2IW80 Software specification and architecture

Structural specification: beyond class diagrams

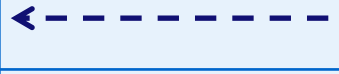
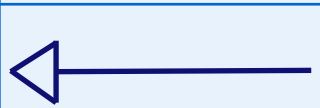
Alexander Serebrenik

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Before we start

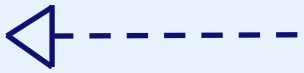
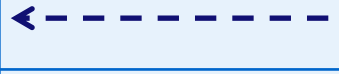
- Match the pairs

1	Association	A	
2	Aggregation	B	
3	Composition	C	
4	Implementation	D	
5	Generalization	E	
6	Dependency	F	

Before we start

- Match the pairs

1E 2C 3F 4A 5D 6B

1	Association	A	
2	Aggregation	B	
3	Composition	C	
4	Implementation	D	
5	Generalization	E	
6	Dependency	F	

Before we start

- A patient must be assigned to only one doctor, and a doctor can have one or more patients.

Determine x and y

This week sources

OMG Unified Modeling Language ™ (OMG UML)

Version 2.5

Slides by

David Meredith,
Aalborg University, DK

Marie-Elise Kontro,
Tampere University, FI

Site by

Kirill Fakhroutdinov
GE Healthcare, USA

Recall

Structural diagram is a diagram that identifies **modules, activities, or other entities** in a system or computer program and **shows how larger or more general entities break down into smaller**, more specific entities.

IEEE Standard Glossary of Software Engineering Terminology 610.12 1990

UML structure diagrams

Class diagram 

TODAY

Object diagram

Packages diagram

Component diagram

Deployment diagram

Composite structure diagram

Between specification and architecture

- **Packages diagram** and **deployment diagram**: the closest UML diagrams come to architecture
  - more about architecture: second half of the quartile

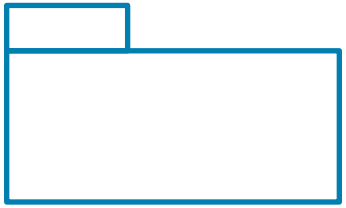
Packages diagram

- Represents the system at a **higher abstraction level**
  - Android SDK – 69 packages vs. 1231 classes
  - less prone to change, ergo better suited for evolution, than lower level representations

- NB: *Packages diagram* (UML standard) is frequently called *package diagram*

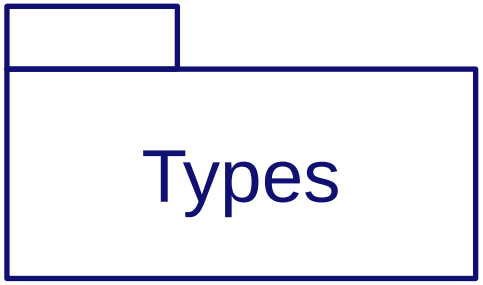
Packages diagram: Packages and Relations

- **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

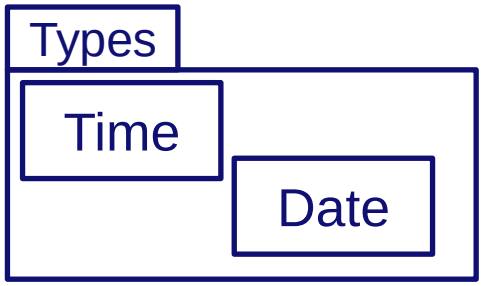


- **Relations**
  - dependencies, implementations, …
  - *imports* and *merges*

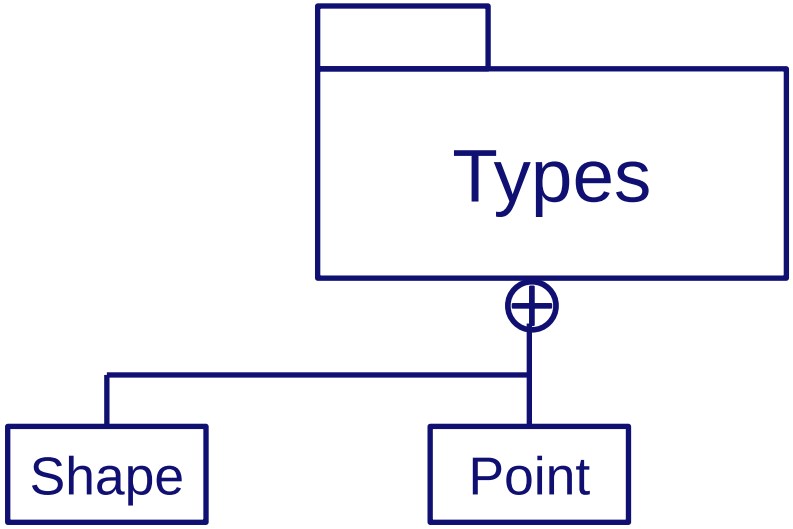
Package representations



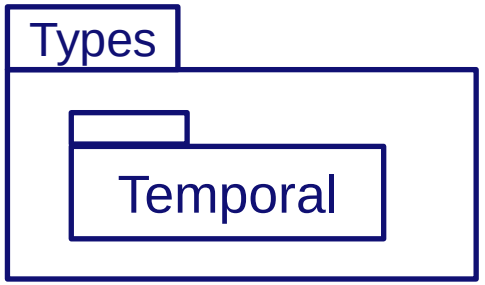
Package Types,
members not shown



Package Types, ***some*** members within the borders of the package



Package Types, ***some*** members shown using ⊕-notation



Nested packages

Relations

- **Dependency**
- **Implementation**
- **Import / access**
- **Merge**

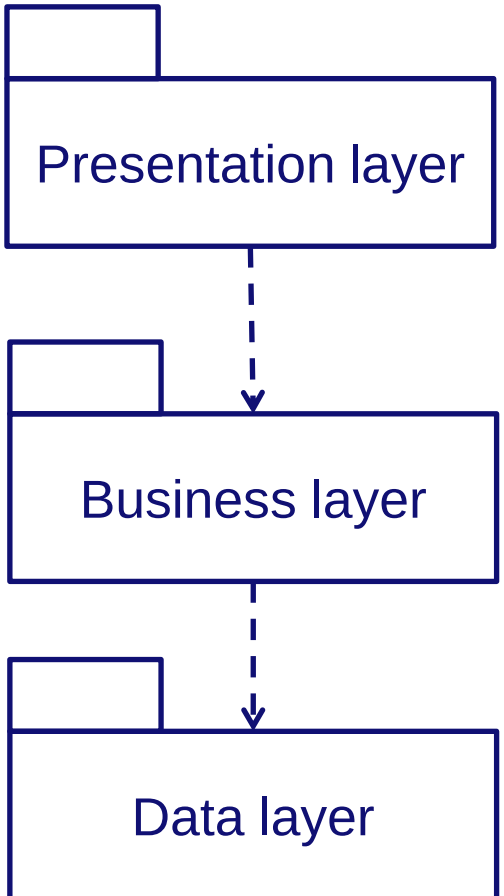
Relations: Dependencies

- Package A **depends** on package B if A contains a class which depends on a class in B
  - Summarise dependencies between classes

- Graphic representation:

Relations: Dependencies

- Package A **depends** on package B if A contains a class which depends on a class in B
  - Summarise dependencies between classes
- Typical 3-tier application (*sketch*):



UI, web-interface, services to other systems

Core calculations, operations, etc

Data storage (DB)

Relations: Dependencies

- Package A **depends** on package B if A contains a class which depends on a class in B
  - Summarise dependencies between classes

- Martin's **Acyclic Dependency Principle**

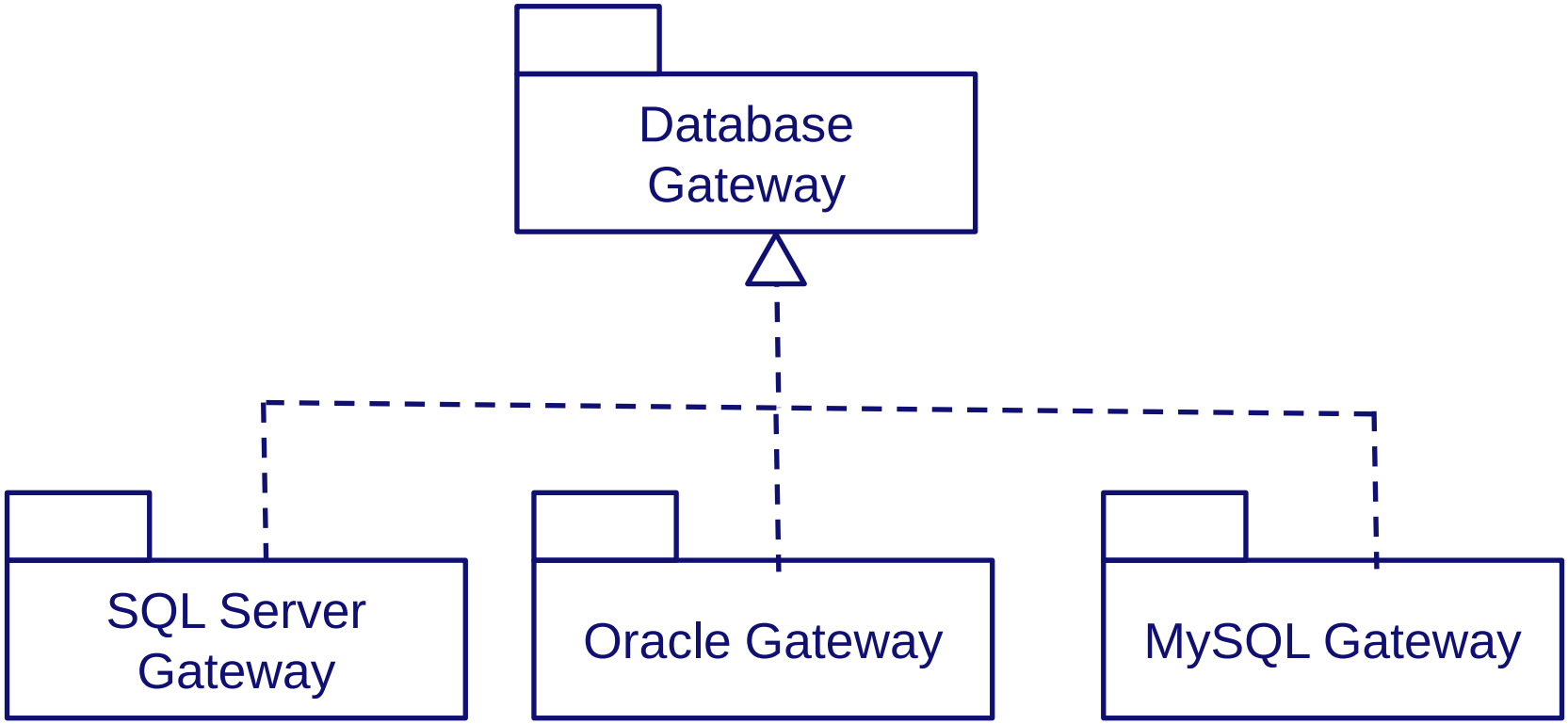
  there should be no cycles in the dependencies

- Fowler:

