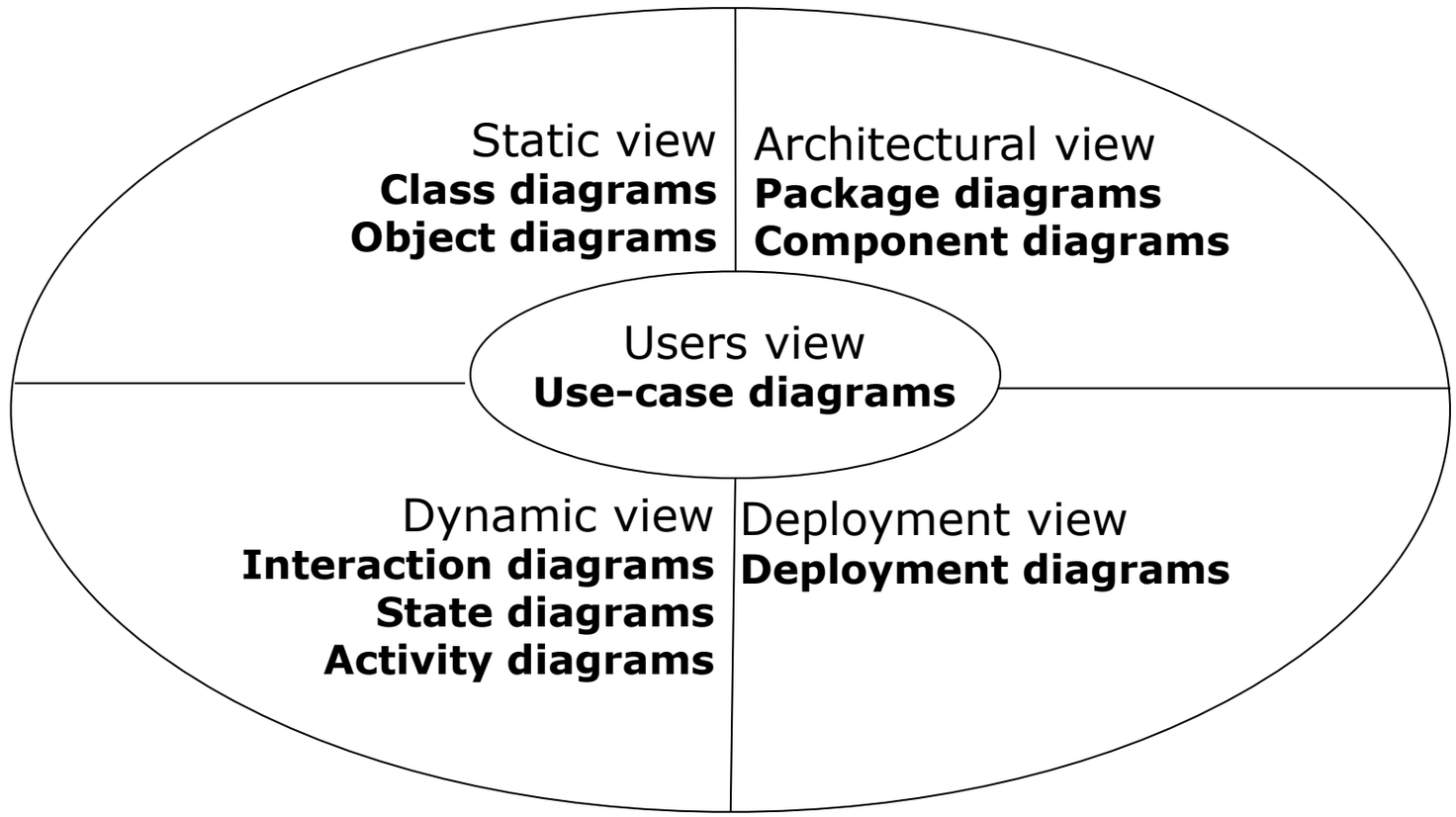
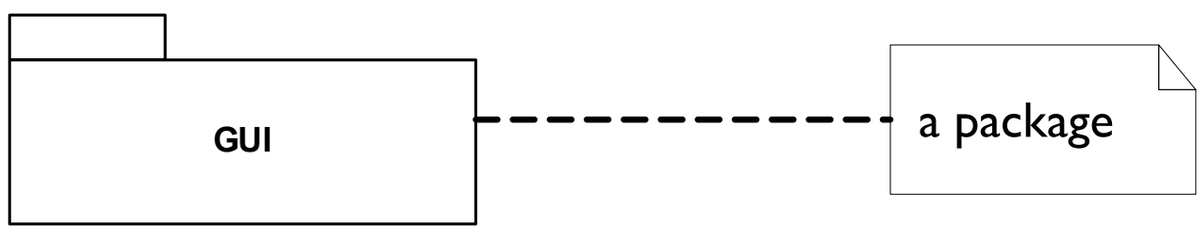
Architectural & Deployment modelling

- Package diagrams
- Component diagrams
- Deployment diagrams

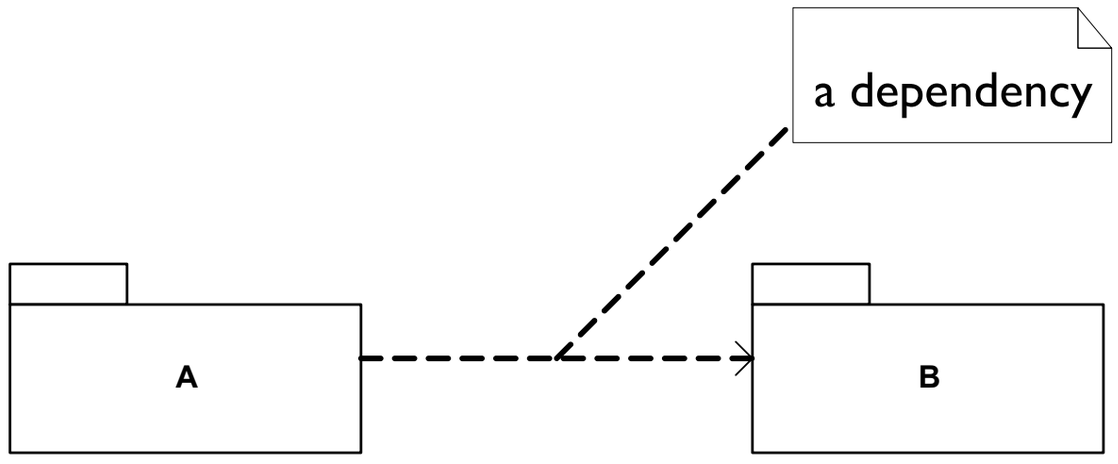


Package diagrams

- A package allows to group related elements
  - Several related classes are grouped together into a package
  - Several related packages are grouped into another package
- Notation

