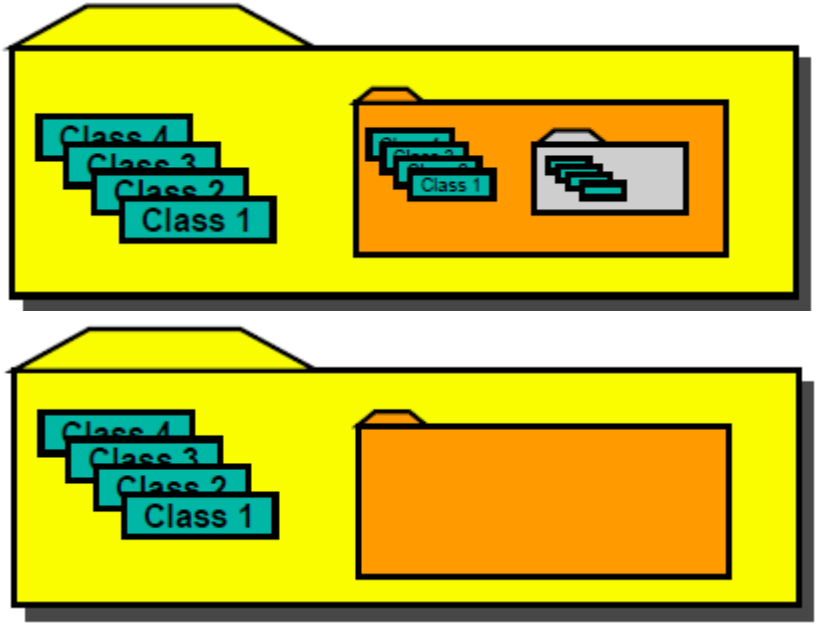


- Dependencies on an overall package represent dependencies on all members of the package.

- Separate dependencies on single members of the overall package can still occur

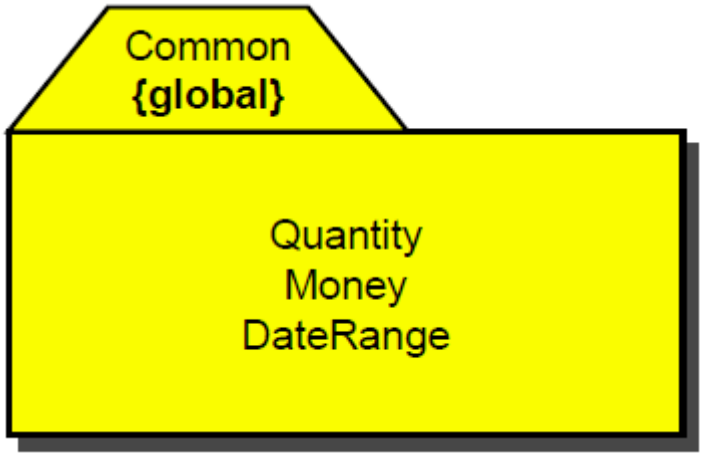
Dependencies: Visibility

- What does it mean to draw a dependency to a package that contains subpackages?
- Convention “transparent”: gives visibility to the classes in the package and in the subpackage.
- Convention “opaque”:gives visibility to the top-level classes only, not to the nested classes.
- Make clear, which convention you use in your project (by use of <<transparent>> or <<opaque>> stereotypes).



Global Dependency

The {global} flag in a package tab signals that all packages in the system are dependent on this package.



Use this notational option sparingly! A change in this package effects all packages!

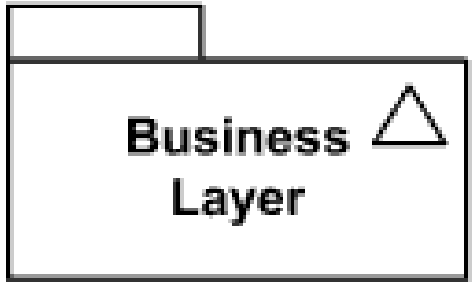
Model

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.

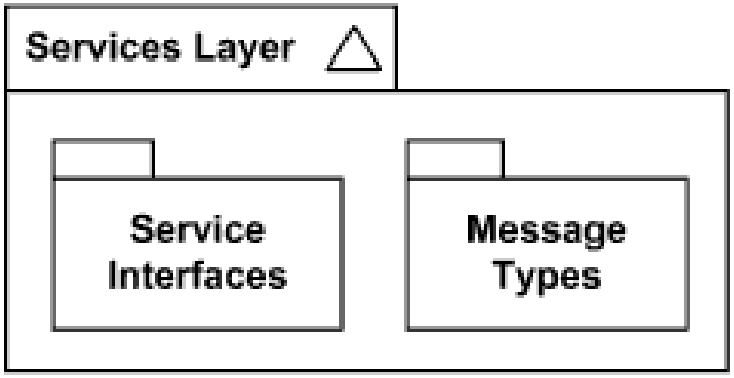
Business layer model :



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.