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

Relations: Implementations

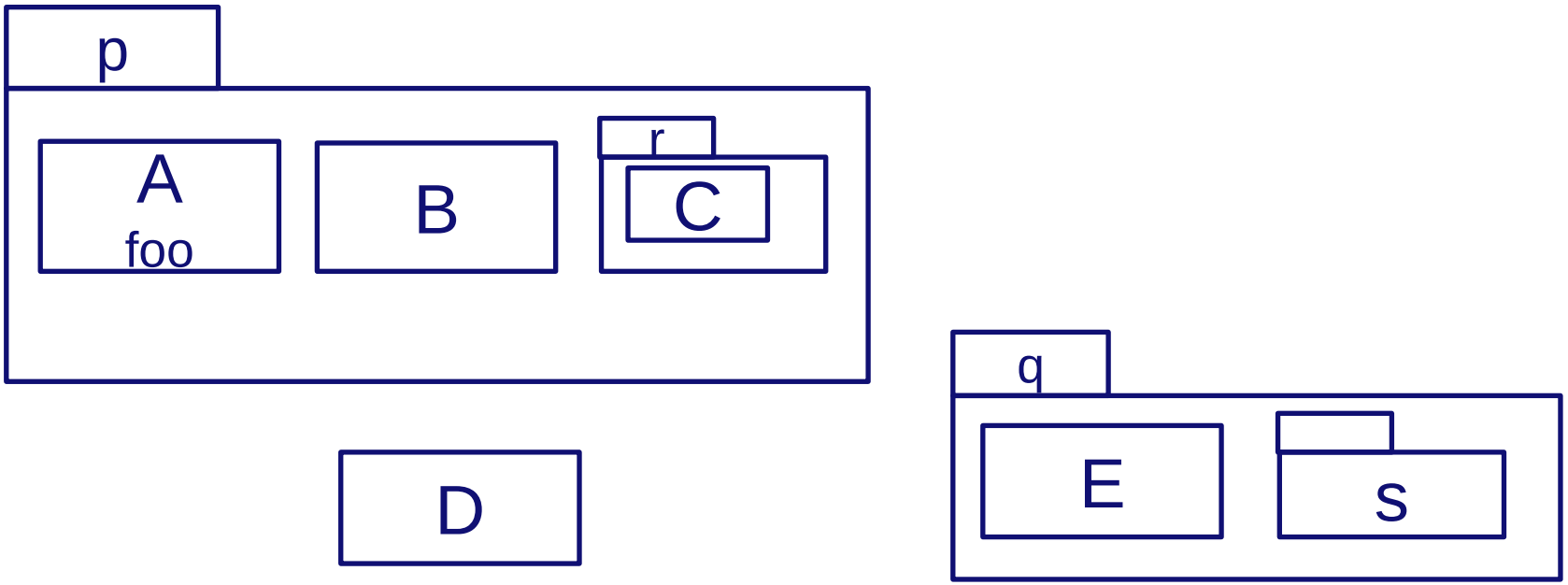
- Meaningful if multiple variants are present



Relations: 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**

How elements can reference each other? (1)



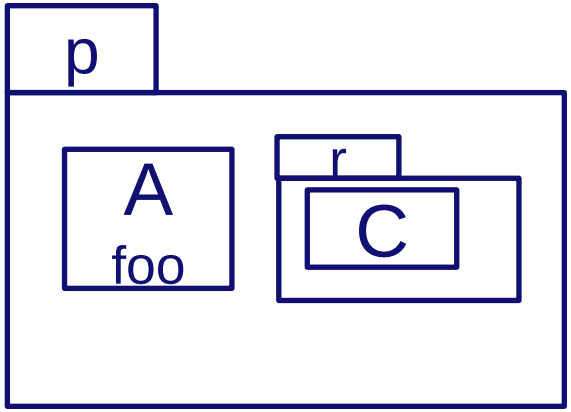
- Element can refer to other elements that are in its own package and in enclosing packages without using fully qualified names

Do you remember?

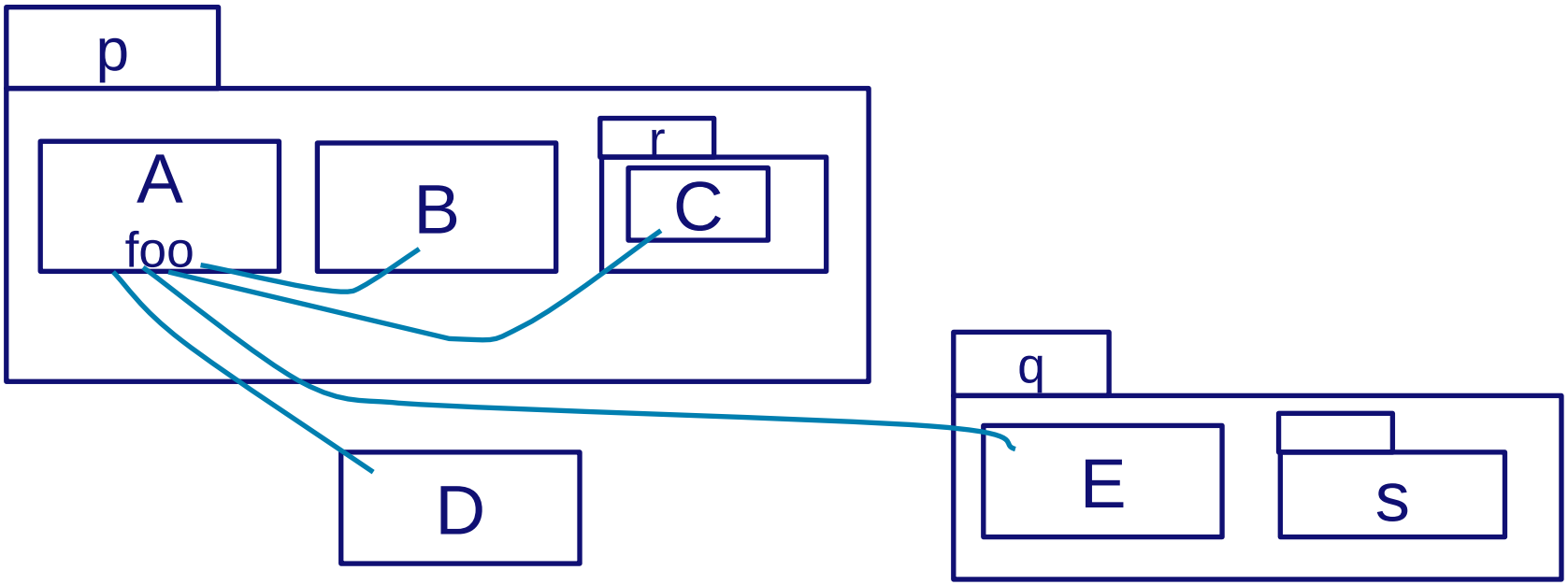
- **Fully qualified name**: a globally unique identifier of a package, class, attribute, method.

- **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**



How elements can reference each other? (2)



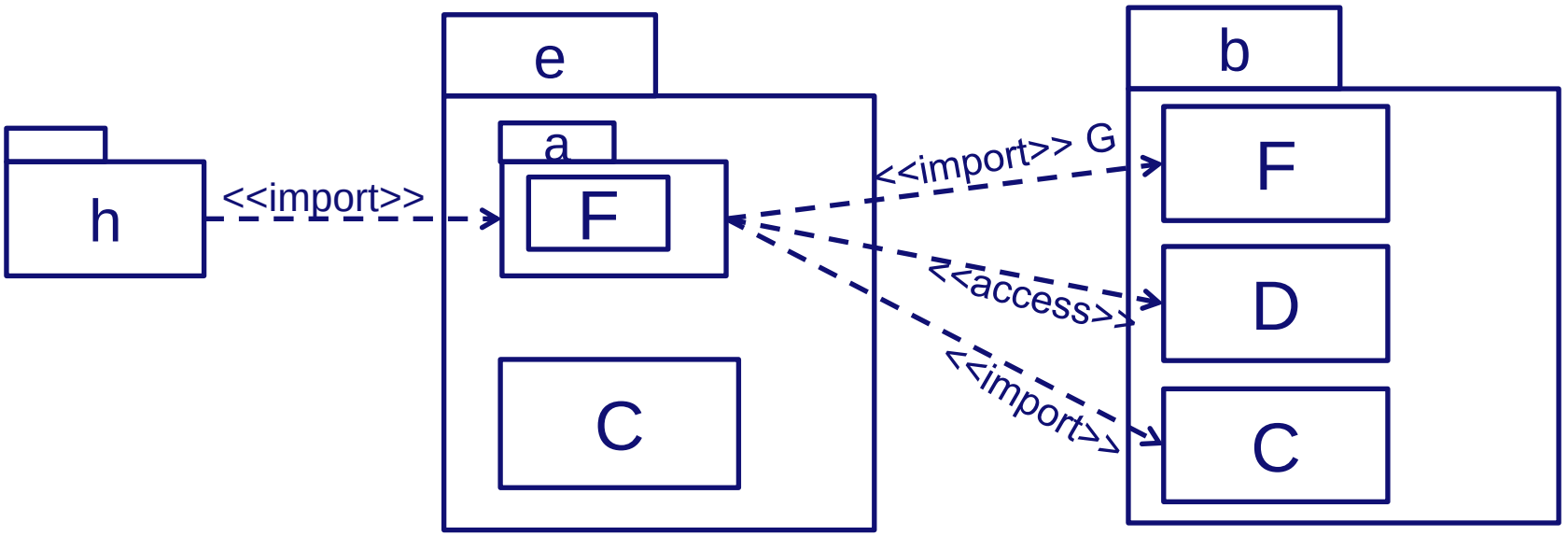
- Element can refer to other elements that are in its own package and in enclosing packages without using fully qualified names

Element Import (1)

- 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

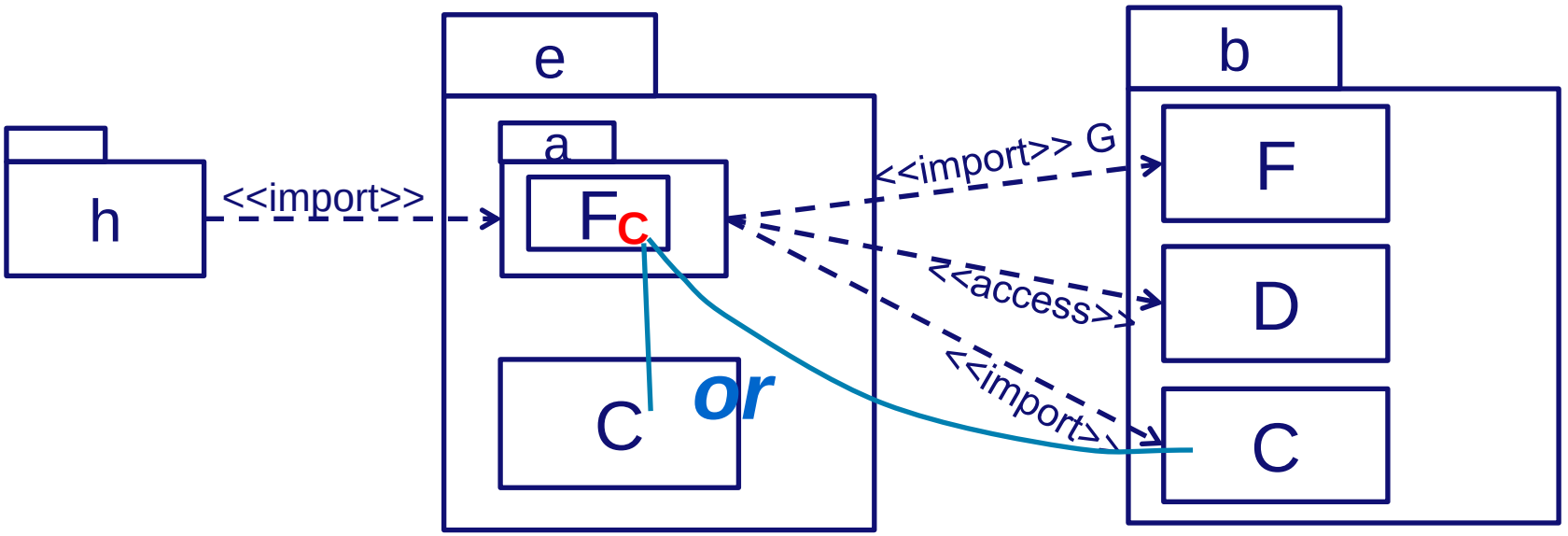
Element Import (2)

