# COMPONENT DIAGRAM

LECTURE 14

# INTRODUCTION



An UML diagram classification:

- Static
  - Use case diagram, Class diagram
- Dynamic
  - State diagram, Activity diagram, Sequence diagram, Collaboration diagram
- Implementation
  - Component diagram, Deployment diagram

UML components diagrams are

- Implementation diagrams: describe the different elements required for implementing a system

# INTRODUCTION



**Another classification:**

- **Behavior diagrams**
  - A type of diagram that depicts behavior of a system

    This includes activity, state machine, and use case diagrams, interaction diagrams
- **Interaction diagrams**
  - A subset of behavior diagrams which emphasize object interactions. This includes collaboration, sequence diagrams
- **Structure diagrams**
  - A type of diagram that depicts the elements of a specification that are irrespective of time. This includes class, composite structure, component, deployment

UML components diagrams are **structure diagrams**

# What is a Component?

- Several definitions of a component in literature, however everyone agrees that a component is a piece of software… But this requires clarification!

# What is a Component?



- Components provide a service without regard to where the component is executing or its programming language
  - A component is an independent executable entity that can be made up of one or more executable objects
  - The component interface is published and all interactions are through the published interface

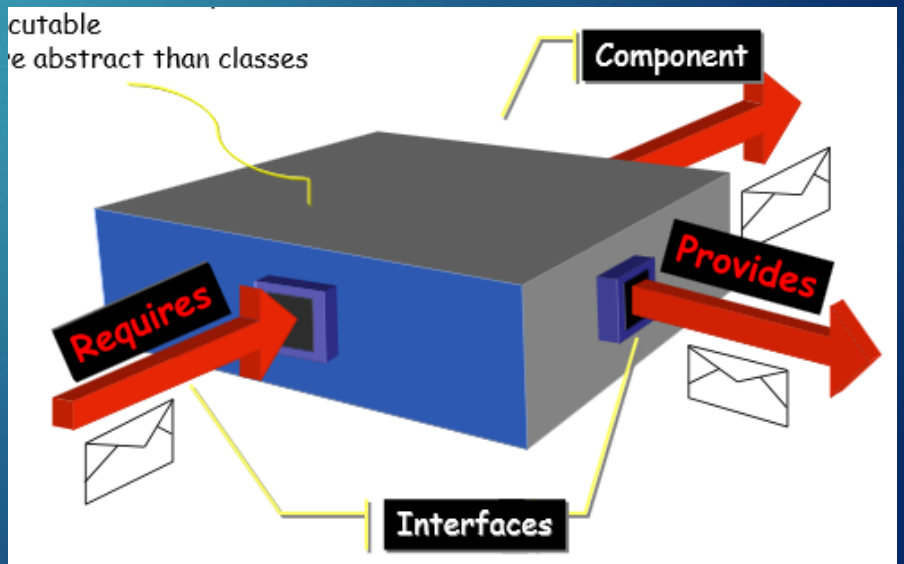
# What is a Component?

- The Object Management Group's "Modeling Language Specification" defines a component as
  - "a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces.
  - A component represents a physical piece of system's implementation, including software code (source, binary or executable) or equivalents, such as scripts or command files.

# Component



- Component diagram is used to model the static implementation view of a system.
  - Provides a service: implementation-independent
  - Need not to be compiled
  - Executable
  - More abstract than classes



# Components



A component is a physical replaceable part of a system.

- Examples of components include:
  - Source code files
  - Executables
  - Libraries
  - Tables
  - Files
  - Documents
- Some more examples:
  - ActiveX Control
  - Java Bean
  - Web Page
  - Web Server
  - Firewall
  - Style Sheet
  - Database etc…

# Components



- Components are the physical elements that contain the logical elements like classes, interfaces etc…
- A component diagram contains:
  - Components
  - Interfaces
  - Dependency, generalization, association and realization relationships

# Components



- Components Vs Classes:
  1. Components are physical things, whereas, classes are logical abstractions.
  2. Components are at a higher level of abstraction than a class.
  3. Components only contain operations, whereas, a class can have both attributes and operations.

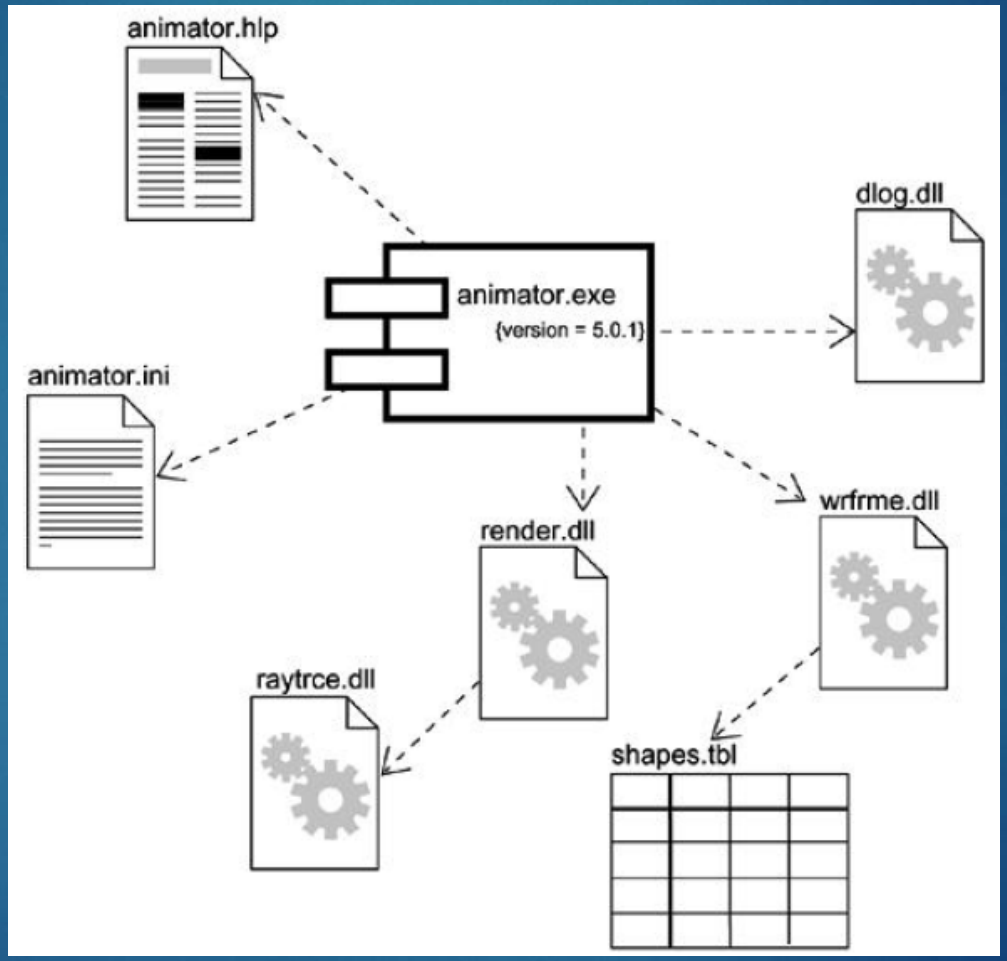
# Components



- An interface contains a set of operations which specify a service to a class or a component.
- There are three kinds of components:
  1. Deployment components (Dlls, Exes)
  2. Work product components (Source code files)
  3. Execution components (COM+ objects)

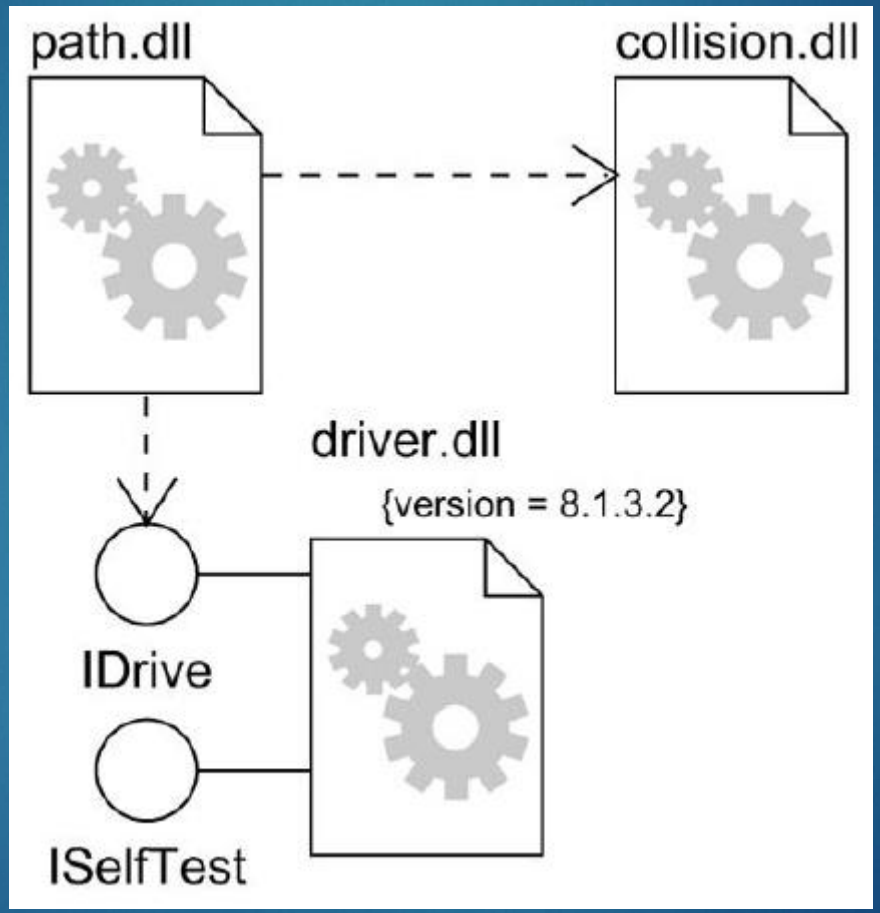
# Components




animator.hlp
dlog.dll
animator.exe
{version = 5.0.1}
animator.ini
wrfrme.dll
render.dll
raytrce.dll
shapes.tbl