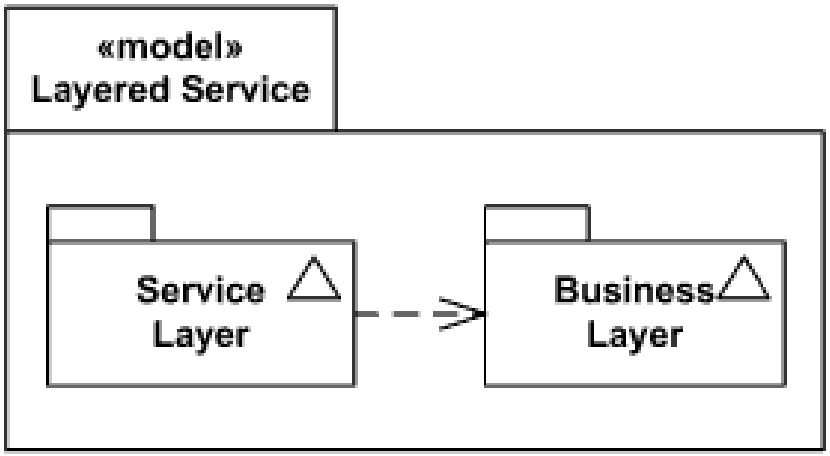
Service Layer model contains service interfaces and message types.



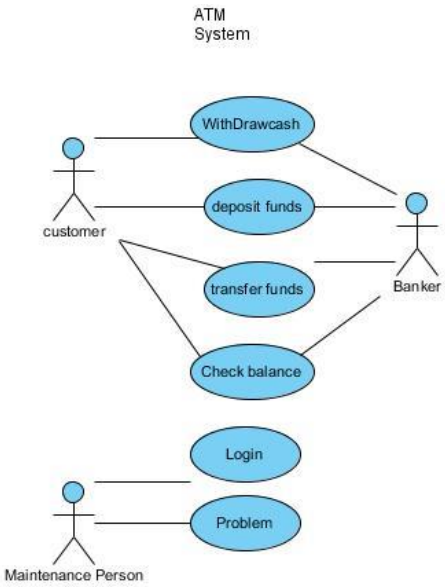
Model

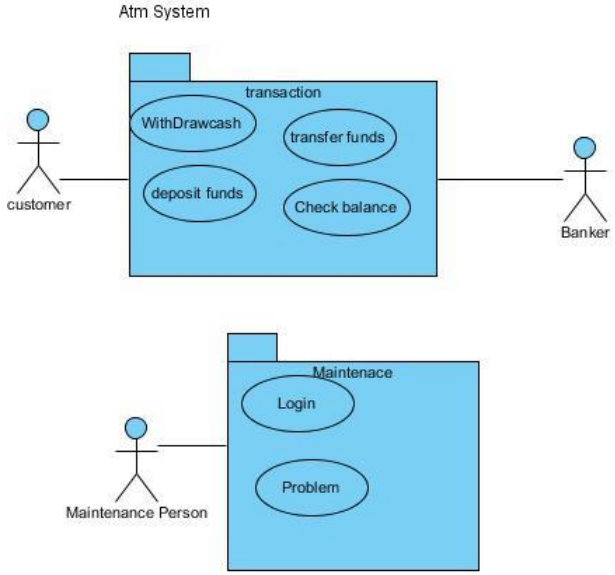
Model could be notated as a package with the keyword «model» placed above the name of the model.

Stereotyped model Layered Service :