- 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



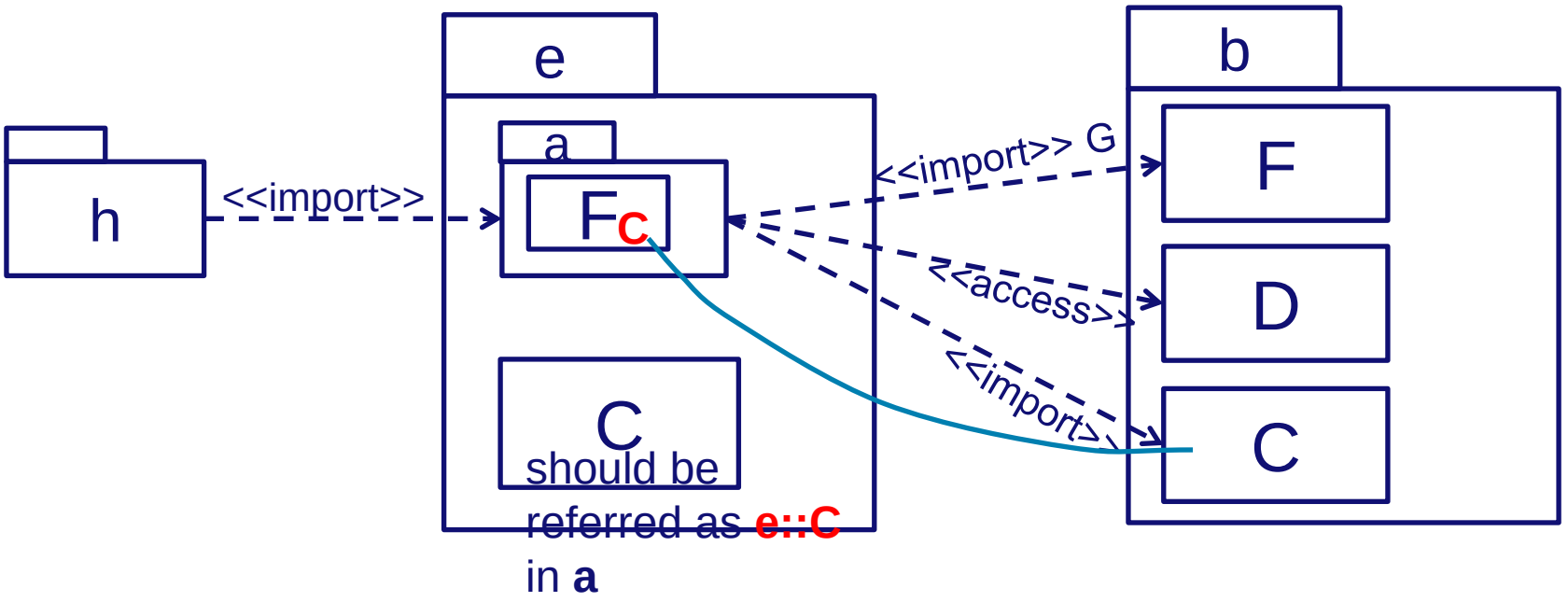
Element Import (3)

- 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



Element Import (4)

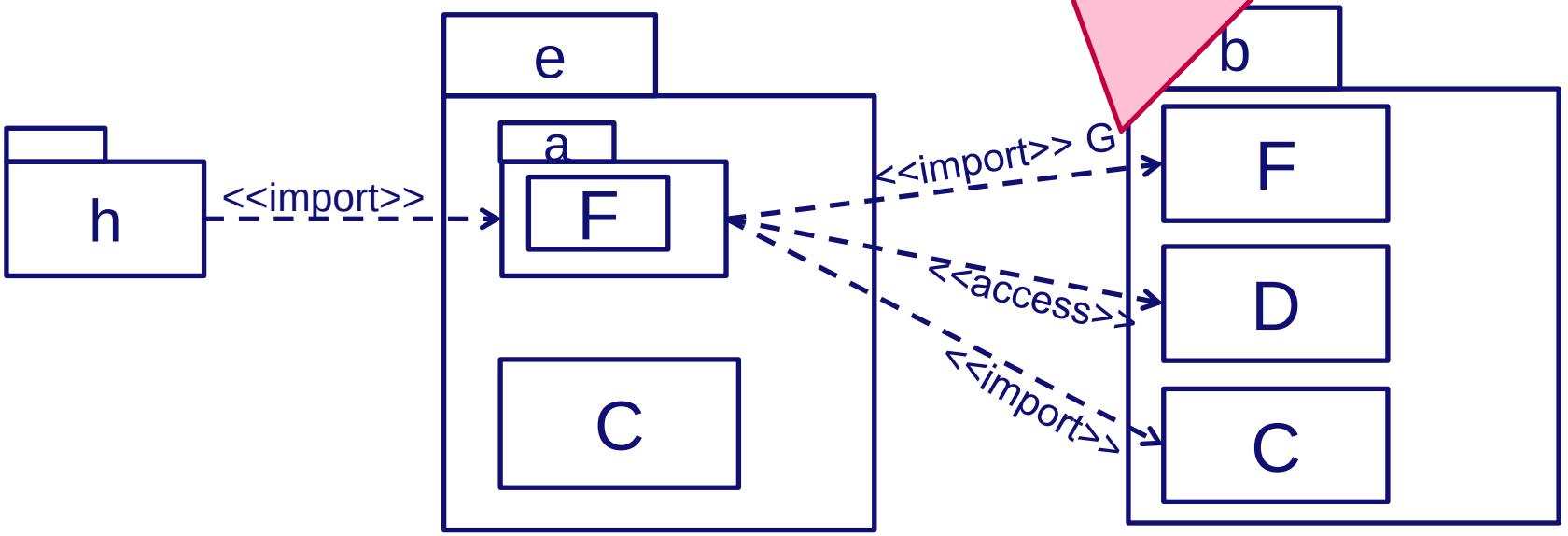
- Element import allows an element in another package to be referenced using its name without a qualifier
  - **<<import>>** imported element within importing package is public
  - **<<access>>** imported element within importing package is private



Element Import (5)

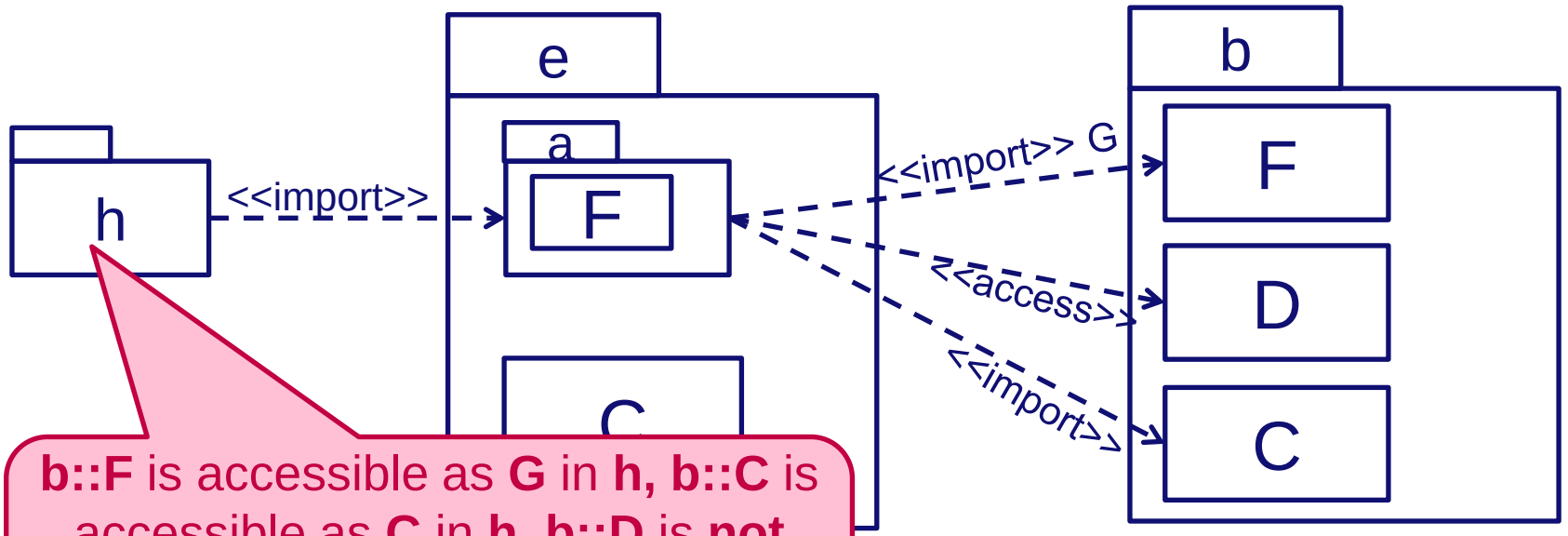
- Element import allows an element in another package to be referenced us
  - **<<import>>** impor s public
  - **<<access>>** impo is private

F cannot be imported to a since there is already an **F** in **a**. Hence, we need to rename **b::F** to **G** in **a**.



Element Import (6)

- 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



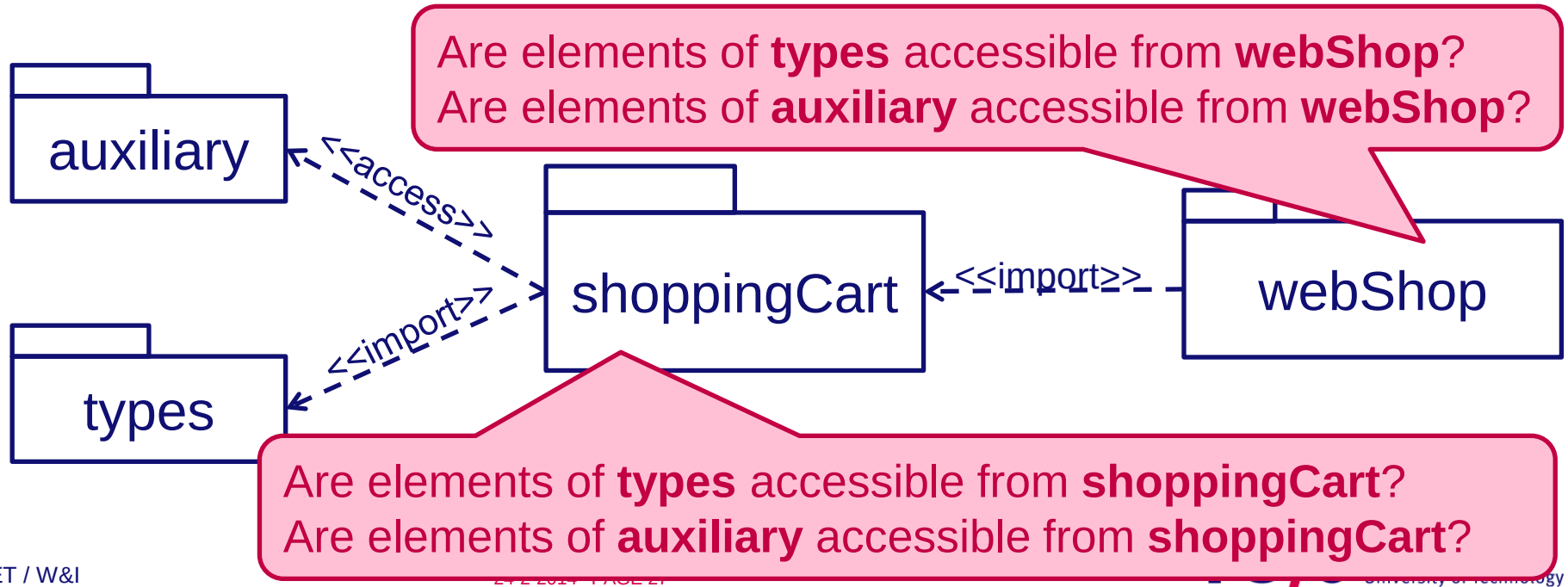
b::F is accessible as **G** in **h, b::C** is accessible as **C** in **h**, **b::D** is **not** accessible in **h** (private visibility of **b::D** in **a** due to **<<access>>**).