# Component Diagram - Example

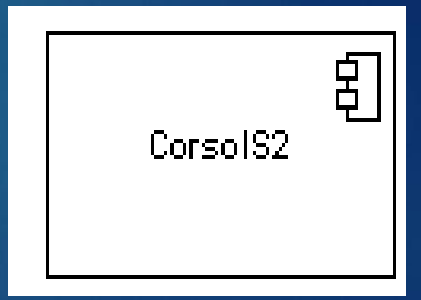




# COMPONENT NOTATION



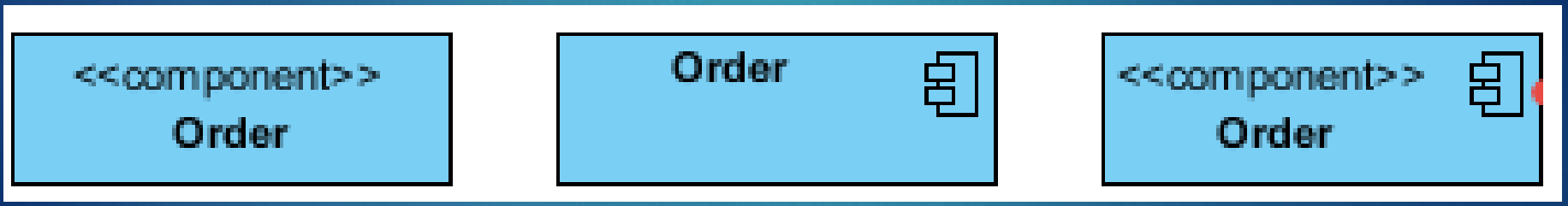
- A component is shown as a rectangle with
  - A keyword <<component>>
  - Optionally, in the right hand corner a component icon can be displayed
    - A component icon  is a rectangle with two smaller rectangles jutting out from the left-hand side
    - This symbol is a visual stereotype
  - The component name
- Components can be labelled with a stereotype there are a number of standard stereotypes ex: <<entity>>, <<subsystem>>

# COMPONENT NOTATION



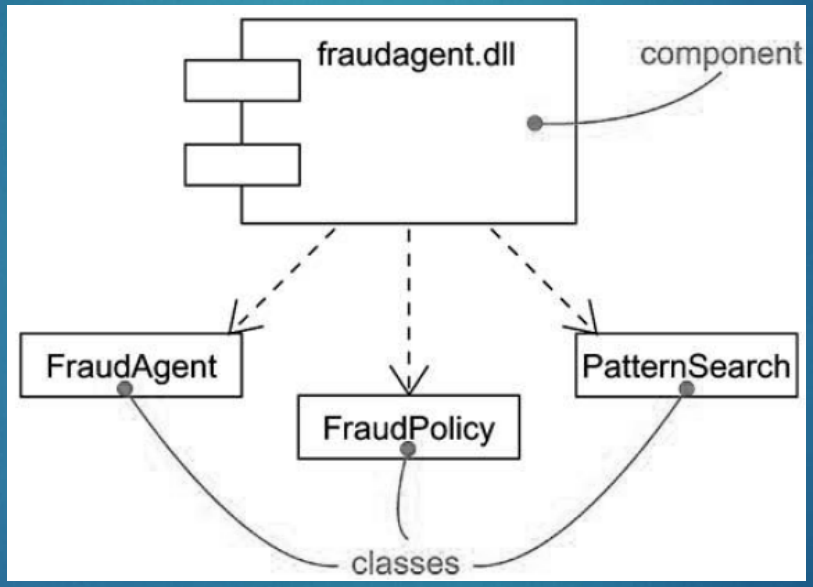
- rectangle with the component's name
- A rectangle with the component icon
- A rectangle with the stereotype text and/or icon



# COMPONENT NOTATION



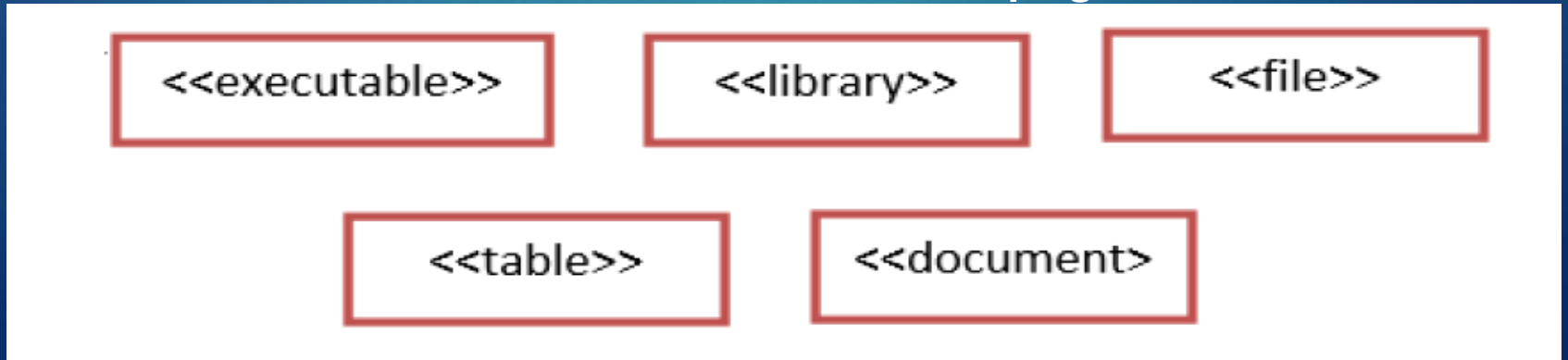
- The relationship between a component and its classes can be represented using dependencies.



# Component Stereotypes



- Components stereotype provides visual cues about roles played by components in a system. Some of component stereotype are as follows.
  - **<<executable>>: executable file (.exe)**
  - **<<library>>: references resources (.dll)**
  - **<<file>>: text file, source code file, etc.**
  - **<<table>>: database file, table file, etc.**
  - **<<document>>: document file, web page file, etc.**



# 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. |

# Component Elements



- A component can have
  - Interfaces
    - An interface represents a declaration of a set of operations and obligations
  - Usage dependencies
    - A usage dependency is relationship in which one element requires another element for its full implementation
  - Ports
    - Port represents an interaction point between a component and its environment
  - Connectors
    - Connect two components
    - Connect the external contract of a component to the internal structure

# Interfaces

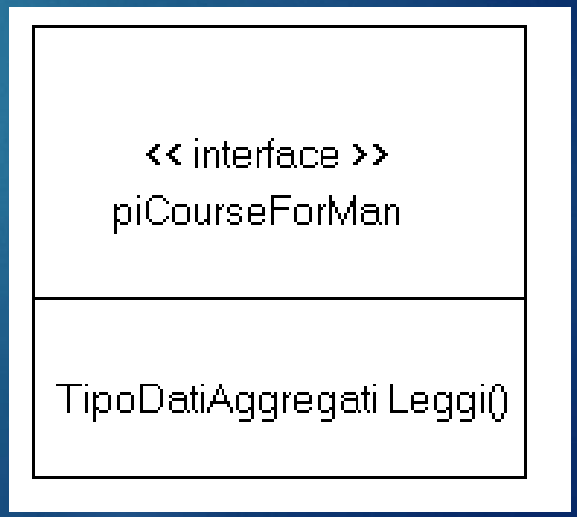


- **An interface**
  - Is the definition of a collection of one or more operations
  - Provides only the operations but not the implementation
  - Implementation is normally provided by a class/ component
  - In complex systems, the physical implementation is provided by a group of classes rather than a single class
- A class can implement one or more interfaces
- An interface can be implemented by 1 or more classes

# Interfaces



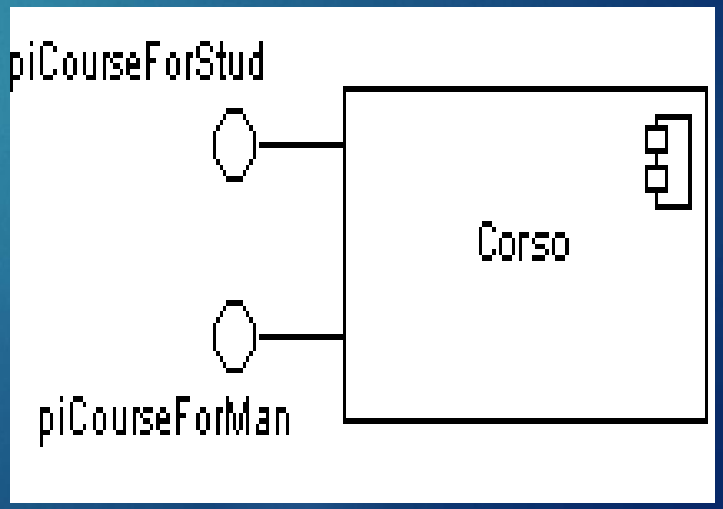
- May be shown using a rectangle symbol with a keyword <<interface>> preceding the name.
- For displaying the full signature, the interface rectangle can be expanded to show details
- Can be
  - Provided
  - Required