Use Case Package Diagram

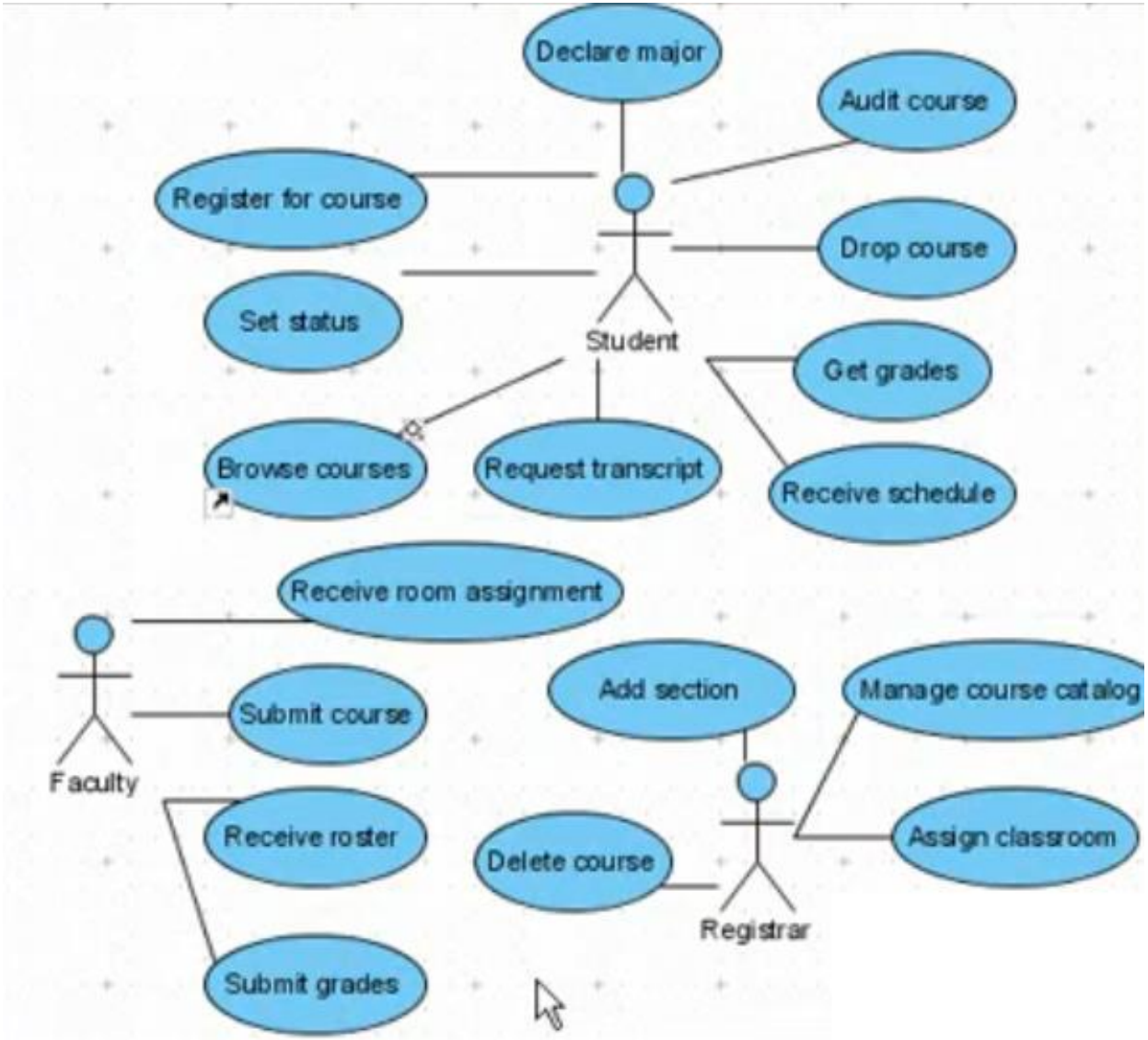




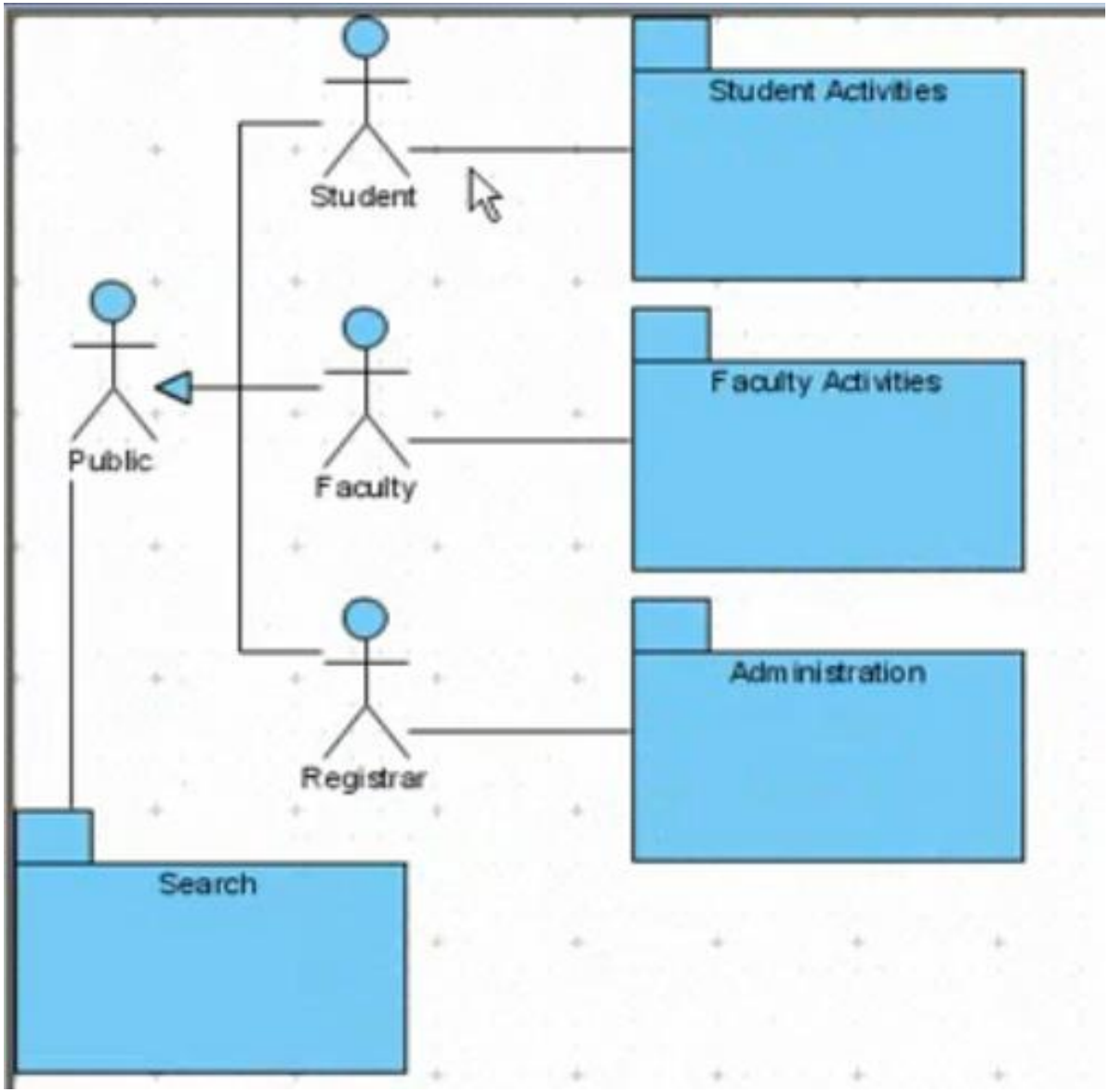
Use case Diagram ➔➔

Use case Package Diagram

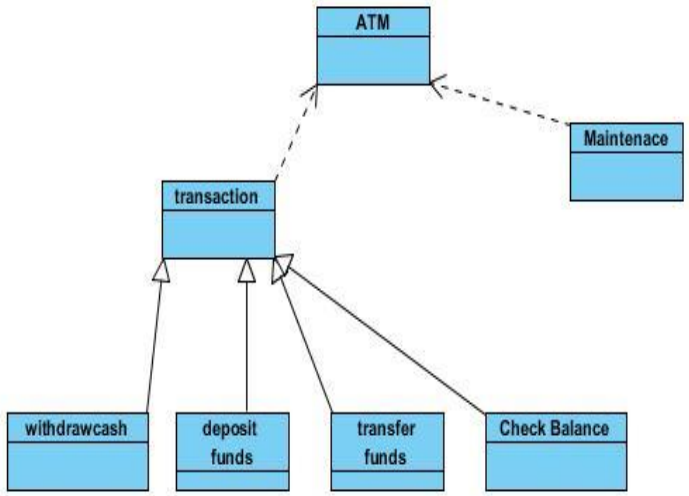