Package import (1)

- A **package import** 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

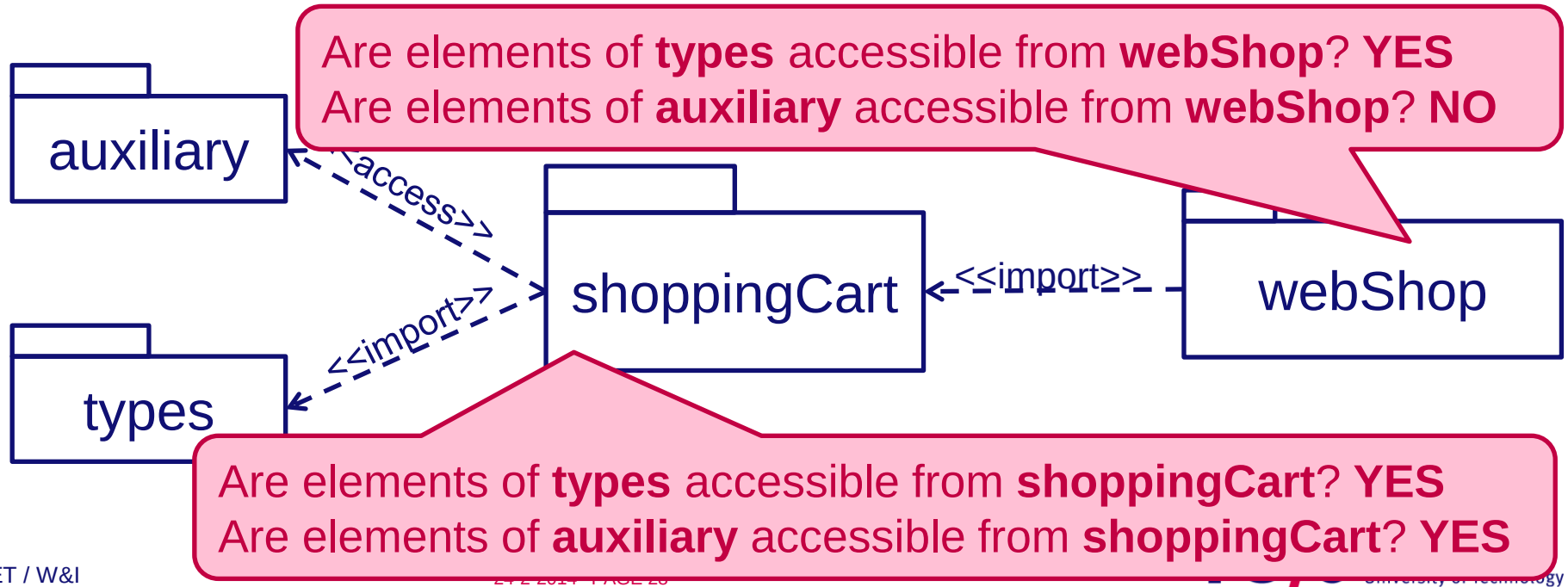
Package import (2)

- 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



Package import (2)

- 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



Relations: Recap

- ✓ Dependency
- ✓ Implementation
- ✓ Import / access
- Merge

Package merge

- A **package merge** indicates that the contents of the two packages are to be combined.
  - A (merged package) is merged into B (receiving package) that becomes B’ (resulting package)

Package merge

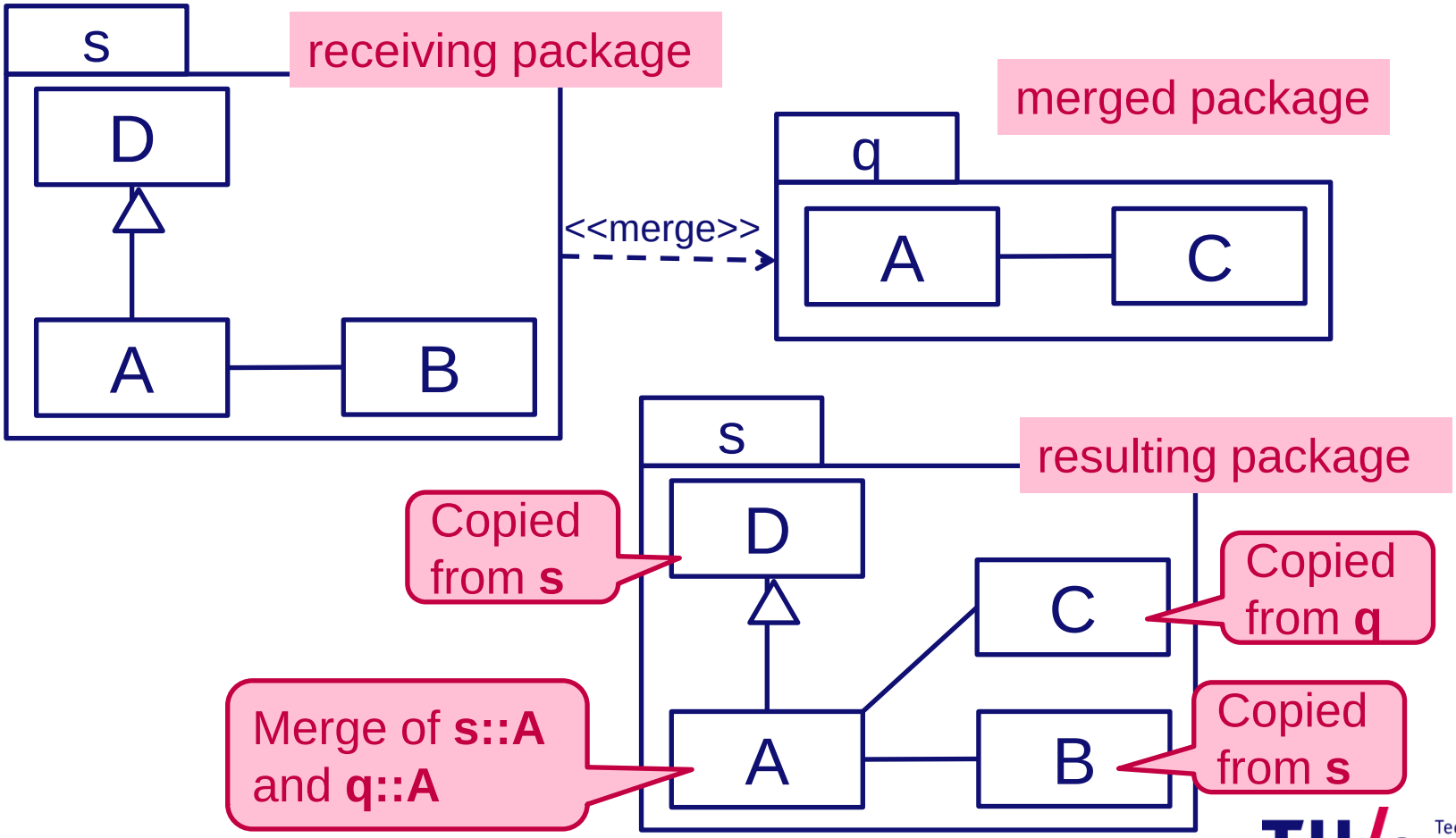
- A **package merge** indicates that the contents of the two packages are to be combined.
  - A (merged package) is merged into B (receiving package) that becomes B' (resulting package)

- 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)