# Provided Interfaces



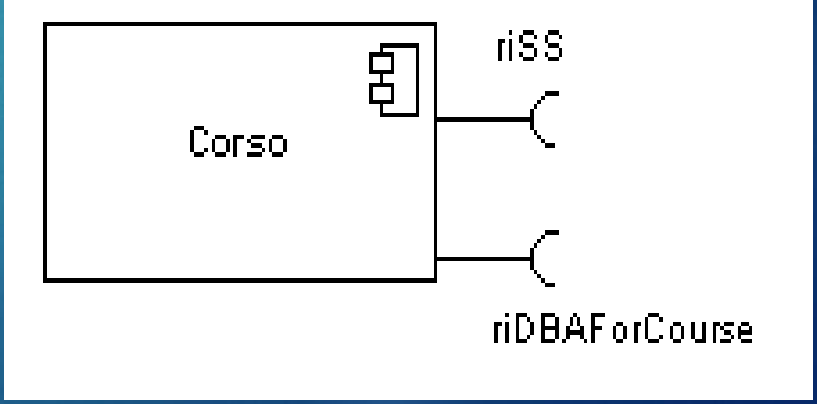
- **Provided Interface:**
- Characterize services that the component offers to its environment
- Is modeled using a ball, labelled with the name, attached by a solid line to the component. Also known as Lollipop notation.



# Required Interfaces



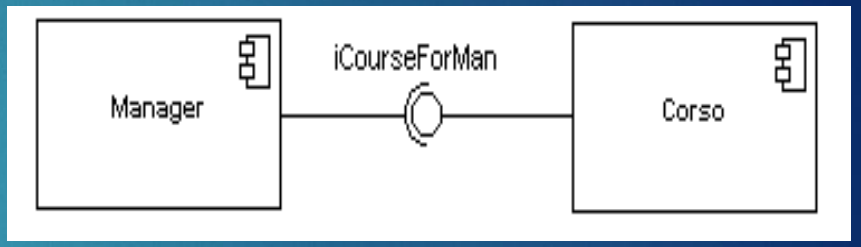
- **Required Interface:**
- Characterize services that the component expects from its environment
- Is modeled using a socket, labelled with the name, attached by a solid line to the component
- In UML 1.x were modeled using a dashed arrow



# Interfaces



- Where two components/classes provide and require the same interface, these two notations may be combined.

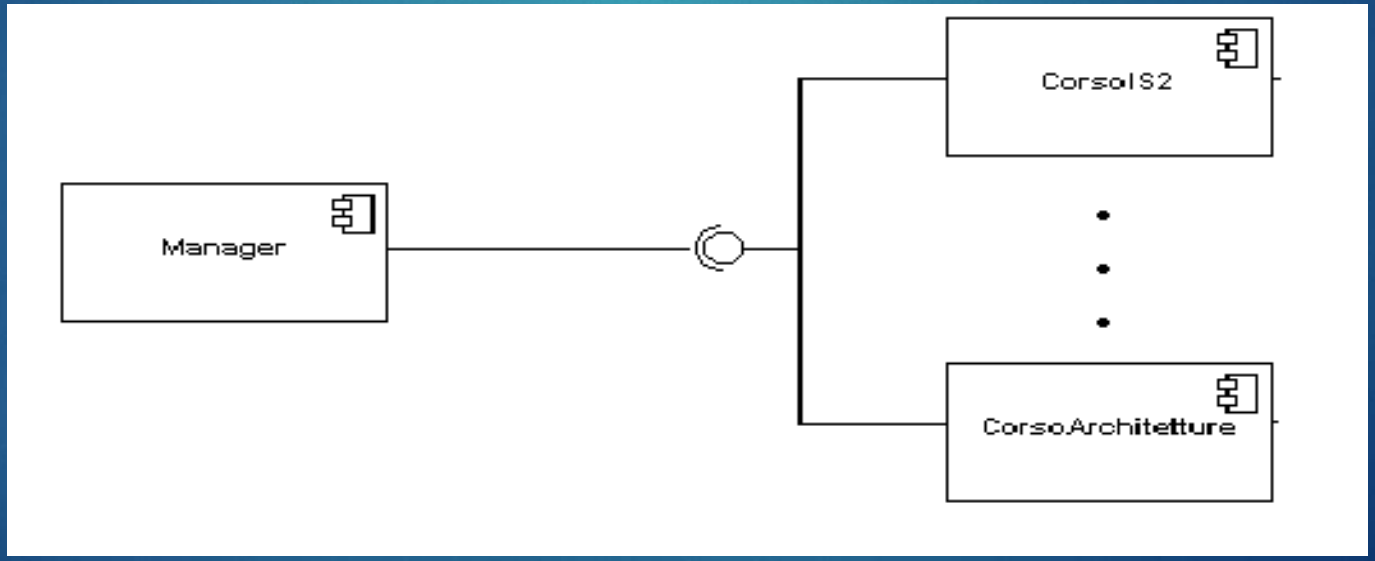



- The ball-and-socket notation hint at that interface in question serves to mediate interactions between the two components
- If an interface is shown using the rectangle symbol, we can use an alternative notation, using dependency arrows

# Interfaces



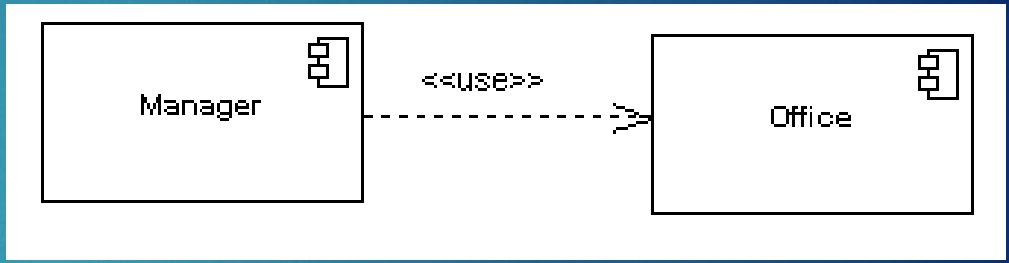
- In a system context where there are multiple components that require or provide a particular interface, a notation abstraction can be used that combines by joining the interfaces.



# Dependencies



- Components can be connected by usage dependencies.




**Usage Dependency**

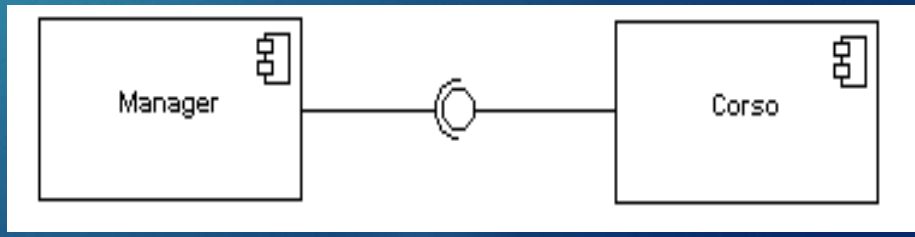
- A usage dependency is relationship where one element requires another element for its full implementation
- Is a dependency in which the client requires the presence of the supplier
- Is shown as dashed arrow with a <<use>> keyword

# Connectors



- Two kinds of connectors:
  - Delegation
  - Assembly
- **ASSEMBLY CONNECTOR**
  - A connector between 2 components defines that one component provides the services that another component requires
  - It must only be defined from a required interface to a provided interface
  - An assembly connector is notated by a "ball-and-socket" connection