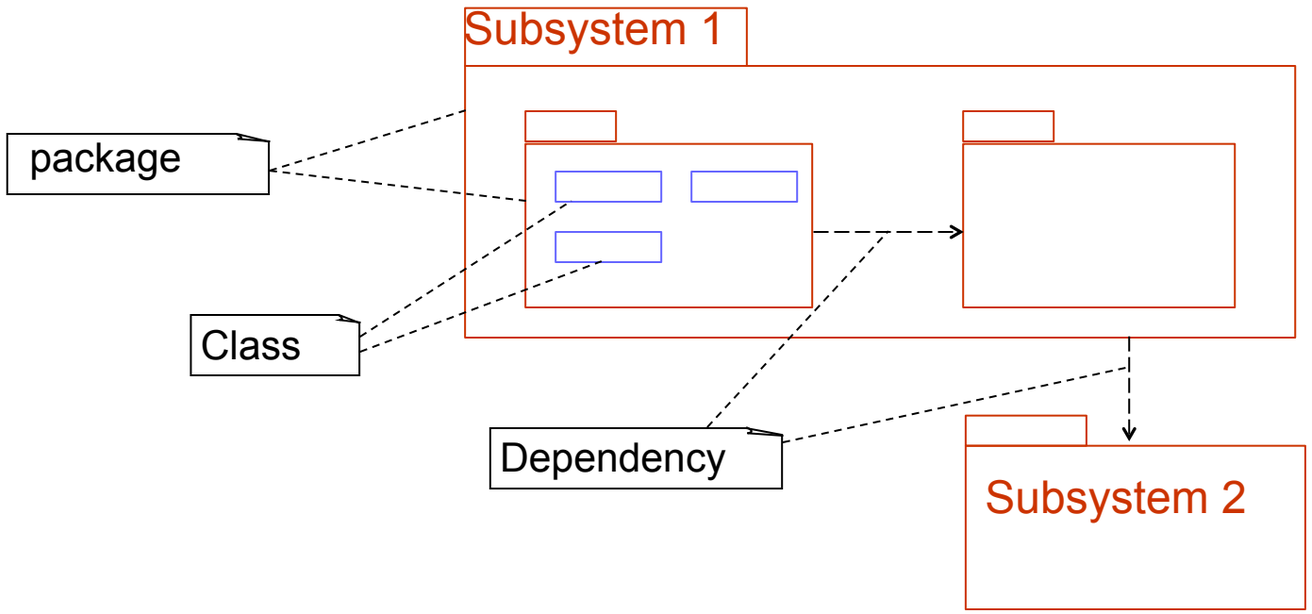


- Dependency
  - A package may depend on another package
    - For example, a package refers to an element of another package
  - Notation



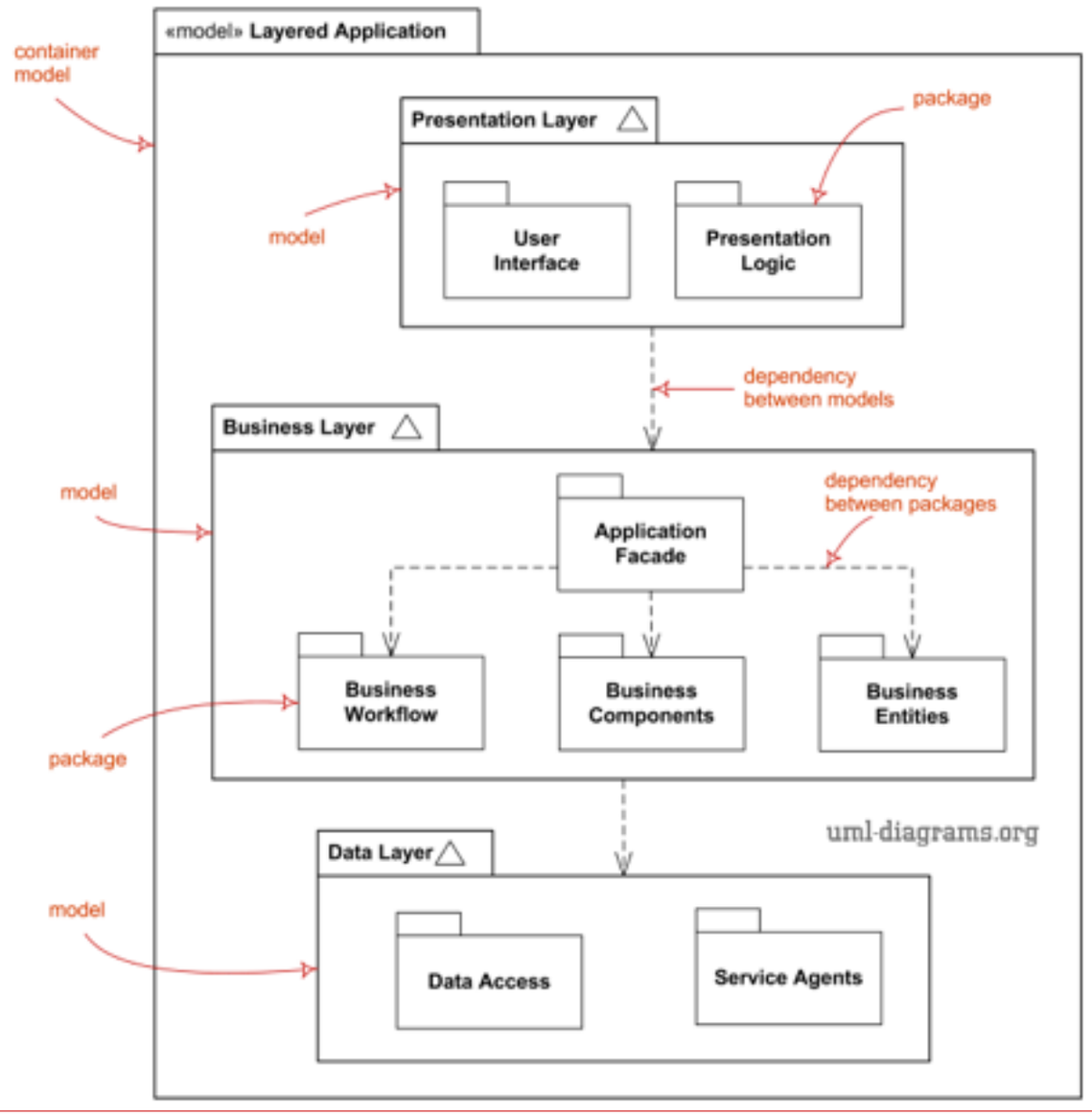
Package diagrams

- Example



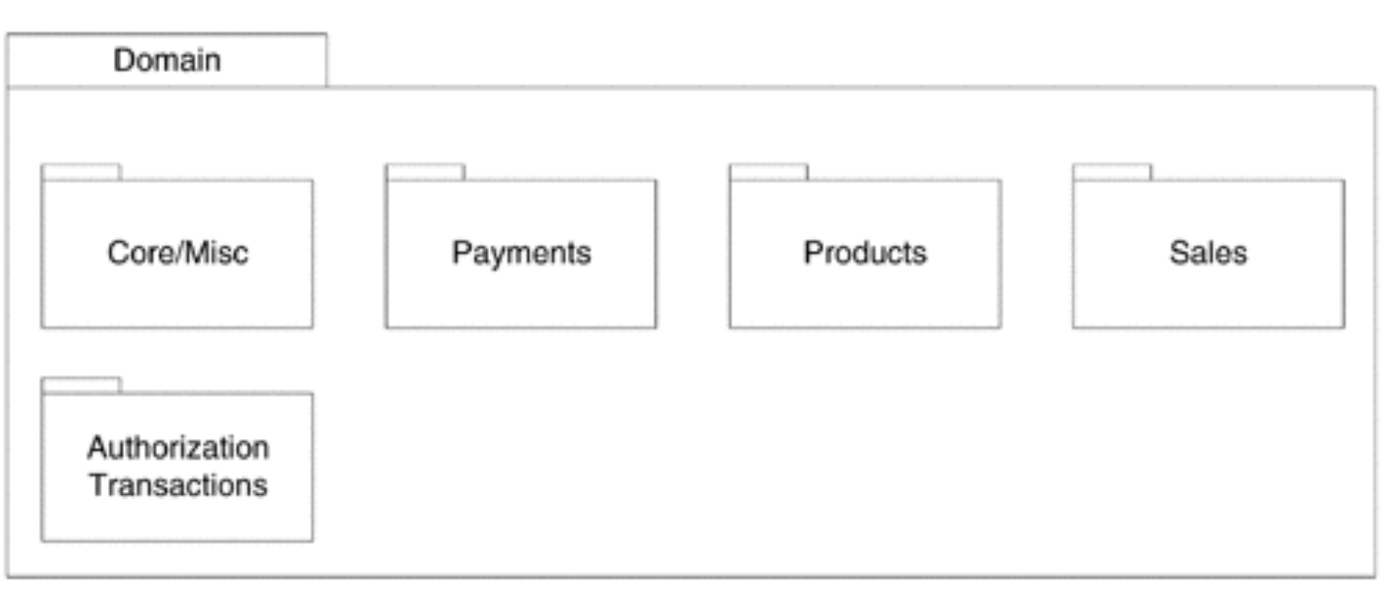
Package diagram

- Why packages?
  - Easy to manage, understand and manipulate
  - Reduce complexity
  - Iterative development: different developers, teams work simultaneously on different packages
- Example

