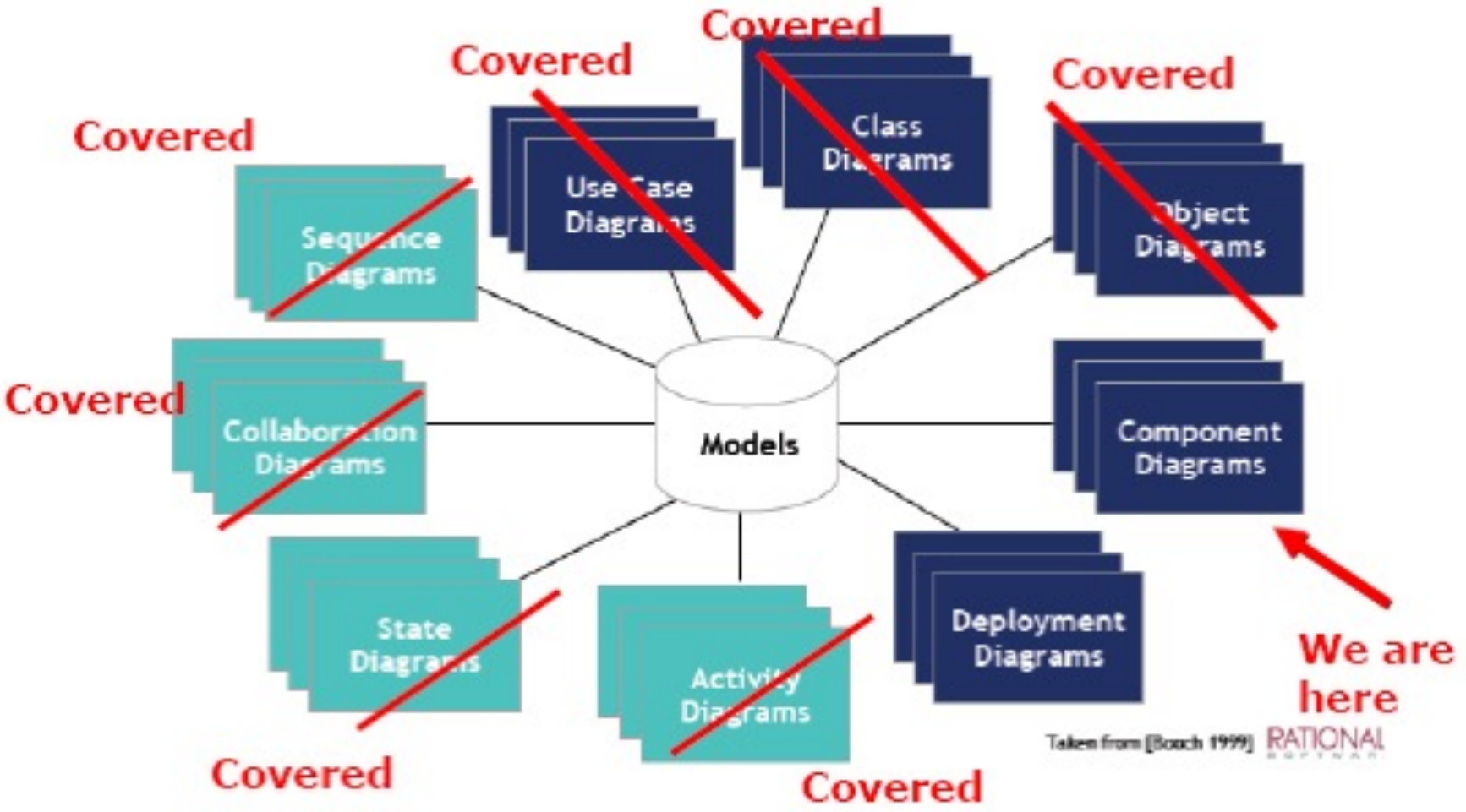
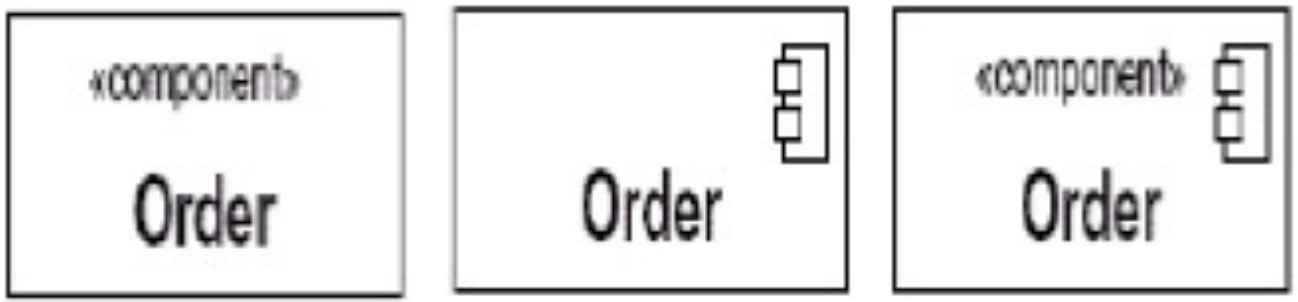
UML Diagrams



Component Diagrams

- The component diagram's main purpose is to show the **structural** relationships between the components of a system
- Component diagrams offer **architects** a natural format to begin modelling a solution
- Component diagrams allow an architect to verify that a system's required functionality is being implemented by components
- Developers find the component diagram useful because it provides them with a high-level**, architectural view** of the system that they will be building/implementing

Component Diagrams



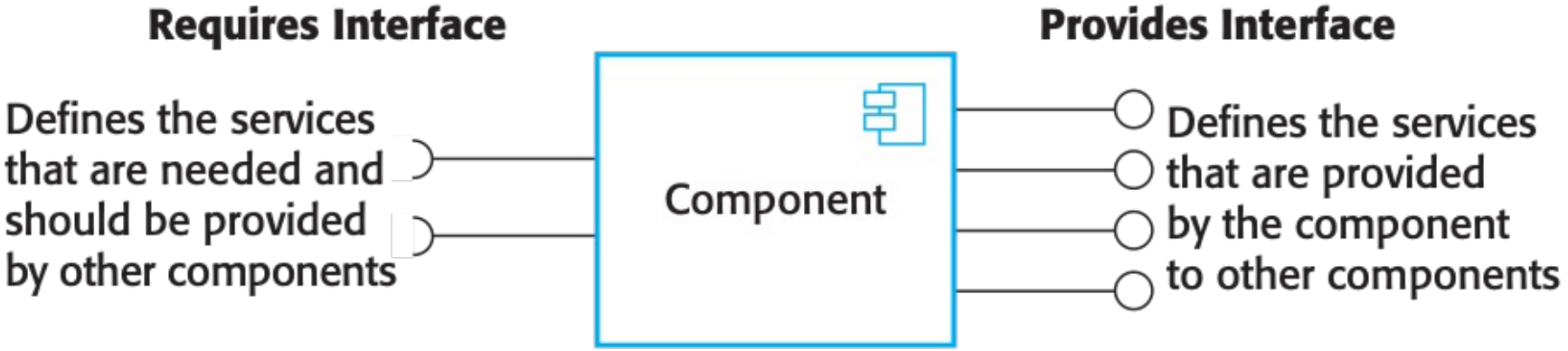
All they mean the same: a component Order

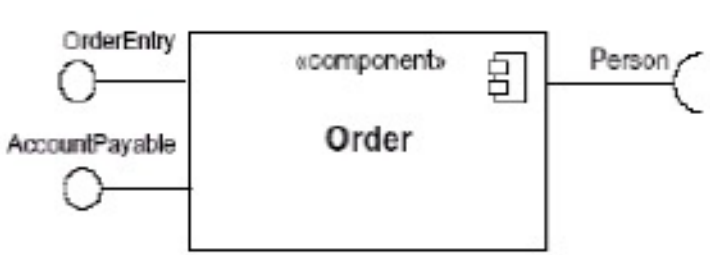
UML version 2.0

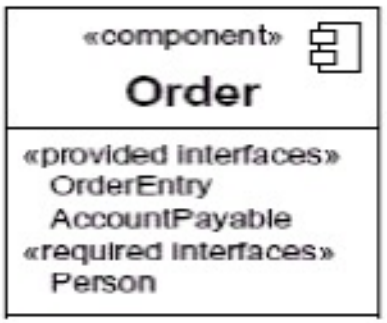
Component Diagrams

- Architectural **connection** in UML 2.0 is expressed primarily in terms of interfaces
- Interfaces are classifiers with operations but no attributes
- Components have **provided** and **required interfaces**
  - Component implementations are said to **realize** their provided interfaces
  - A provided and required interface can be connected if the operations in the latter are a subset of those in the former, and the signatures of the associated operations are '**compatible**'
- **Ports** provide access between external interfaces and internal structure of components
- UML components can be used to model complex architectural connectors (like a CORBA ORB)