Merge rules

UML 2.5 Beta 2, pp. 252-262

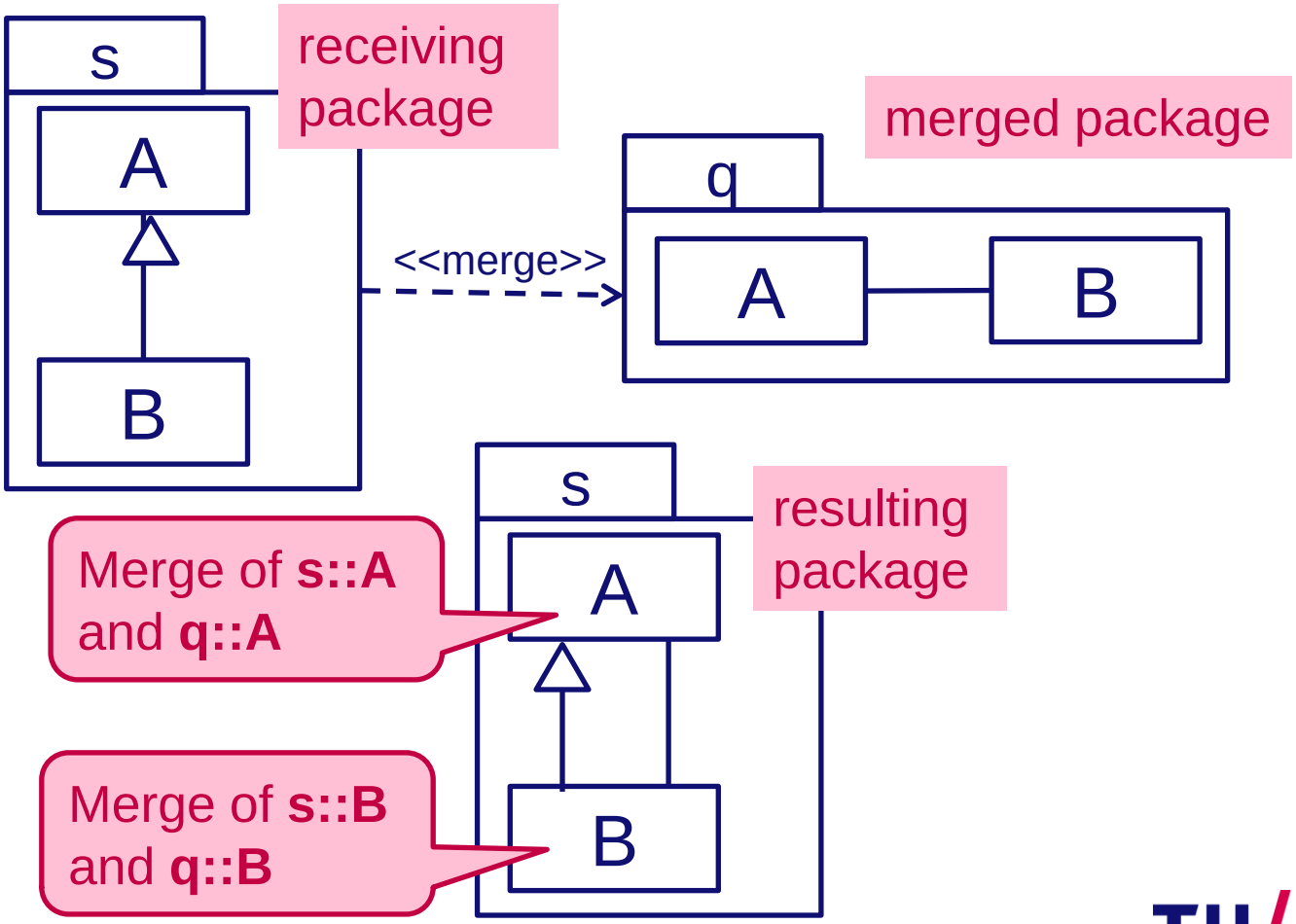
http://www.omg.org/spec/UML/2.5/Beta2/



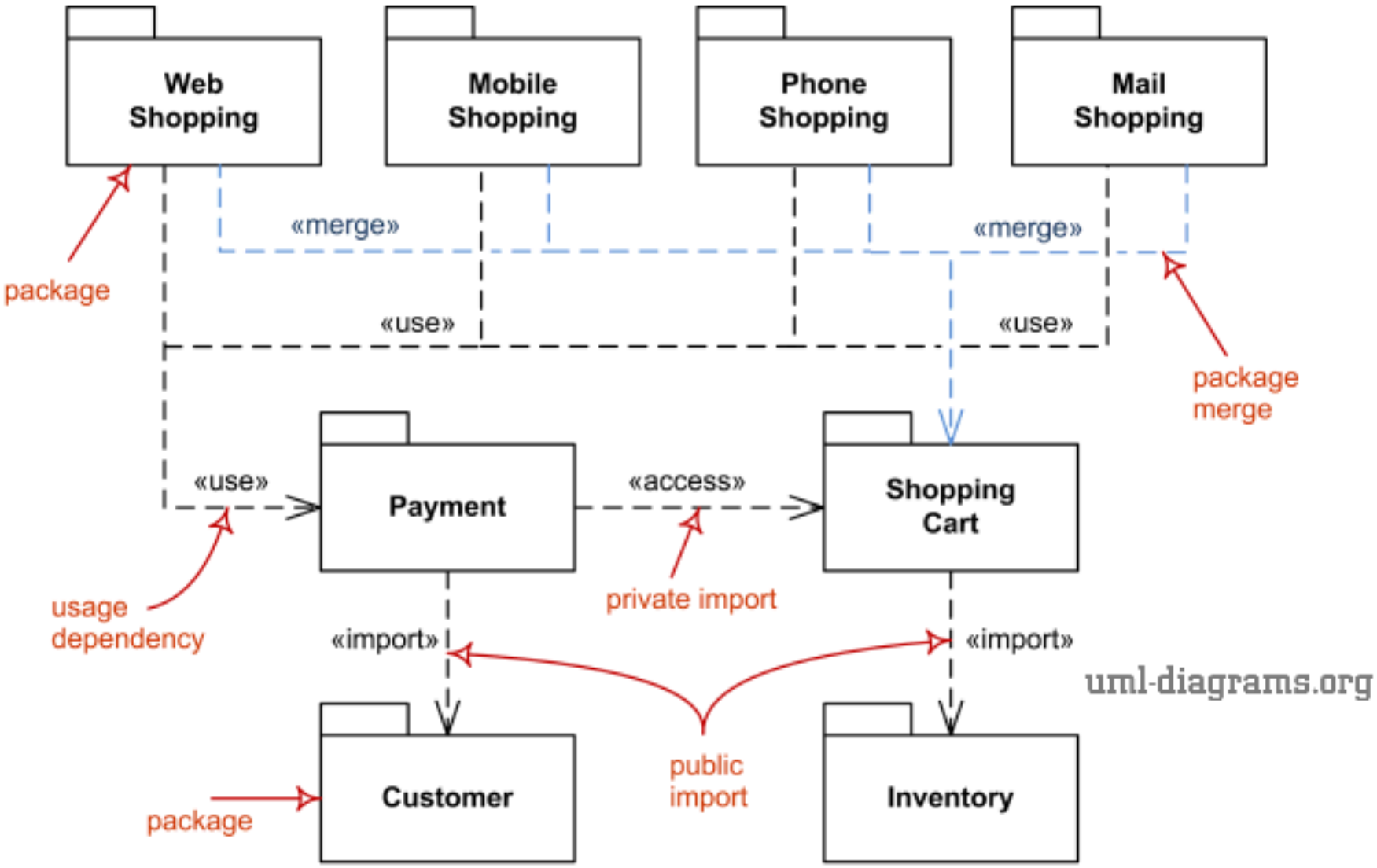
Merge rules

UML 2.5 Beta 2, pp. 252-262

http://www.omg.org/spec/UML/2.5/Beta2/



Summary: UML package diagrams



http://www.uml-diagrams.org/package-diagrams-overview.html

How do we organize classes/use-cases in packages?

- **General**: try to give packages meaningful names

- Two special cases:
  - **Class package diagrams**
    - "basic elements" are class diagrams
    - The most popular special case
  - **Use-case package diagrams**
    - "basic elements" are use-case diagrams
    - Useful for larger projects to organize requirements

Class Package Diagrams

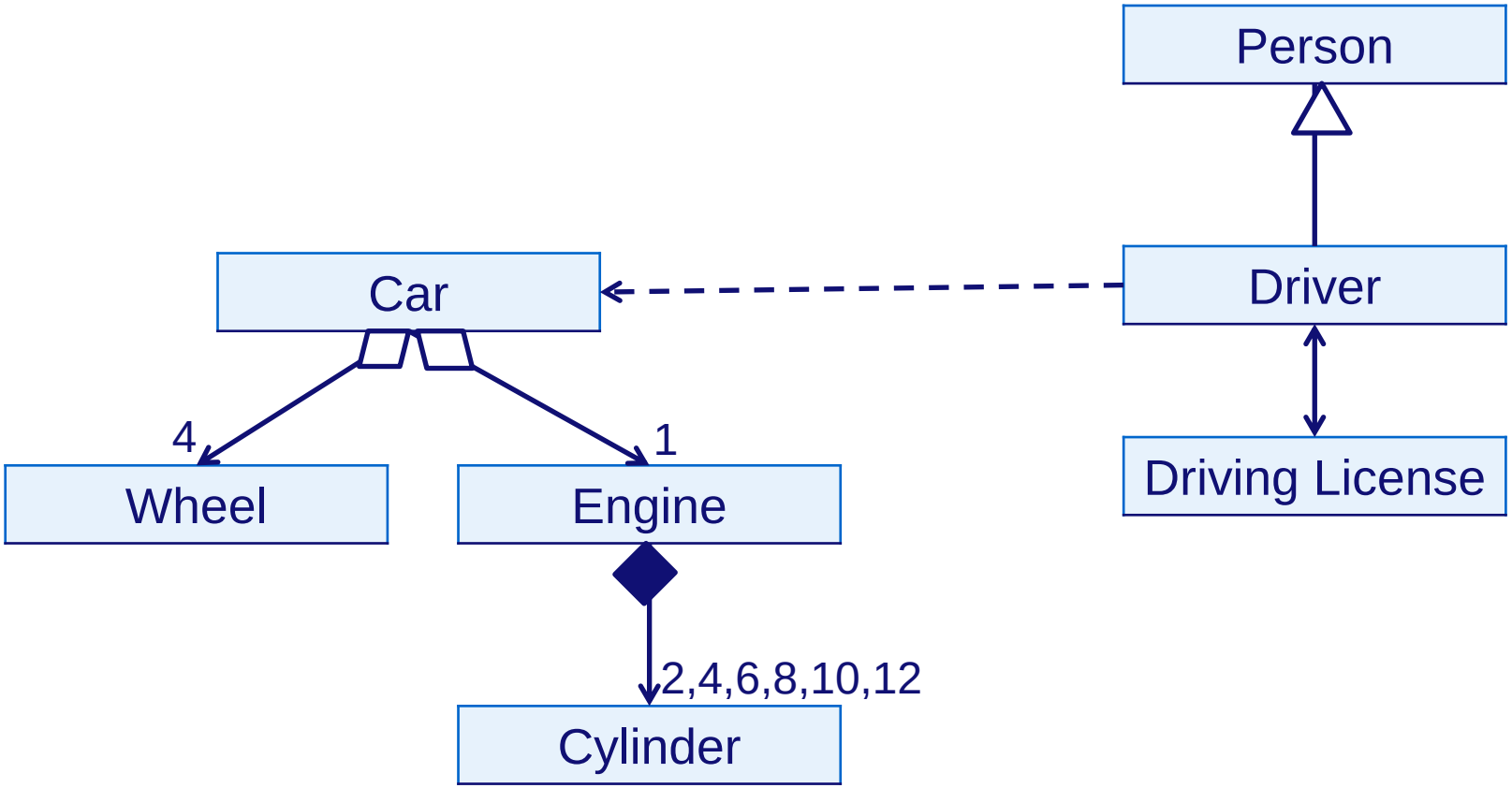
- **Heuristics** to organize classes into packages:
  - Classes of a framework belong in the same package.
  - Classes in the same inheritance hierarchy typically belong in the same package.
  - Classes related to one another via aggregation or composition often belong in the same package.
  - Classes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

How would you organize into 2 packages?

- Car, Cylinder, Driver, Driving License, Engine, Person, Wheel

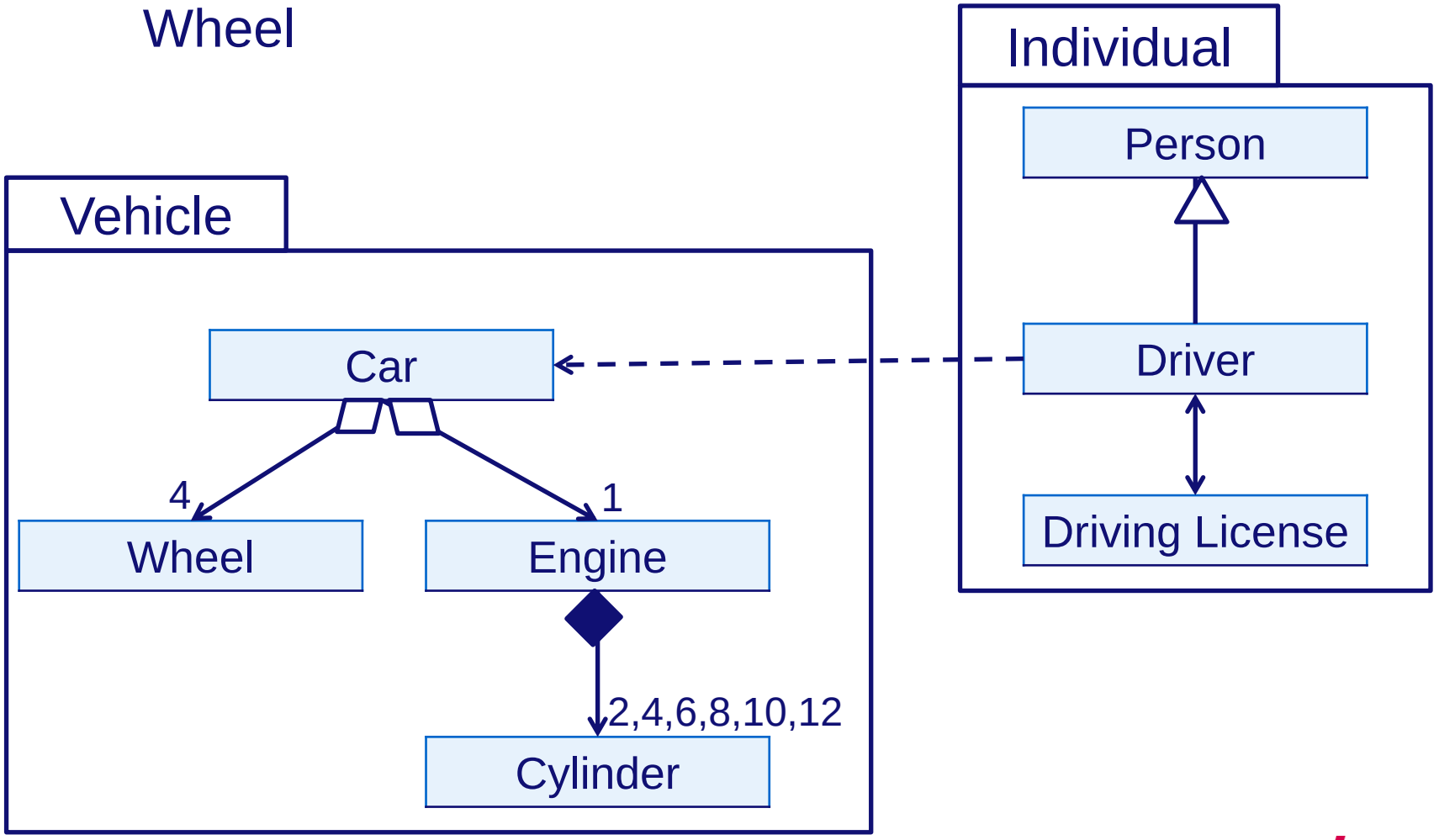
How would you organize into 2 packages?