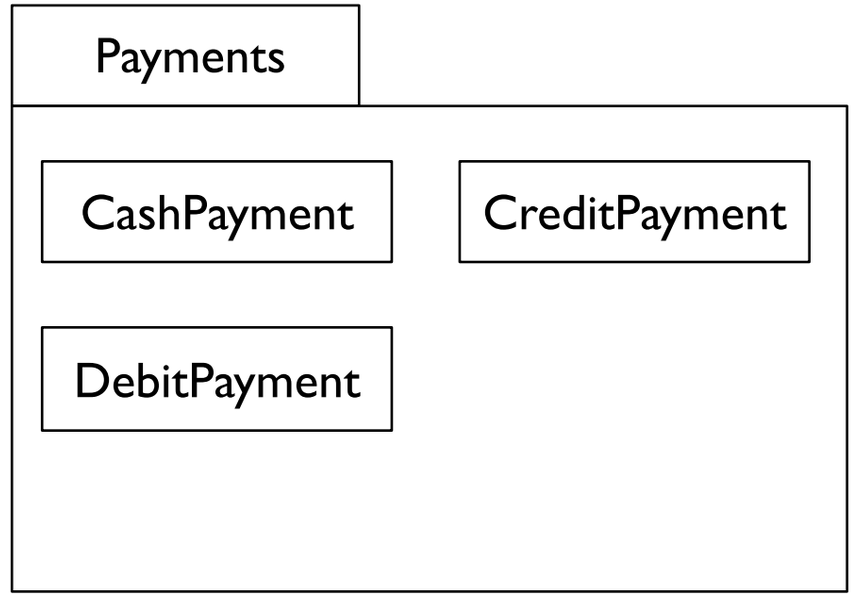


Package diagrams

- Organizing principles of packages
  - **Functional cohesion**
    - Classes/interfaces that are grouped are strongly associated in terms of purpose, service, collaboration, function
      - Example: all elements of the “payment” package are related to the payment of the products



“Domain” package



“Payments” package

Package diagrams

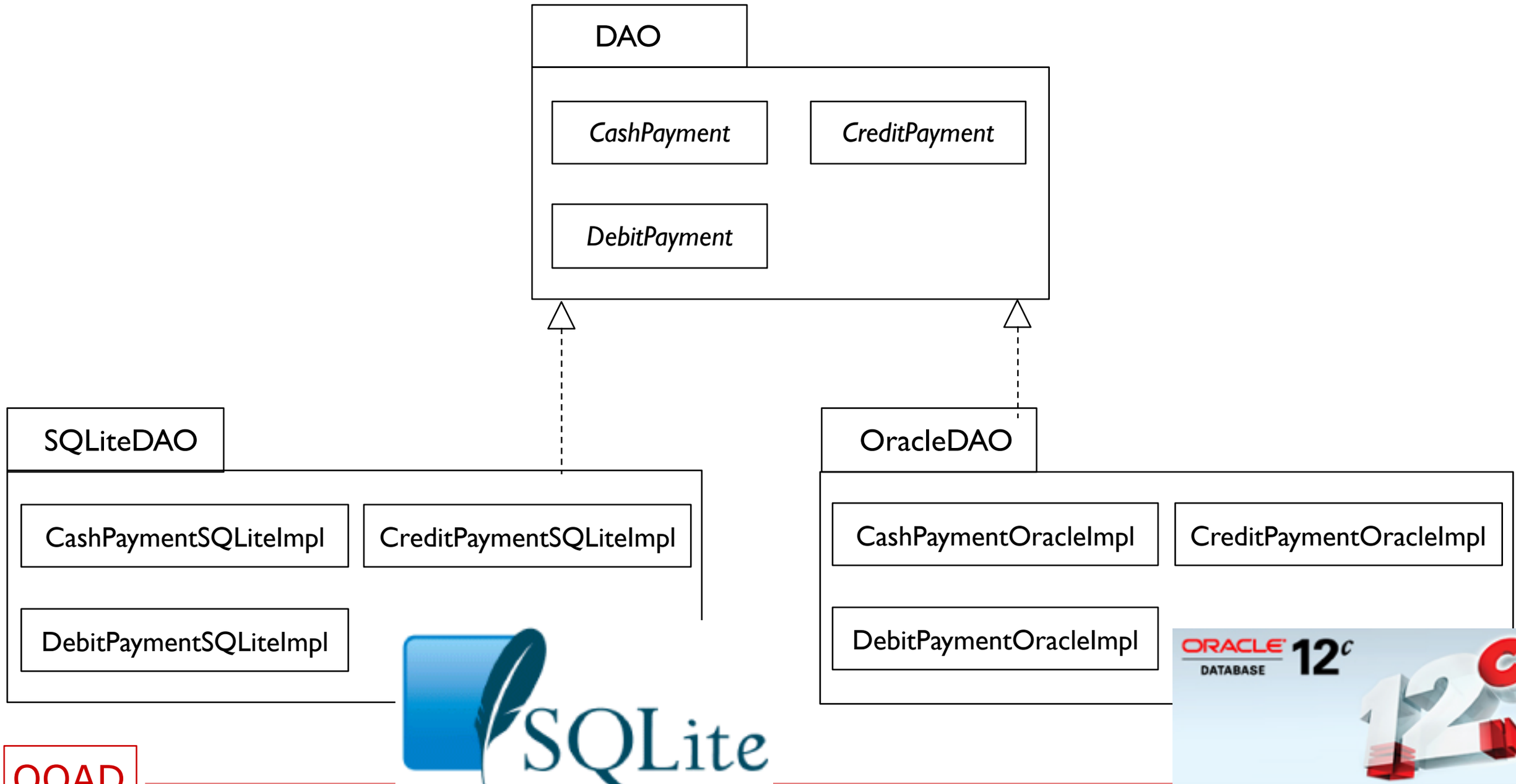
- Organizing principles of packages
  - Functional cohesion
    - **The cohesion of a package is quantified by**

$$C = \frac{\text{Number Of Internal Relationships}}{\text{Number Of Elements}}$$

      - A small value of C can say that
        - The package contains too many non-related items, is not well organised
        - The package contains no-related items with the intent of the designer
          - for example a “tools” packages
        - The package contains a subset of highly cohesive elements, but the whole is not cohesive

Package diagrams

- Organizing principles of packages (continued)
  - **Package of interface**
    - Related interfaces are placed in a package
    - The implementation classes are separated