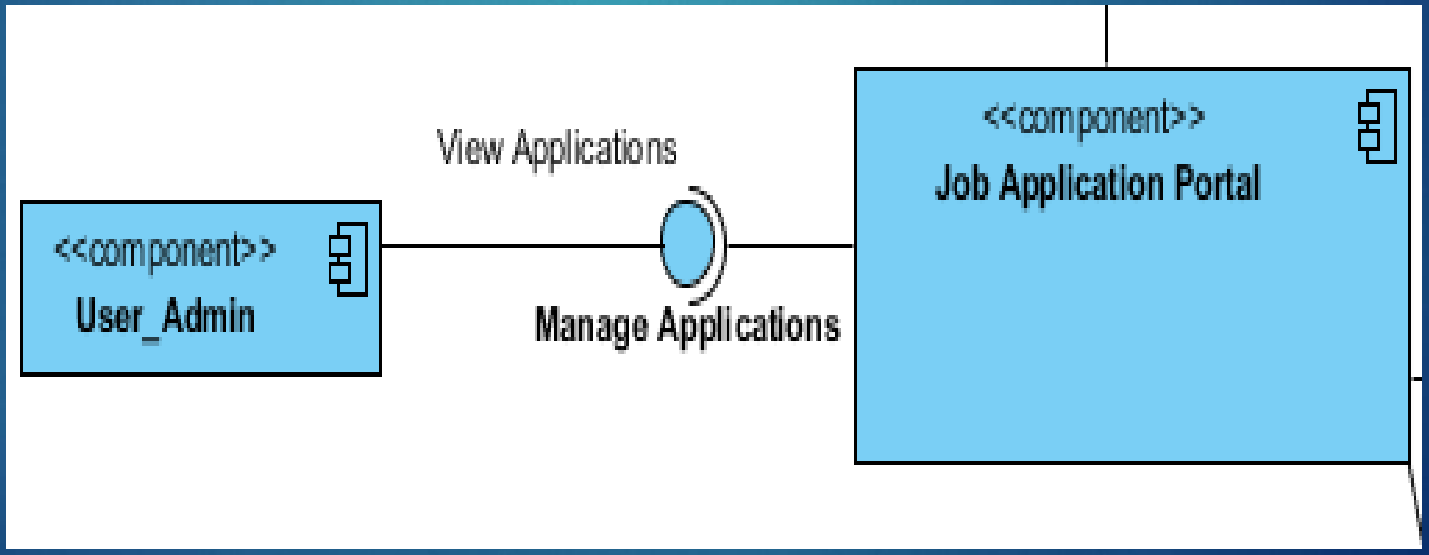
This notation allows for succint grafical wiring of components



# Assembly Connector



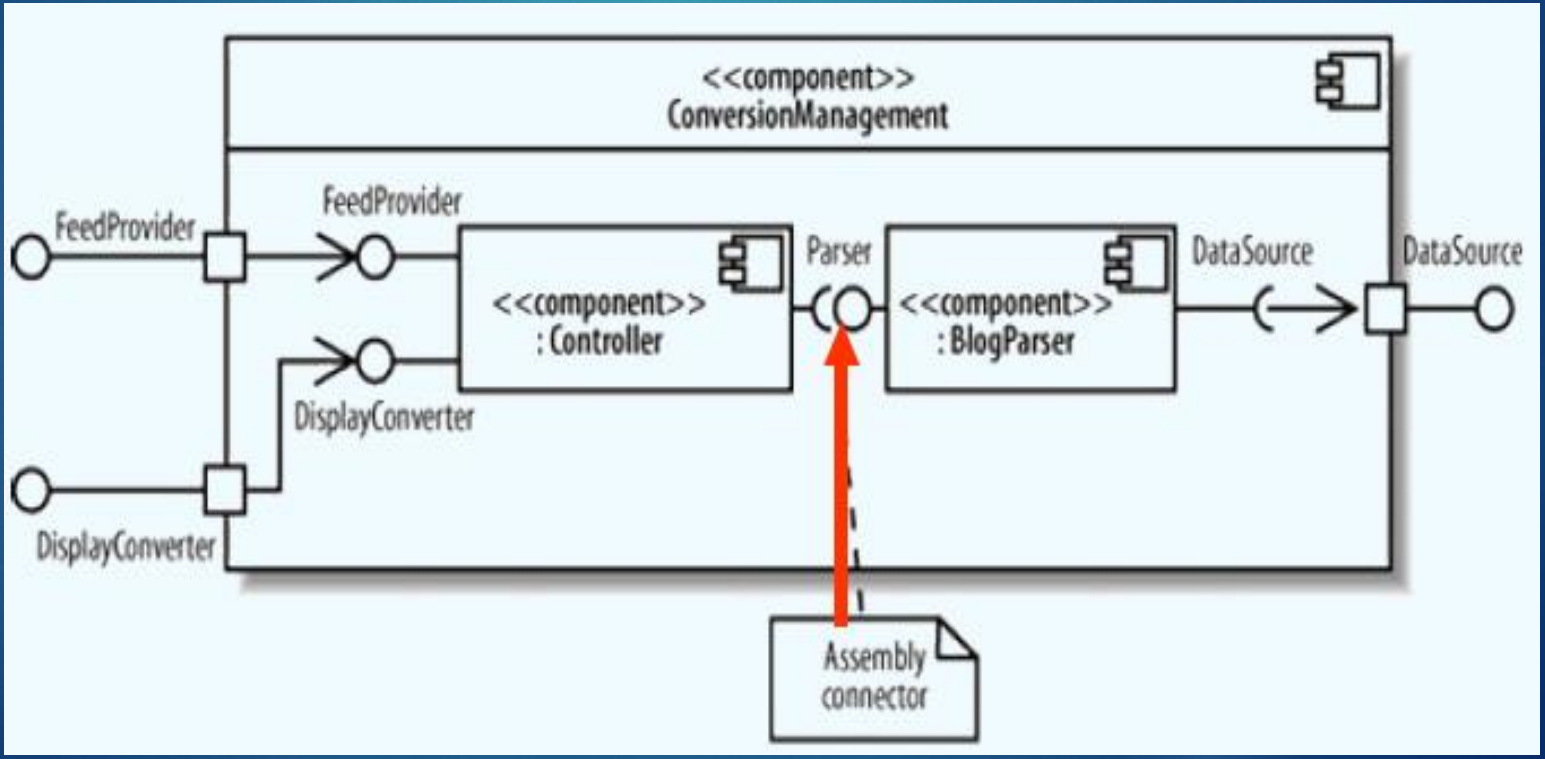
- The assembly connector bridges a component's required interface (Job Application portal) with the provided interface of another component (User Applicant).



# Assembly Connector



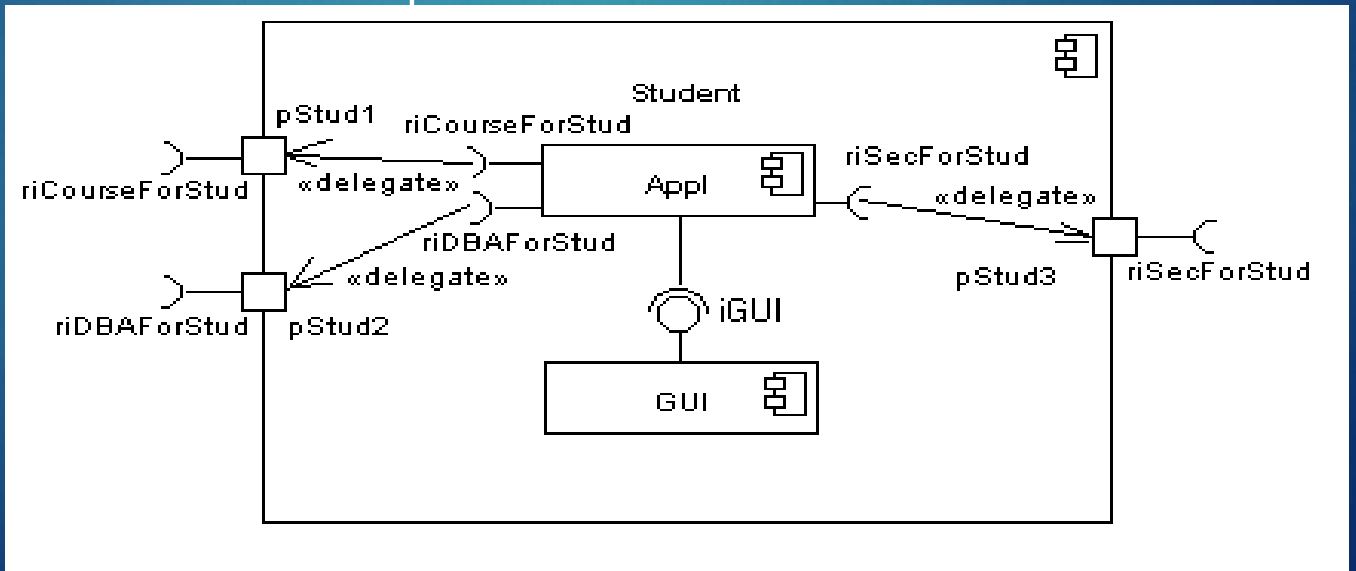
- Used to show components within another component working together through interfaces.



