Object Oriented Analysis & Design

Engr. Abdul-Rahman Mahmood
DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com	alphasecure@gmail.com
alphapeeler.sf.net/pubkeys/pkey.htm	http://alphapeeler.sourceforge.net
pk.linkedin.com/in/armahmood	http://alphapeeler.tumblr.com
www.twitter.com/alphapeeler	armahmood786@jabber.org
www.facebook.com/alphapeeler	alphapeeler@aim.com
abdulmahmood-sss alphasecure	mahmood_cubix 48660186
armahmood786@hotmail.com	alphapeeler@icloud.com
http://alphapeeler.sf.net/me	http://alphapeeler.sf.net/acms/

Concepts
Revision
Case Study

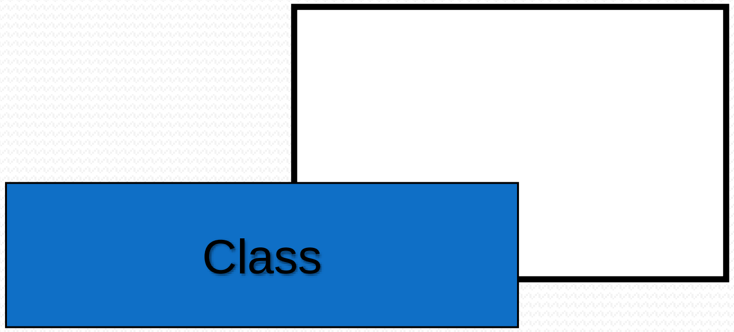
Objectives

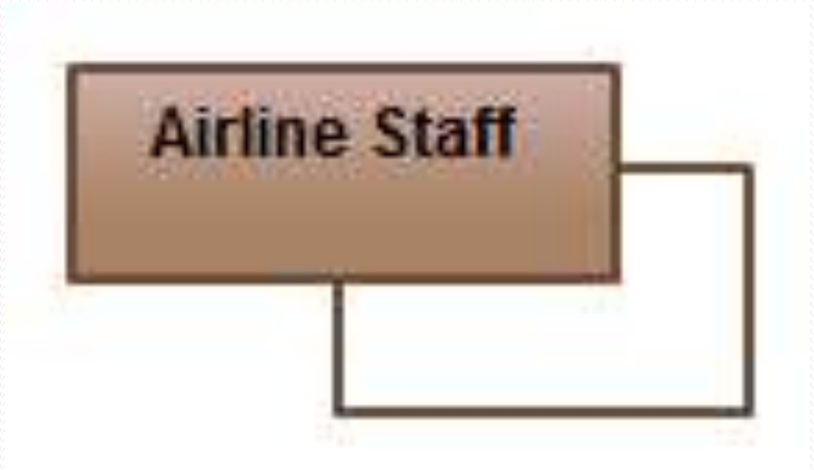
- Concept of reflexive relationship
- Concept of association class
- Homogenization of the system
- Why is homogenization required
- Changing the class model through homogenization
- Case study for homogenization
- What are components and why we need them

Concept of reflexive relationship

A class can have a *self association*.

reflexive relationship

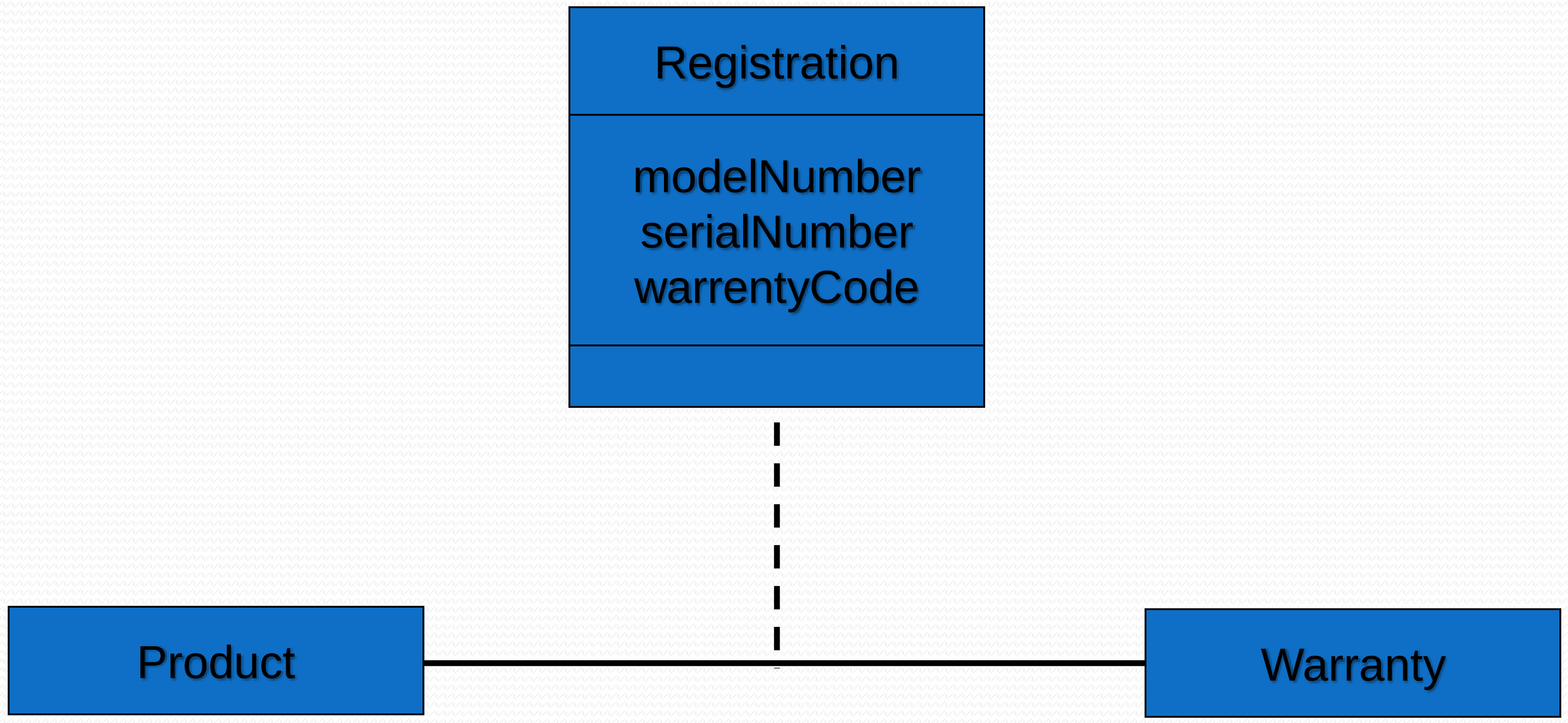




a staff working in an airport may be a pilot, aviation engineer, a ticket dispatcher, a guard, or a maintenance crew member.

Concept of association class

Associations can also be objects themselves, called *link classes* or an *association classes*.



Homogenization of the system

- In parallel design process, several stimuli with the same purpose or meaning are defined by several designers. These stimuli should be consolidated to obtain as few stimuli as possible. It is called *homogenization.*
- Homogenize - to change (something) so that its parts are the same or similar.