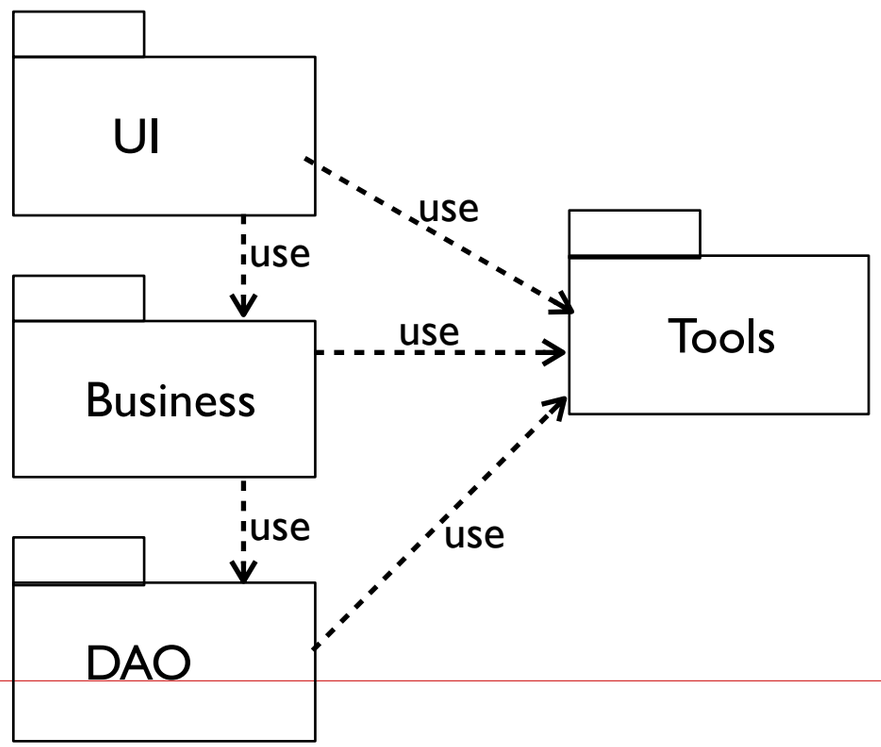


Package diagrams

- Organizing principles of packages (continued)
  - **Package of unstable elements**
    - Stable elements are grouped in a package
    - Unstable elements are grouped together in a package
      - These are items that are often modified and redistributed
      - To reduce the impact on the stable elements
      - Example: a package includes twenty classes of which ten are often modified and redistributed. It is best to separate them into two packages: one includes ten stable classes, the other is unstable.

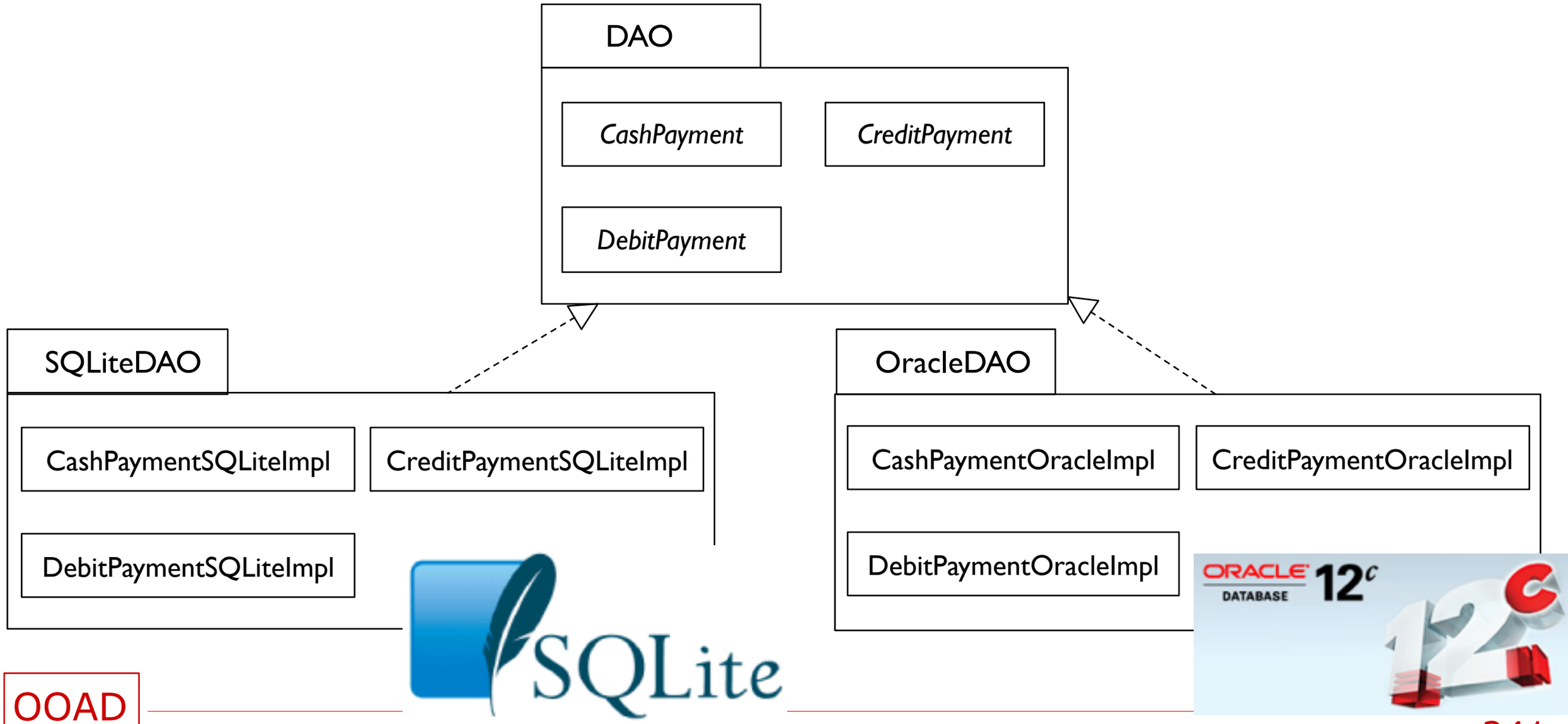
Package diagrams

- Organizing principles of packages (continued)
  - The more dependent the more stable
    - The more dependent a package has, the more stable it should be
      - Since changes on these packages has big impact on other packages
      - For example, a “tools” package needs to be stable.
    - Some ways to improve the stability of a package
      - It contains only interfaces or abstract classes
      - It doesn’t depend on other packages, or depends only on very stable packages
      - It contains stable code (well implemented)



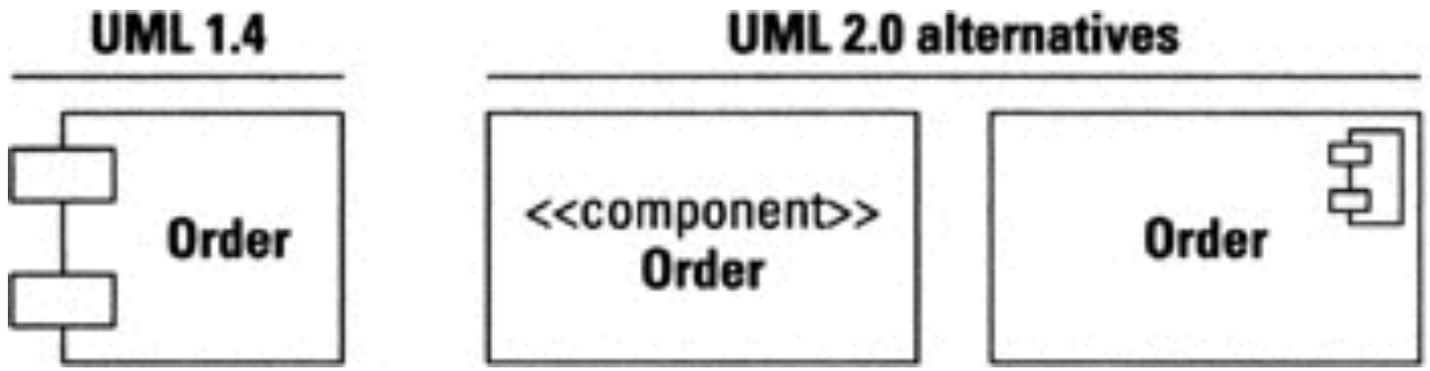
Package diagrams

- Organizing principles of packages (continued)
  - **Package of independent elements**
    - The elements that are used independently or in different contexts are separated into different packages
    - Example: SQLLite & Oracle for development/testing & production environments respectively