# Delegation Connector



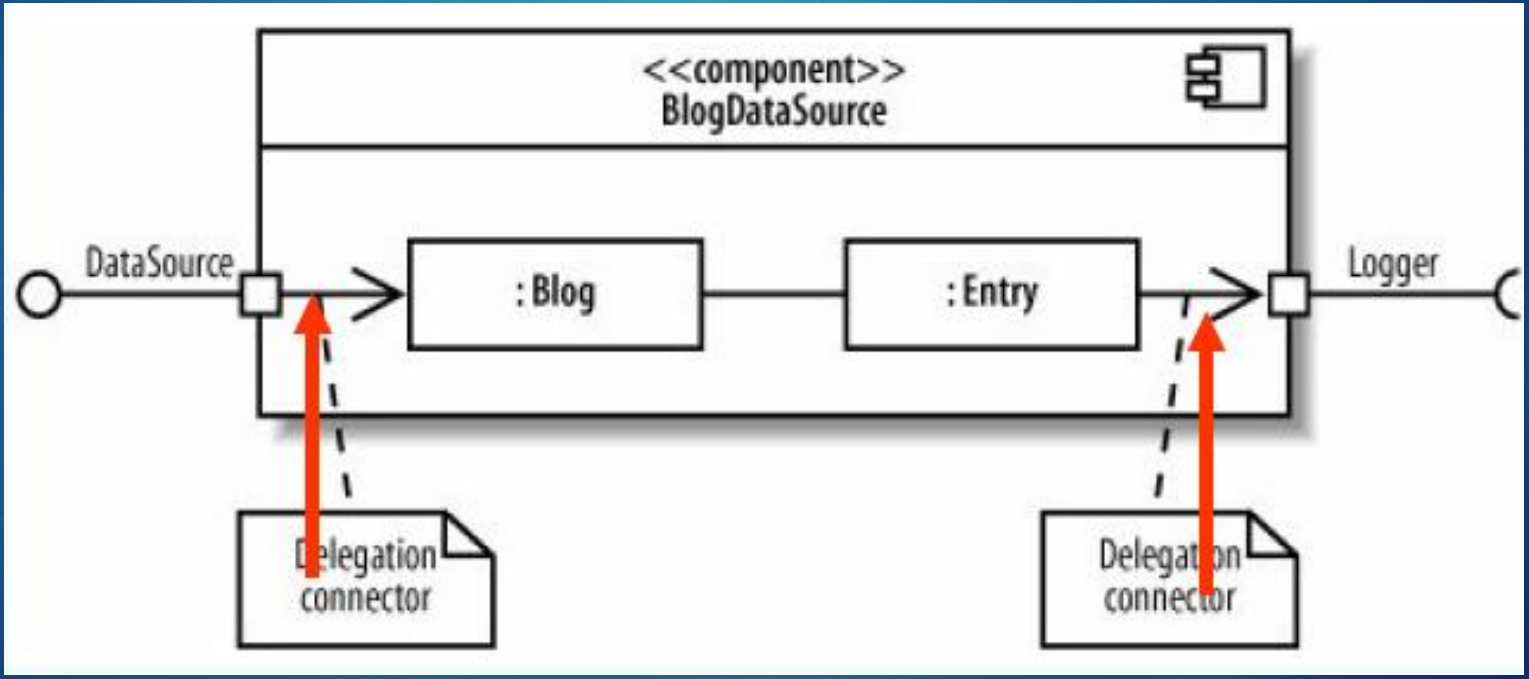
- Links the external contract of a component to the internal realization
- Represents the forwarding of signals
- It must only be defined between used interfaces or ports of the same kind



# Delegation Connector



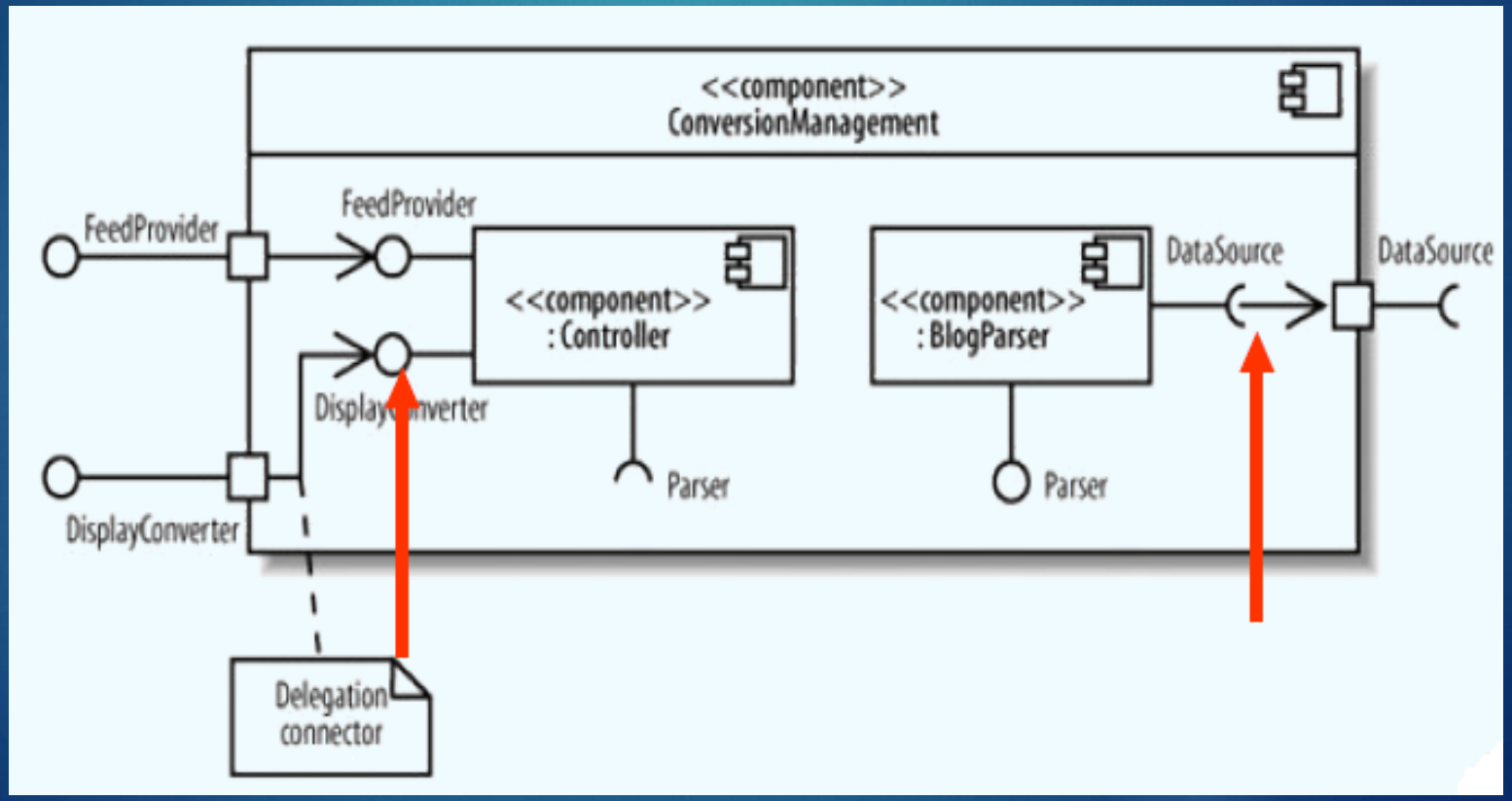
- Used to show that internal parts realize or use the component's interfaces.



# Delegation Connector



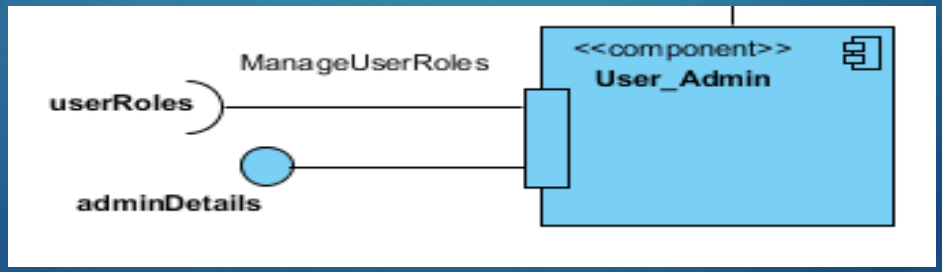
- Delegation connectors can also connect interfaces of internal parts with ports



