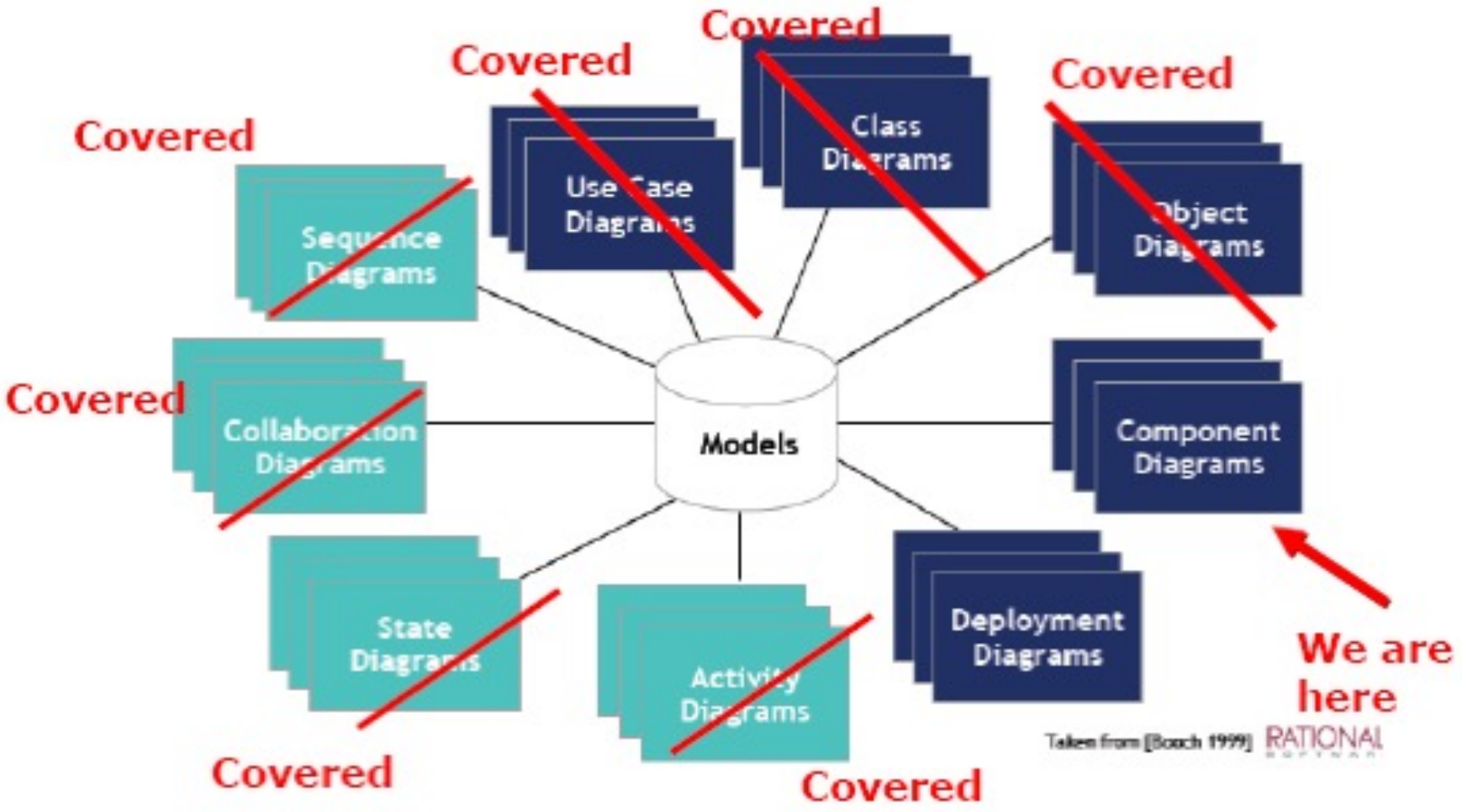
UML Diagrams



Component Diagrams

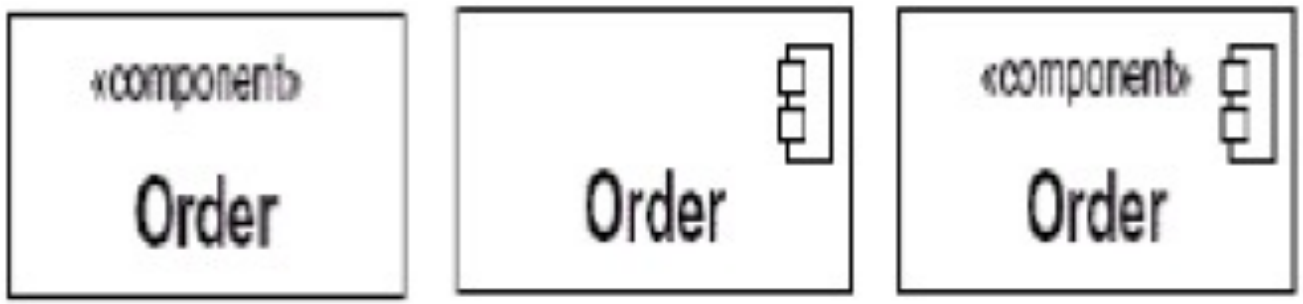
The component diagram's main purpose is to show the structural relationships between the components of a system