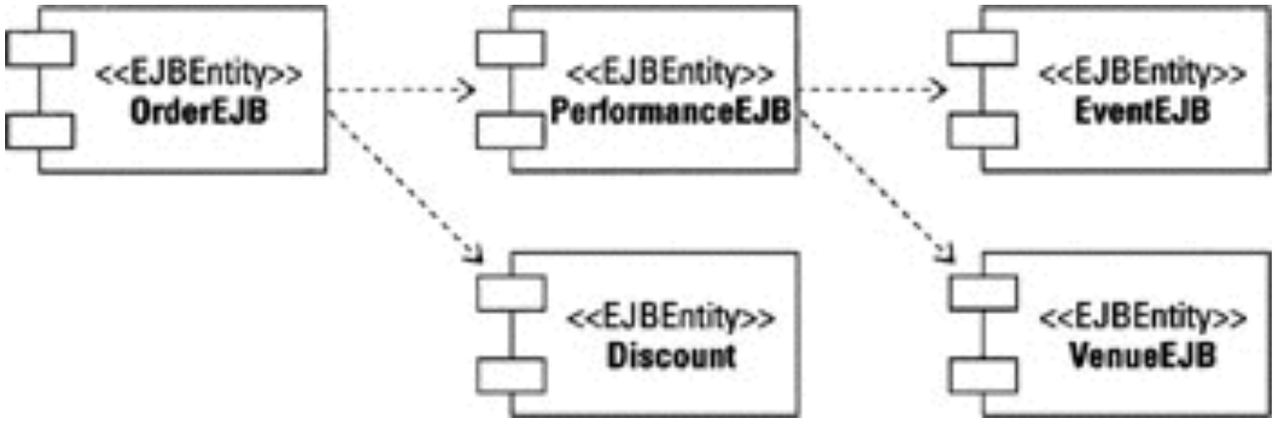
Component diagrams

- A component diagram models the **architectural view** of the system, the physical implementation of the system
  - Classes describe the logical organization, while *Components* describe the physical implementations
- Component diagrams define physical software modules and their relationships to one another
- *Components* may represent anything from a single class to applications, subsystems, and systems
- *Artifacts* that *implement* components
- The artifacts may be any type of code that can reside in any type of memory-source code, binary files, scripts, executable files, databases, or applications
- *Dependencies* represent the types of relationships that exist between components on a Component diagram
- Notation

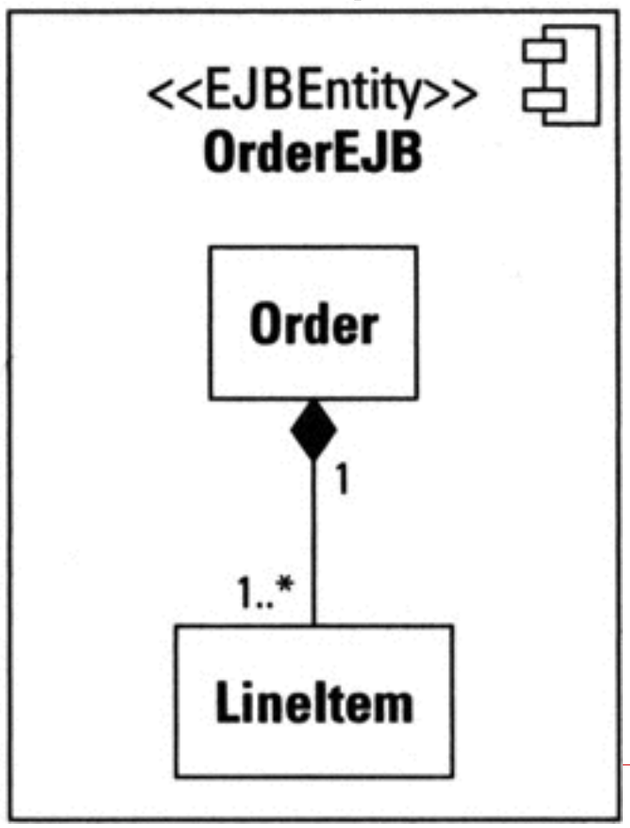


Component diagrams

- Modeling dependencies

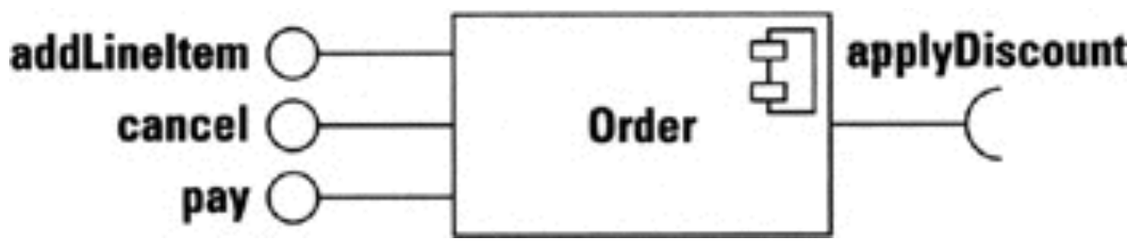


- A component is an abstraction, a representation of a requirement for the physical software.
- *Realization* refers to any implementation of a requirement.
- Component can contain the realizations (implementations)

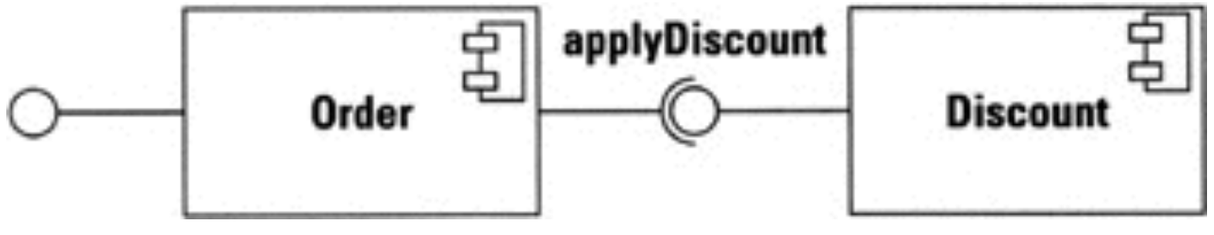


Component diagrams

- A fundamental feature of components is the ability to define interfaces
- Interfaces come in two types: *required* and *provided*
  - *Provided interface*: defines how another component must ask for access to a provided service
  - *Required interface:* defines an interface exactly what it needs
- Example: Order component *provides* the services "add line items to the order", "cancel the order", and "pay for the order"; and *requires* discount (from other component)