Use Case Package Diagram

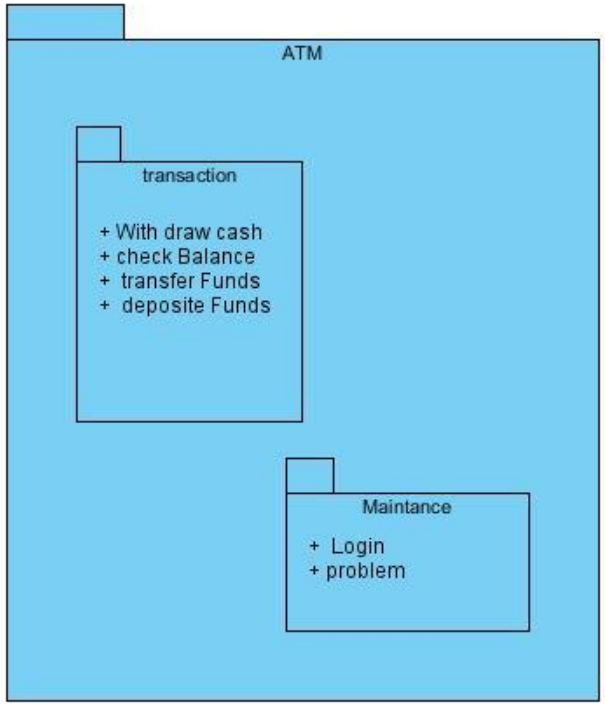


Use Case Package Diagram



Class Package Diagram





class Diagram ➔➔ class Package Diagram

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.