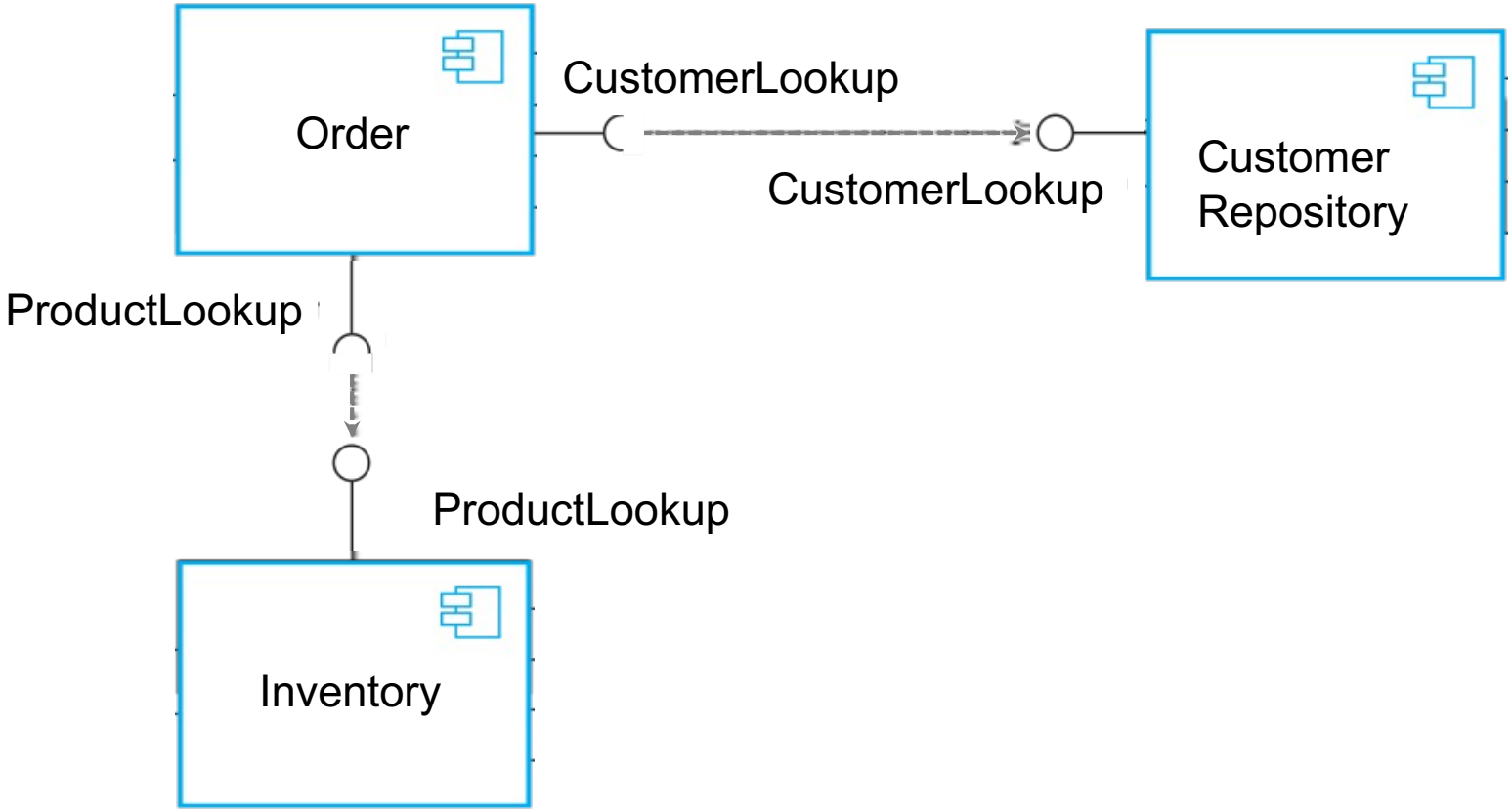
Required/Provide Interface





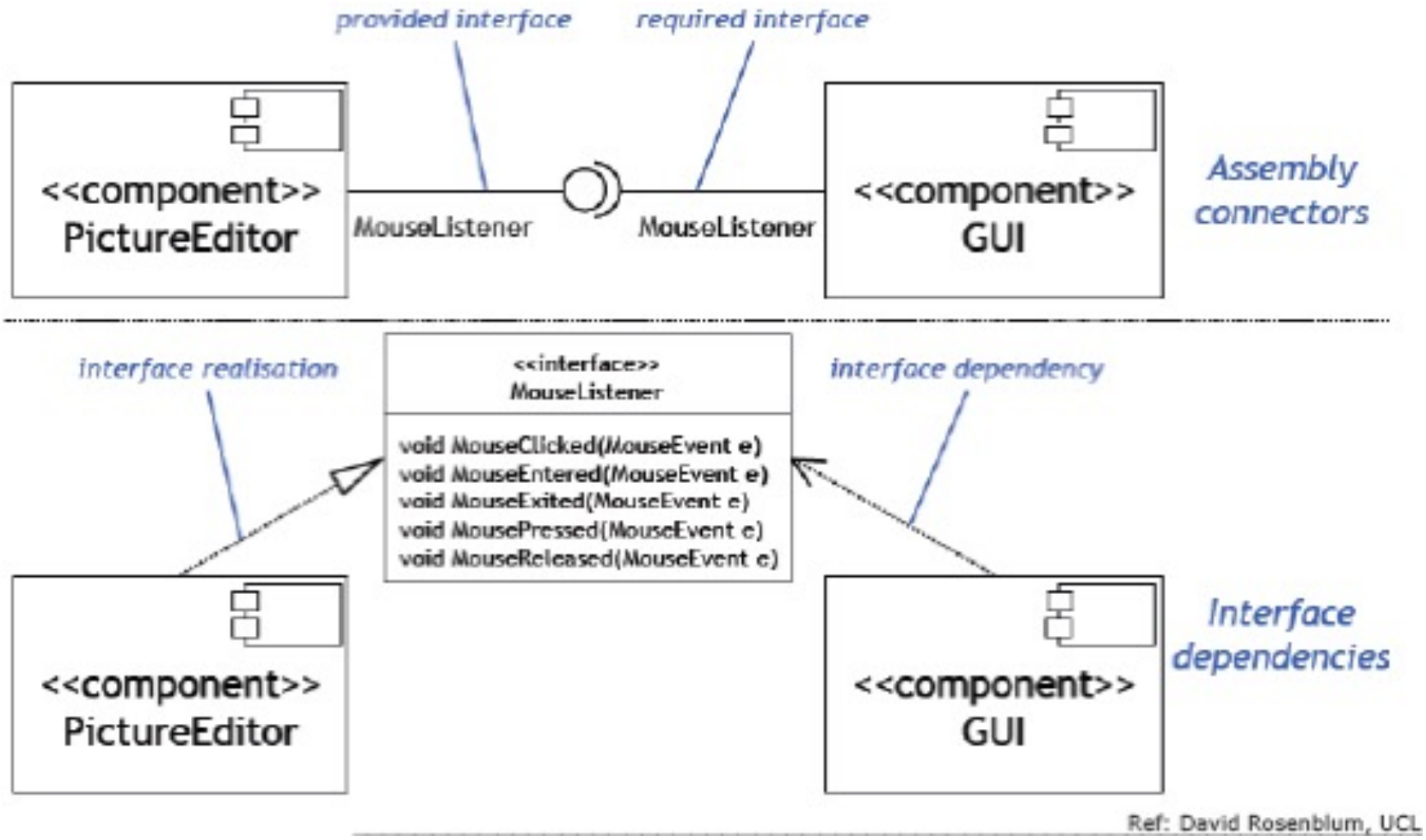


Component Diagrams

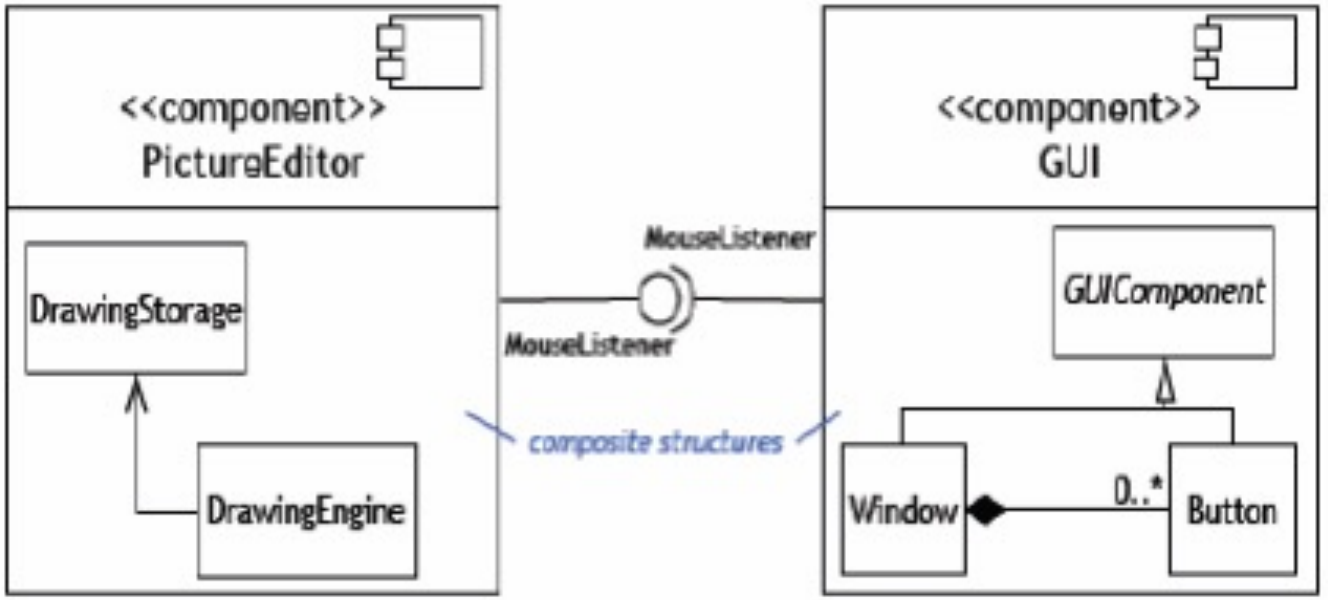


showing a component's relationship with other components, the lollipop and socket notation must also include a dependency arrow (as used in the class diagram). On a component diagram with lollipops and sockets, note that the dependency arrow comes out of the consuming (requiring) socket and its arrow head connects with the provider's lollipop

Component Diagrams



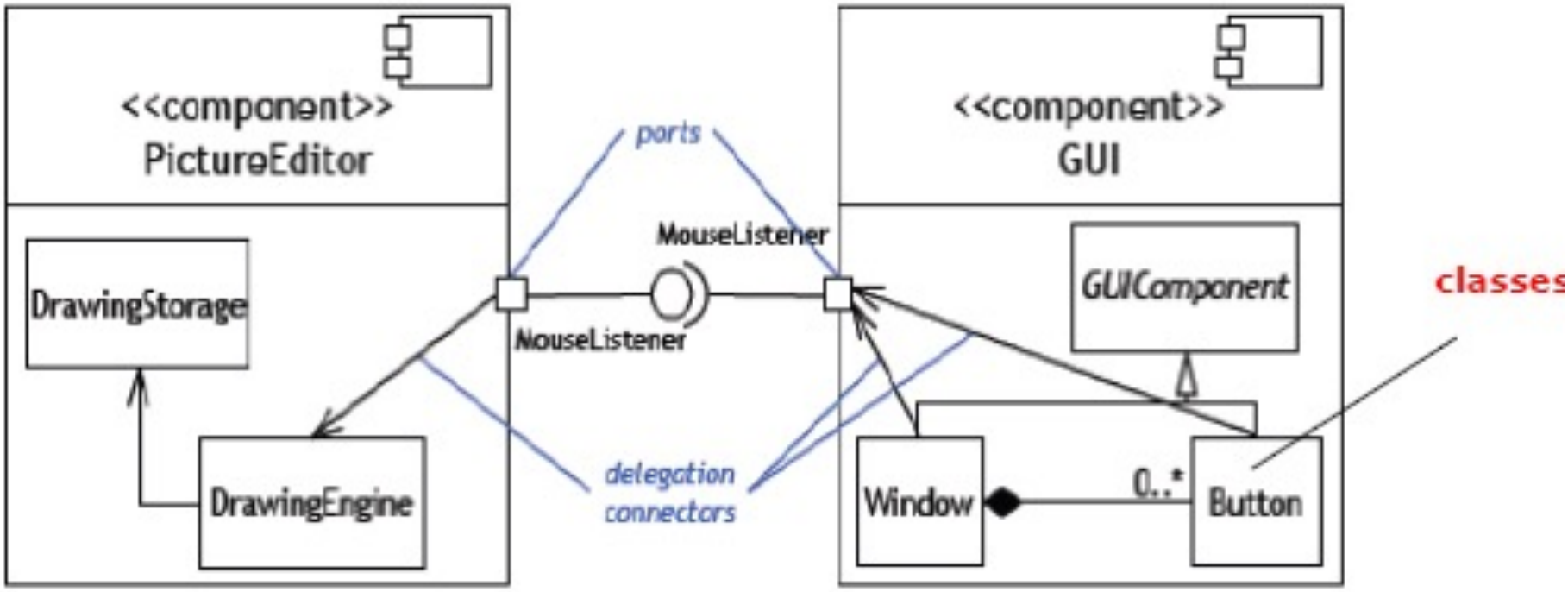
Composite Structure in Component Diagrams



Ref: David Rosenblum, UCL

A composite structure depicts the internal realisation of component functionality

Ports

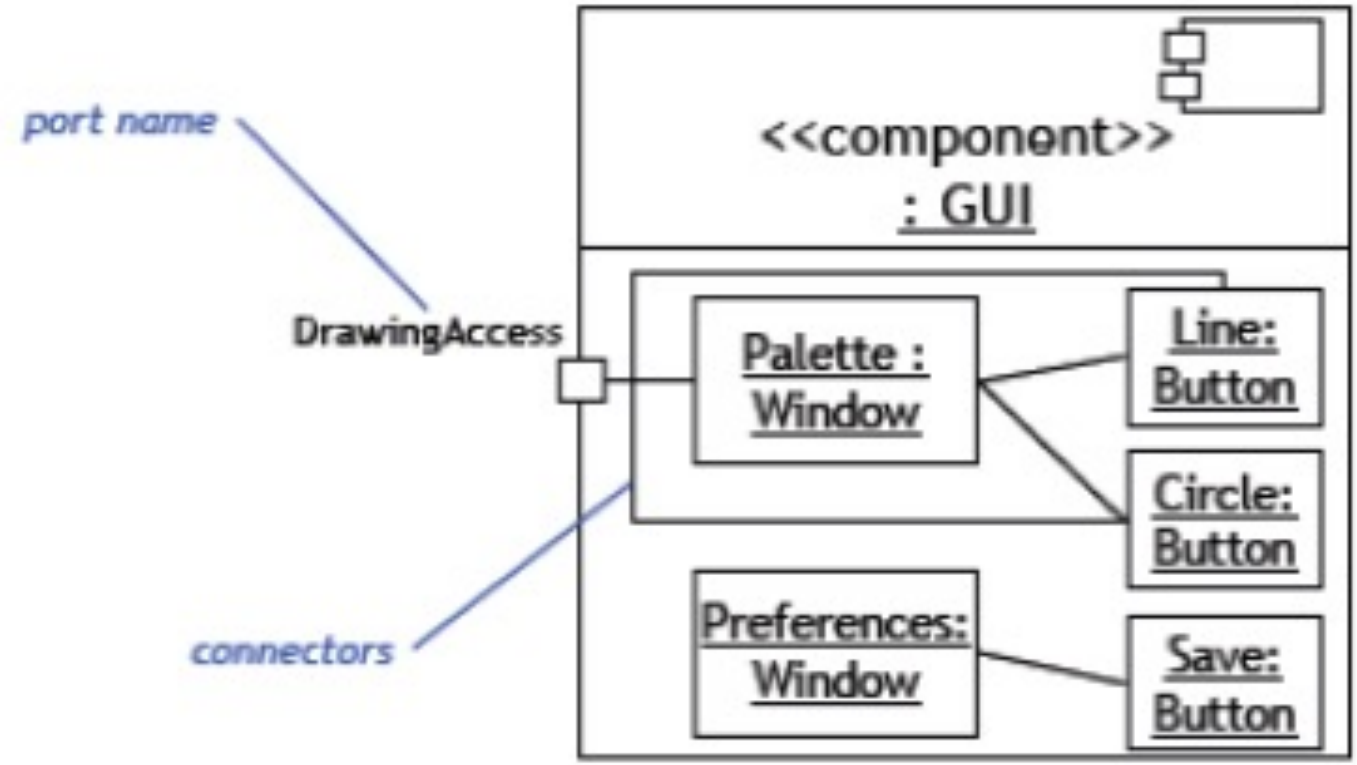


Ref: David Rosenblum, UCL

The ports and connectors specify how component interfaces are mapped to internal functionality

Note that these 'connectors' are rather limited, special cases of the ones in software architectures