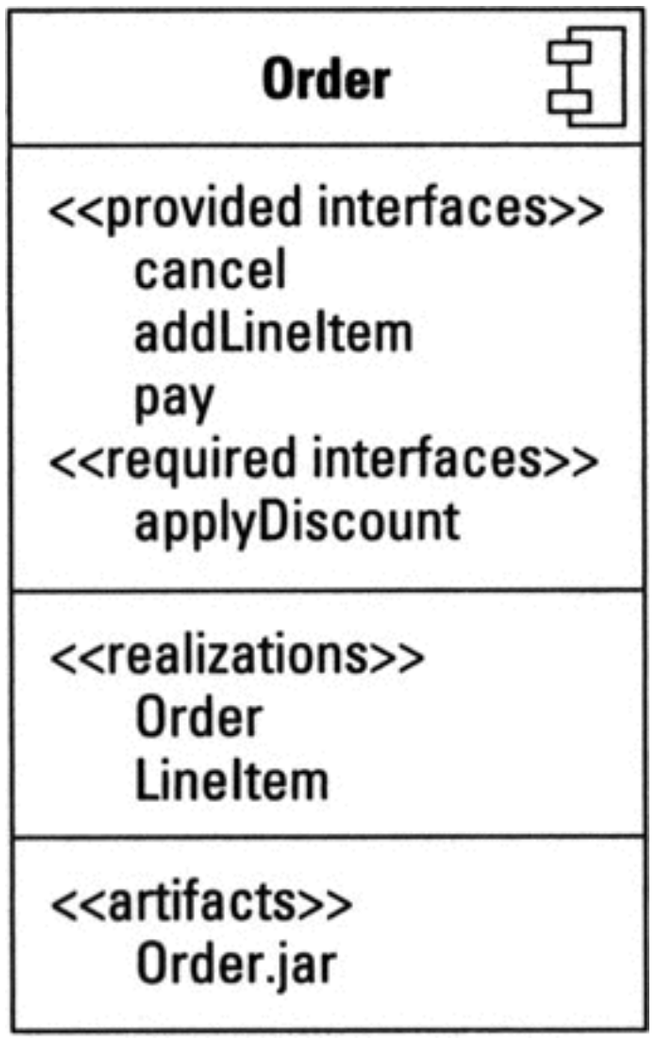


- Connecting required and provided interfaces to form a partnership between components



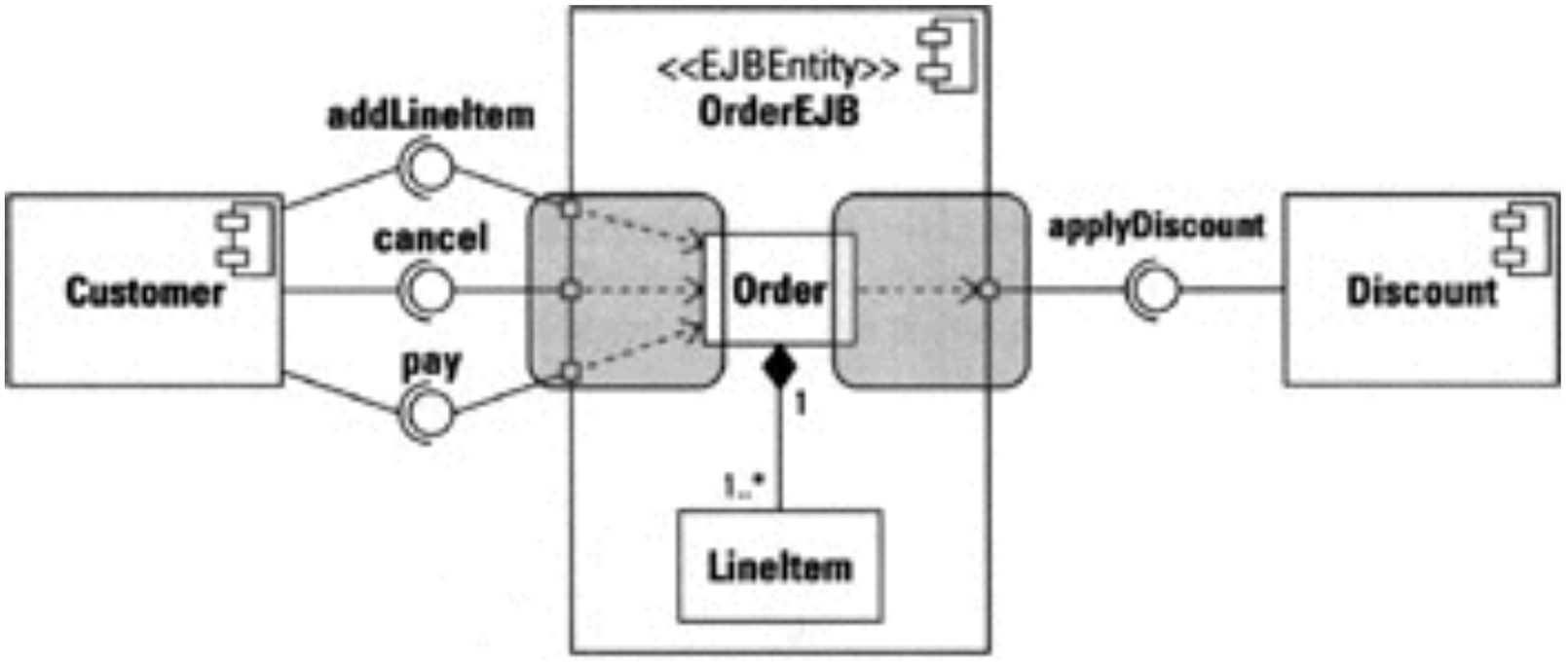
Component diagrams

- UML 2.0 provides a way to represent all of the information defined so far for a component, including interfaces, realizations, and artifacts
  - white box view



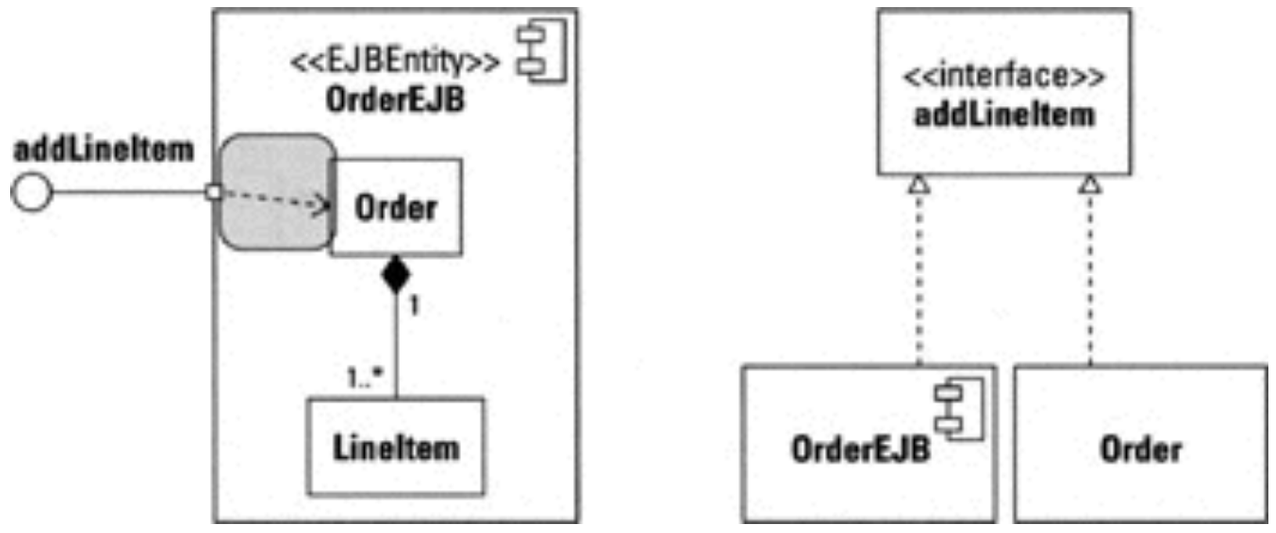
Components diagrams

- **Ports** map the interfaces of the component to the realizations that support them
- A port appears as a small square in the edge of the component