# Port

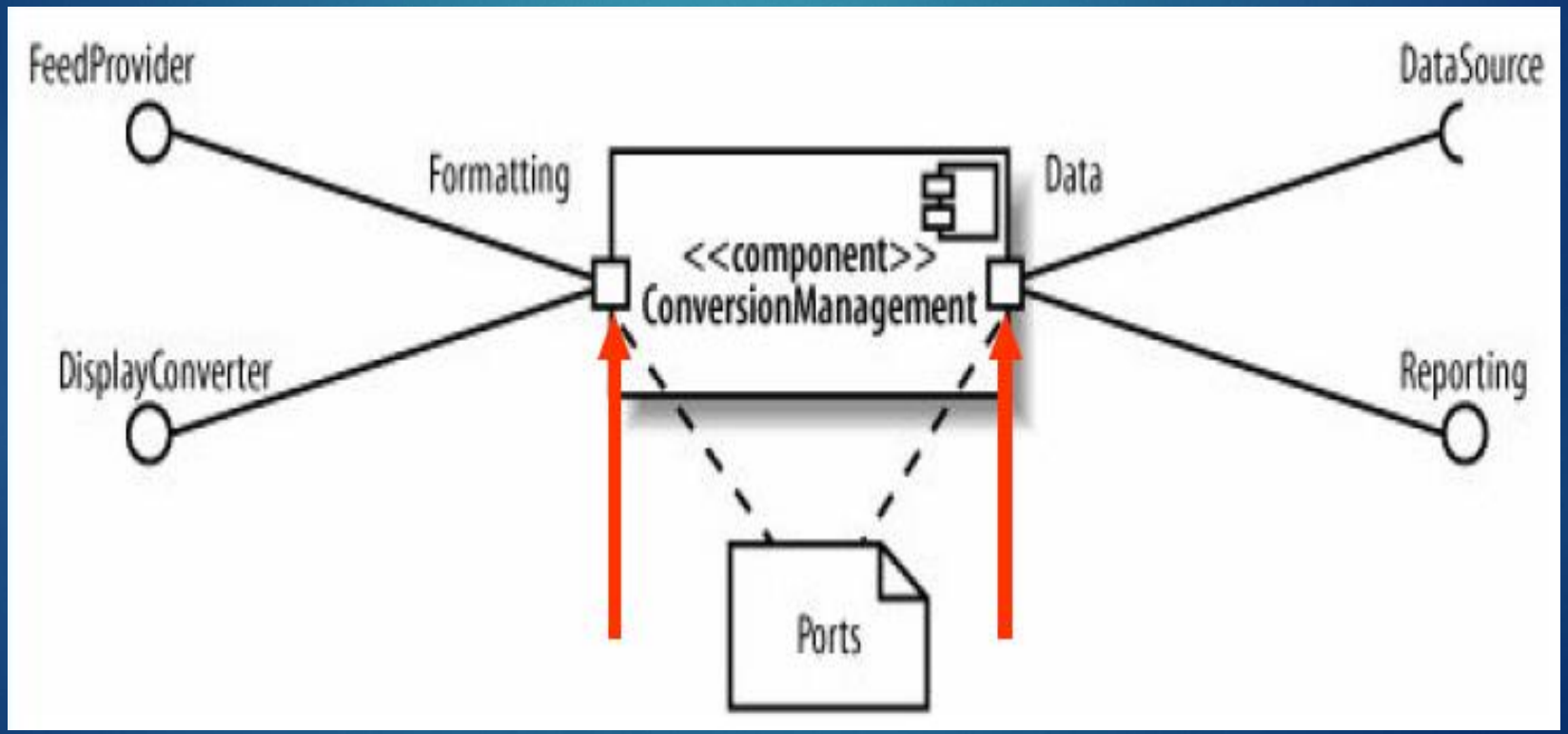


- Specifies a distinct interaction point between that component and its environment –
- Between that component and its internal parts – Is shown as a small square symbol –
- Ports can be named, and the name is placed near the square symbol – Is associated with the interfaces
- Library Services class has port searchPort.



# PORT

- Used to model distinct ways that a component can be used with related interfaces attached to the port



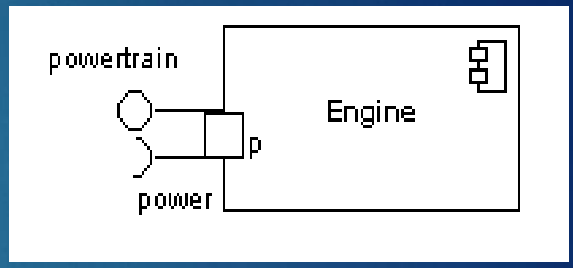


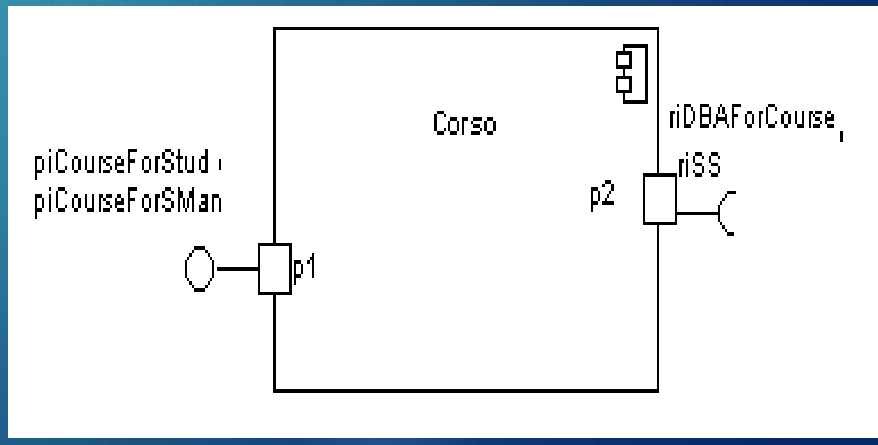

34

# PORT



- Ports can support unidirectional communication or bi-directional communication
- If there are multiple interfaces associated with a port, these interfaces may be listed with the interface icon, separated by a commas



# Views of a Component

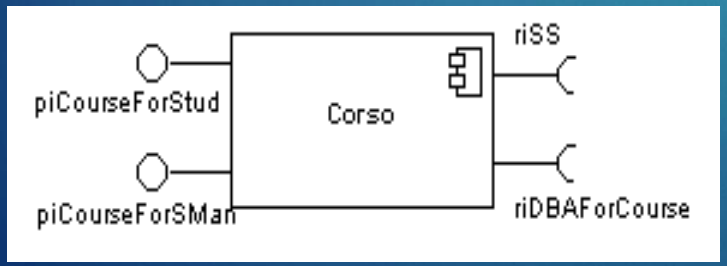


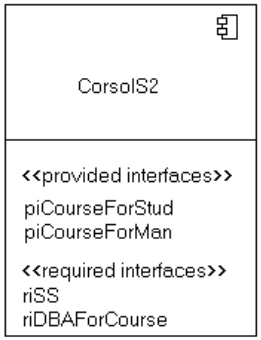
- A component have an
  - external view and
  - an internal view

# EXTERNAL VIEW



- An external view (or black box view) shows publicly visible properties and operations
- An external view of a component is by means of interface symbols sticking out of the component box
- The interface can be listed in the compartment of a component box



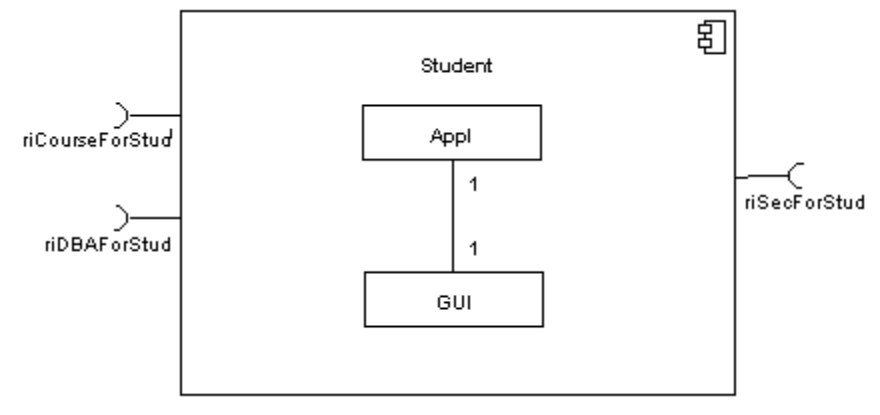
# Internal View

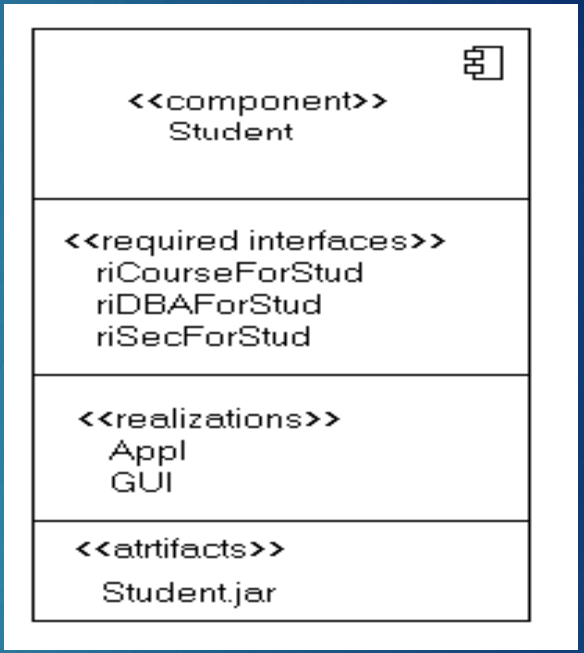


- An internal, or white box view of a component is where the realizing classes/components are nested within the component shape
- The internal class that realize the behavior of a component may be displayed in an additional compartment
- Compartments can also be used to display parts, connectors or implementation artifacts
- An artifact is the specification of a physical piece of information