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

Component Diagram

Component Diagram

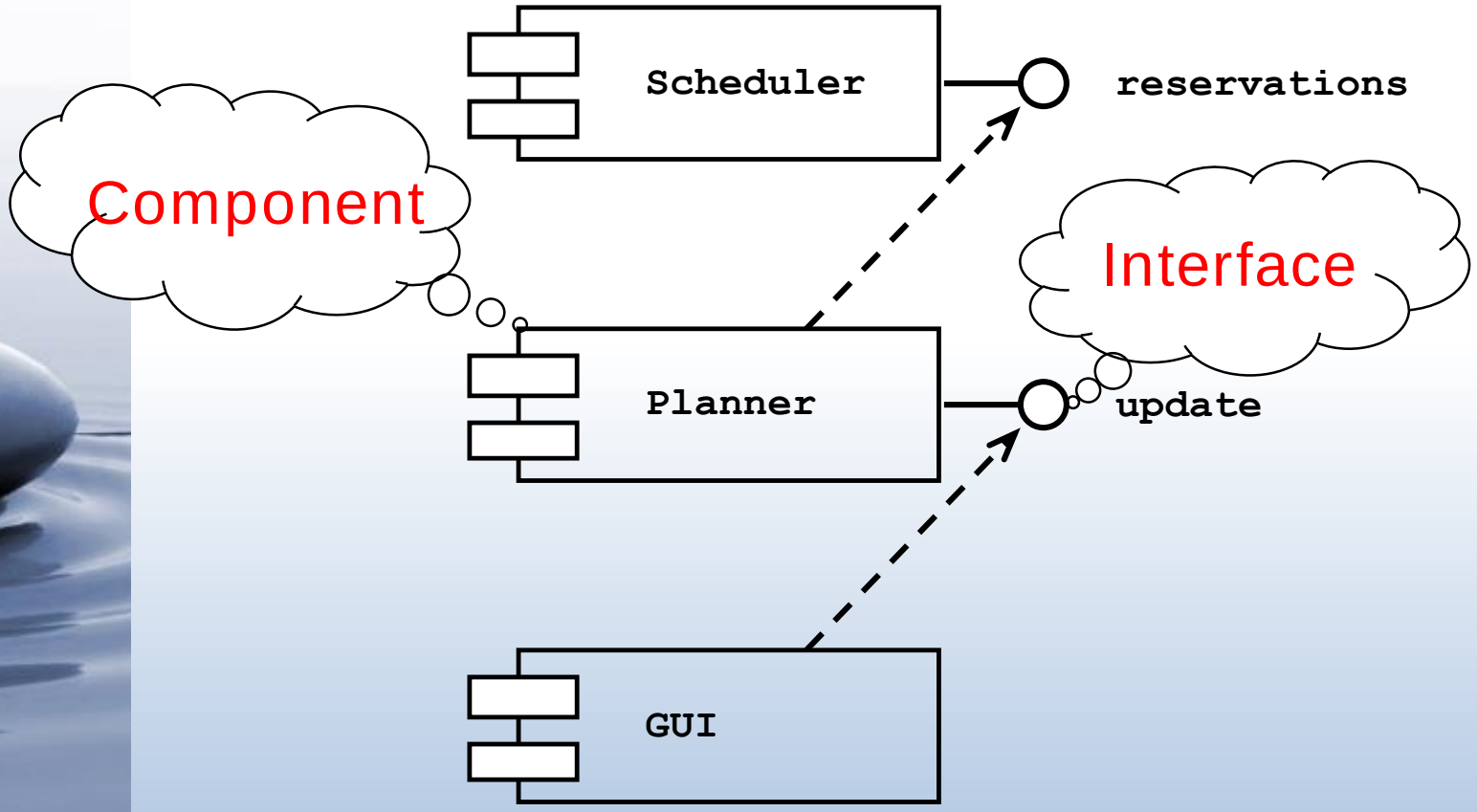
A graph of components connected by dependency relationships.