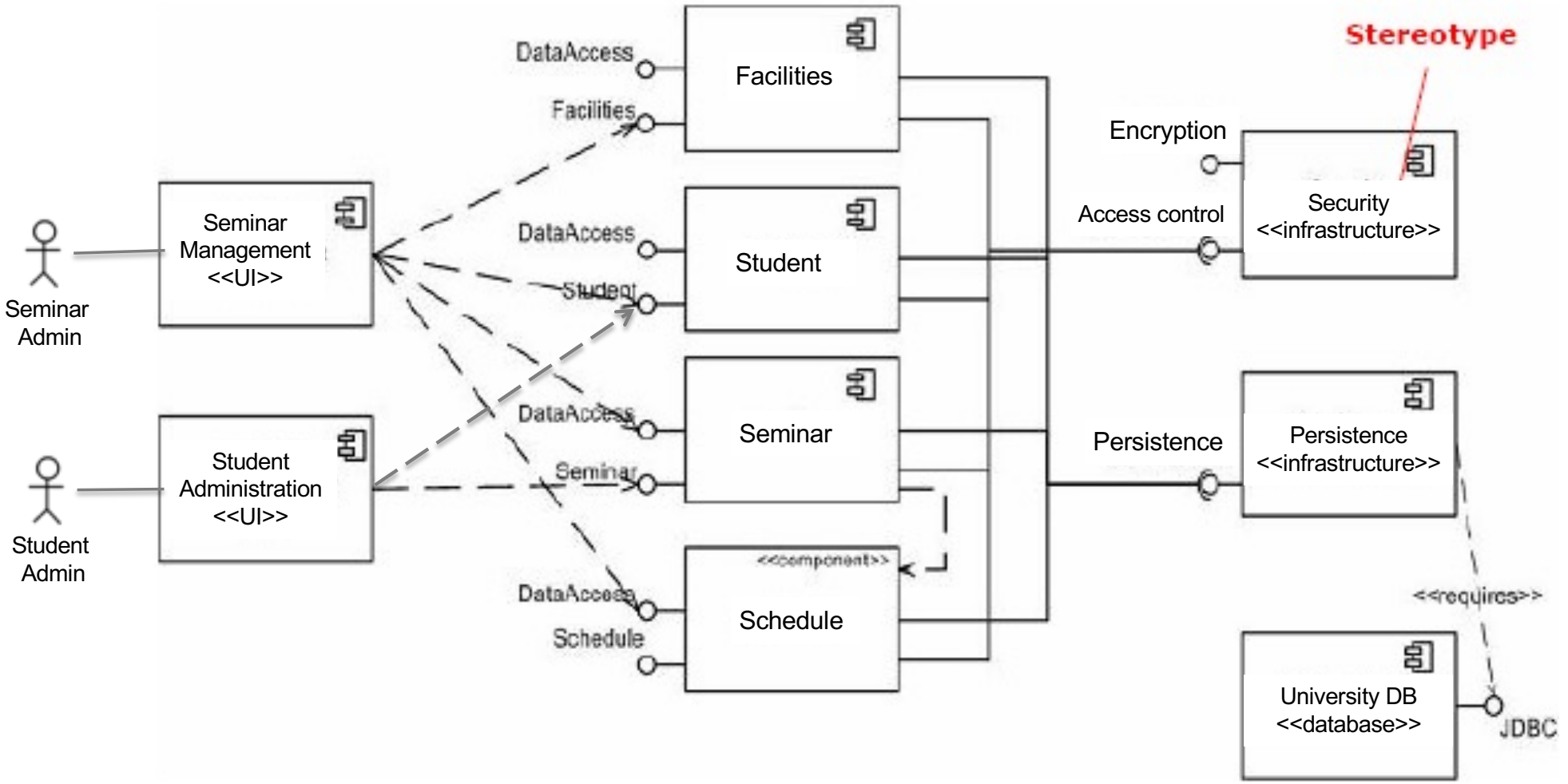
Ports



Ref: David Rosenblum, UCL

Connectors and ports also can be used to specify structure of component ***instantiations***

Example


Stereotype
Facilities
DataAccess
Facilities
Seminar Management <<UI>>
Seminar Admin
Student
DataAccess
Student
Seminar
DataAccess
Seminar
Student Administration <<UI>>
Student Admin
<<component>>
Schedule
DataAccess
Schedule
Encryption
Access control
Security <<infrastructure>>
Persistence
Persistence <<infrastructure>>
<<requires>>
University DB <<database>>
JDBC

Componentization Guidelines

"Keep components *cohesive*". i.e a component should implement a single, related set of functionality.

This may be the user interface logic for a single user application, business classes comprising a large-scale domain concept, or technical classes representing a common infrastructure concept.

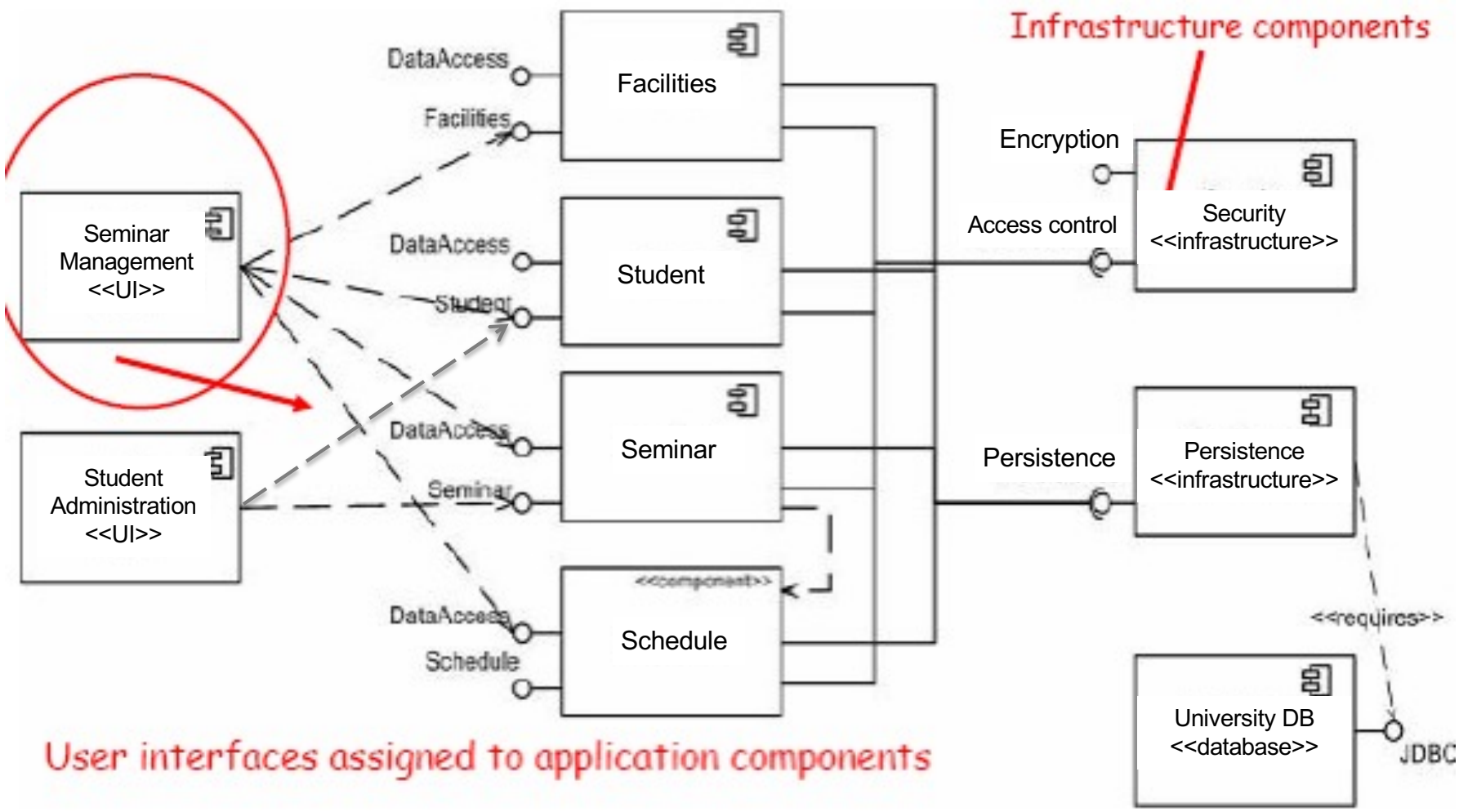
User *interface* classes assigned as application components.

User interface classes, those that implement screens, pages, or reports, as well as those that implement "glue logic".

Assign common technical classes to *infrastructure components*.

Technical classes, e.g. that implement system-level services such as security, persistence, or middleware should be assigned to components which have the *infrastructure stereotype*.

Example


Infrastructure components
DataAccess
Facilities
Facilities
Seminar Management <<UI>>
DataAccess
Student
Student
Student Administration <<UI>>
DataAccess
Seminar
Seminar
DataAccess
Schedule
Schedule
Encryption
Access control
Security <<infrastructure>>
Persistence
Persistence <<infrastructure>>
<<requires>>
University DB <<database>>
JDBC
User interfaces assigned to application components

Componentization Guidelines

Assign *hierarchies* to the same component.

99.9% of the time it makes sense to assign all of the classes of a hierarchy, either an *inheritance hierarchy* or an aggregation/*composition hierarchy*, to the same component.

Identify business domain components.

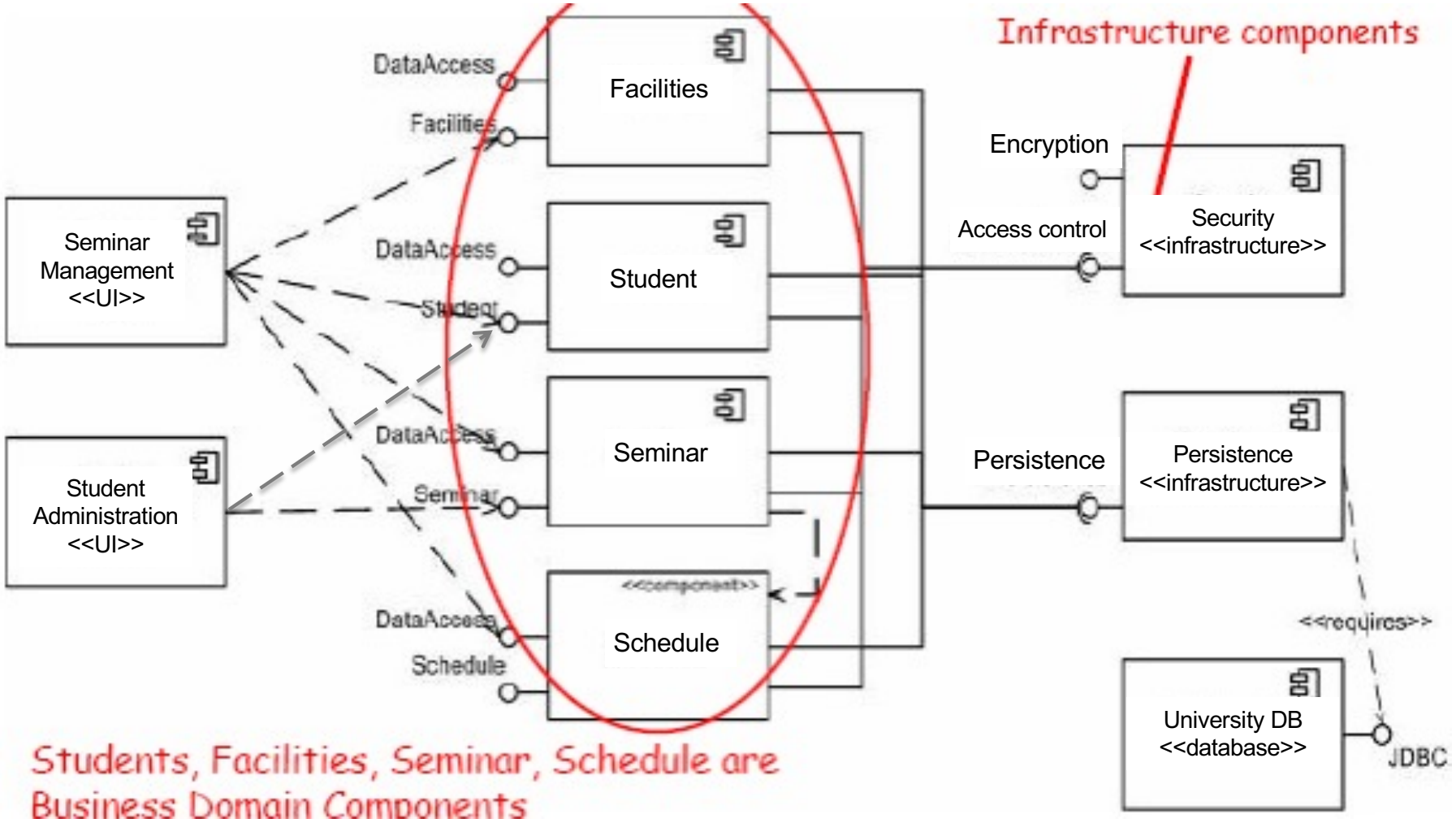
Because you want to minimize network traffic to reduce the response time of your application, you want to design your business domain components in such a way that most of the *information flow* occurs *within* the components and not *between* them.

Business domain components = business services

Identify the "collaboration type" of business classes.

Once you have identified the collaboration type of each class (e.g. server, client or both), you can start identifying potential business domain components.

Example


Infrastructure components
DataAccess
Facilities
Facilities
Seminar Management <<UI>>
DataAccess
Student
Student
Student Administration <<UI>>
DataAccess
Seminar
Seminar
DataAccess
Schedule
Schedule
Encryption
Access control
Security <<infrastructure>>
Persistence
Persistence <<infrastructure>>
<<requires>>
University DB <<database>>
JDBC
Students, Facilities, Seminar, Schedule are Business Domain Components

Componentization Guidelines

Highly coupled classes grouped in the same component.

When two classes collaborate frequently, this is an indication they should be in the same domain business component to reduce the network traffic between the two classes.