Why is homogenization required

- If different teams are working on different scenarios, a class may be called by different names.
- These name conflicts must be resolved.
- Homogenization is an ongoing process.

Elements of Homogenization

- **COMBINING CLASSES**
- **SPLITTING CLASSES**
- **ELIMINATING CLASSES**
- **SCENARIO WALK THROUGH**
- **EVENT TRACING**
- **DOCUMENTATION REVIEW**

Homogenization

- **COMBINING CLASSES**
- examine each class with its definition.
- Examine attributes and functions of the classes.
- Look for the use of synonyms.
- Once you determine that the 2 classes are doing the same thing, choose the name that is closest to the language used by the customer.

Homogenization

- **SPLITTING CLASSES**
  - Classes should be examined if they are following the golden rule of OO, that is A class should do one thing and it should do really well.

  - Example :
  - A StudentInformation class contains:
    - Info about student Actor
    - And info about the courses that student has successfully completed.
  - Resolve into 2 classes: StudentInfo and Transcript.

Homogenization

- **ELIMINATING CLASSES**
  - When?
    - A class does not have any structure or behavior
    - The class does not participate in any use cases
  - Guideline: Look for the control classes in particular. Lack of the sequencing responsibility may lead to the deletion of the class.
  - Example:
    - A control class is only a pass through or a forwarder. I.e., it received information from boundary class and only passes it to the entity class without the need of sequencing logic.

Homogenization

- **CONSISTENCY CHECKING**
- Why ?
  - Because static view of the system and dynamic view of the system and interaction view of the system are under development in parallel.
  - Consistency checking is necessary to check if all the 3 views of the system has the same assumptions / decisions and terminology used.
  - Consistency Checking is done by forming a small team.
  - Team members 5-6.
  - Analyst, Designer, Customer / Customer Rep., Domain experts.

Homogenization

- **SCENARIO WALK THROUGH**
- A simple method of consistency checking is walk through of **high risk scenario** as represented by **sequence / collaboration** diagrams.
- Specially check the **reflexive relationships** because they are likely to miss during analysis.
- Reflexive relationships are needed when multiple objects of the same class interacts during a single scenario.
- Finally verify that each object in sequence / collaboration diagram belongs to a class on the class diagram.

Homogenization

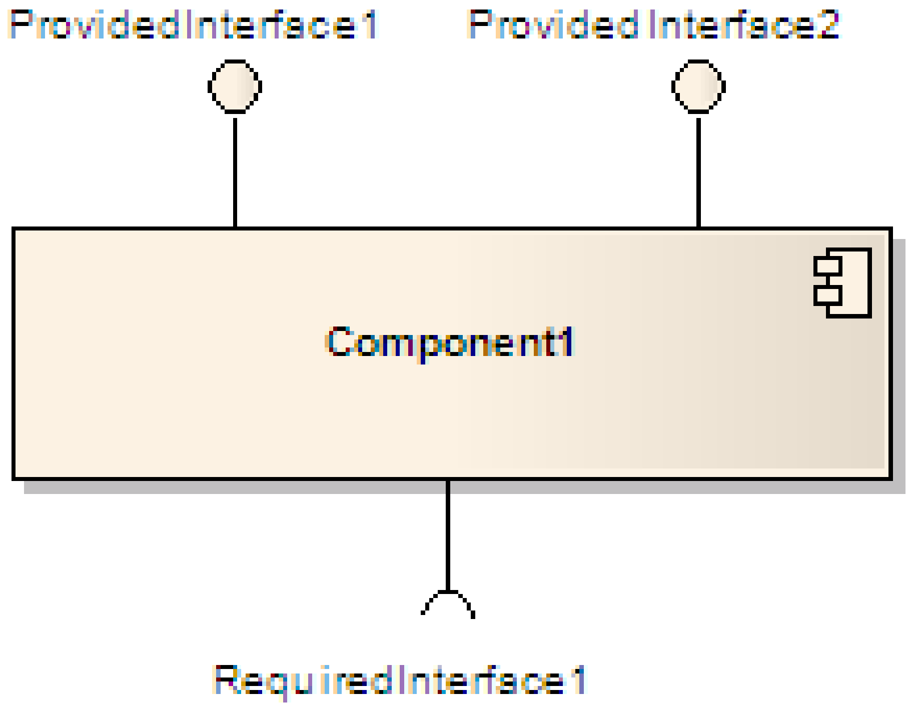
- **EVENT TRACING**
- For every message on the sequence / collaboration diagrams:
    - Verify that an operation is responsible to **send** the message in the sending class
    - Verify that an operation is responsible to **receive** the message in the receiving class and handles it properly.
    - Verify that there exists some kind of **association** between sending and receiving classes on the class diagram.
    - Verify each message in the **state transition** diagram.

Homogenization

- **DOCUMENTATION REVIEW**
  - Each class should be documented.
  - Verify that each method and attribute is defined properly.
  - Finally check for the establishment of documentation standards, format specifications, and content rules.
  - Make sure that all the standards established are followed.

Component (UML)

- A **component** in the Unified Modeling Language "represents a modular part of a system, that encapsulates its content and whose manifestation is replaceable within its environment. A component defines its behavior in terms of *provided* and *required* interfaces".[1]



Manifestation : an event, action, or object that clearly shows or embodies something abstract or theoretical.

Component (UML)

- A component may be **replaced** by another if and only if their provided and required interfaces are identical.
- This idea is the underpinning for the **plug-and-play** capability of component-based systems and promotes **software reuse**.
- UML places **no restriction on the granularity** of a component. Thus, a component may be as small as a *figures-to-words converter*, or as large as an entire *document management system*.
- Such assemblies are illustrated by means of **component diagrams**.

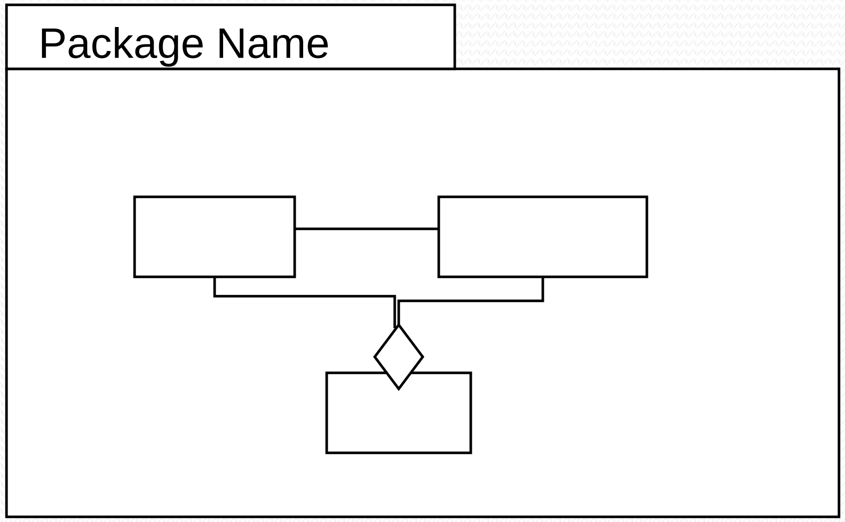
Implementation Diagrams

Implementation Diagrams

- Both are structural diagrams
- **Component Diagrams**:
  - set of **components** and their **relationships**
  - Illustrate **static** implementation view
  - Component maps to one or more **classes**, **interfaces**, or **collaborations**
- **Deployment Diagrams**:
  - Set of **nodes** and their **relationships**
  - Illustrate **static** deployment view of architecture
  - **Node** typically encloses **one or more components**