Shows the dependencies among software components source code, linkable libraries, executables

Dependencies are shown as dashed arrows from the client component to the supplier component.

The kinds of dependencies are implementation language specific.

Component Diagram Example



Deployment Diagram

Deployment diagrams are useful for showing a system design after the following decisions are made

Subsystem decomposition

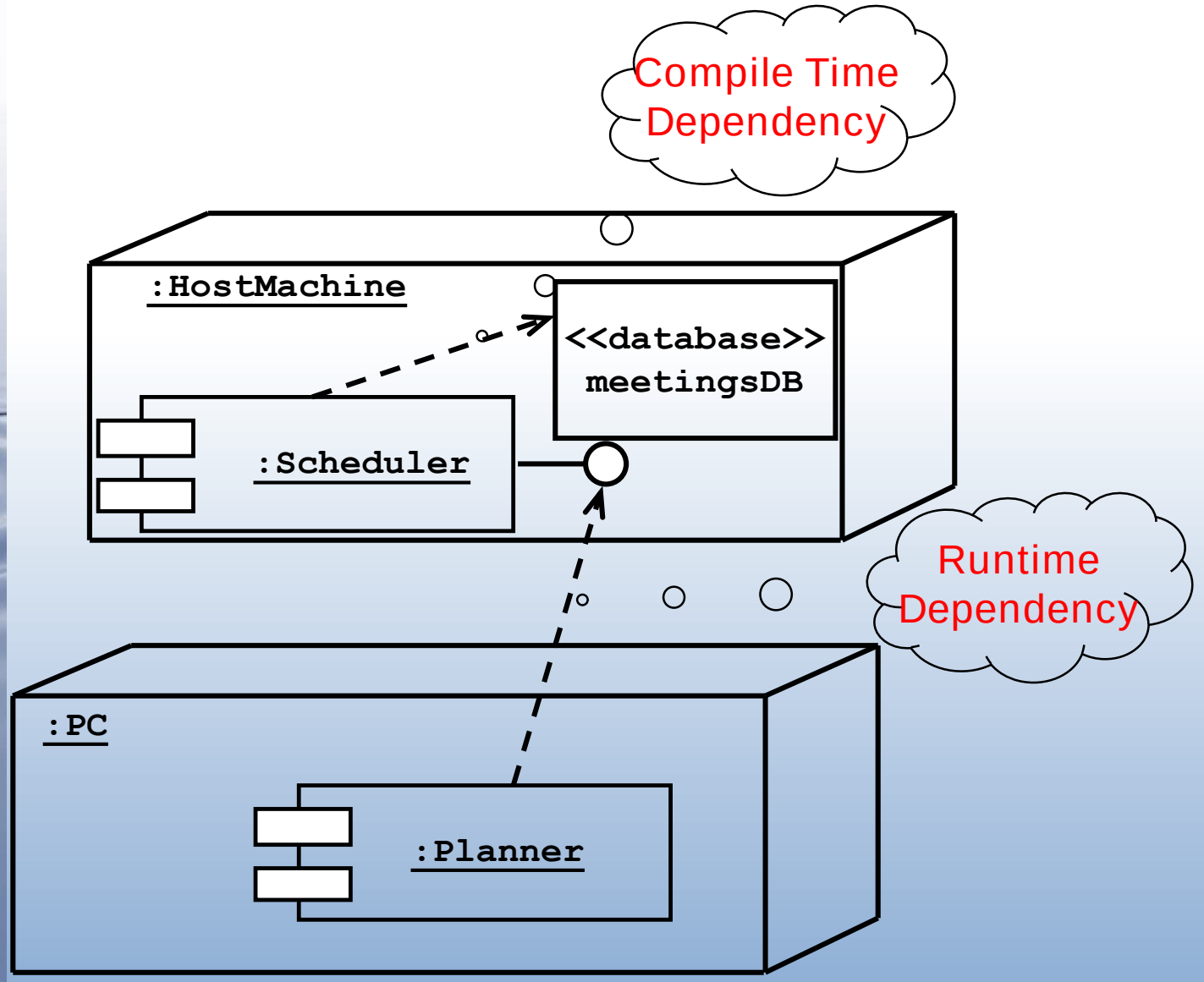
Concurrency

Hardware/Software Mapping

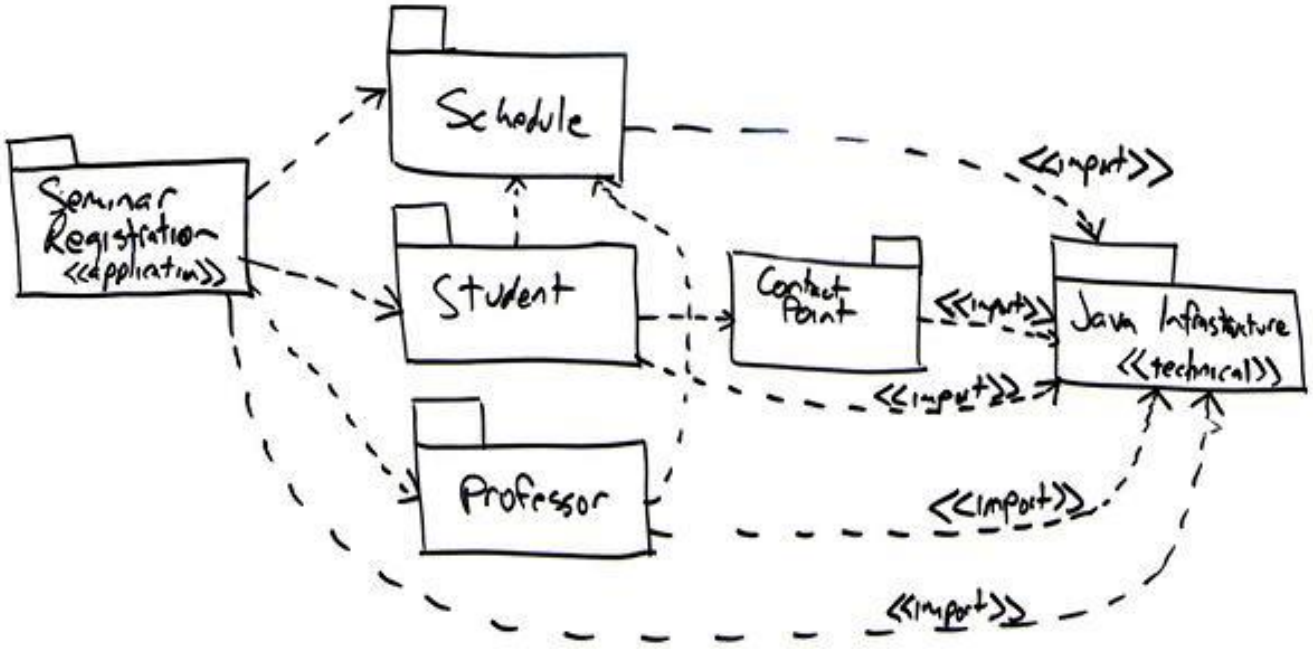
A deployment diagram is a graph of nodes connected by communication associations.

- Nodes are shown as 3-D boxes.
- Nodes may contain component instances.
- Components may contain objects (indicating that the object is part of the component)