- Car, Cylinder, Driver, Driving License, Engine, Person, Wheel



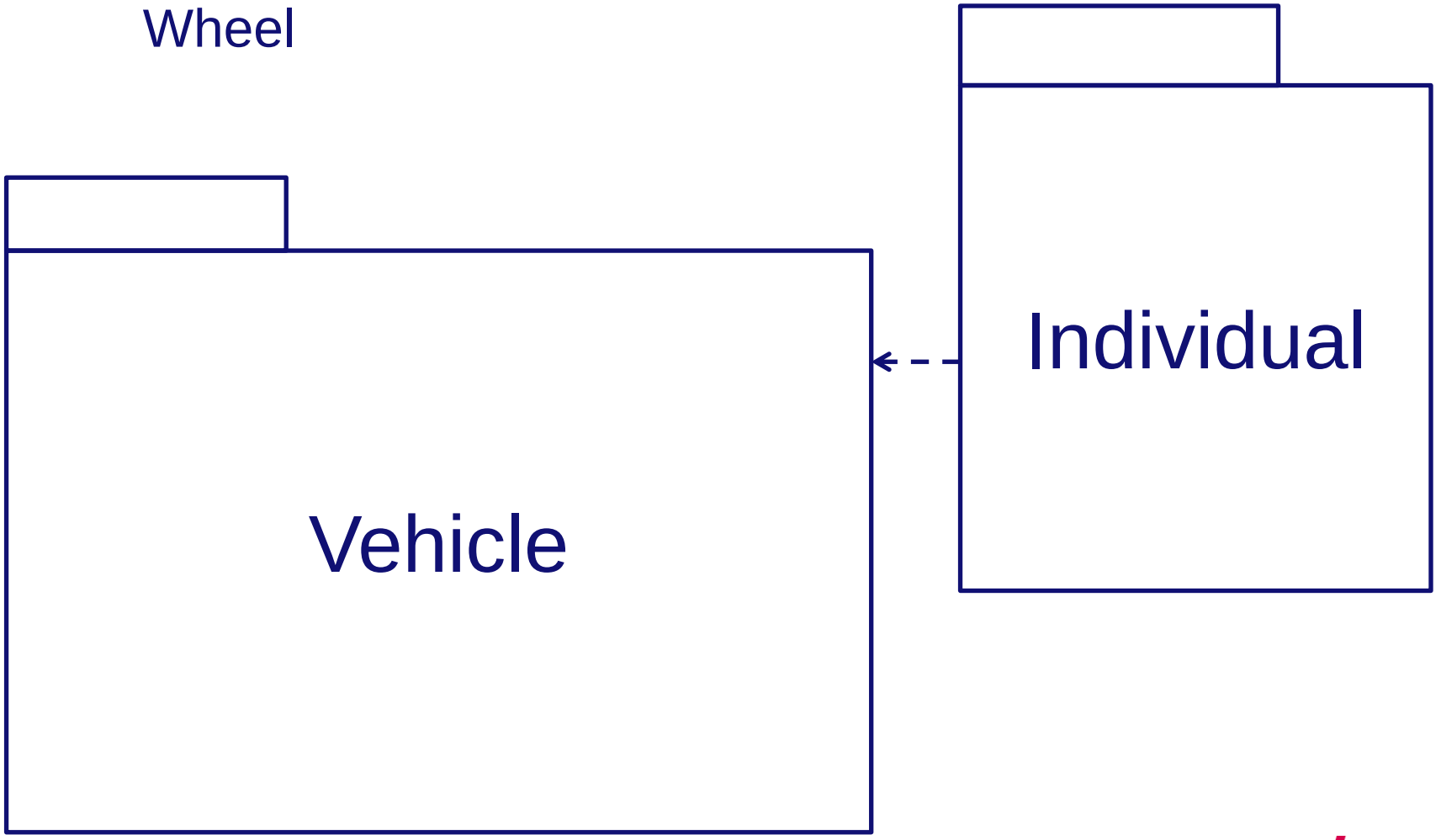
How would you organize into 2 packages?

- Car, Cylinder, Driver, Driving License, Engine, Person, Wheel



How would you organize into 2 packages?

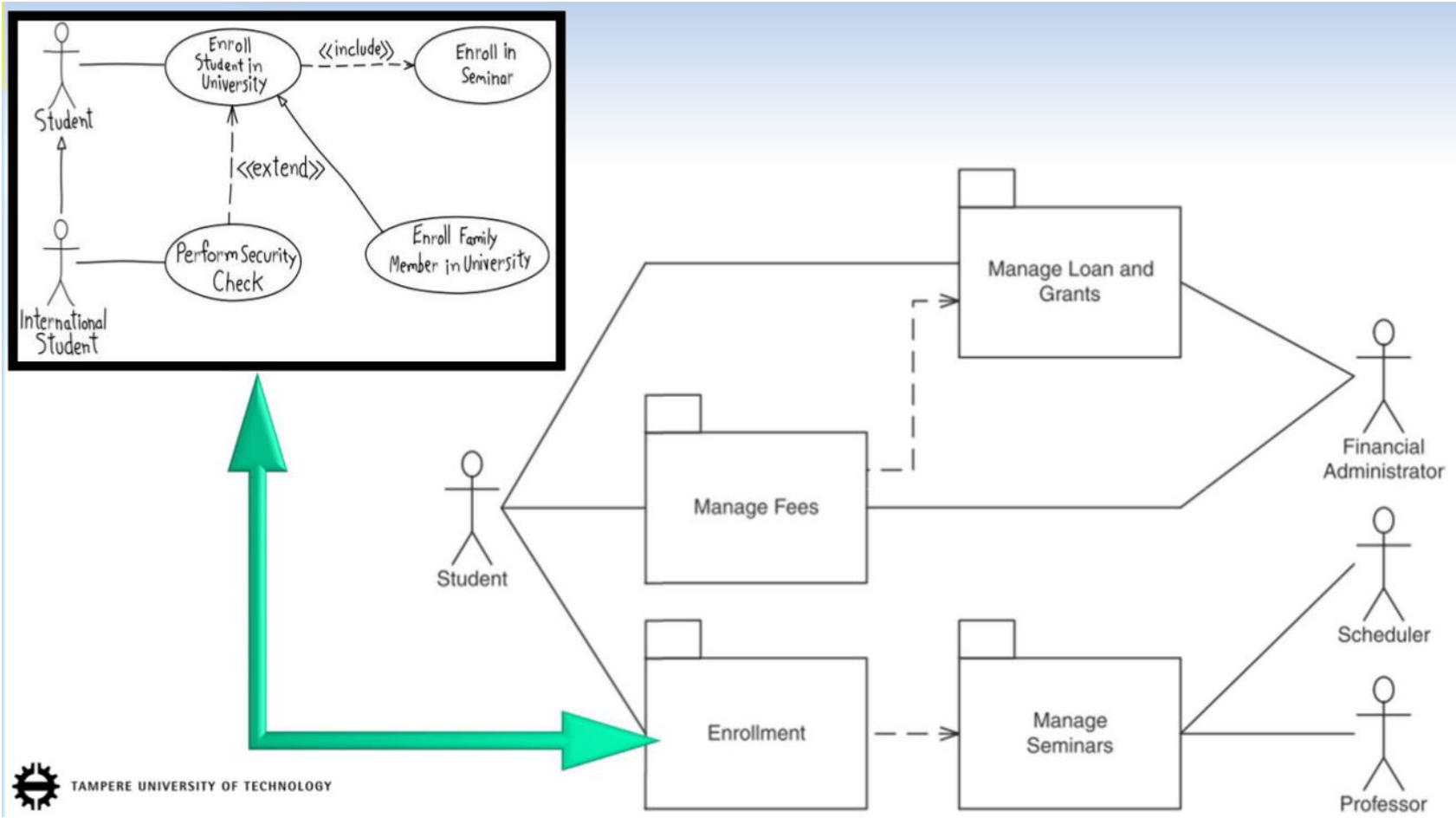
- Car, Cylinder, Driver, Driving License, Engine, Person, Wheel



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 Example



http://www.students.tut.fi/~kontrom/files/Lecture6.pdf

UML structure diagrams

Class diagram

Object diagram

Packages diagram

Component diagram

TODAY

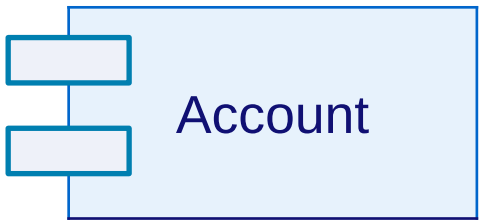
Deployment diagram

Composite structure diagram

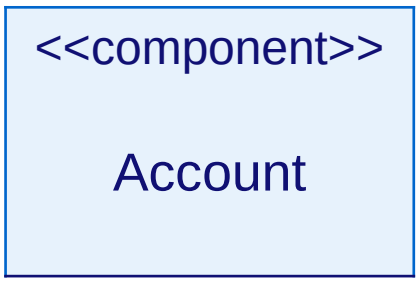
Component diagrams

- **Component**: a modular unit with well-defined interfaces that is replaceable within its environment (UML Superstructure Specification, v.2.0, Chapter 8)
  - fosters reuse
  - stresses interfaces

- Graphical representation: **special kind of class**



UML 1

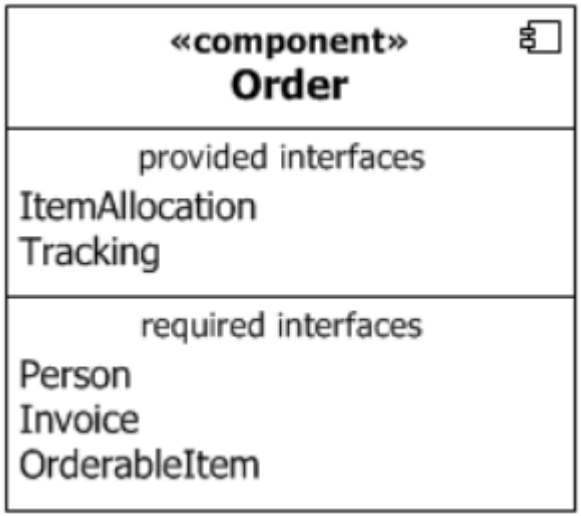


UML 2

Component diagrams

- **Component**: a modular unit with well-defined interfaces that is replaceable within its environment (UML Superstructure Specification, v.2.0, Chapter 8)
  - fosters reuse
  - stresses interfaces

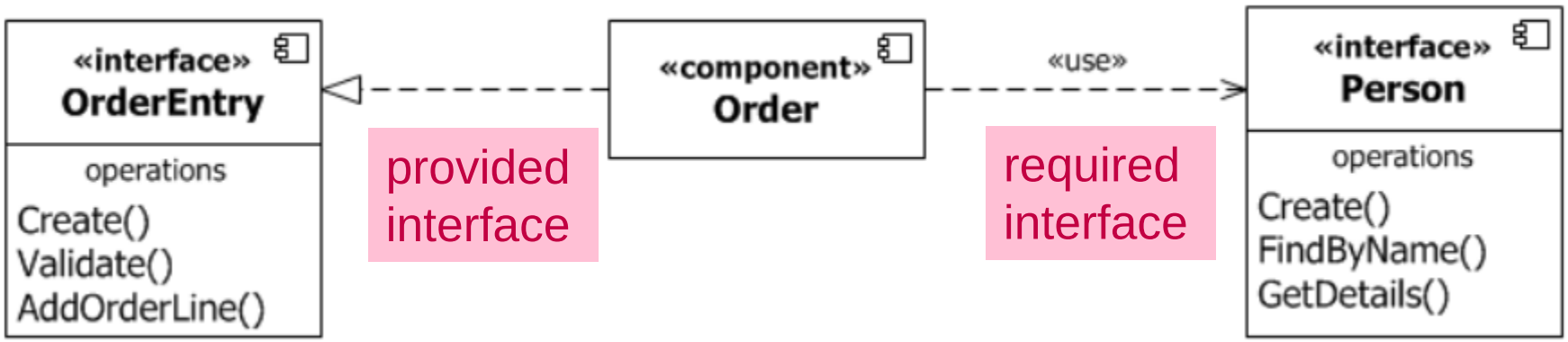
- Two views: black-box and white-box
  - **Black-box** view: interfaces provided and required only