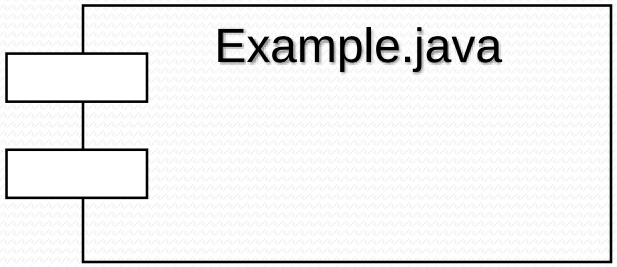
Package

- General purpose mechanism for organizing elements into groups
- Can group classes or components.



Component Diagram

- Classes
- Interfaces
- Dependency, generalization, association, and realization relationships



Special kind of class diagram focusing on system’s components.

Interfaces

- Definition:
  - Collection of operation signatures and/or attribute defns
  - Defines a cohesive set of behaviors
- Realized by:
  - Implemented by classes and components
  - Implement operations/attributes defined by interface
- Relationships:
  - A class can implement one or more interfaces
  - An interface can be implemented by 1 or more classes
- Notation:
  - Lollipop
  - Dashed arrow

UML Component Diagrams

- **Component diagram** shows components, provided and required interfaces, ports, and relationships between them. This type of diagrams is used in **Component-Based Development** (**CBD**) to describe systems with **Service-Oriented Architecture** (**SOA**).
- Component-based development is based on assumptions that previously constructed components could be **reused** and that components could be **replaced** by some other "equivalent" or "conformant" components, if needed.

UML Component Diagrams

- Component are intended to be capable of being <u>**deployed**</u> and <u>**re-deployed**</u> ***independently***, for instance to update an existing system.
- **Components** in UML could represent
  - **logical components** (e.g., business components, process components), and
  - **physical components** (e.g., CORBA components, EJB components, ATL, COM+ and .NET components, WSDL (Web Service Definition Language) components, etc.),

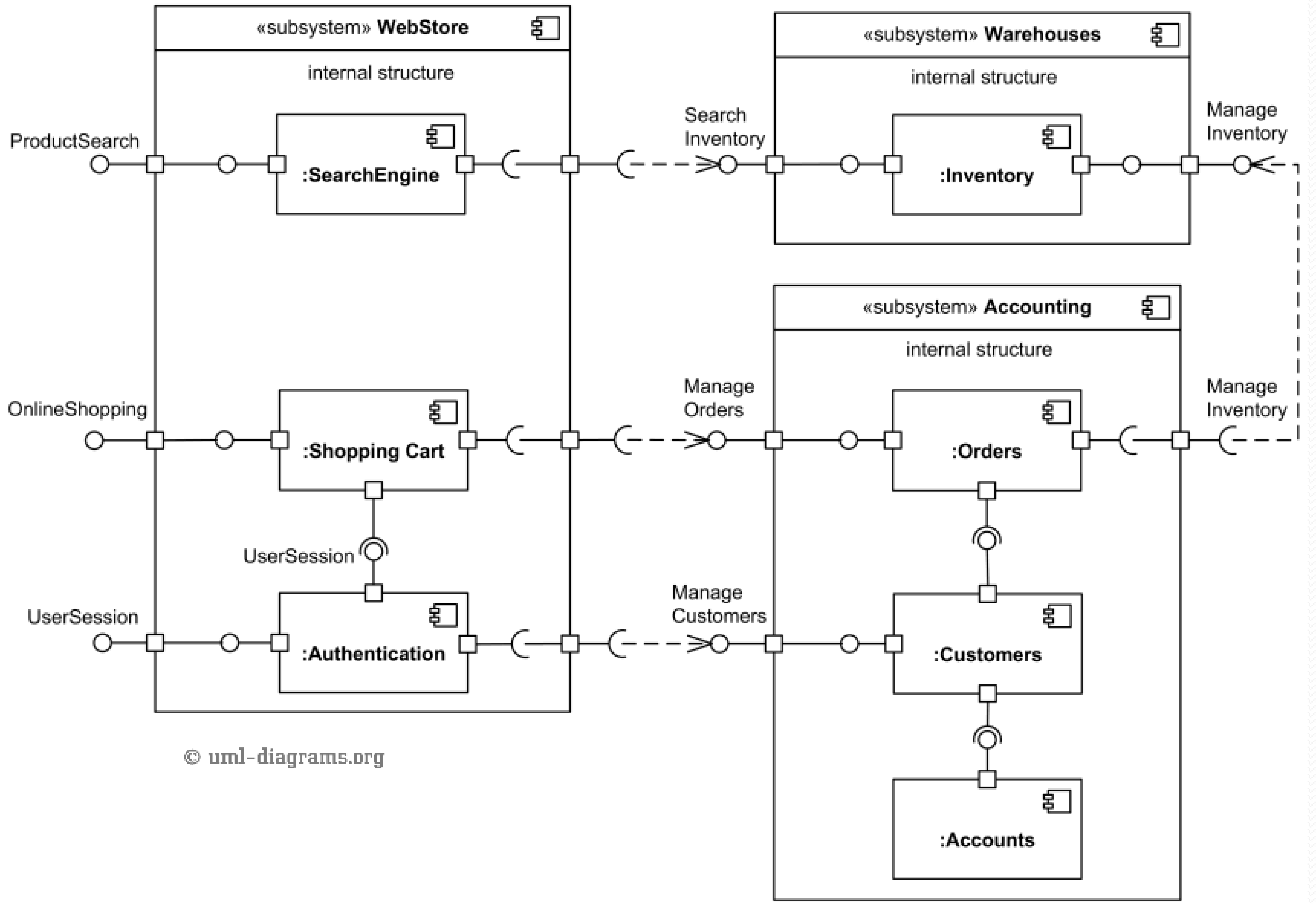
Online Shopping - Components

- ***Summary**: The diagram shows "white-box" view of the internal structure of three related **subsystems**: WebStore, Warehouses, and Accounting.*
- ***WebStore** subsystem contains three components related to online shopping - **Search Engine**, **Shopping Cart**, and **Authentication**.*
- ***Accounting** subsystem provides two interfaces: **Manage Orders** and **Manage Customers**.*
- ***Warehouses** subsystem provides two interfaces: **Search Inventory** and **Manage Inventory** used by other subsystems.*