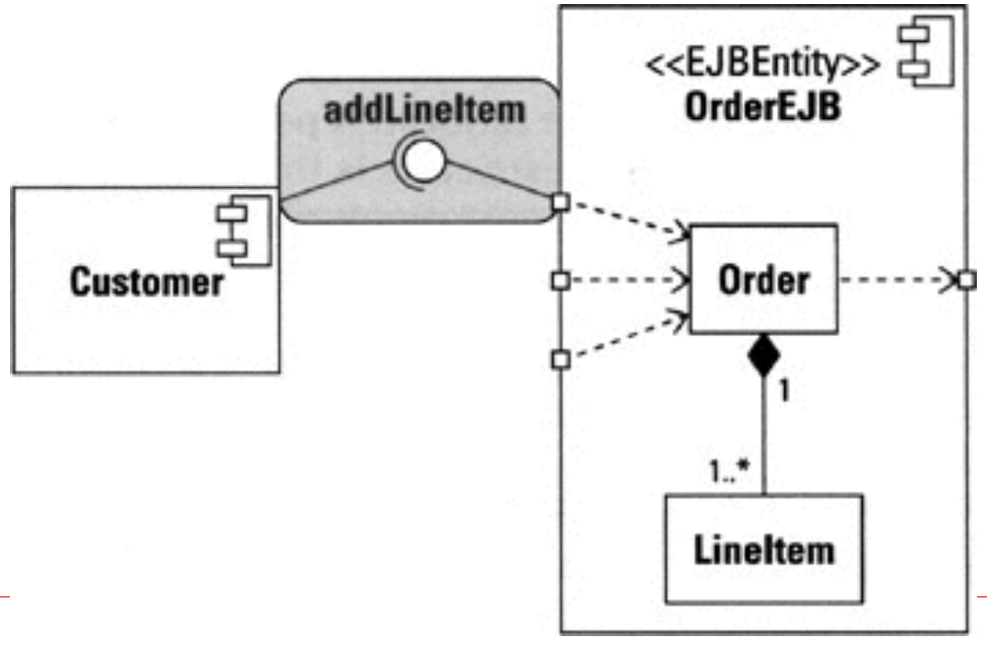


Component diagrams

- A **connector** is a **link** that enables communication between two or more components
  - Two types of connection: delegation connector & assembly connector
- A *delegation connector* maps a request from one port to either another port or to a realization that provides the implementation for the request

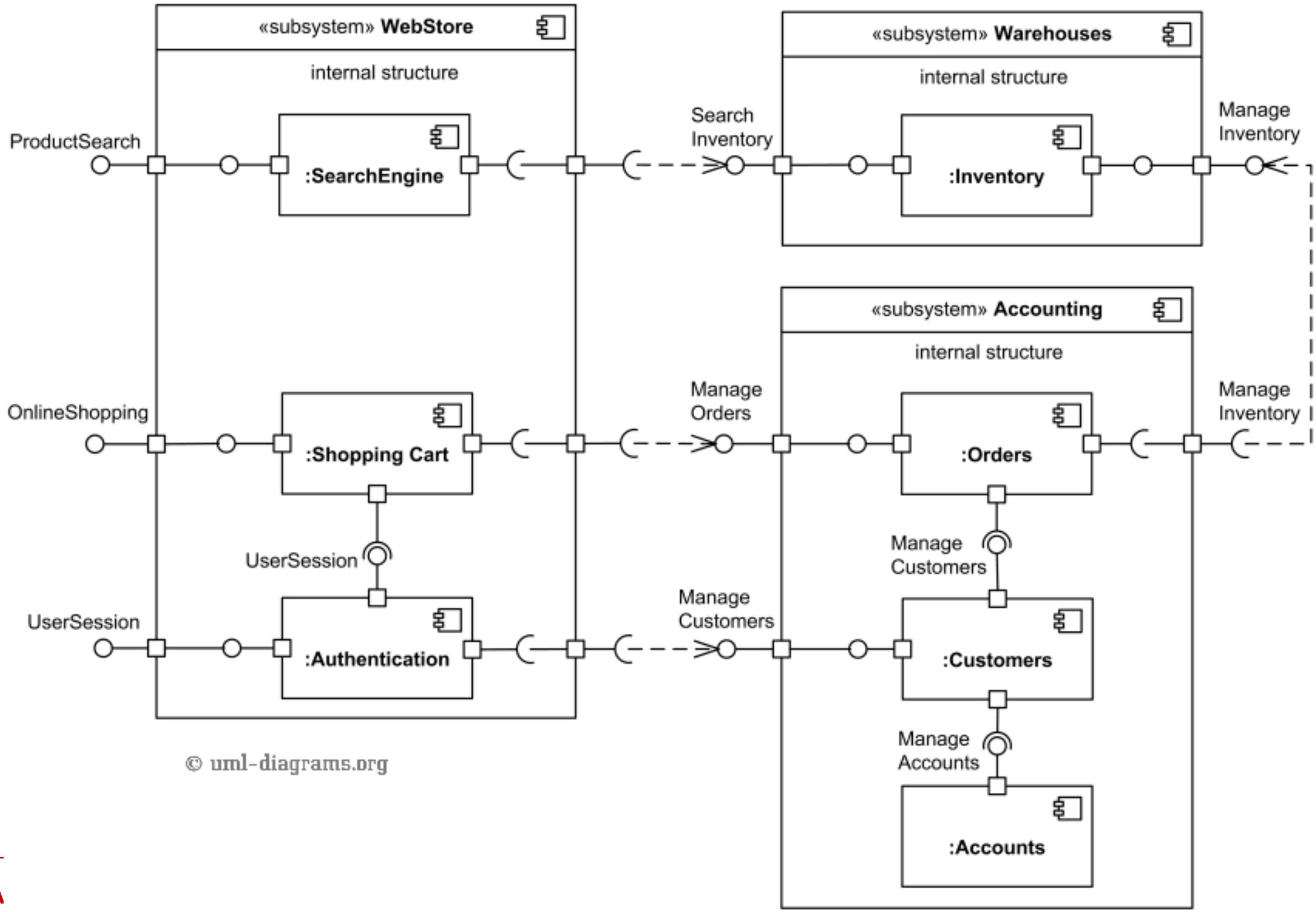


- An *assembly connector* is used to map a required interface to a provided interface



Component diagrams

- Example: Online shopping example with three related subsystems - Webstore, Warehouses and Accounting