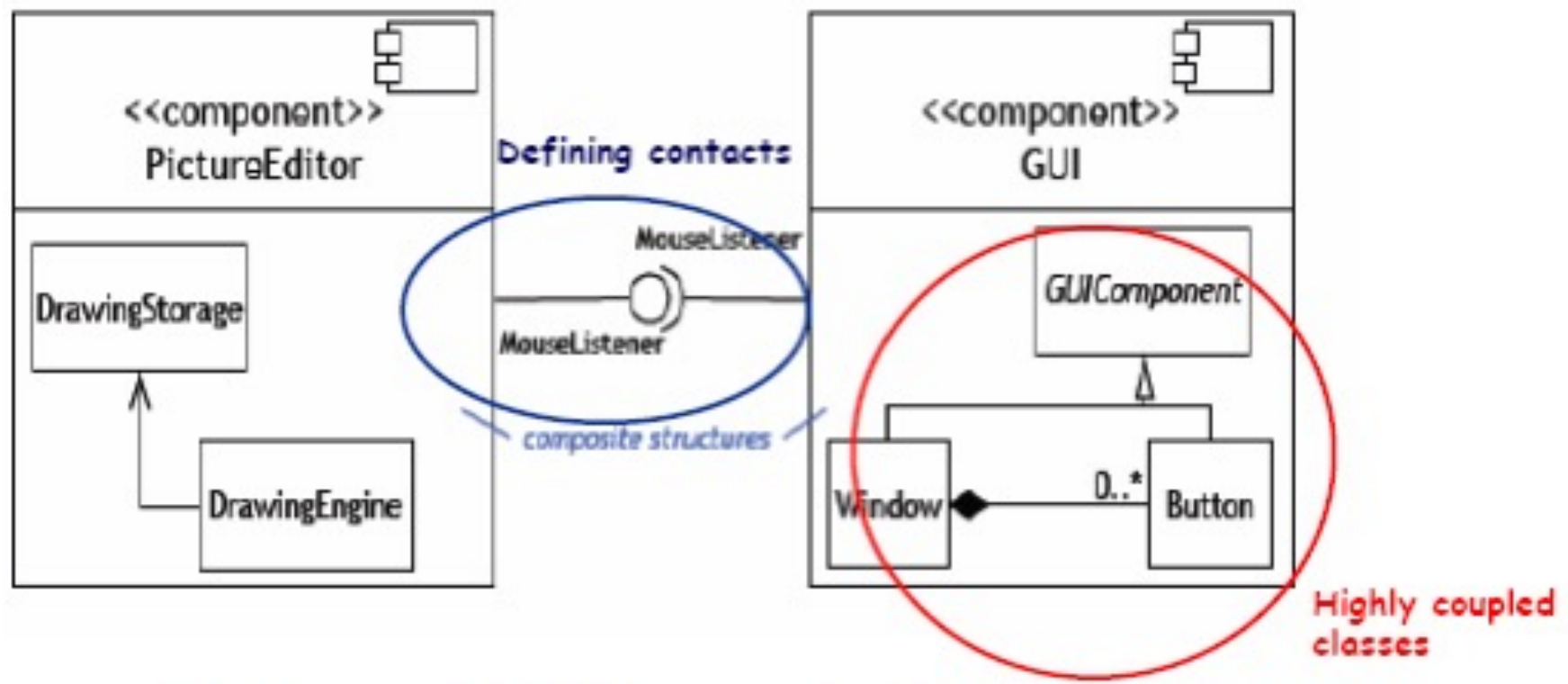
Minimize the size of the *message flow* between components.

If you have domain components, one as a server to only the other as a client, you may decide to combine or merge the two components.

Define component *contracts,* as interfaces.

Each component will offer services to its client components, each such service is a component contract.

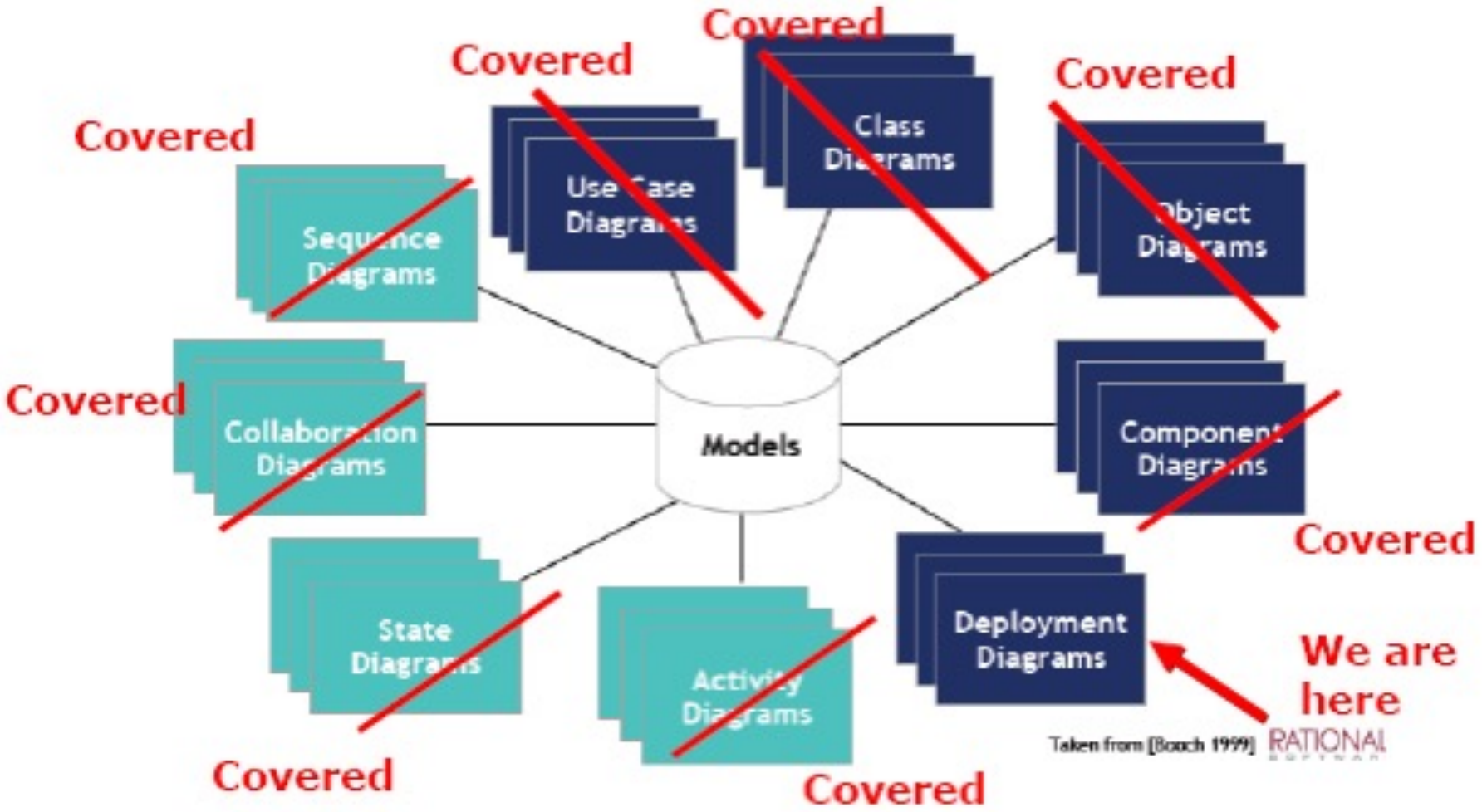
Example



Highly coupled classes belong in the same component

Ref: David Rosenblum, UCL

UML Diagrams



Deployment Diagram

Models the *run-time* configuration in a static view and visualizes the distribution of components in an application

It helps map between software components and hardware

A component is deployed part of the *software system architecture*

In most cases, it involves modelling the *hardware* configurations together with the *software* components that run on

Deployment Diagram

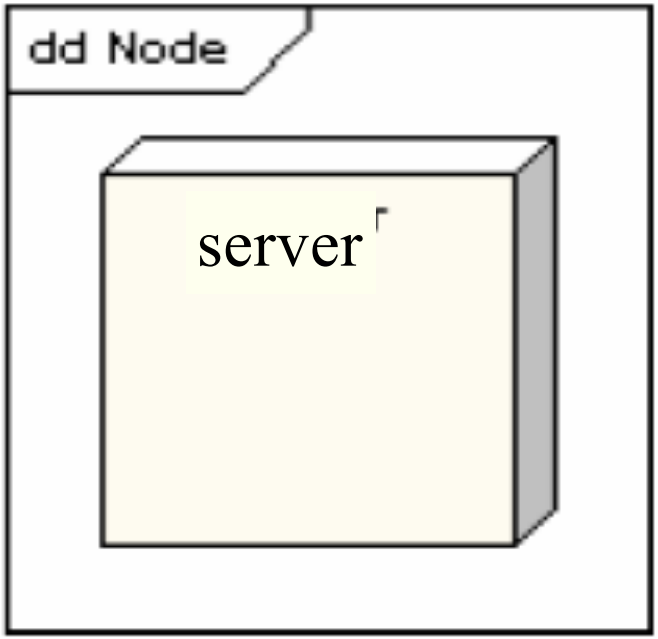
Deployment diagram depicts a *static view* of the run-time configuration of processing nodes and the components that run on those nodes
  Node: server, client etc.

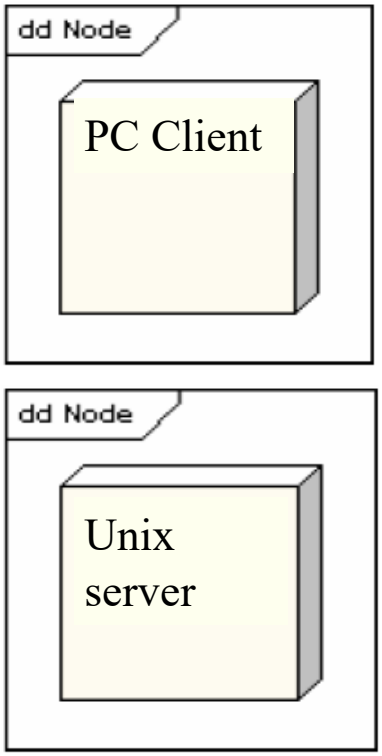
Deployment diagrams show the *hardware* for your system, the *software* that is installed on that hardware, and the *middleware* used to connect the disparate *machines* to one another!

Visualizes the distribution of components in an application, it shows the configuration of the *hardware* elements (nodes) and shows how software elements and artifacts are mapped onto those nodes.

Node

A Node is either a hardware or software element. It is shown as a three-dimensional box shape, as shown below.

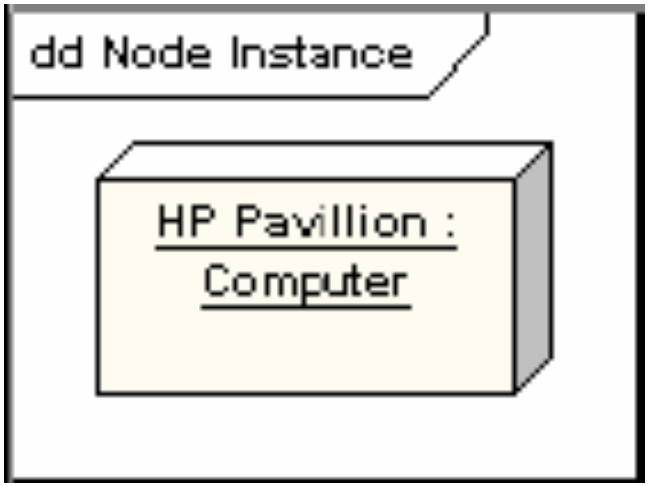




Node Instance

An **instance** can be distinguished from a node by the fact that its name is underlined and has a colon before its base node type. An instance may or may not have a name before the colon.

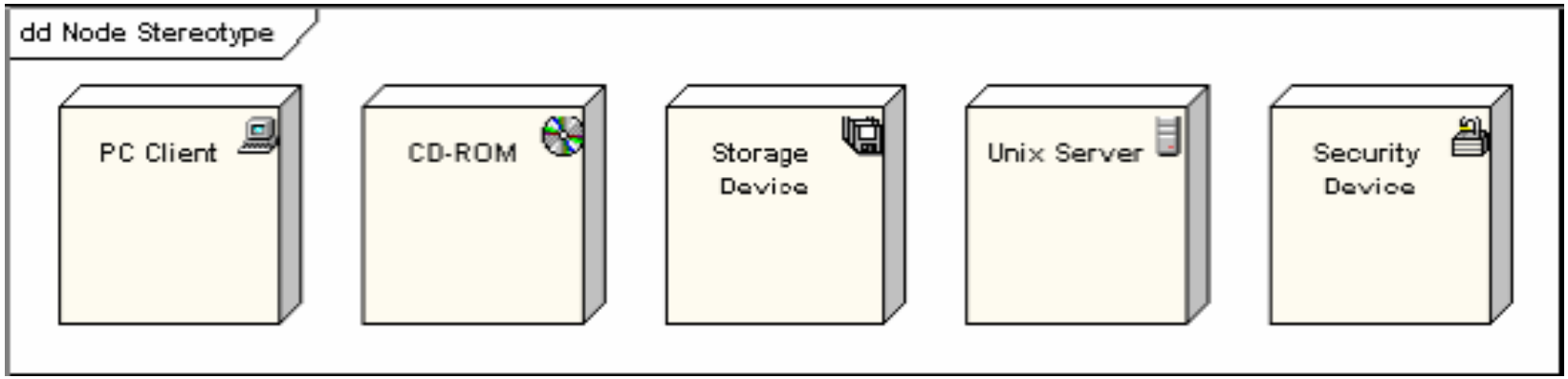
The following diagram shows a named instance of a computer



Node Sterotypes

In UML, a number of standard **stereotypes** are provided for nodes, namely «cdrom», «cd-rom», «computer», «disk array», «pc», «pc client», «pc server», «secure», «server», «storage», «unix server», «user pc».