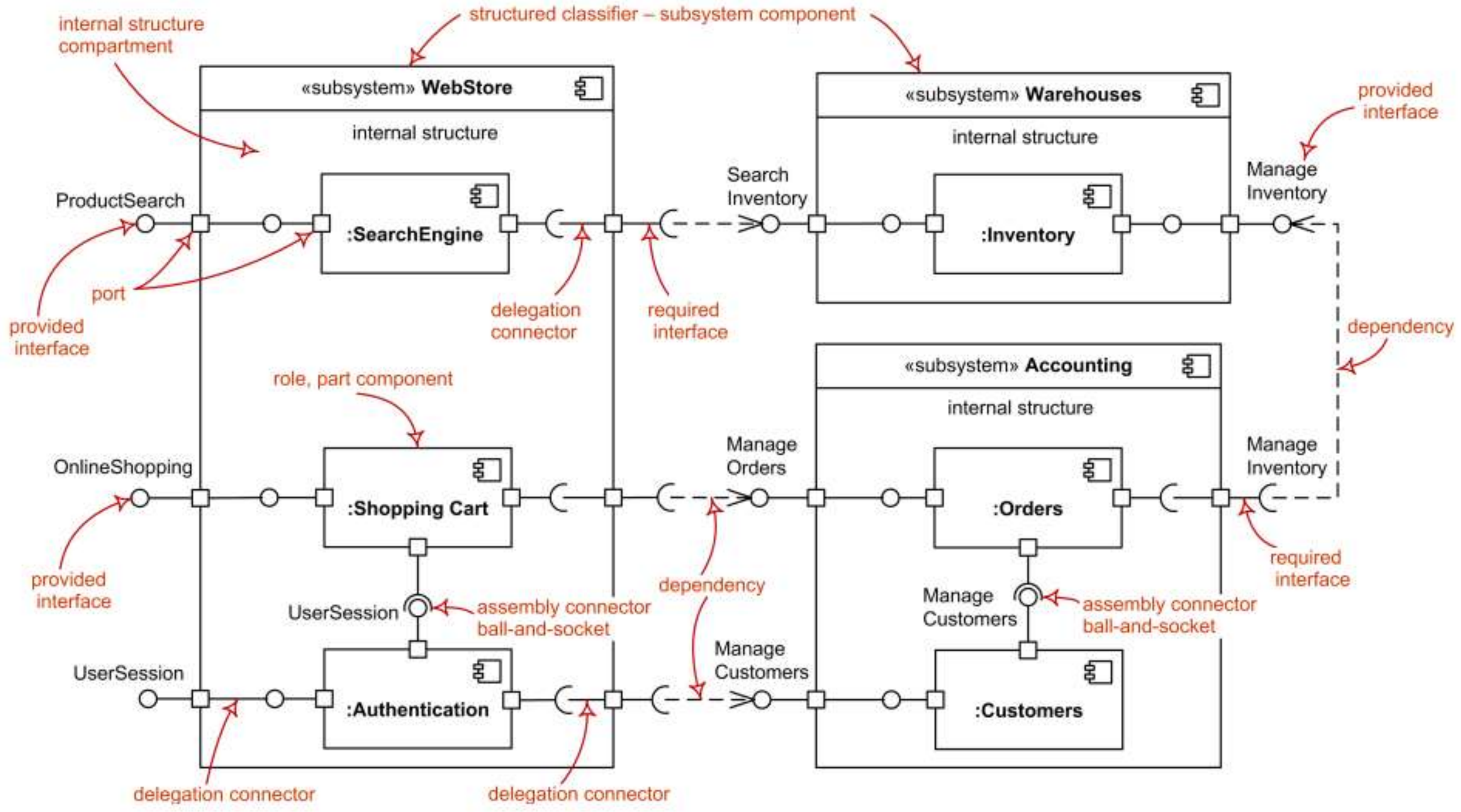
Online Shopping - Components

- **WebStore** subsystem contains three components related to online shopping - **Search Engine**, **Shopping Cart**, and **Authentication**.
- **Search Engine** component allows to search or browse items by exposing **provided interface Product Search** and uses **required interface Search Inventory** provided by **Inventory** component.
- **Shopping Cart** component uses **Manage Orders interface** provided by **Orders** component during checkout.
- **Authentication** component allows customers to create account, login, or logout and binds customer to some account.
- **Accounting** subsystem provides two interfaces - **Manage Orders** and **Manage Customers**.