# Internal View

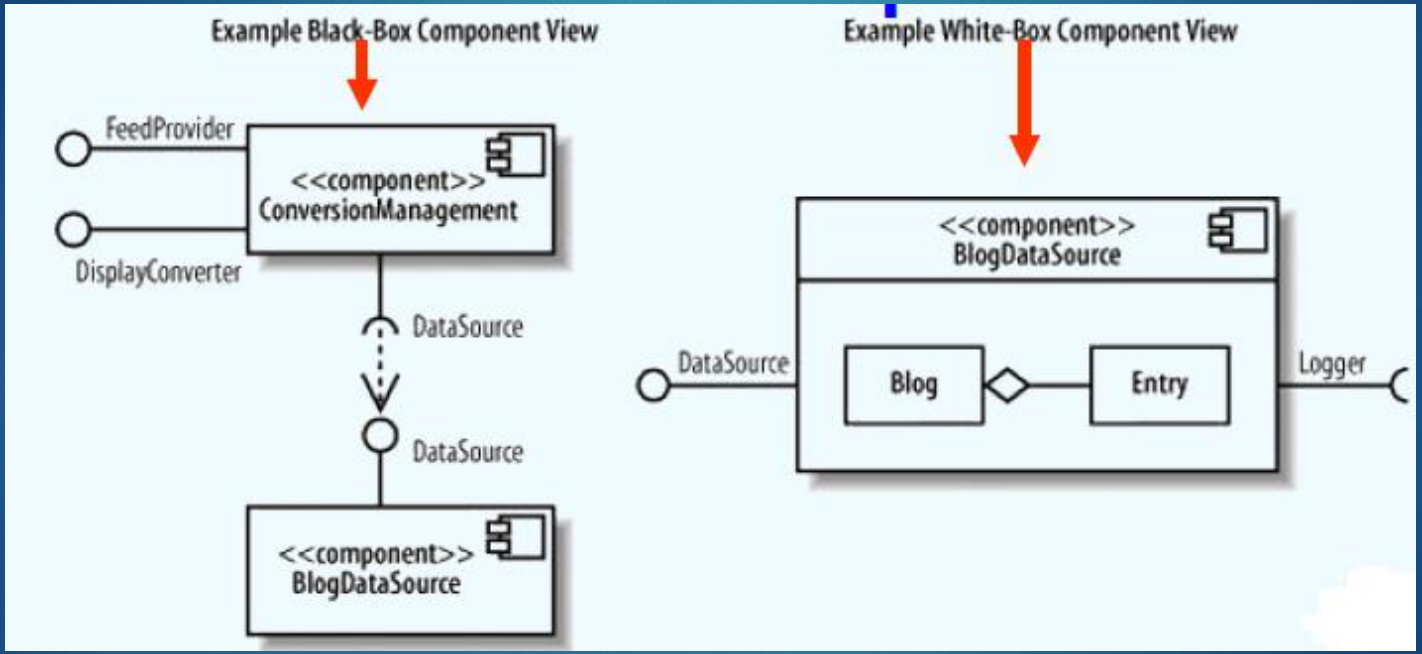




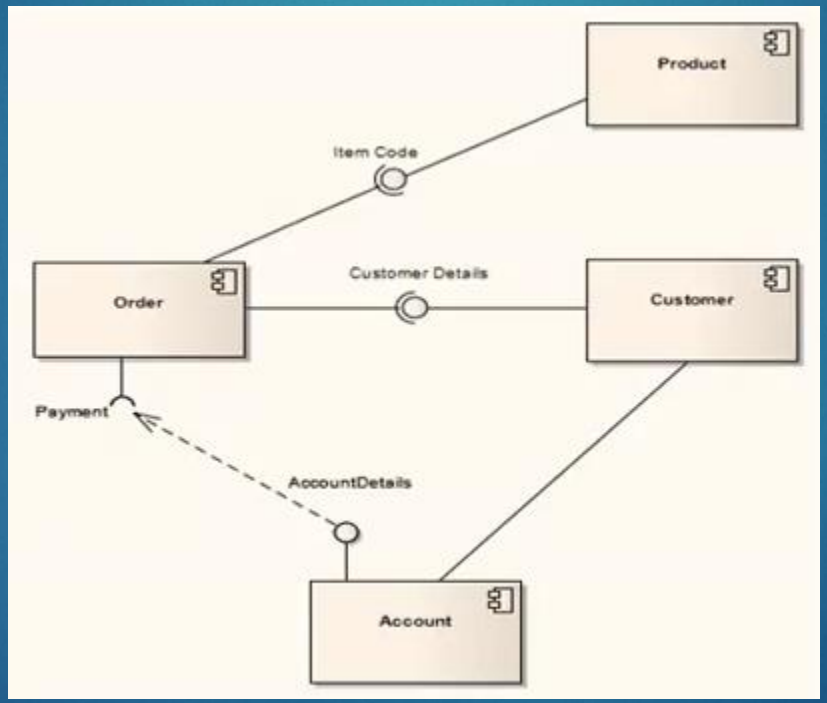
# Black-Box and White-Box Views





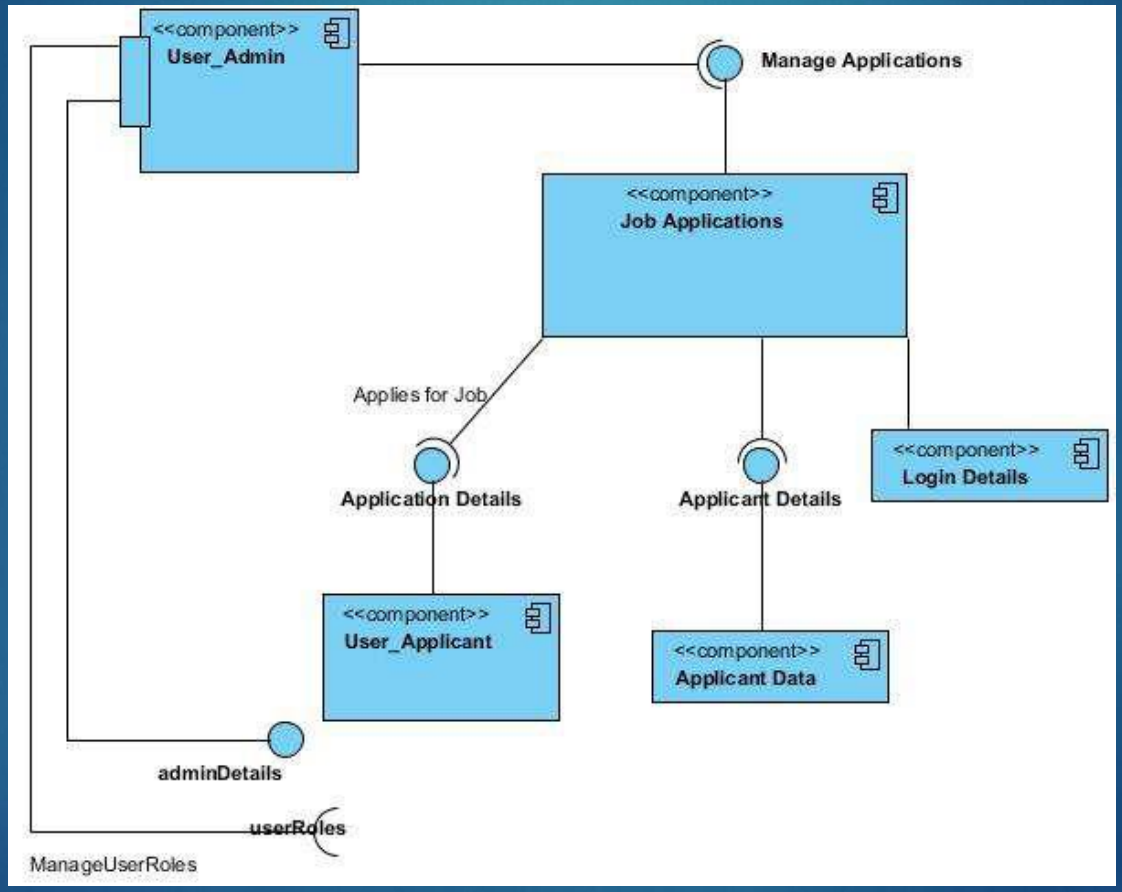
# Example of Component Diagram





# Sample Component Diagram





# Component Diagram Guidelines



- **Use Descriptive Names for Architectural Components**
  - Use Environment-Specific Naming Conventions for Detailed Design Components
  - Apply Textual Stereotypes to Components Consistently
- **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