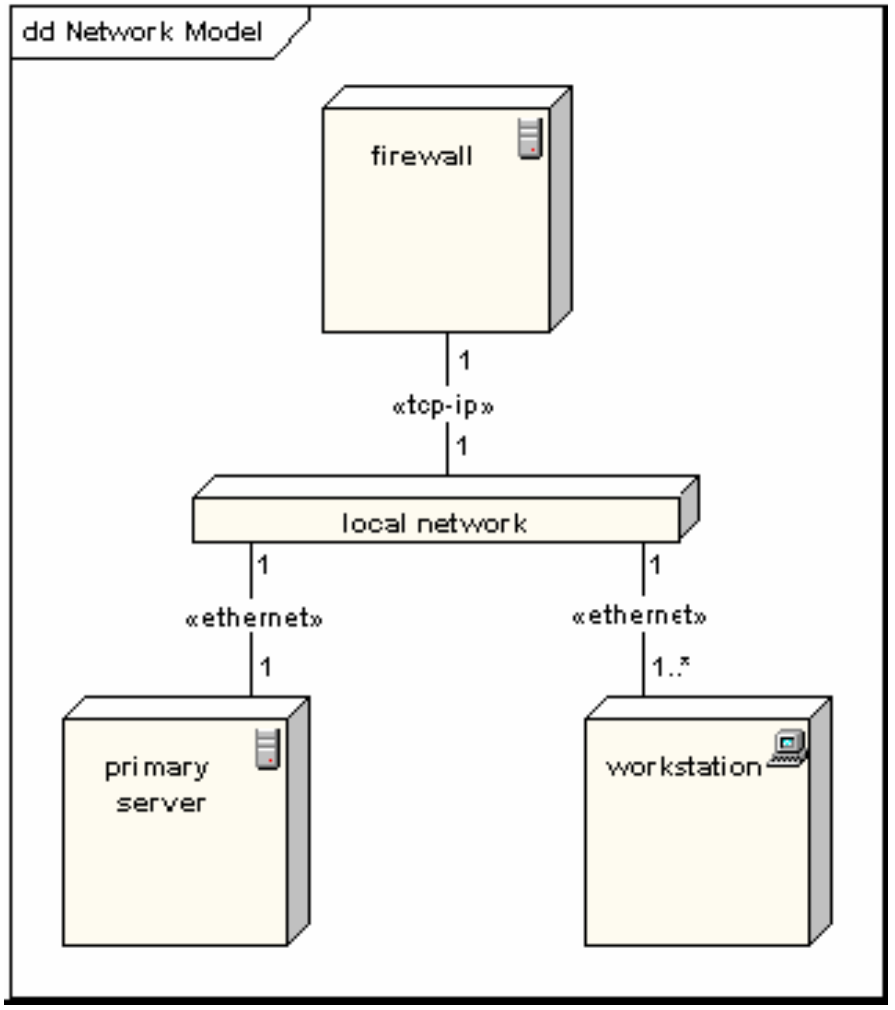
These will display an appropriate icon in the top right corner of the node symbol



Association

In deployment diagram, an association represents a communication path between nodes.

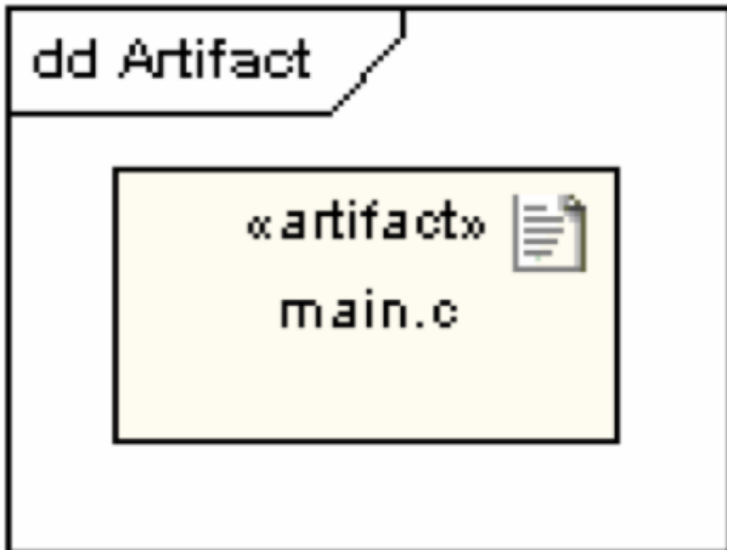
The diagram shows a deployment diagram for a network, depicting network protocols as stereotypes, and multiplicities at the association ends.



Artifact

An **artifact** is a product of the software development process. That may include process models (e.g. use case models, design models etc.), source files, executable files, design documents, test reports, prototypes, user manuals, etc.

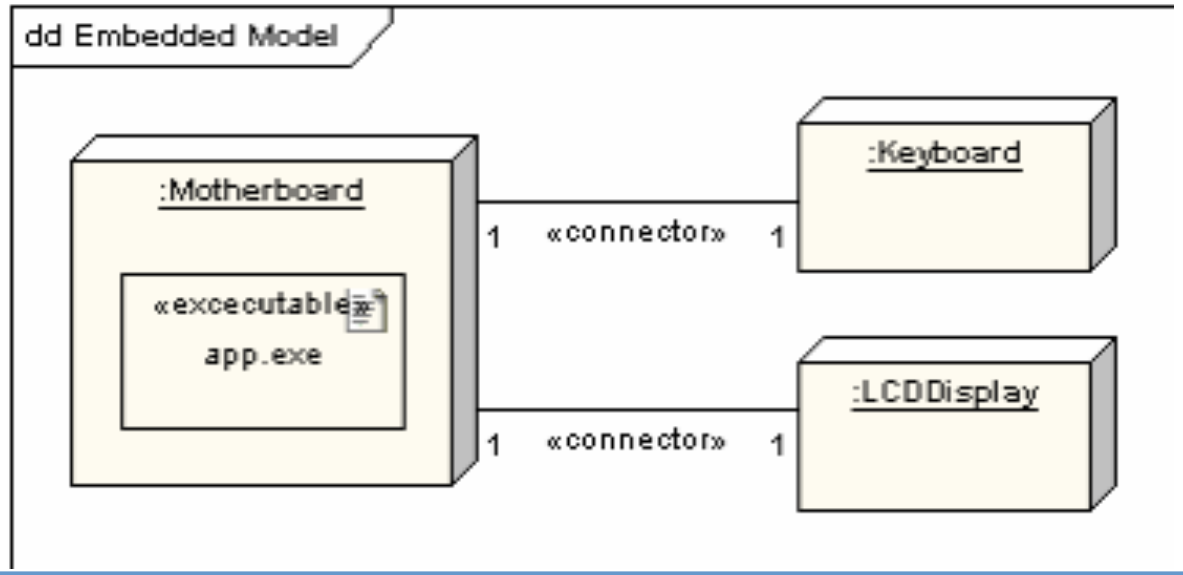
An artifact is denoted by a rectangle showing the artifact name, the «artifact» keyword and a document icon, as shown.



Node as Container

A node can contain other elements, such as components or artifacts.

The diagram shows a deployment diagram for part of an embedded system, depicting an executable artifact as being contained by the motherboard node.



Architectural Style vs Architecture

Architectural Style:

A pattern for a system layout

Software Architecture:

Instance of an architectural style.

Examples of Architectural Styles

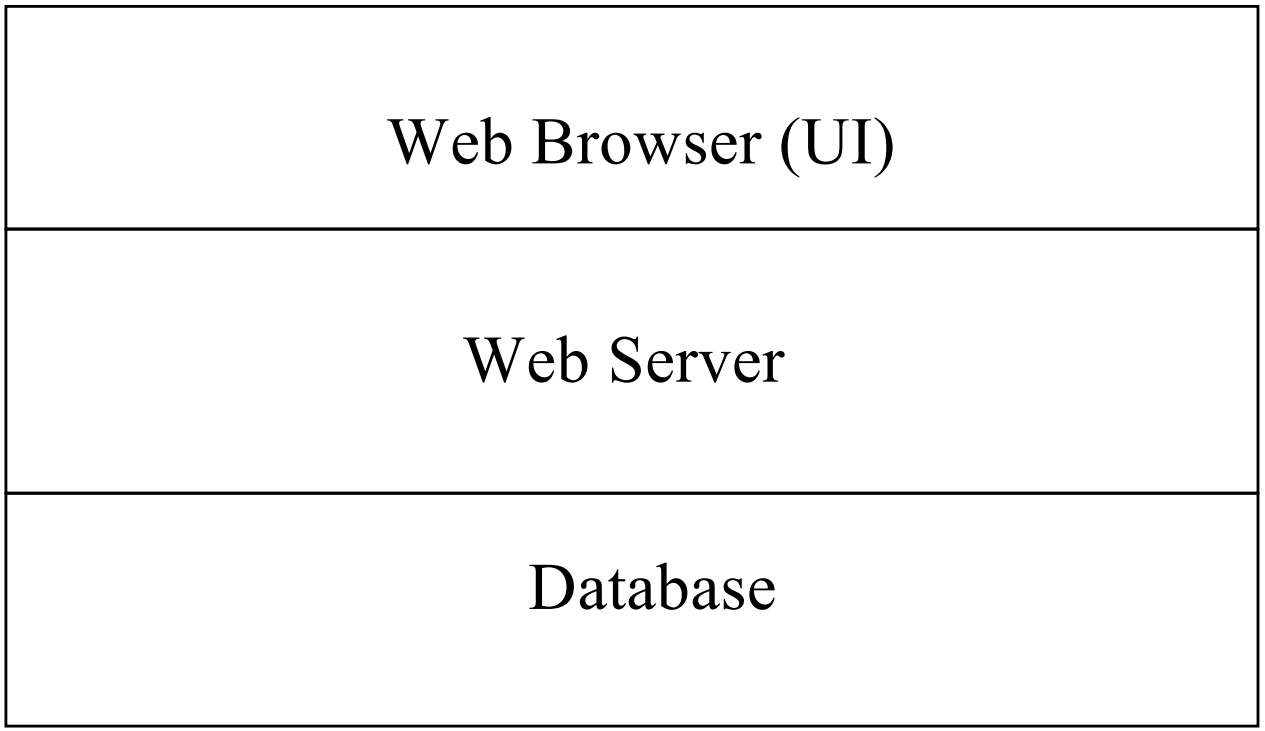
- Layered Architectural style
  - Service-Oriented Architecture (SOA)
- Client/Server
- Peer-To-Peer
- Three-tier, Four-tier Architecture
- Repository
- Model-View-Controller
- Pipes and Filters

Example: Layered Architecture

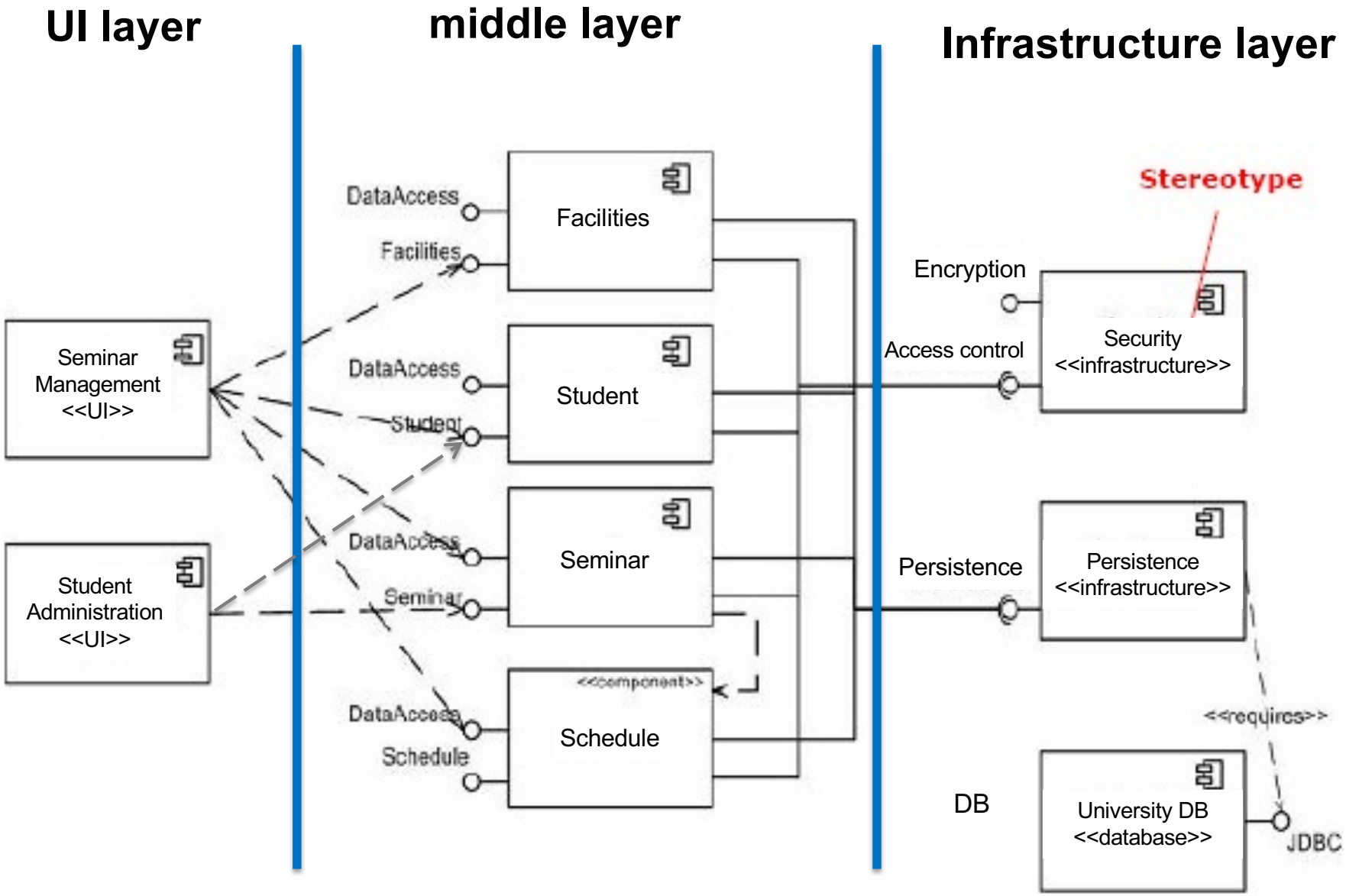
Simplified architecture diagram

3-Layered Architecture:

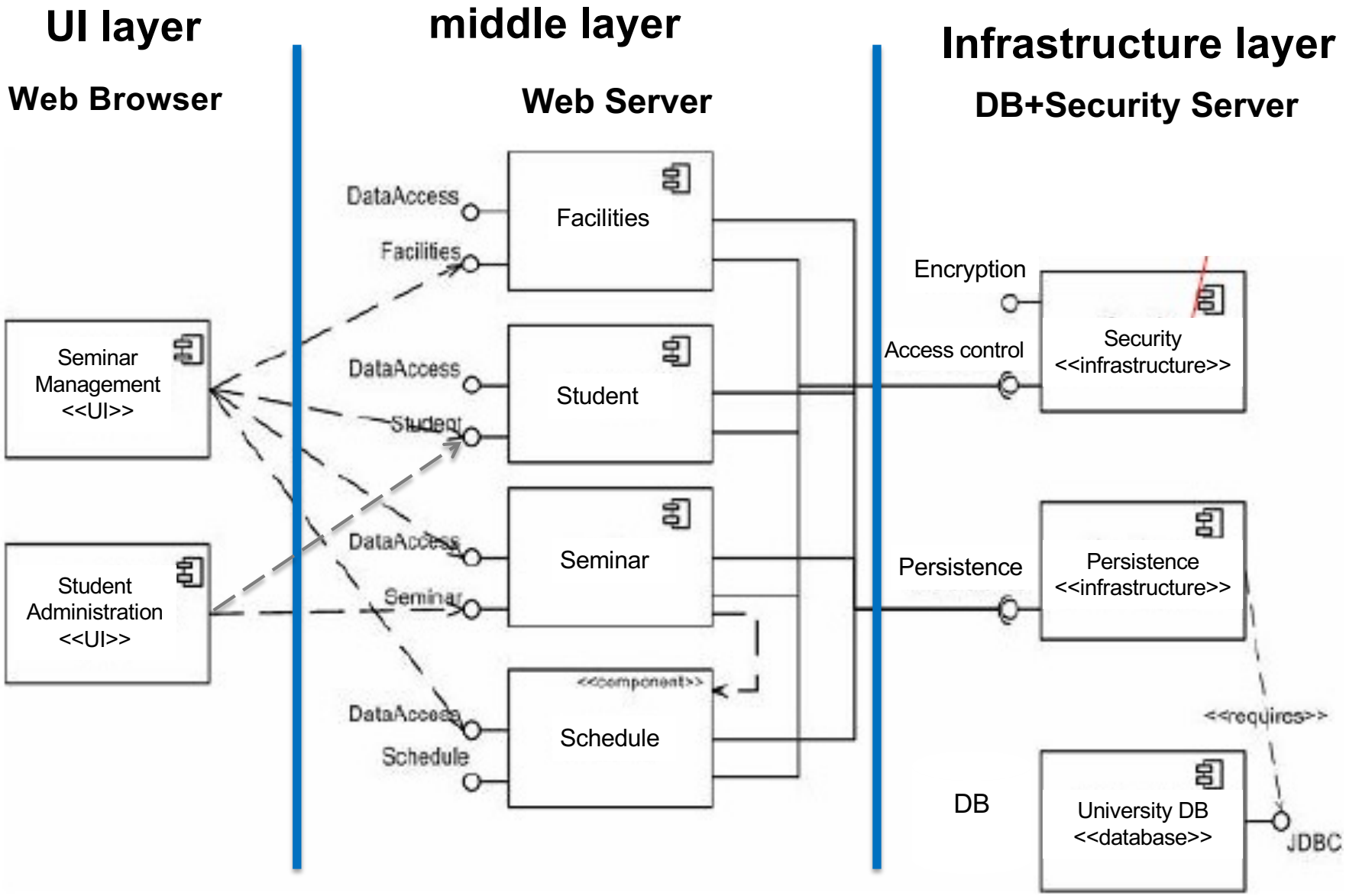
often used for the development of Web-based Applications



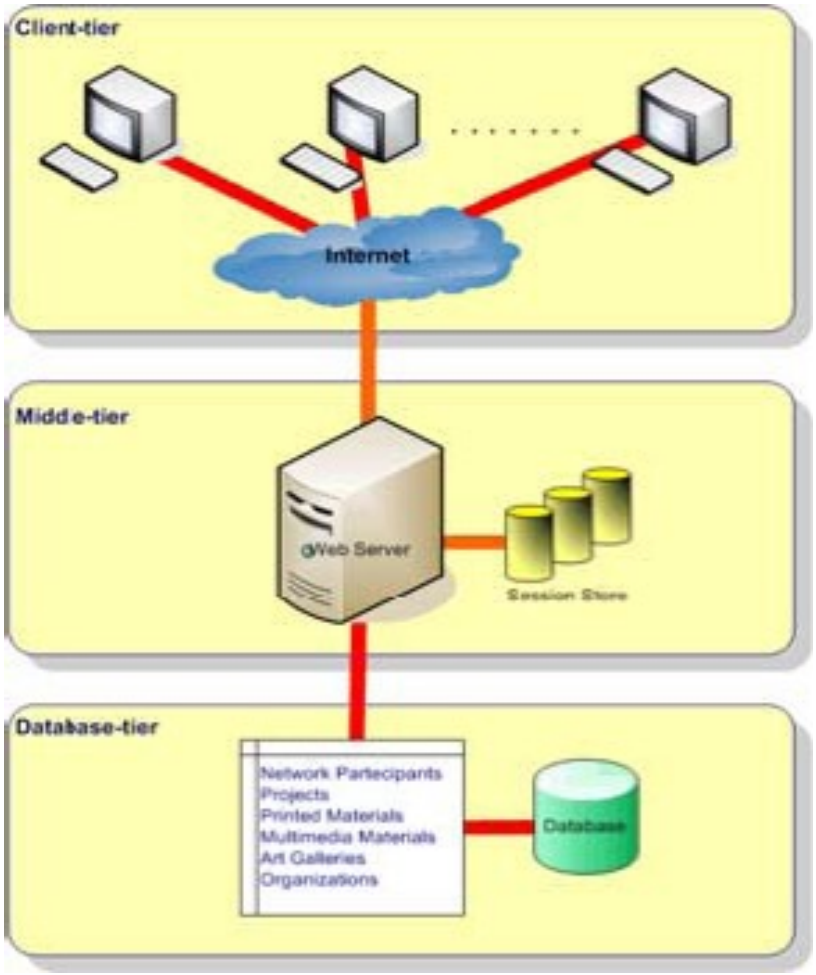
Example: Layered Architecture

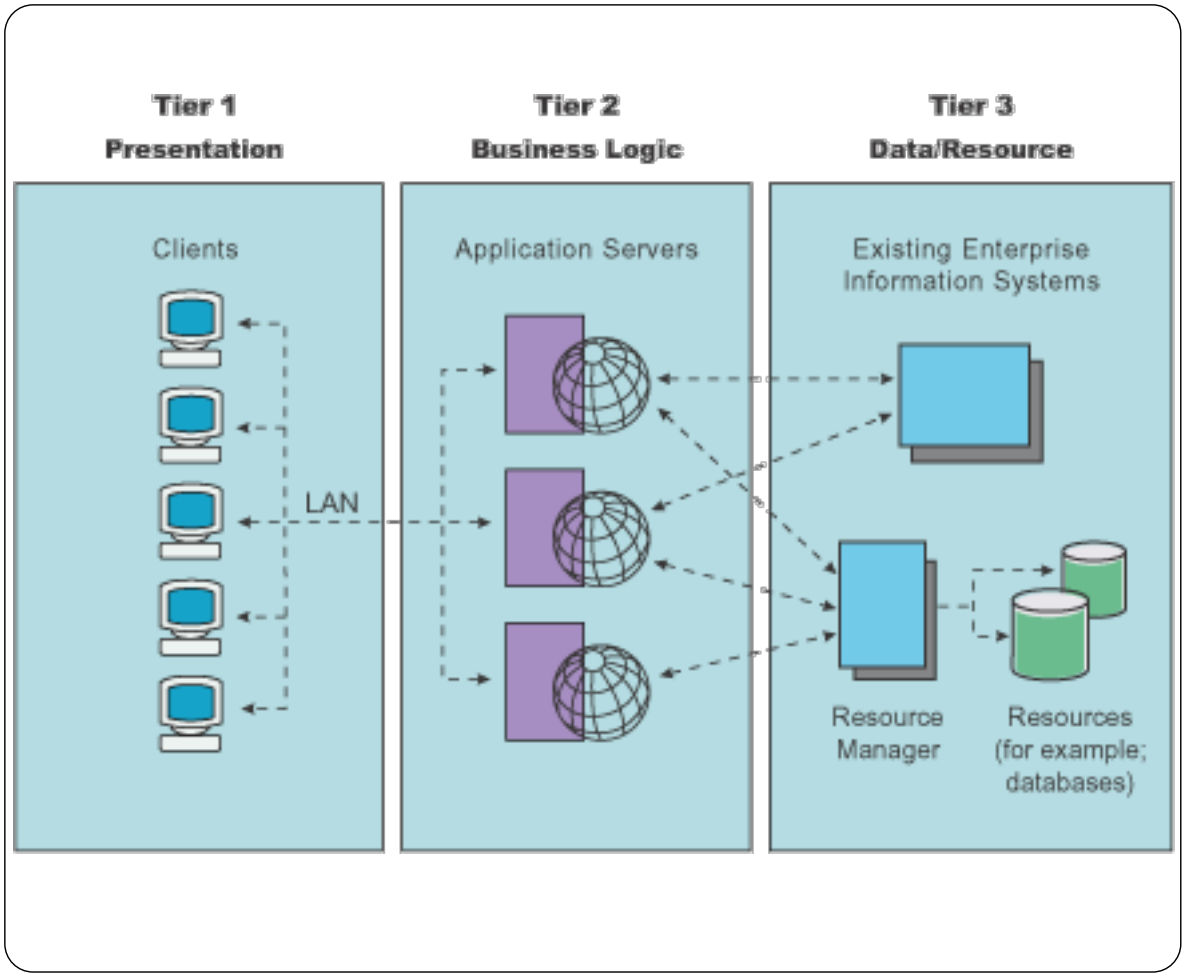


Example: Layered Architecture (Web App)