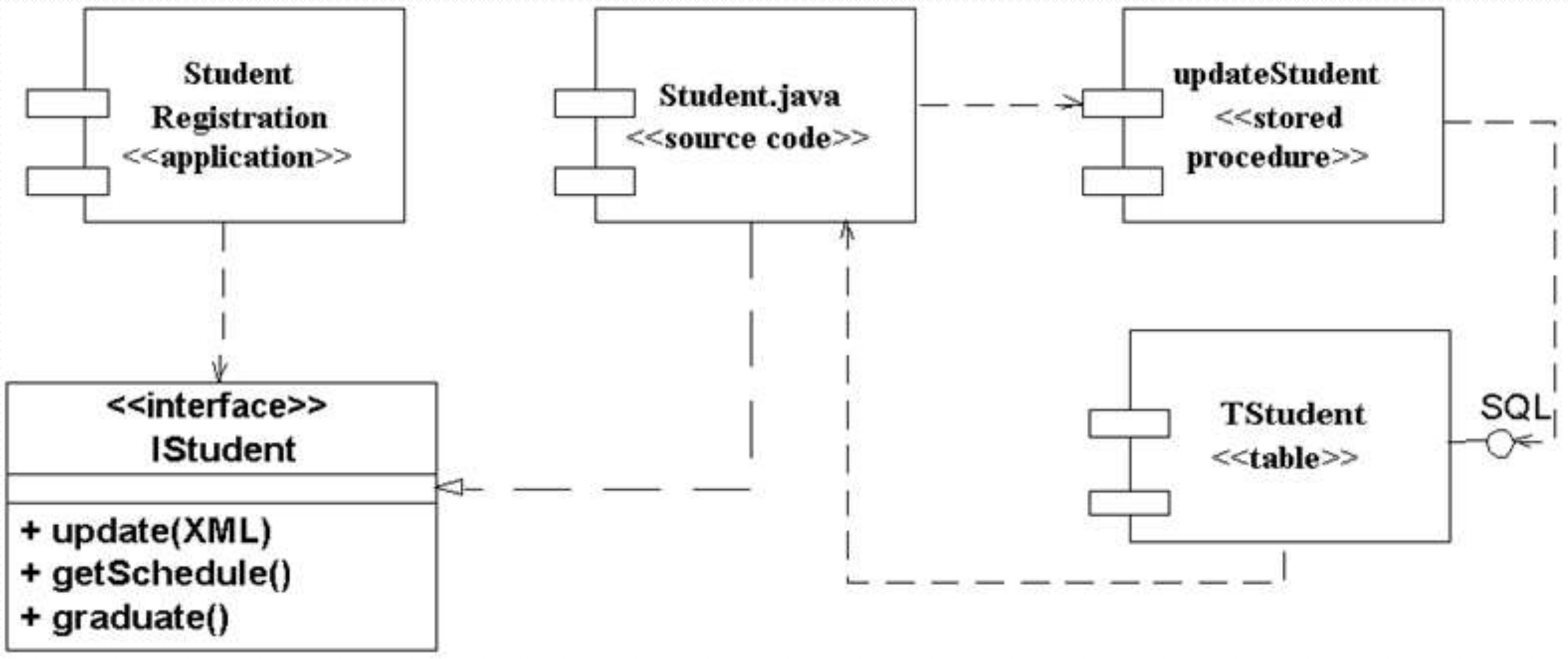
Ex1: Component Diagram



Ex1: UML Component Diagram



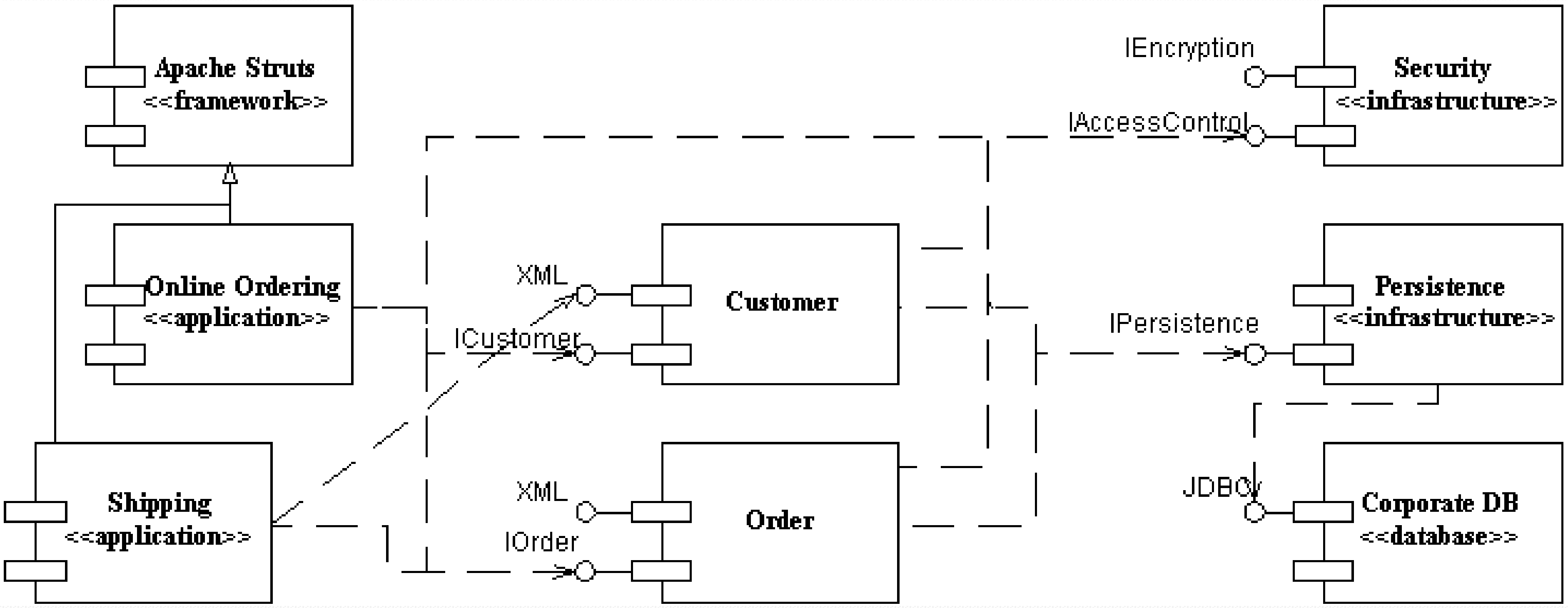
Ex2: Sample interfaces



Ex3: - Online ordering / shipping

- Online **Ordering** and **Shipping** applications uses **Apache Struts** framework for MVC. There is a **Customer** component which provides XML (data) and ICustomer interfaces. There is an **Orders** component which provides XML (data) and IOrder interfaces.
- **Ordering** App uses IOrder & ICustomer interfaces to place orders. **Shipping** App requires XML data from **Customer** component & uses IOrder to link respective orders for shipping.
- **Security** infrastrure provides IAccesControl & IEncryption interfaces. IAccesControl is used by **Ordering**, **Shipping**, **Customers** and **Orders**.
- **Persistence** infrastructure has a *dependency* on **Corporate DB** component via JDBC. **Persistence** infrastructure provides IPersistence used by **Customers** and **Orders** to provide storage persistency for customer orders.