Component diagrams

- **Component**: a modular unit with well-defined interfaces that is replaceable within its environment (UML Superstructure Specification, v.2.0, Chapter 8)
  - fosters reuse
  - stresses interfaces

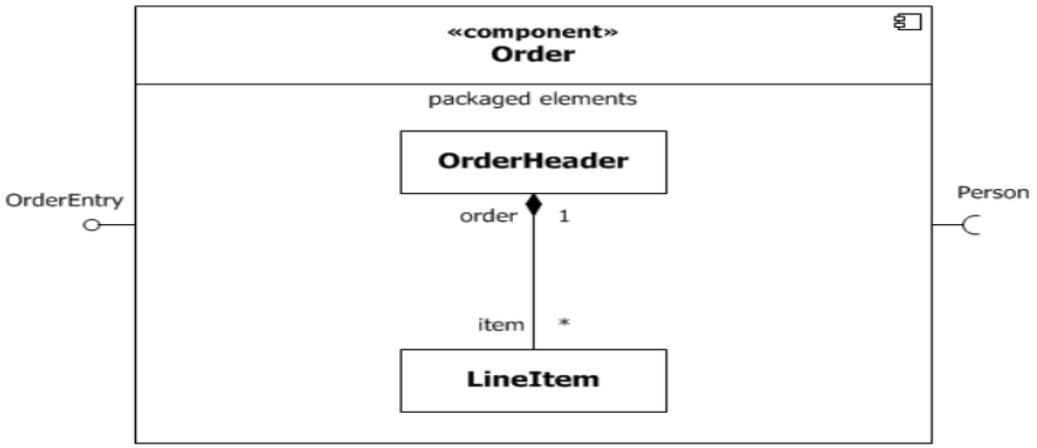
- Two views: black-box and white-box
  - **Black-box** view: interfaces provided and required only
  - **White-box** view: *structure of interfaces* and/or internal structure



Component diagrams

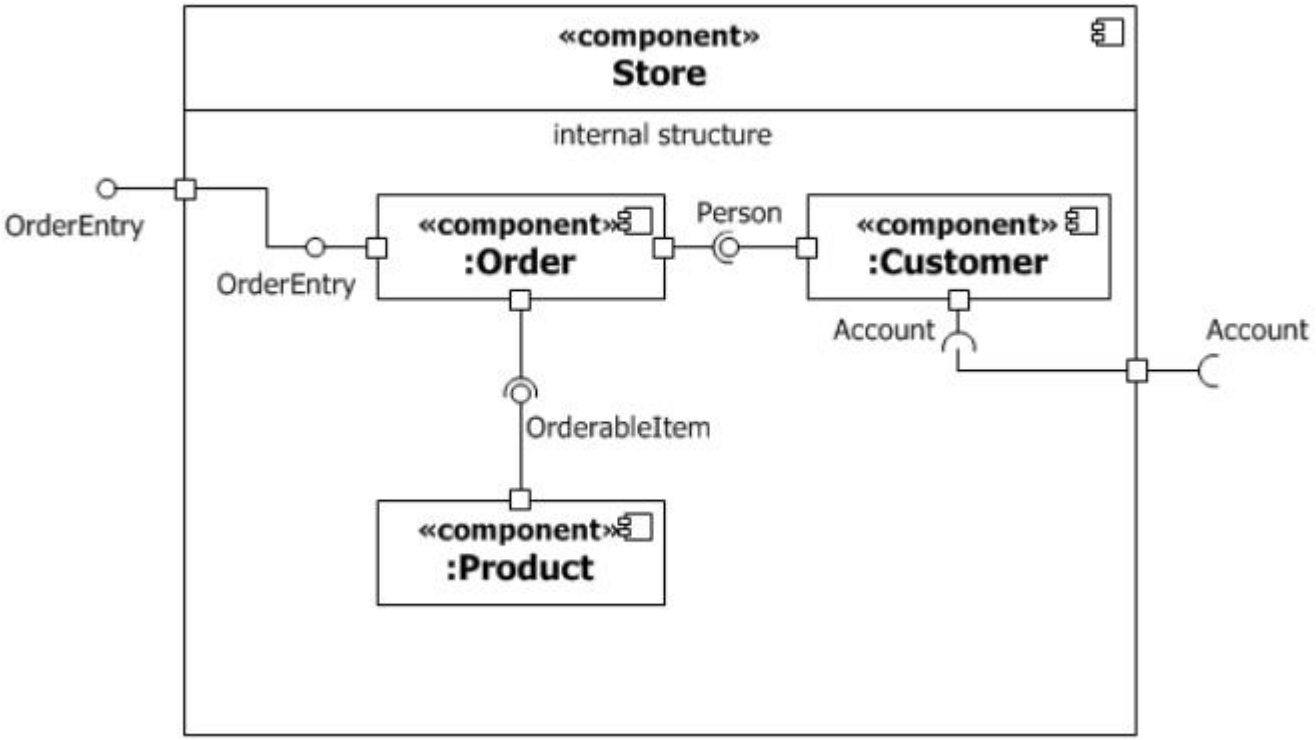
- **Component**: a modular unit with well-defined interfaces that is replaceable within its environment (UML Superstructure Specification, v.2.0, Chapter 8)
  - fosters reuse
  - stresses interfaces

- Two views: black-box and white-box
  - **Black-box** view: interfaces provided and required only
  - **White-box** view: structure of interfaces and/or *internal structure*

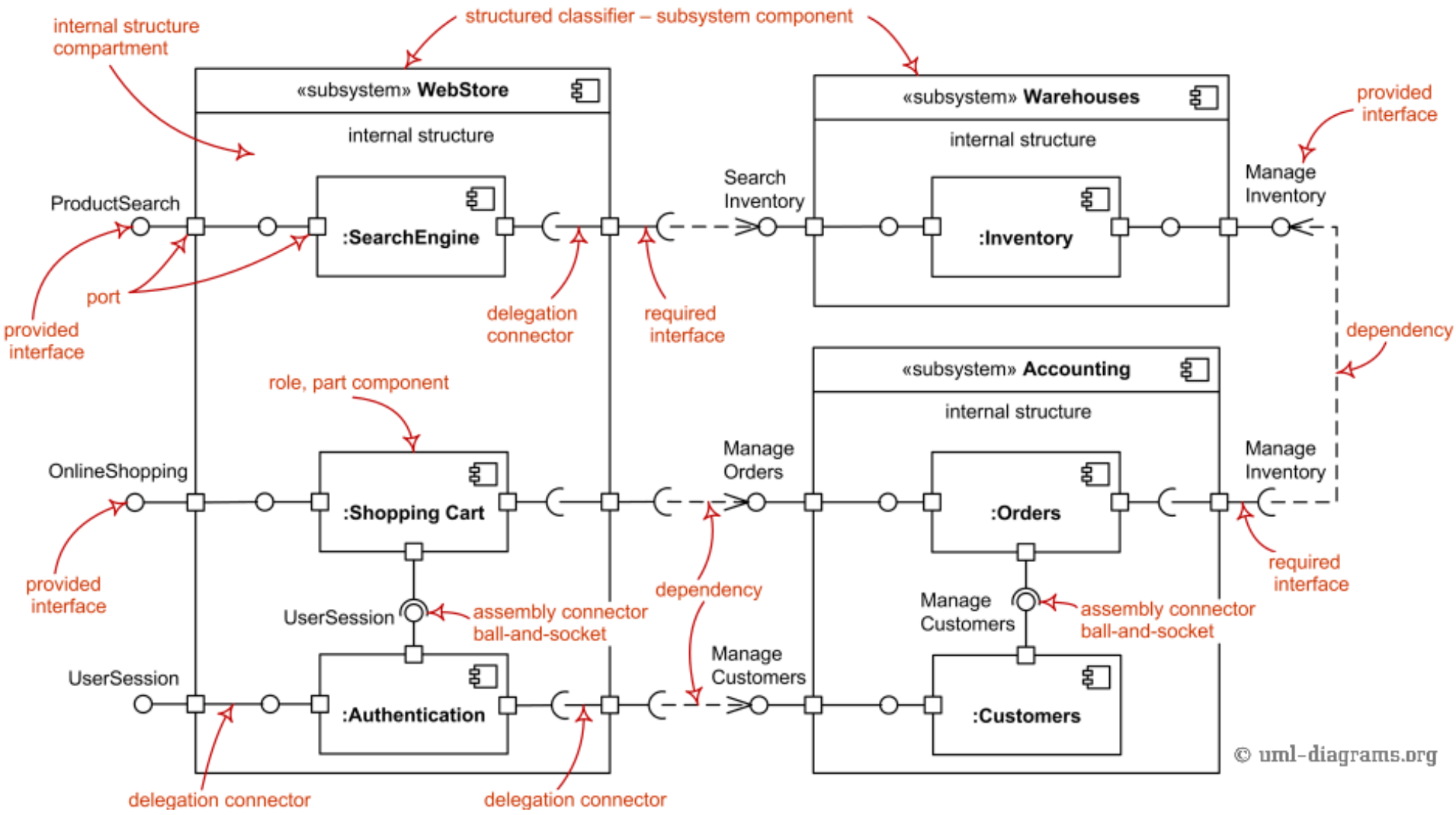


Nested components

- Components can be **contained** in other components
- Interfaces can then be **delegated** through **ports**



Summary: UML component diagrams



http://www.uml-diagrams.org/component-diagrams.html

UML structure diagrams

Class diagram

Object diagram

Packages diagram

Component diagram

Deployment diagram TODAY

Composite structure diagram

Deployment

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

Deployment

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

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

Deployment

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

- **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

Deployment

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

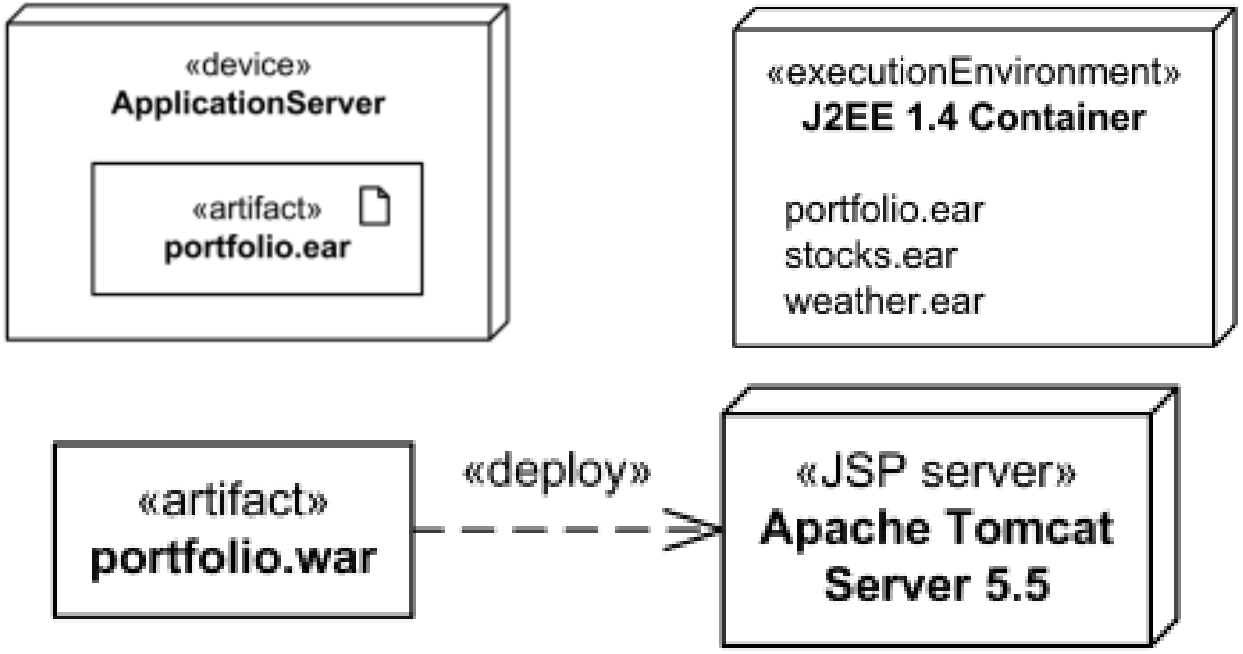
- **Artefacts**: information items produced during software development or when operating the system
  - model files, source files, scripts, executable files, database tables, word-processing documents, mail messages, …
  - Graphical representation: “class-like”