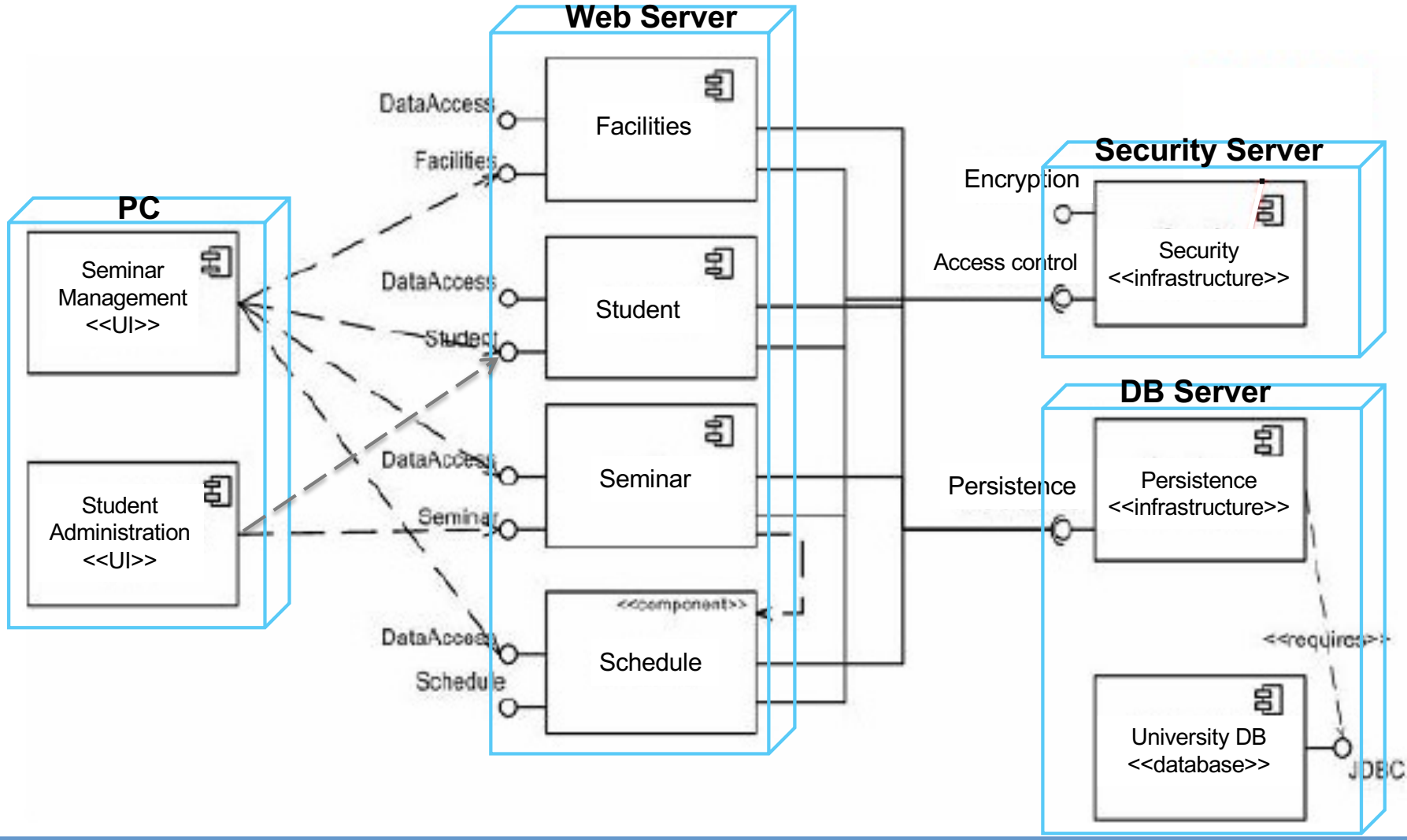
Example of three-tiers architectures



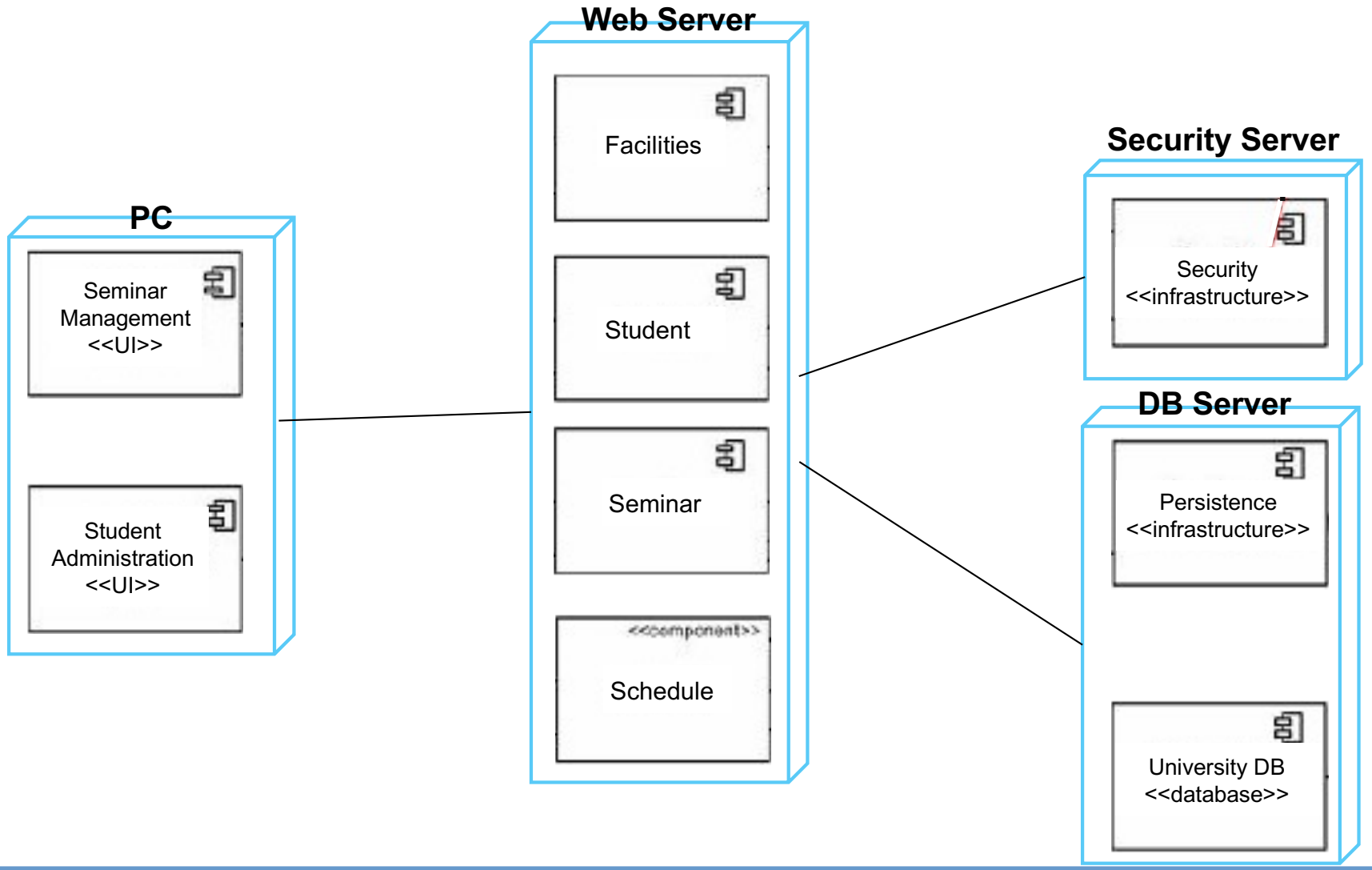


Many of real life web applications have three tier architectures

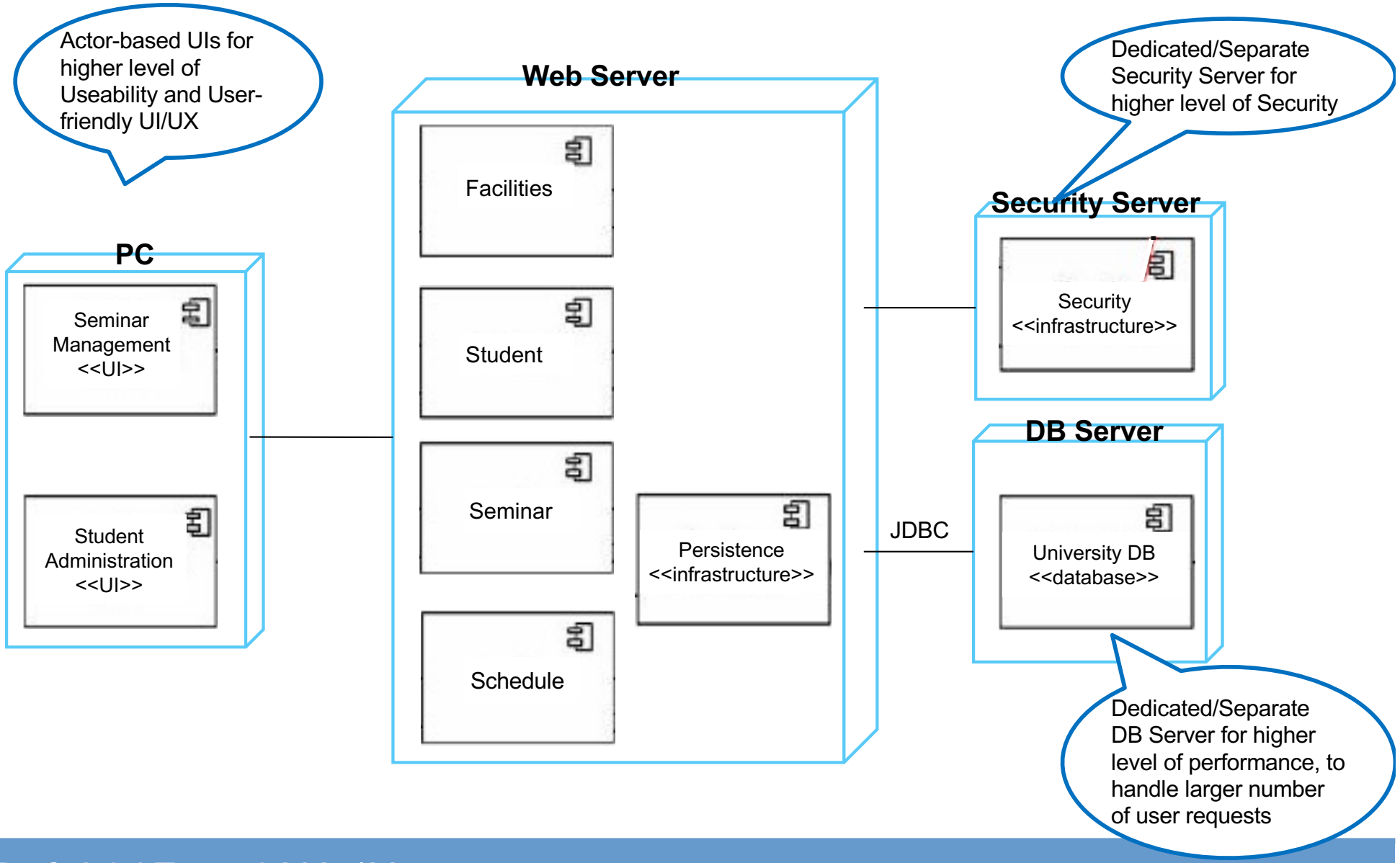
Component-Deployment mapping: as a Web Application



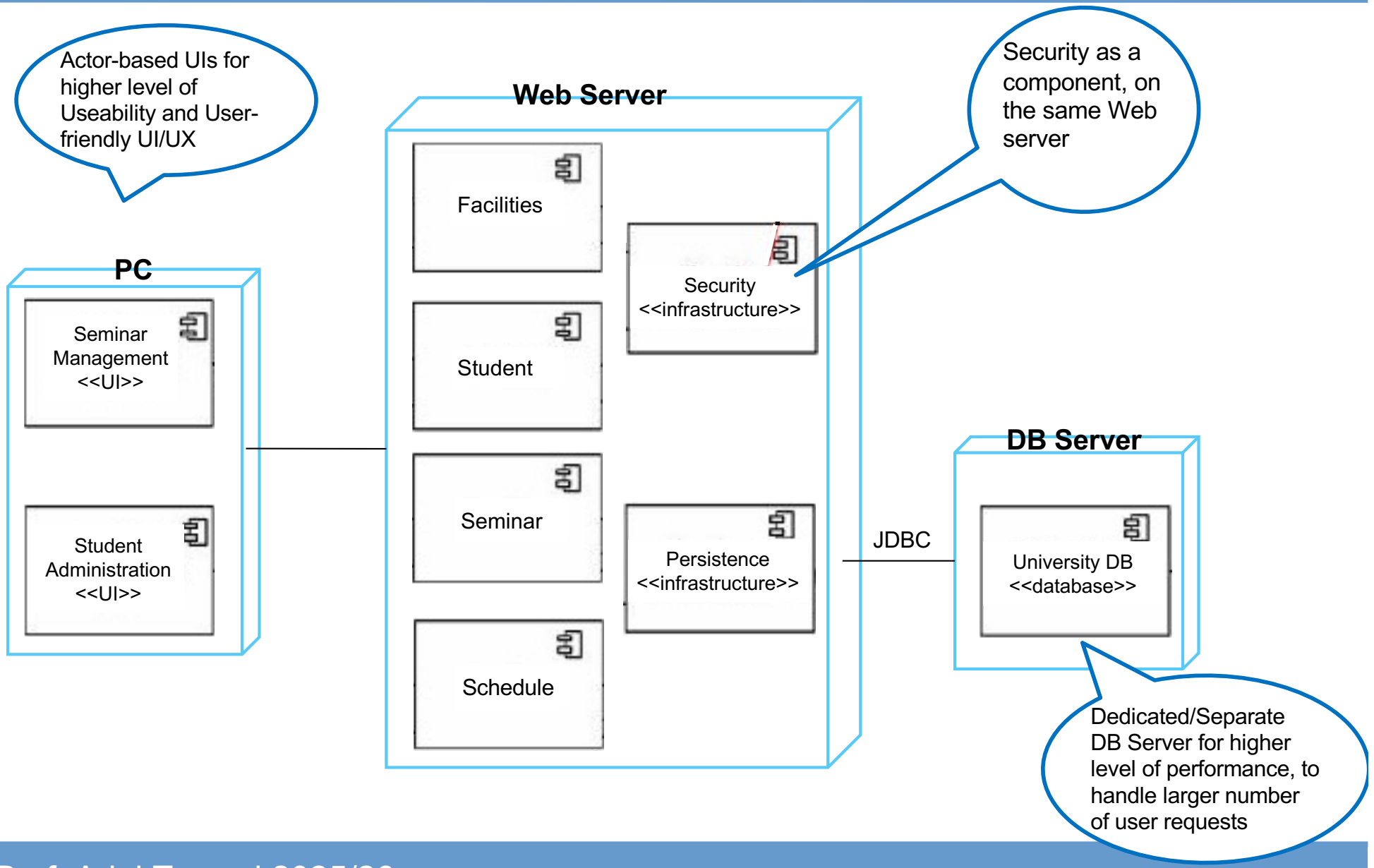
Deployment: as a Web Application (layered architecture)



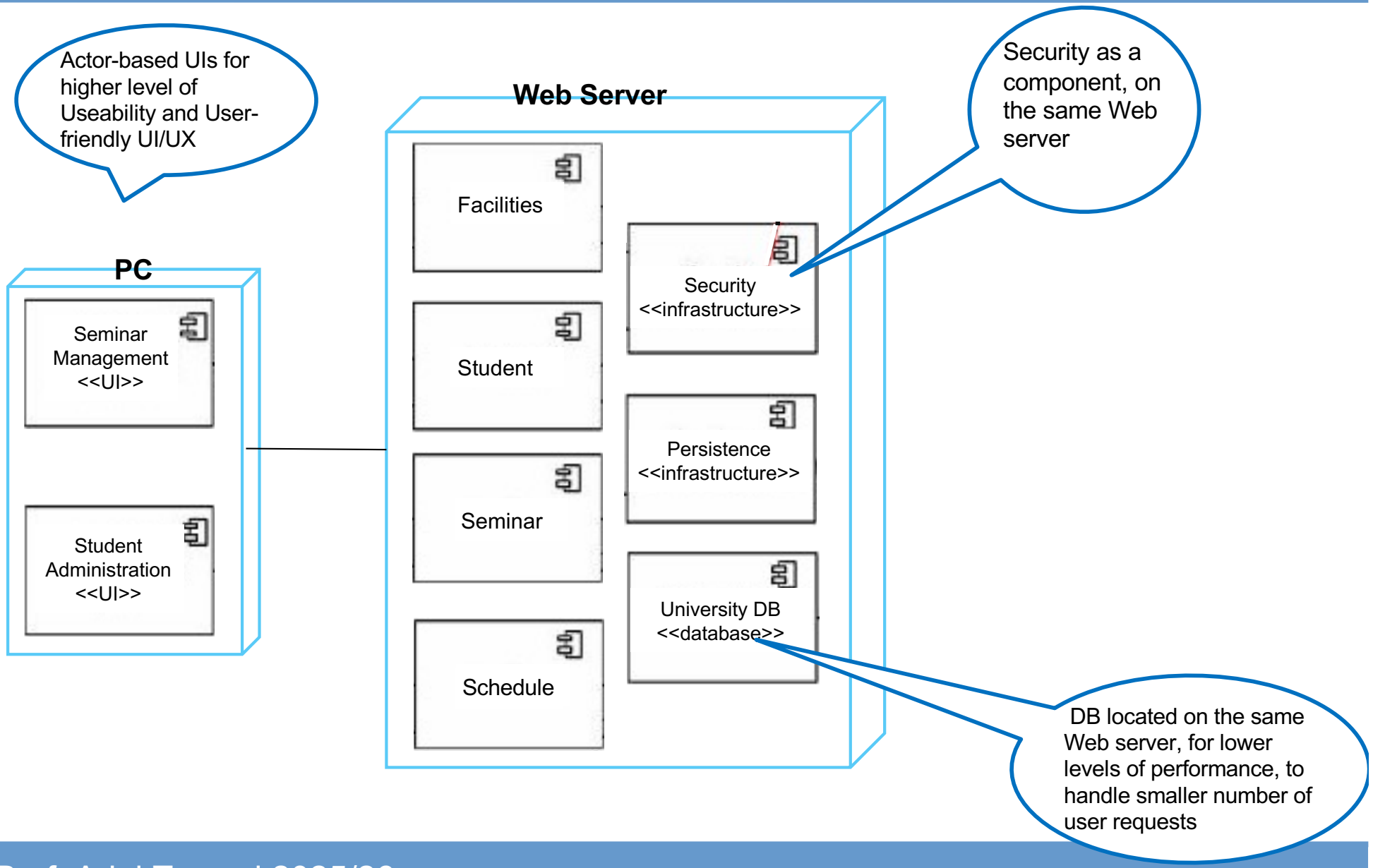
Deployment: as a Web Application (layered architecture- 3 Tier)