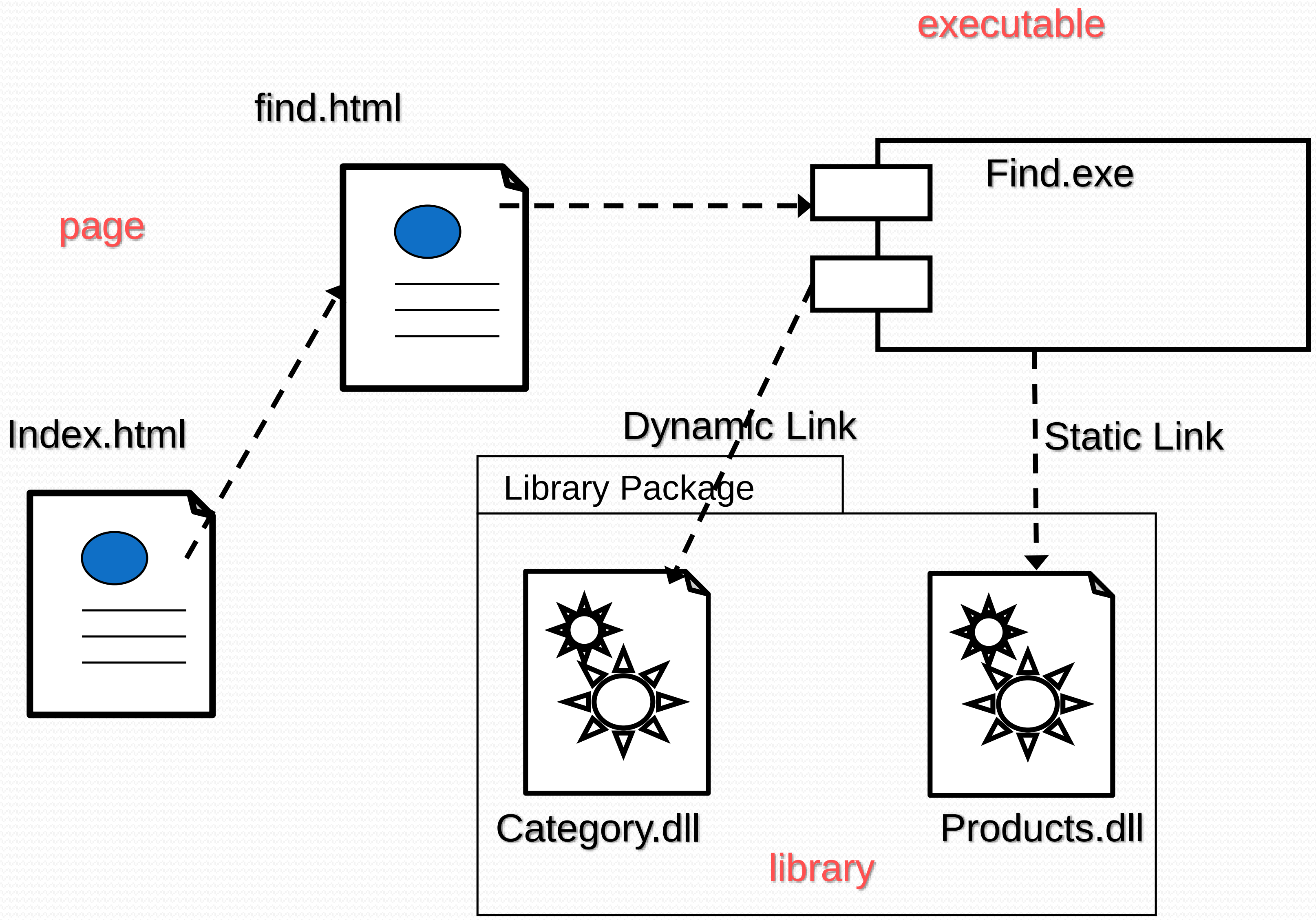
Ex3: Component Diagram



Ex4: Component Diagram

- Consider a scenario of a web application where initially customer visits **index.html** page and from there he has two options: (1) select a category then goto **find.html** page to search in that category (2) or he can directly jump to the **find.html** page for direct product search without category. **find.html** uses **Find.exe** application, which in-turn uses a **Products** DLL with static binding and **Categories** with dynamic binding.
- DLL (***Dynamic Link Library***) , a library of executable functions or data that can be used by a Windows application. Typically, a DLL provides one or more functions and a program can accesses functions by creating either a **static** or **dynamic** link to DLL. Static link remains constant during program execution while a dynamic link is created by the program as needed.

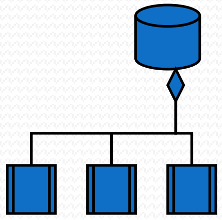
Ex4: Component Diagram



Common Uses:

- Model source code:
  - model configuration mgmt
- Model executable releases
  - Release is relatively complete and consistent set of artifacts delivered to user
  - Release focuses on parts necessary to deliver running system
  - Component Diagram visualizes, specifies, and documents the decisions about the physical parts that define the software.

Common Uses: (cont'd)



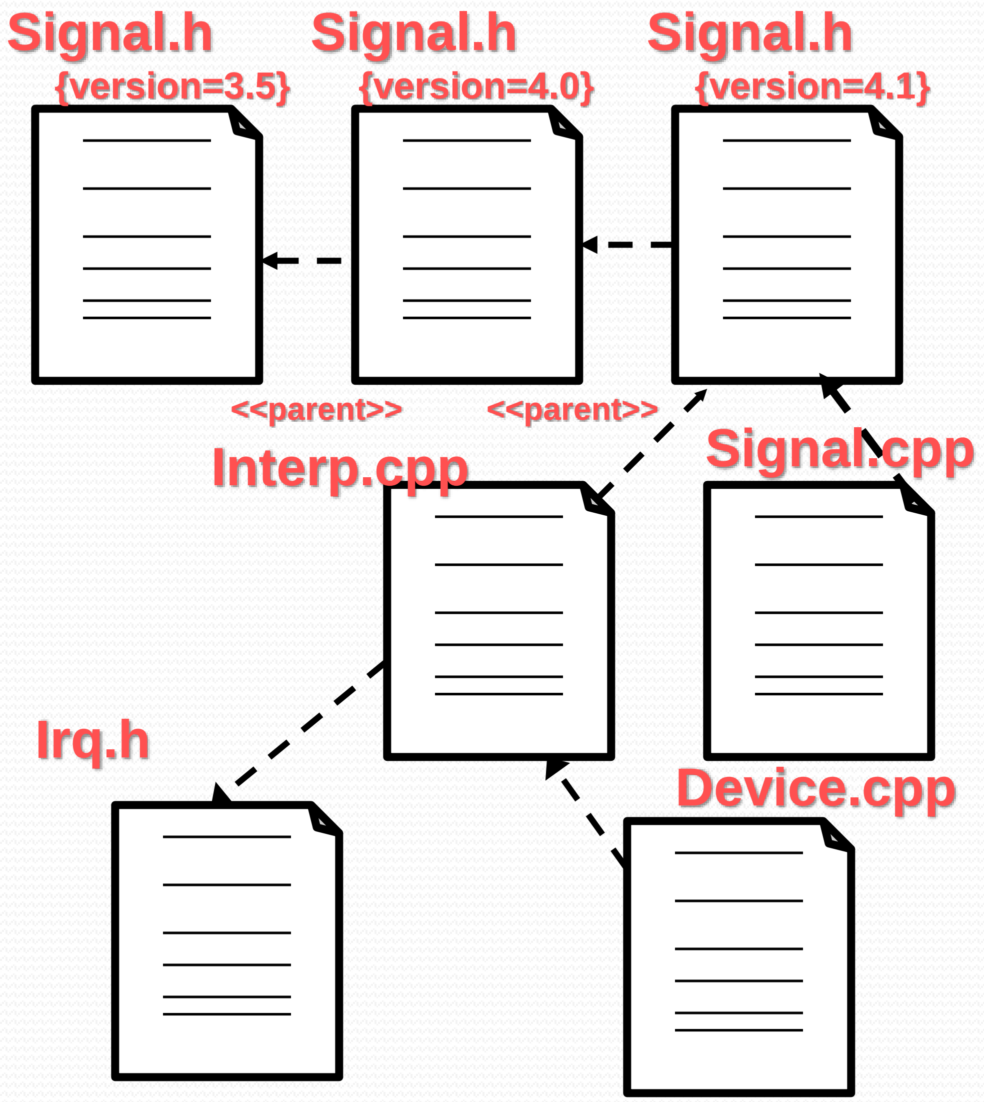
- Model Physical databases:
  - Component Diagram can represent physical DBs
  - database is concrete realization of schema
  - schemas offer an API to persistent information
  - model of physical DBs represents storage of information in tables of a relational DB or pages of an OO dbase.
- Model Adaptable systems:
  - Adaptable Models support changeable domain modules.
  - Allow users to add new rules / features to system without new programming.
  - can model **static aspects** of adaptable systems
  - can model **dynamic aspects** (in conjunction with behavioral models)

Modeling Source Code

- (Forward/Reverse Eng): identify set of source code files of interest
  - model   as components stereotyped as files
- Larger systems: use packages to show groups of source code files
- Model compilation dependencies among files