- Relations: dependencies

Deployment

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

- Deployment: three equally valid representations



http://www.uml-diagrams.org/deployment-diagrams.html

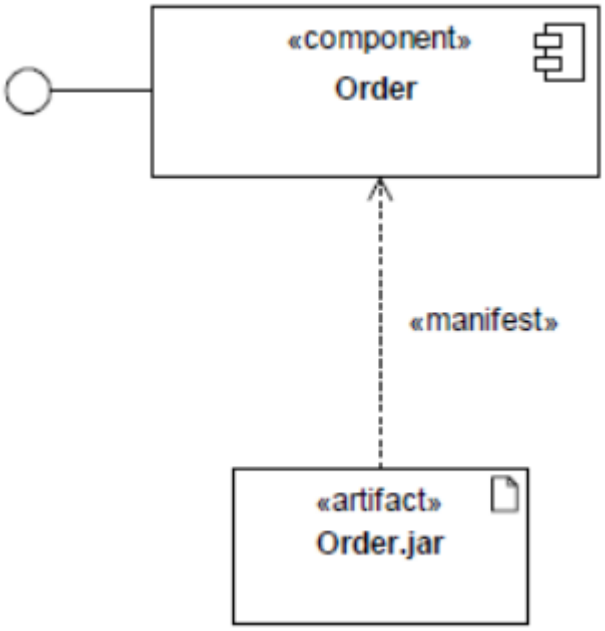
Deployment: missing piece

- 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

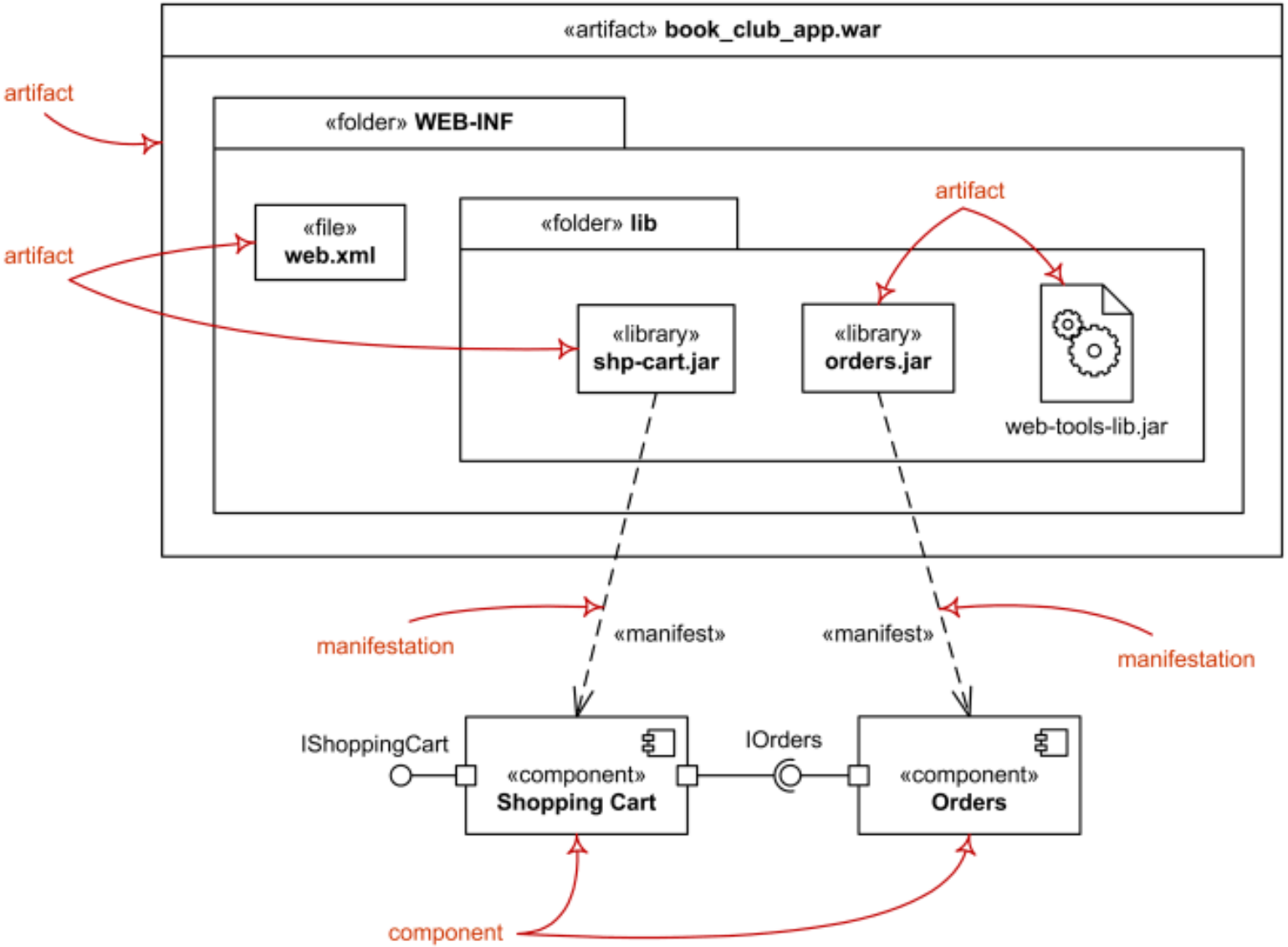
Deployment: missing piece

- 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



Summary: deployment diagrams

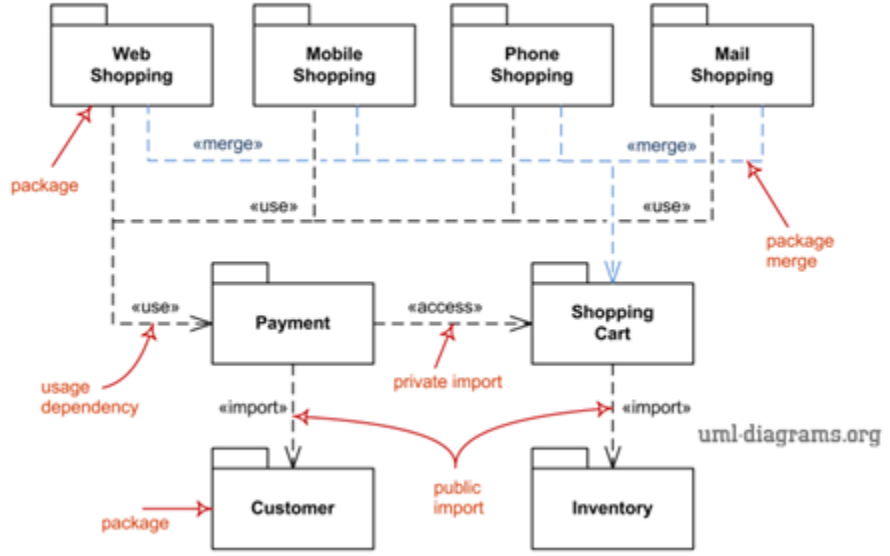


http://www.uml-diagrams.org/deployment-diagrams-overview.html

UML structure diagrams

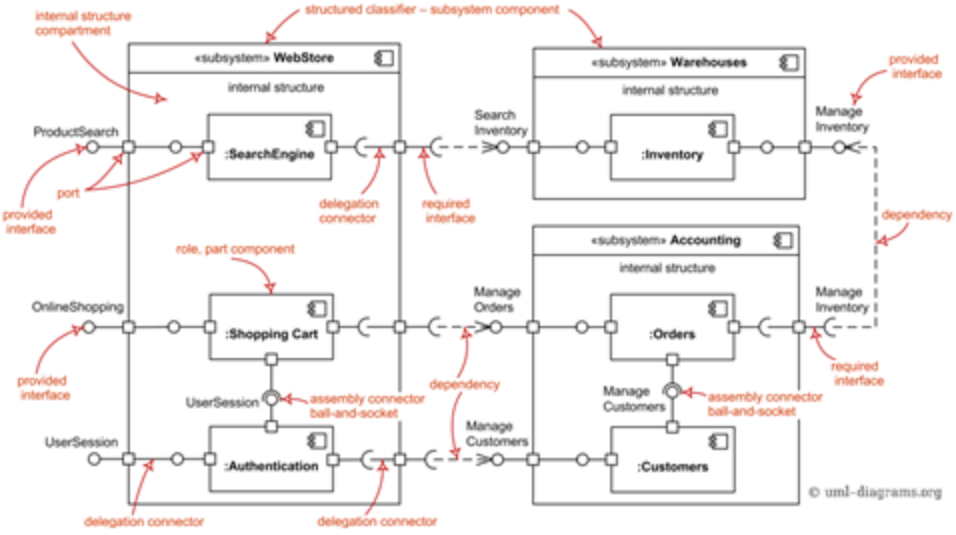
- Class diagram ✓
- Object diagram
- Packages diagram ✓
- Component diagram ✓
- Deployment diagram ✓
- Composite structure diagram

Summary: UML package diagrams



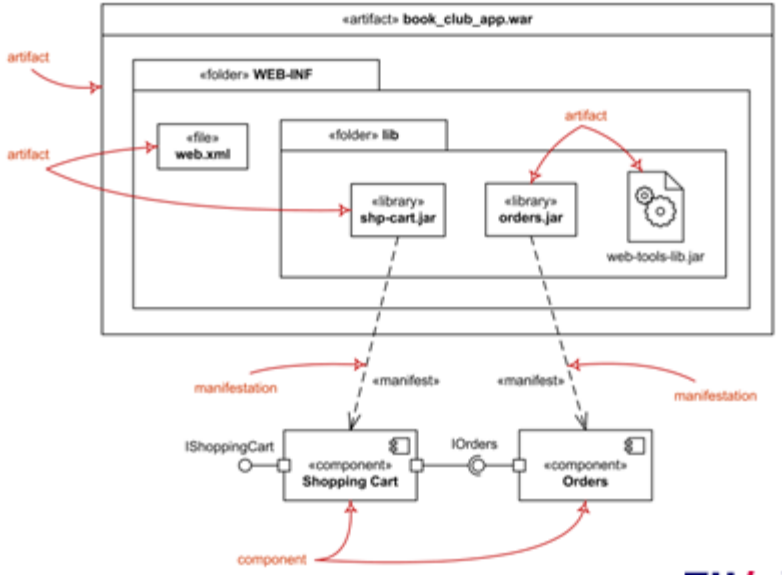
http://www.uml-diagrams.org/package-diagrams-overview.html

Summary: UML component diagrams



http://www.uml-diagrams.org/component-diagrams.html

Summary: deployment diagrams



http://www.uml-diagrams.org/deployment-diagrams-overview.html