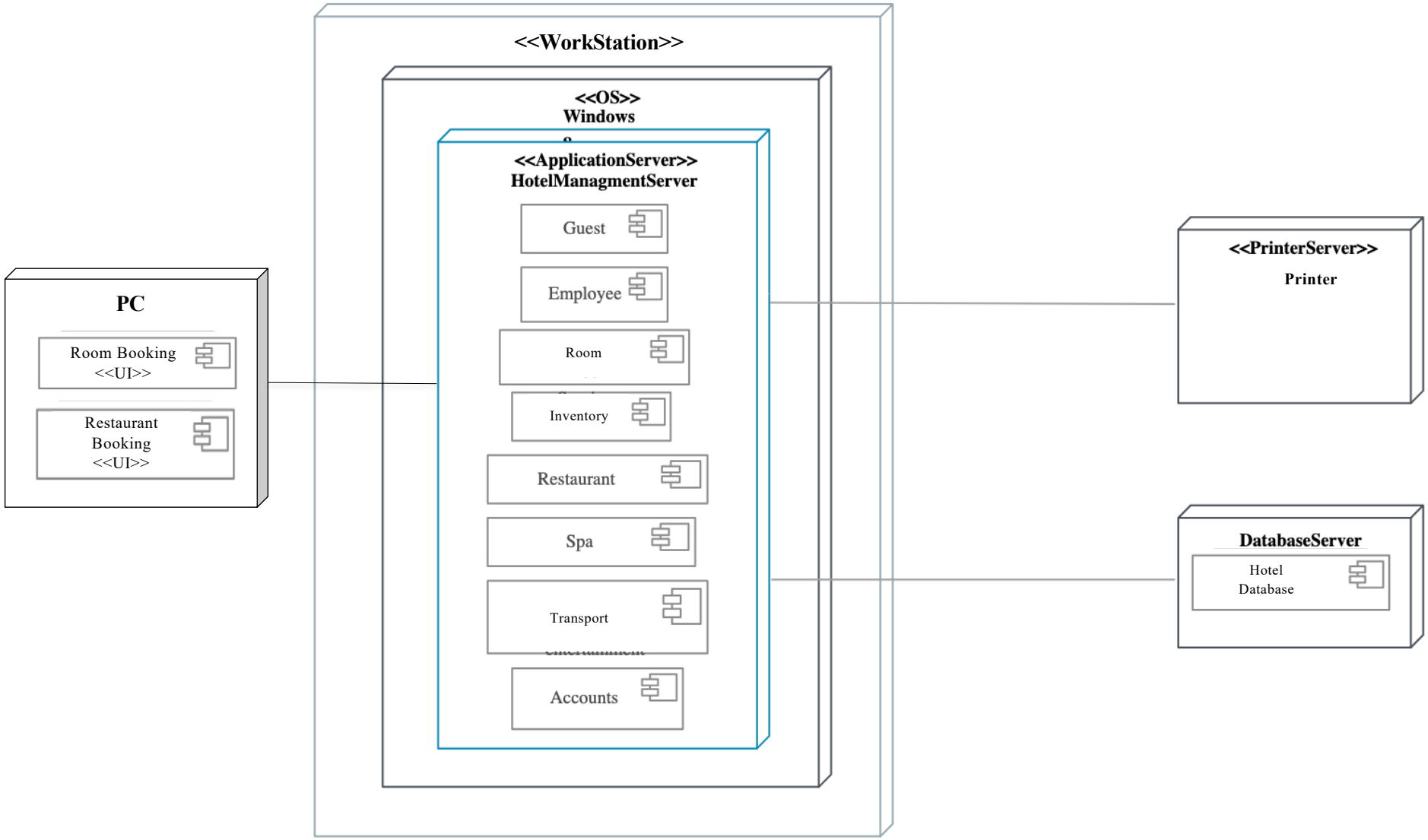
Deployment: as a Web Application (layered architecture – 3 Tier)



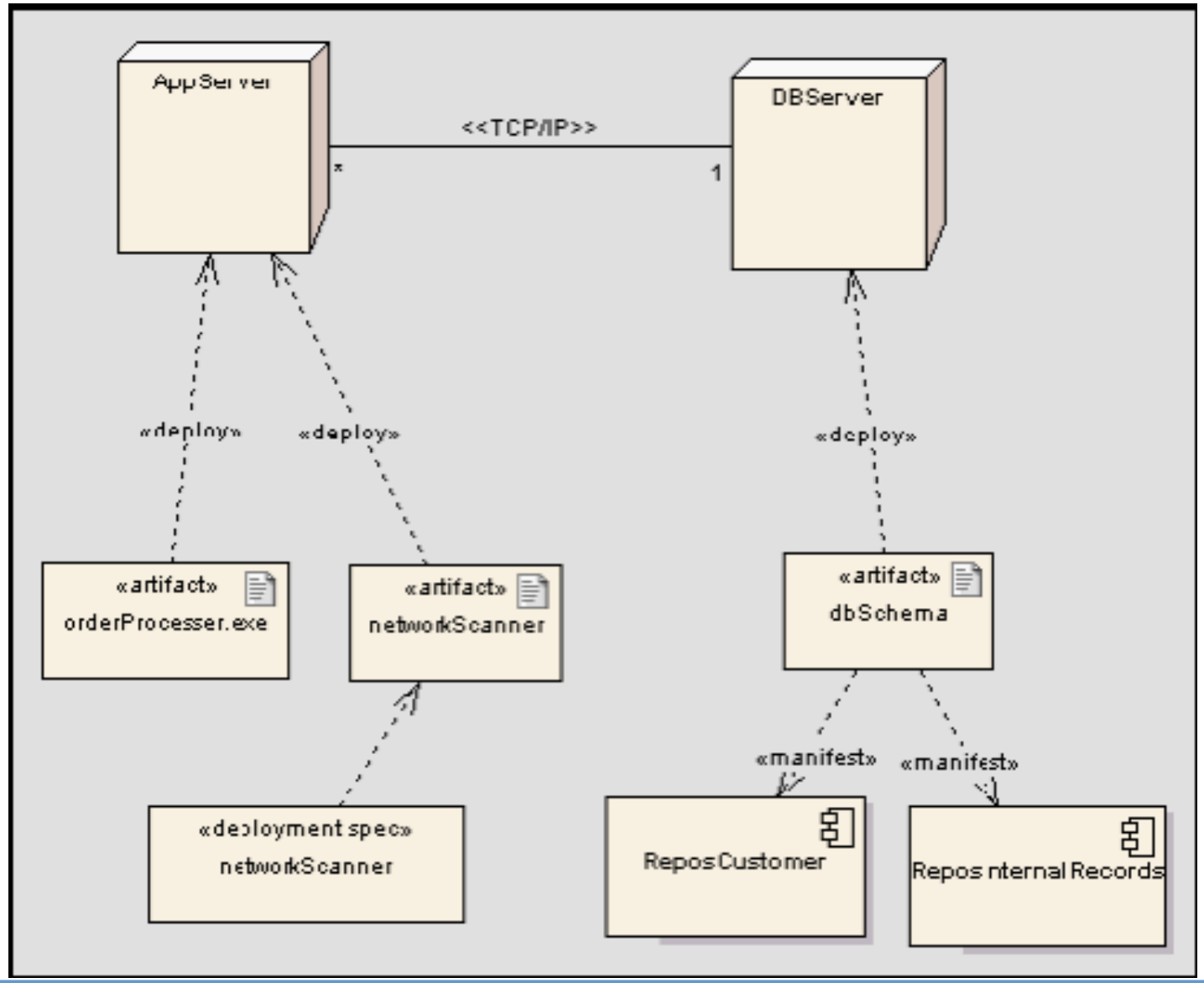
Deployment: as a Web Application (layered architecture – 2 Tier)



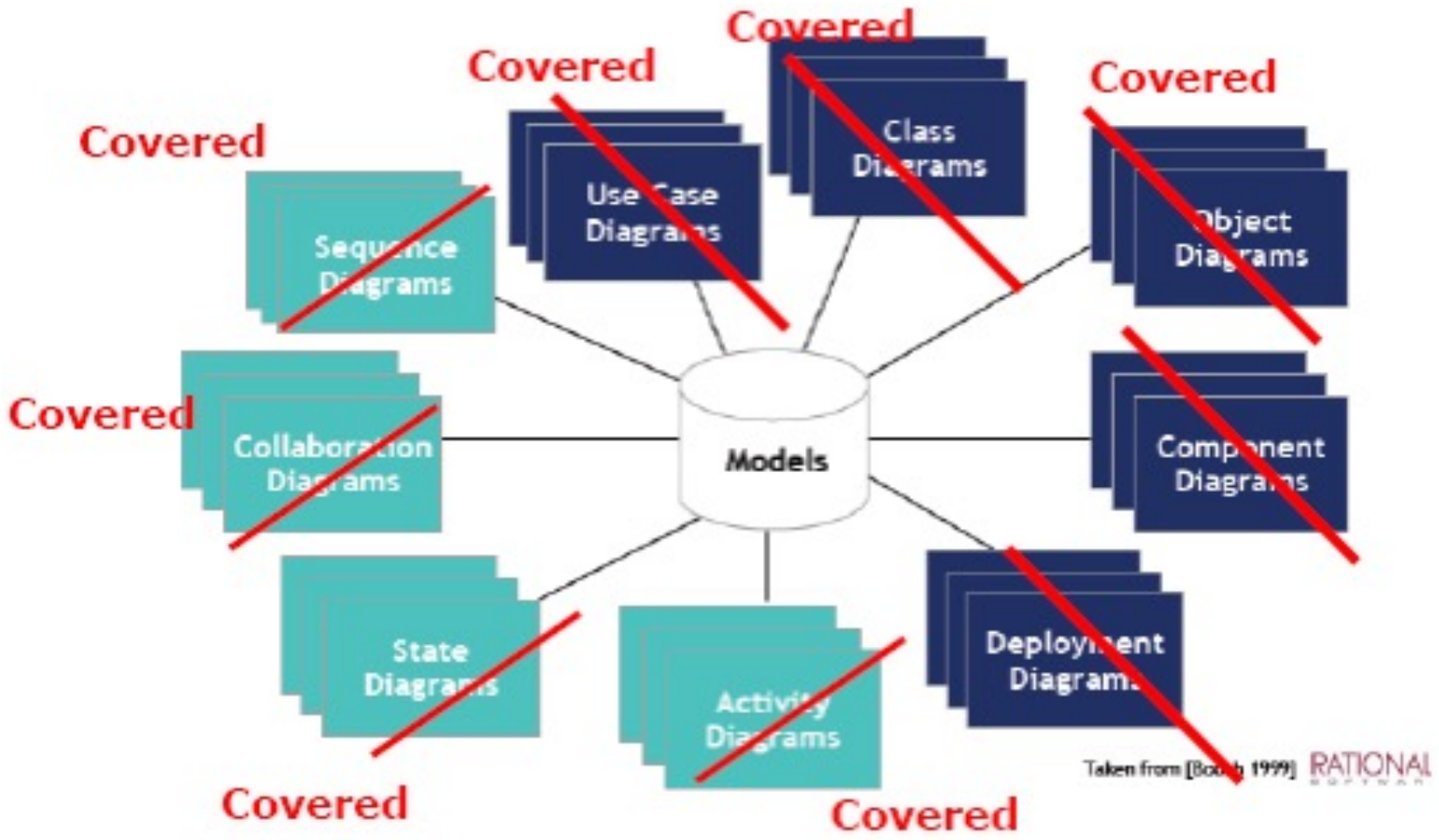
Deployment: Multi-Container Nodes



Example: Deployment Diagram for client server architectures



Covered!?


Covered
Covered
Covered
Covered
Covered
Covered
Covered
Covered
Use Case Diagrams
Class Diagrams
Sequence Diagrams
Object Diagrams
Collaboration Diagrams
Models
Component Diagrams
State Diagrams
Activity Diagrams
Deployment Diagrams
Taken from [Booch 1999]
RATIONAL

Key points

A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behaviour.

Context models show how a system that is being modeled is positioned in an environment with other systems and processes.

Structural models show the organization and architecture of a system. Use cases describe interactions between a system and external actors. Class diagrams are used to define the static structure of classes in a system and their associations using both data-driven and executable view points.

Behavioural models show how how system elements interactions. Use case diagrams, activity diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed taking the business view points or needs. Activity diagrams show how a business achieve its business process through interactions between use cases. Sequence diagrams show how a system achieve use cases through interactions between system objects.