# 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.*

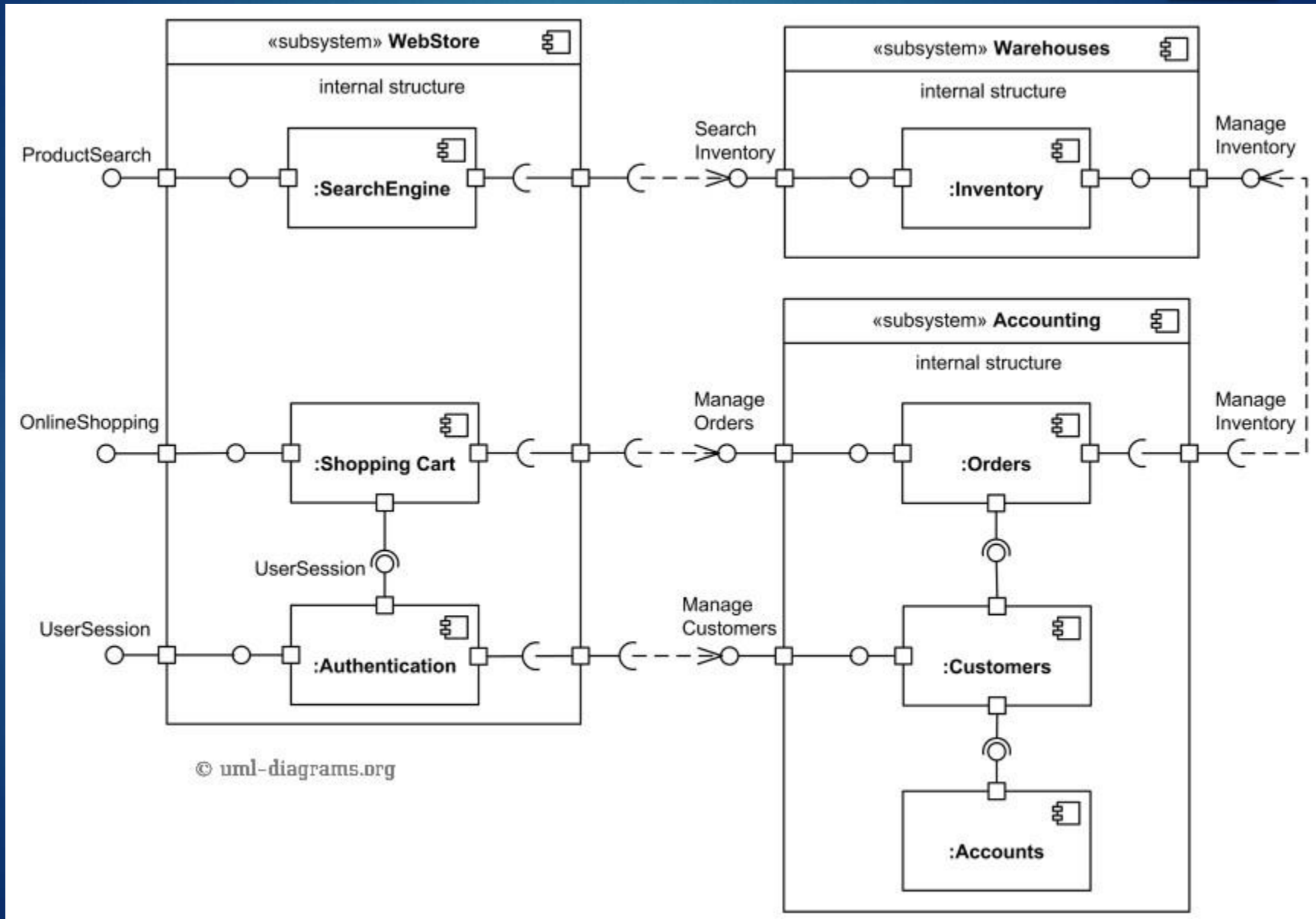
# Online Shopping - Components



- **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**.

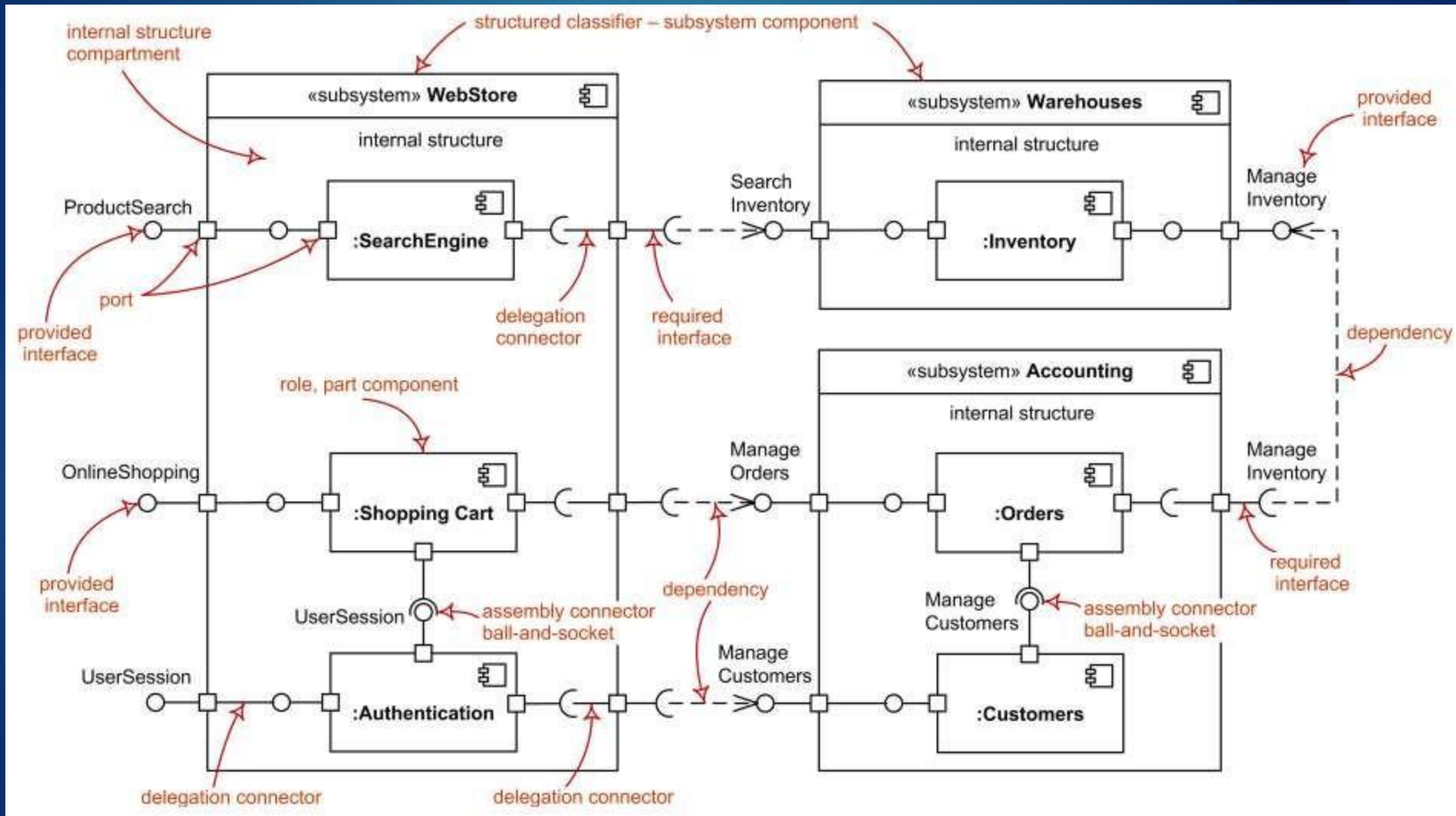
# Ex1: Component





# Ex1: UML Component





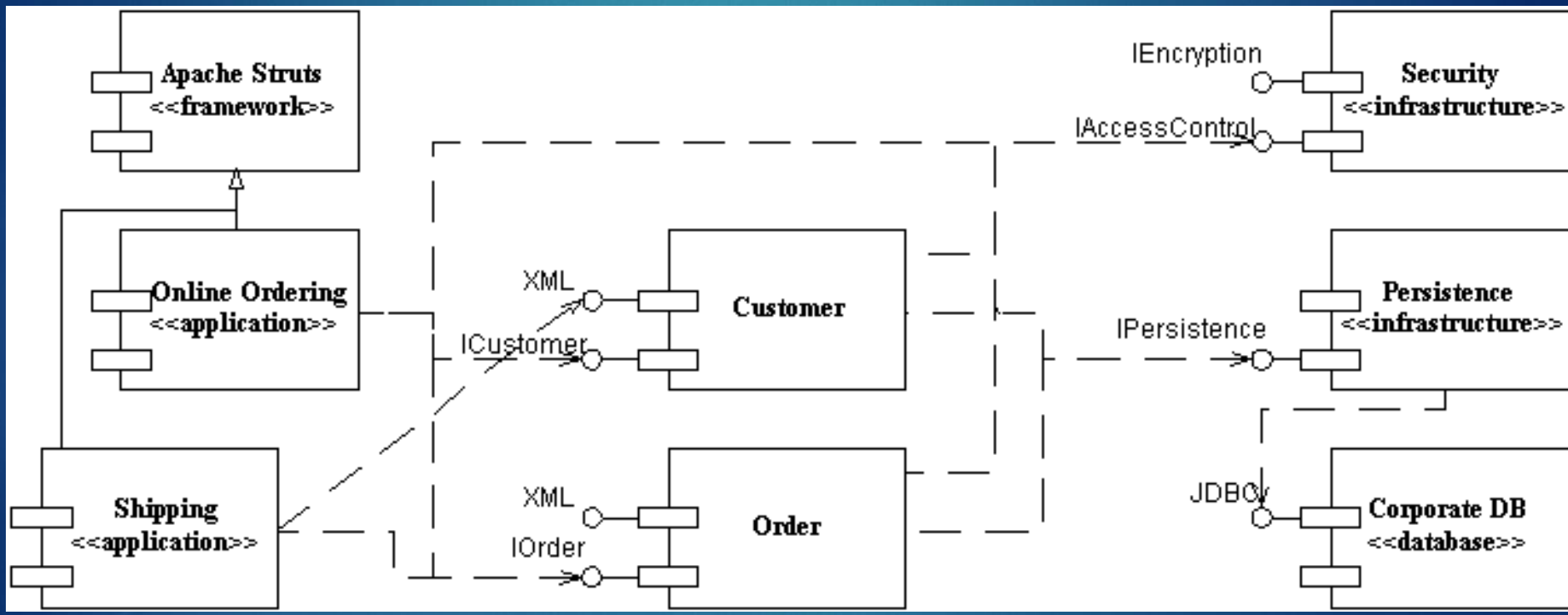
# Ex3: - Online ordering / shipping



- 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 **CorporateDB** component via JDBC. **Persistence** infrastructure provides IPersistence used by **Customers** and **Orders** to provide storage persistency for customer orders.

# Ex3: Component Diagram





# 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