Ex5: Modeling Source Code

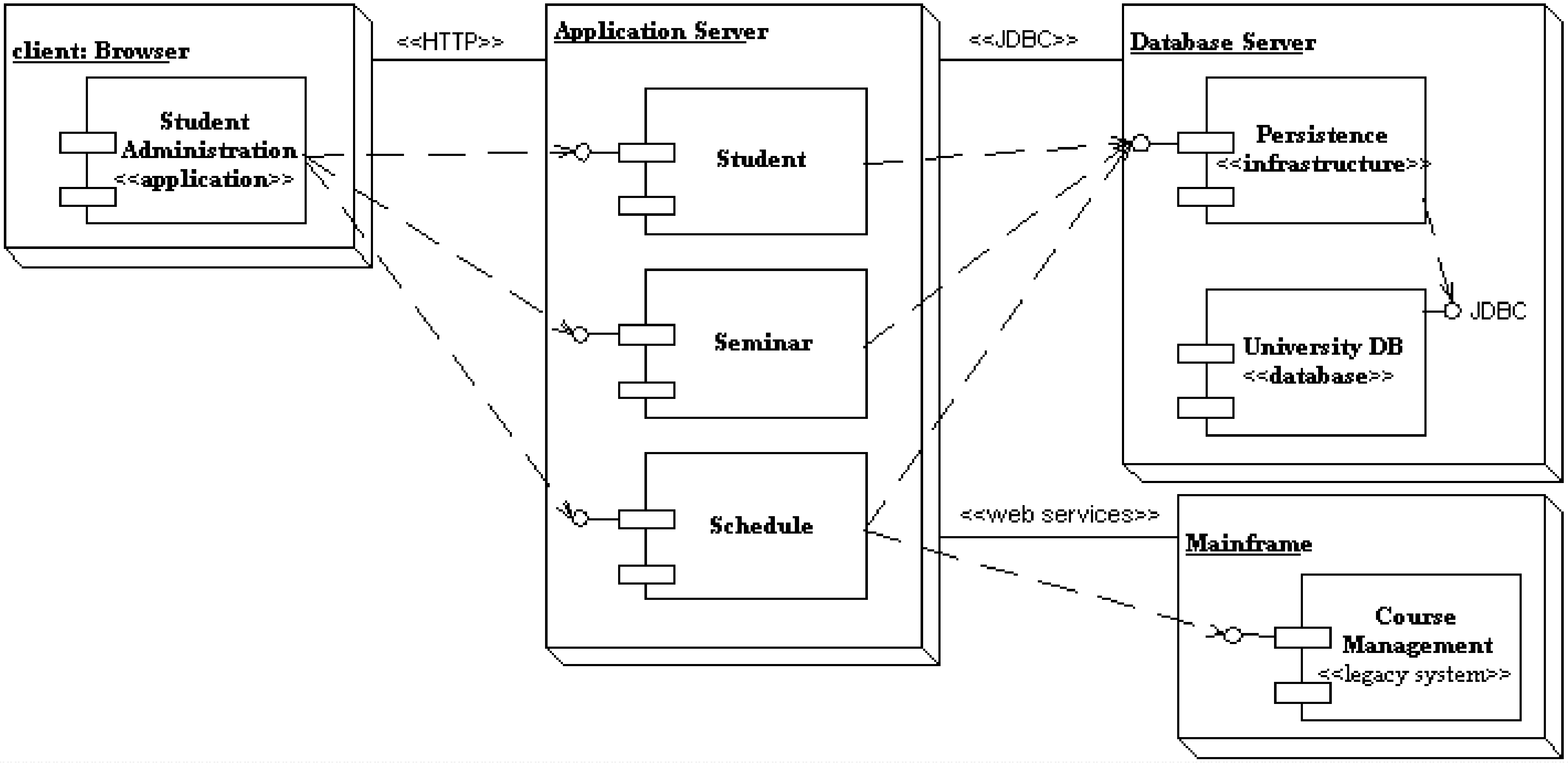
- 5 source code files
  - signal.h (header)
  - used by 2 other files (signal.cpp, interp.cpp)
  - interp.cpp has compilation dependency to header file (irq.h)
  - device.cpp compilation dependency to interp.cpp



Ex6: Communication Links

- Communication associations support one or more communication protocols, each of which should be indicated by a UML stereotype. In Figure you see that the HTTP, JDBC, and web services protocols are indicated using this approach.

Ex6: Communication Links


client: Browser
Student Administration <<application>>
<<HTTP>>
Application Server
Student
Seminar
Schedule
<<JDBC>>
Database Server
Persistence <<infrastructure>>
JDBC
University DB <<database>>
<<web services>>
Mainframe
Course Management <<legacy system>>

Common stereotypes for communication associations.

- Asynchronous: An asynchronous connection, perhaps via a message bus or message queue.
- HTTP: HyperText Transport Protocol.
- JDBC: Java Database Connectivity, Java API for DB access.
- ODBC: Open Database Connectivity, (MS API for DB).
- RMI: Remote Method Invocation, (Java comm. Protocol).
- RPC: Communication via remote procedure calls.
- Synchronous: A synchronous connect where the senders waits for a response from the receiver.
- web services: Communication is via Web Services protocols such as SOAP and UDDI

Component Diagram Guidelines

- **Use Descriptive Names for Architectural Components**
  - Use Environment-Specific Naming Conventions for Detailed Design Components
  - Apply Textual Stereotypes to Components Consistently
  - Avoid Modeling Data and User Interface Components
- **Interfaces**
  - Prefer Lollipop Notation To Indicate Realization of Interfaces By Components
  - Prefer the Left-Hand Side of A Component for Interface Lollipops
  - Show Only Relevant Interfaces
- **Dependencies and Inheritance**
  - Model Dependencies From Left To Right
  - Place Child Components Below Parent Components
  - Components Should Only Depend on Interfaces
  - Avoid Modeling Compilation Dependencies

Common Stereotypes

Stereotype	Indicates
<<application>>	A "front-end" of your system, such as the collection of HTML pages and ASP/JSPs that work with them for a browser-based system or the collection of screens and controller classes for a GUI-based system.
<<database>>	A hierarchical, relational, object-relational, network, or object-oriented database.
<<document>>	A document. A UML standard stereotype.
<<executable>>	A software component that can be executed on a node. A UML standard stereotype.
<<file>>	A data file. A UML standard stereotype.
<<infrastructure>>	A technical component within your system such as a persistence service or an audit logger.
<<library>>	An object or function library. A UML standard stereotype.
<<source code>>	A source code file, such as a .java file or a .cpp file.
<<table>>	A data table within a database. A UML standard stereotype
<<web service>>	One or more web services.
<<XML DTD>>	An XML DTD.