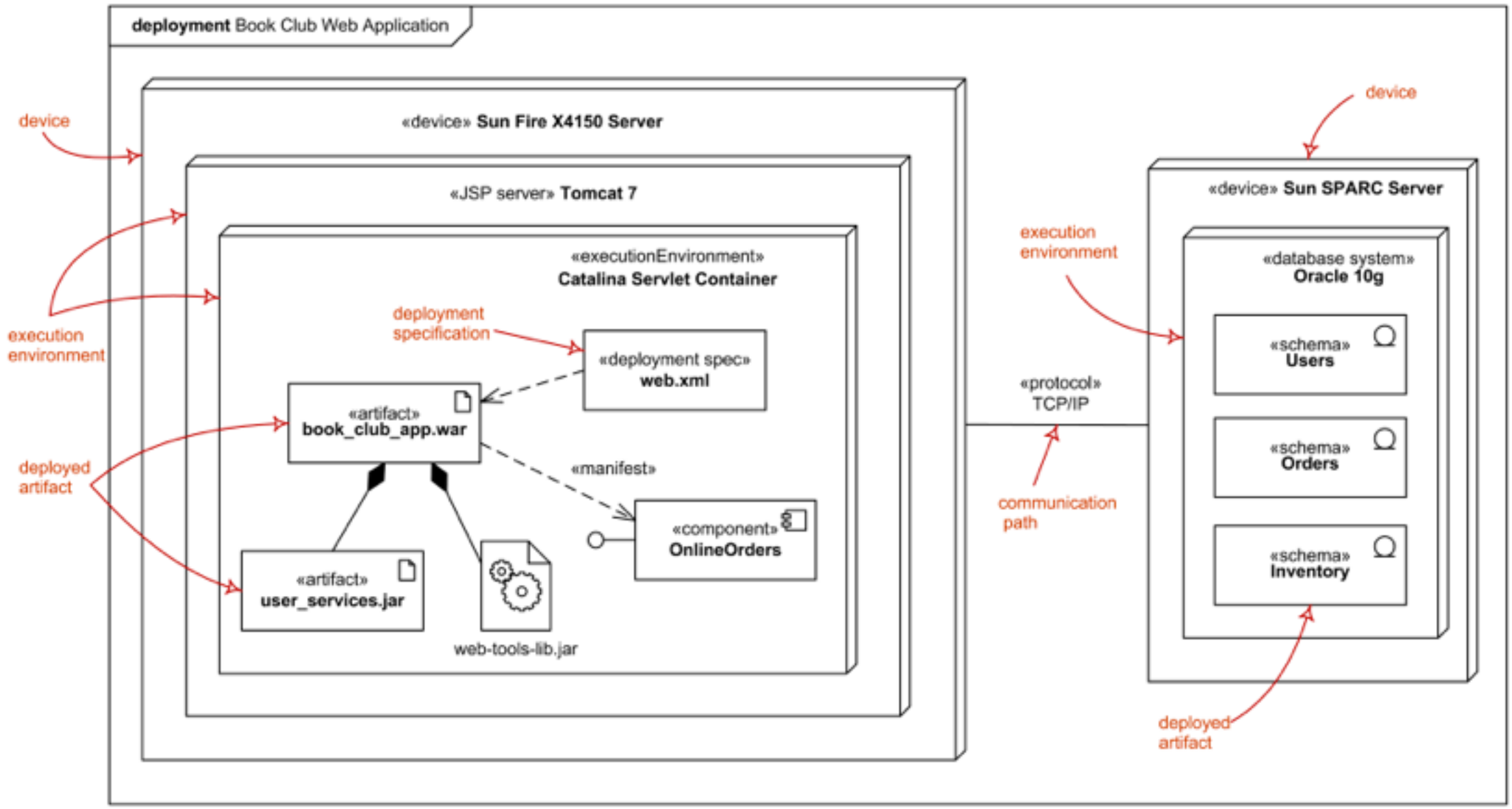


Deployment diagrams

- **Deployment diagram** shows the architectural of a system as deployment (distribution) of software artifacts to deployment targets
  - **Artifacts** represent concrete elements in physical world such as executable files, libraries, archives, database schemas, configuration files, …
  - Deployment target is usually represented by a **node** which is either hardware device or software execution environment. **Nodes** could be connected
- Deployments diagrams could describe architecture at **specification level** (also call type level) or at **instance level** (similar to class diagrams and object diagrams).

Deployment diagrams

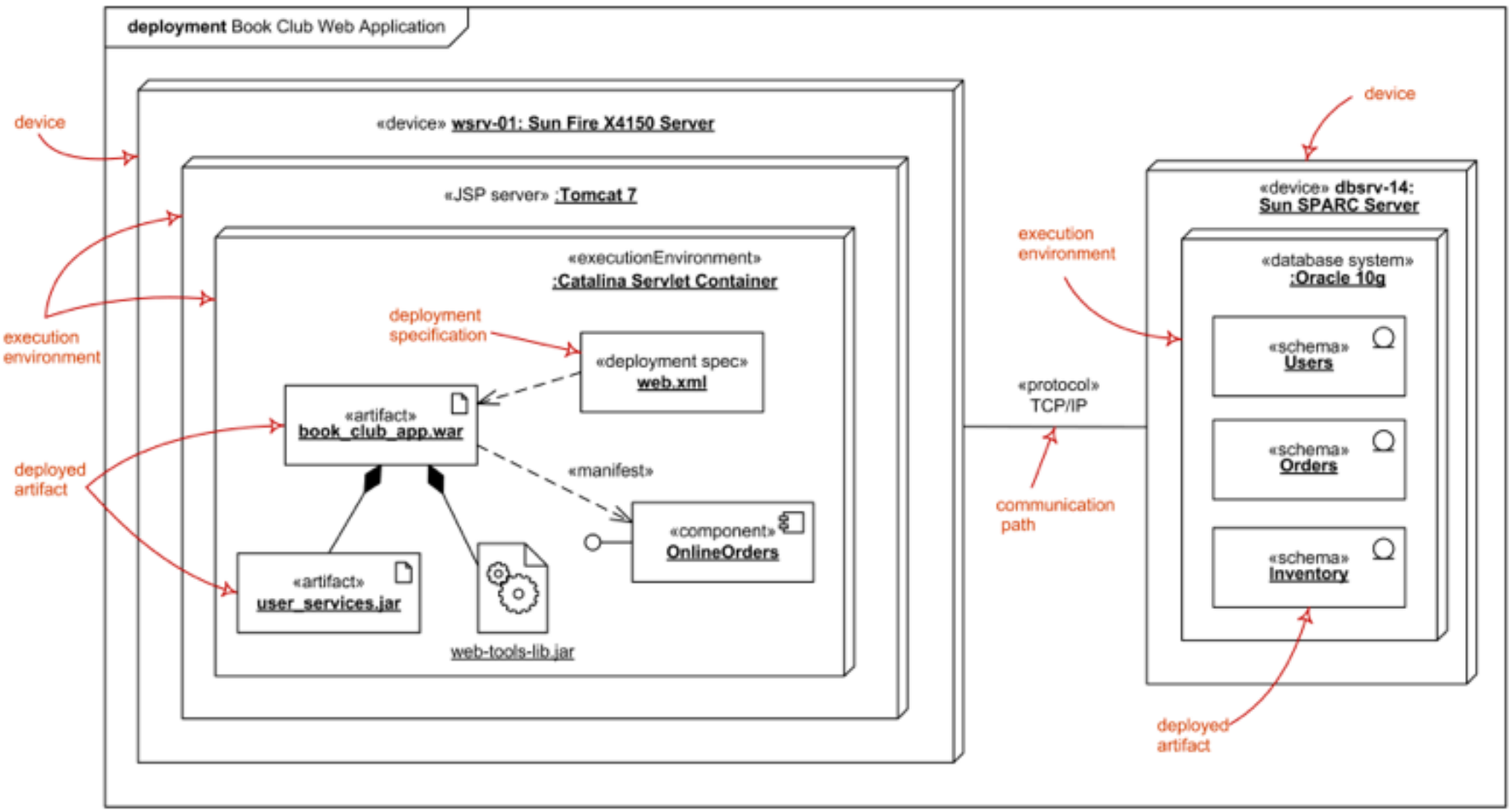
- **Specification level** (also call type level) deployment diagram shows some overview of deployment of artefacts to deployment targets, without referencing specific instances of artefacts or nodes.



Specification level deployment diagram -
web application deployed to Tomcat JSP server and database schemas - to database system

Deployment diagrams

- **Instance level** deployment diagram shows deployment of instances of artefacts to specific instances of deployment targets. It could be useful for example to show the differences in deployments to development, staging or production environments with the names/ids specific to deployment services or devices



Instance level deployment diagram - web application deployed to Tomcat JSP server and database shemas - to database system