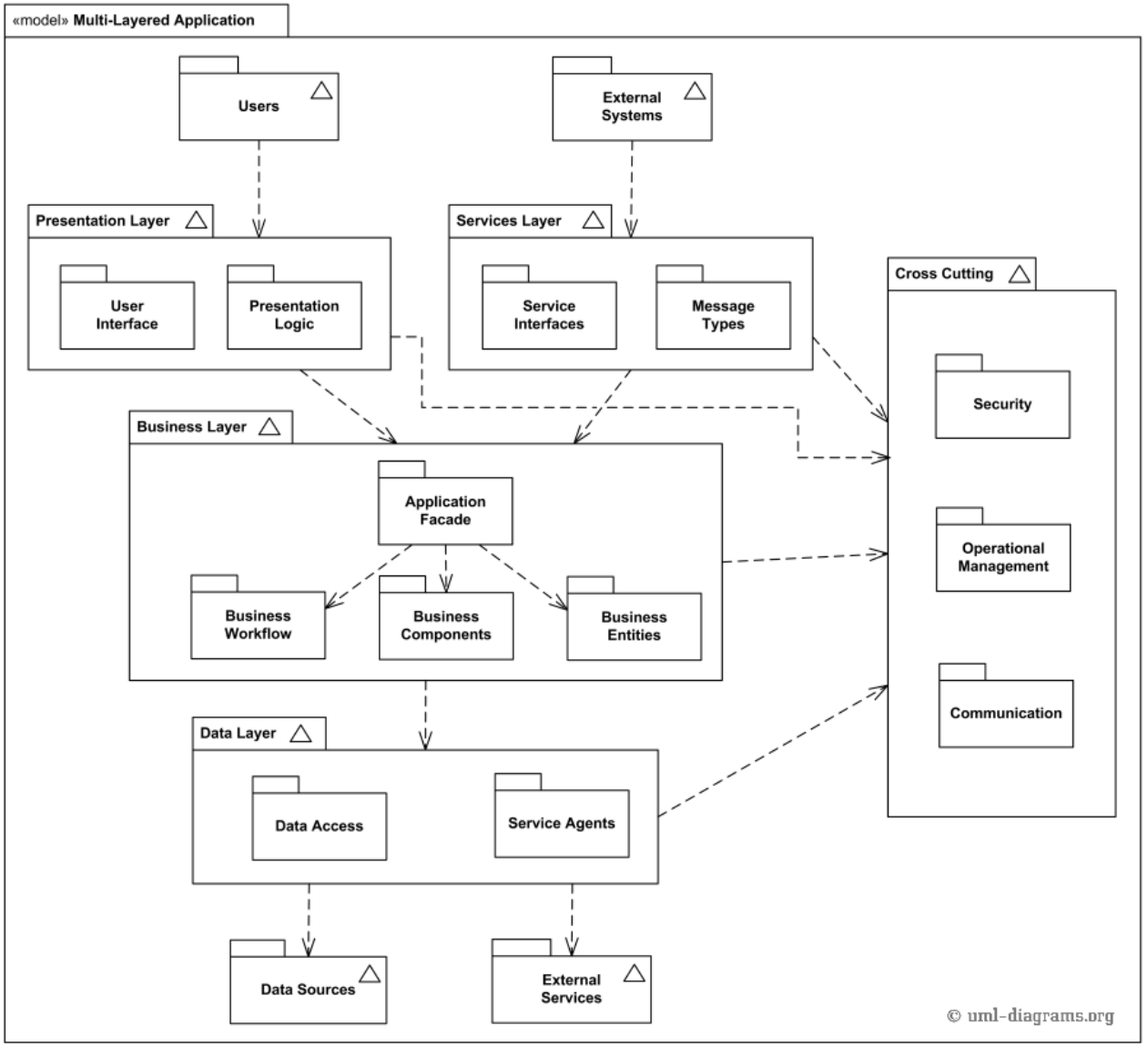
Deployment Diagram Example



Exercise #1

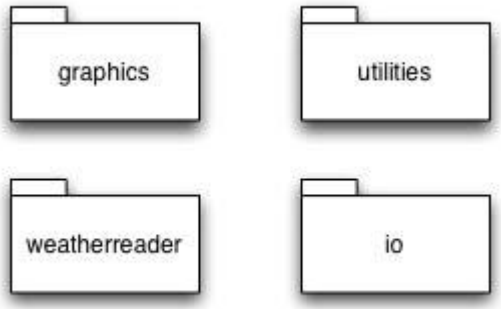

Schedule
Seminar Registration
«application»
Student
Contact Point
Java Infrastructure
«technical»
Professor
«import»
«import»
«import»
«import»

Exercise #2


«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

Exercise #3

You're writing a program to display current weather which contains the following packages:



Draw the package diagram

Exercise #4

Draw a deployment diagram given that the nodes are three client PCs, a server and a printer. The communications protocol between the clients and server is TCP/IP; and between the server and the printer is a standard parallel printer protocol. The user interface and the control objects will run on the clients.

Exercise #5

Supermarket Warehouse Information System (SWIS)

SWIS will maintain information about stock levels and will be able to communicate to headquarters when stock levels of any product fall dangerously low. The current inventory system will be retained as a legacy sub-system. SWIS will keep track of both vehicle loading and route dispatch to ensure only the minimum necessary numbers of trips are made to supermarkets. SWIS will look after staff rosters and the payment of staff salaries into their bank accounts. Layout of stock in the warehouse will also be a responsibility of SWIS.

Exercise #6

Three entity classes are used in a collaboration – CarSharer, Journey and Address. Each of these classes will be implemented by a (.java) source file. These classes are used across a number of use cases and are grouped together into a CarSharing component as Java (.class) files. here we are just dealing with the source files. There are two other classes MCSUserInterface and MCSControl. Each of these will be implemented by a (.java) file. The MCSControl component has a dependency on the CarSharing component and on the MCSUserInterface component

A) Draw a component diagram showing the source code dependencies. The .class files are grouped together into two Java archive (.jar) files. The MCSControl.class component will need to read a configuration file (MCS.ini) and display a help file (MCS.hlp) when required. The MCSControl (.class) file also has dependencies on the MCSUserInterface (.java) file and the CarSharing (.jar) components.

B) Re-draw the component diagram to show the run-time component dependencies.

Any Questions?

✉ hvusynh@hcmiu.edu.vn