Deployment Diagrams

Deployment Diagram

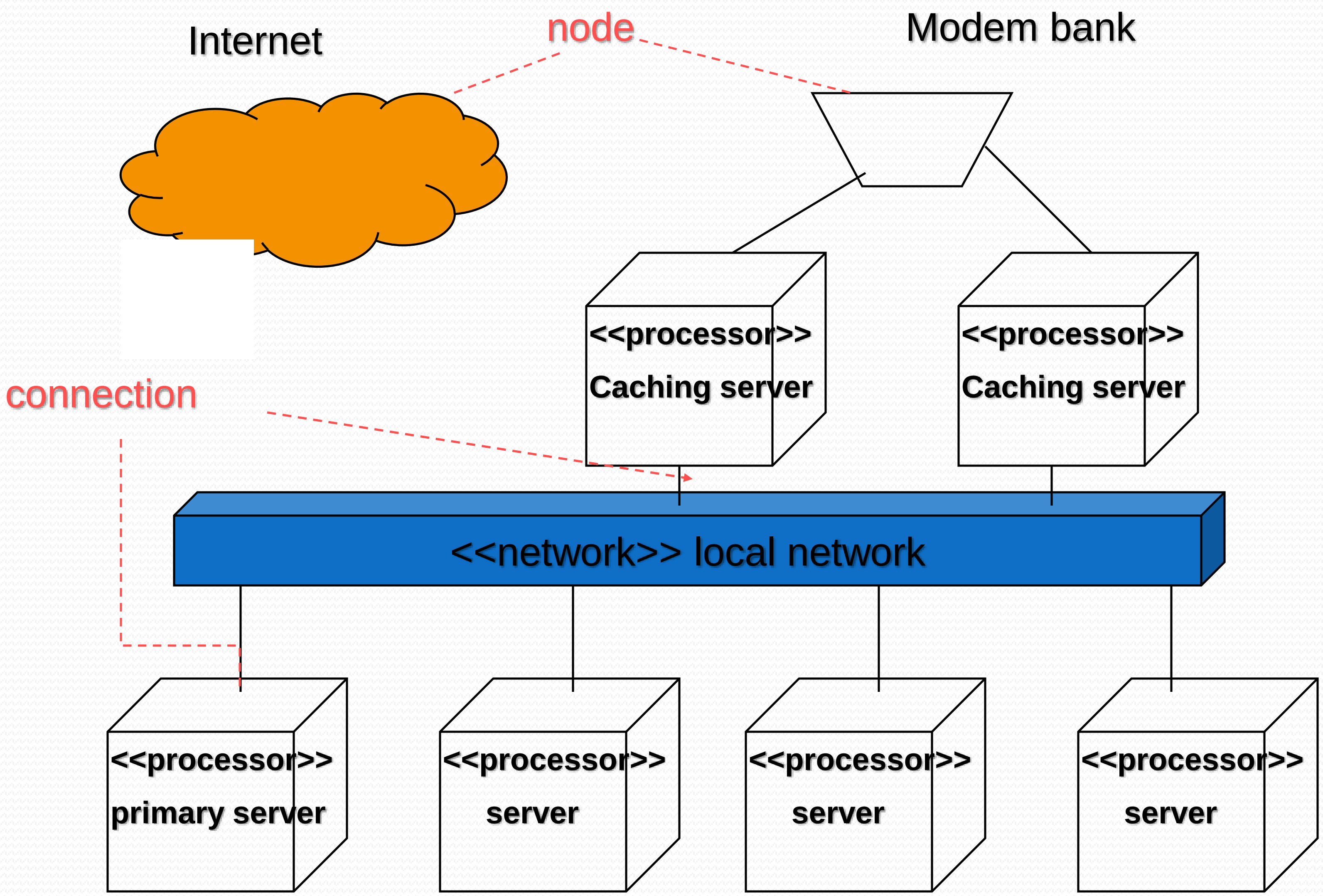
- Shows the configuration of:
  - run time processing nodes and
  - the components that live on them

- Graphically: collection of vertices and arcs

Contents

- Deployment diagrams contain:
  - Nodes
  - Dependency and association relationships
  - may also contain components, each of which must live on some node.

A Deployment Diagram

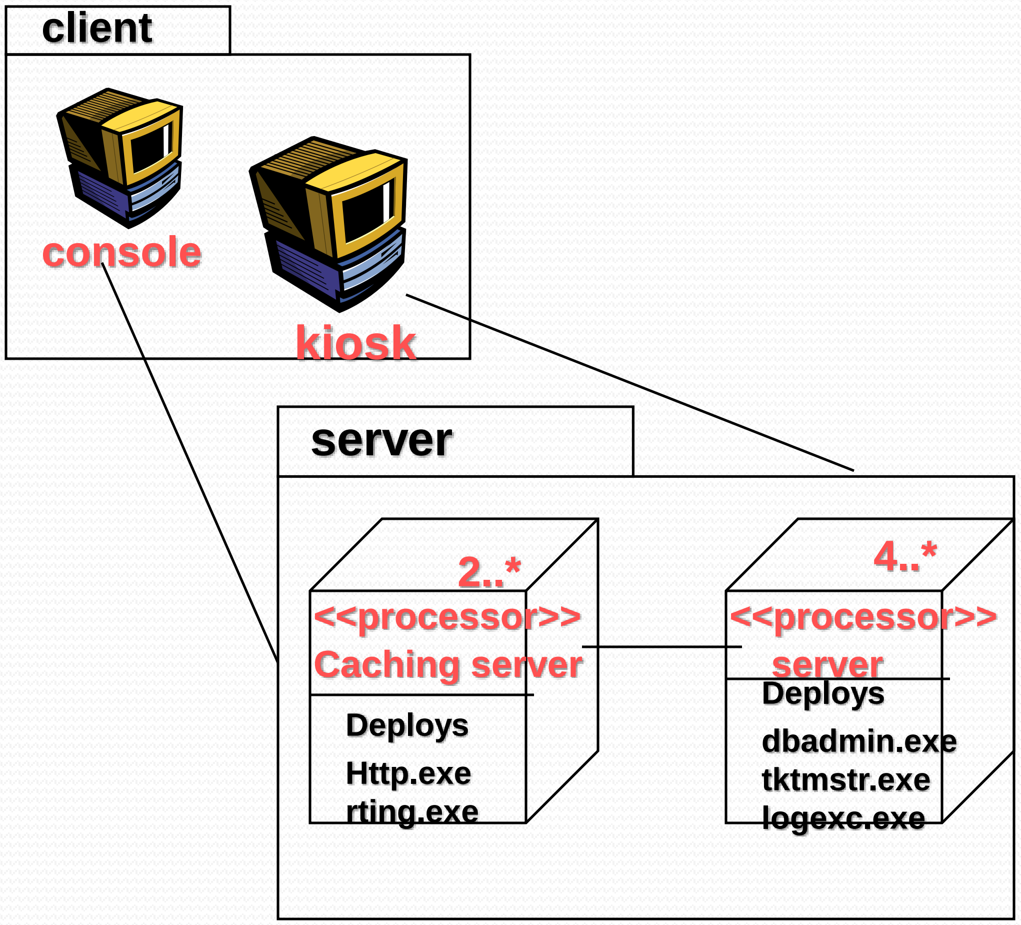


Modeling Client-Server Architecture

- Identify nodes that represent system's client and server processors
- Highlight those devices that are essential to the behavior
  - E.g.: special devices (credit card readers, badge readers, special display devices)
- Use stereotyping to visually distinguish

Client-Server System

- Human resource system
- 2 pkgs: client, server
- Client: 2 nodes
  - console and kiosk
  - stereotyped, distinguishable
- Server: 2 nodes
  - caching server and server
  - Multiplicities are used



Guidelines for Deployment Diagrams

- **General** [Ambler 2002-2005]
  - Indicate Software Components on Project-Specific Diagrams
  - Focus on Nodes and Communication Associations on Enterprise-Level Diagrams
- **Nodes and Components**
  - Name Nodes With Descriptive Terms
  - Model Only Vital Software Components
  - Apply Consistent Stereotypes to Components
  - Apply Visual Stereotypes to Nodes
- **Dependencies and Communication Associations**
  - Indicate Communication Protocols Via Stereotypes
  - Model Only Critical Dependencies Between Components