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

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**

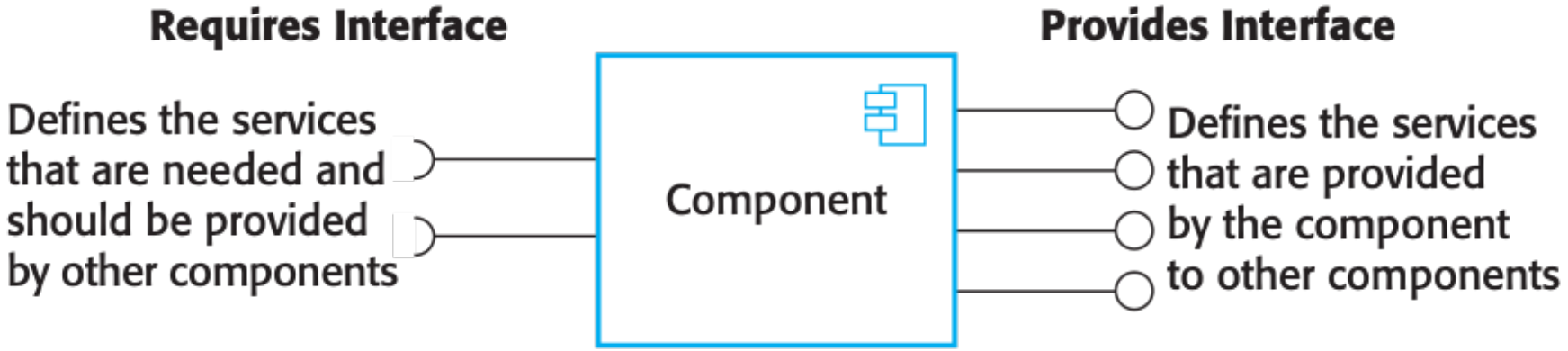
Component implementations are said to **realize** their provided interfaces

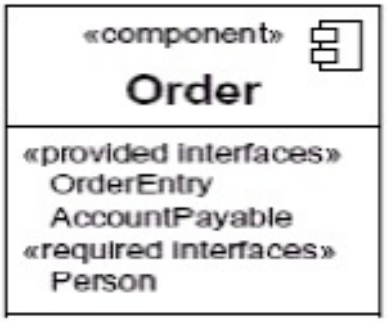
A provided and required interface can be connected if the operations in the latter are a subset of those in the former, and the signatures of the associated operations are '**compatible**'

Ports provide access between external interfaces and internal structure of components

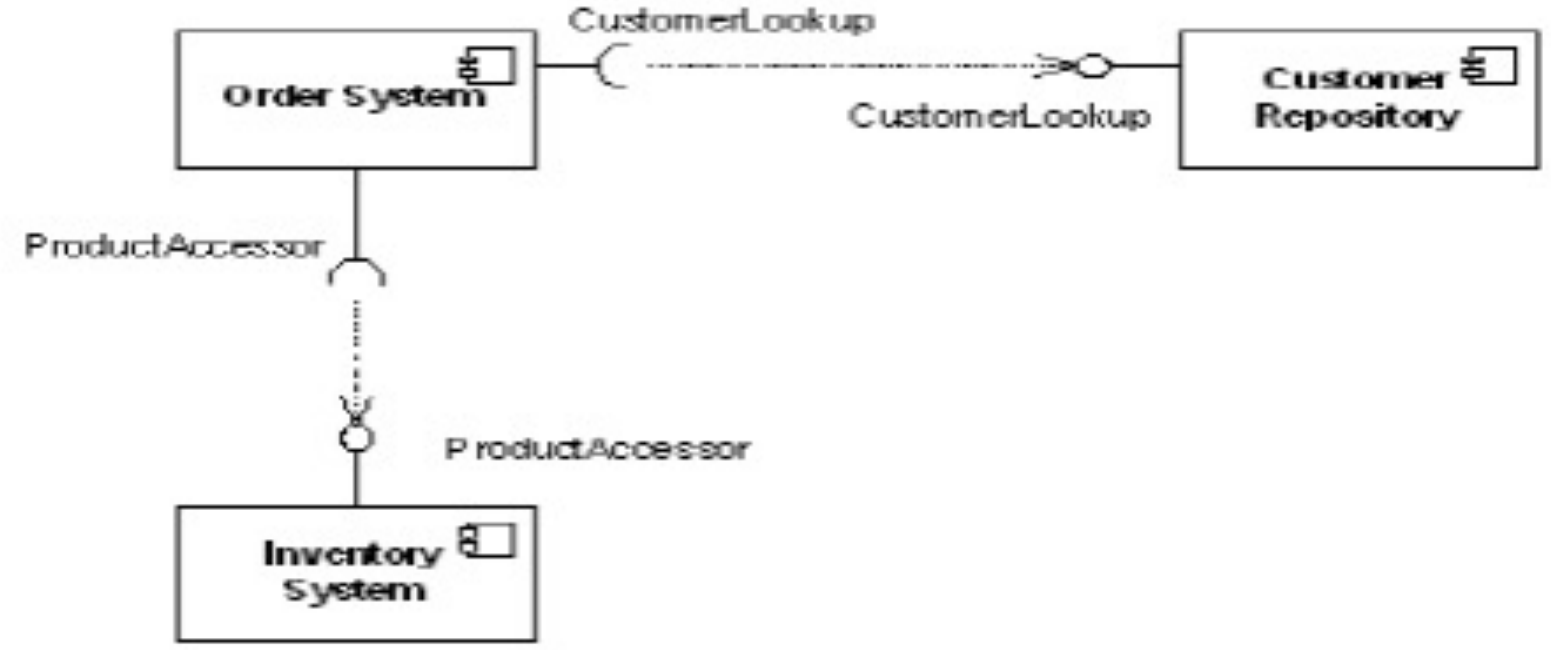
UML components can be used to model complex architectural connectors (like a CORBA ORB)

Required/Provide Interface



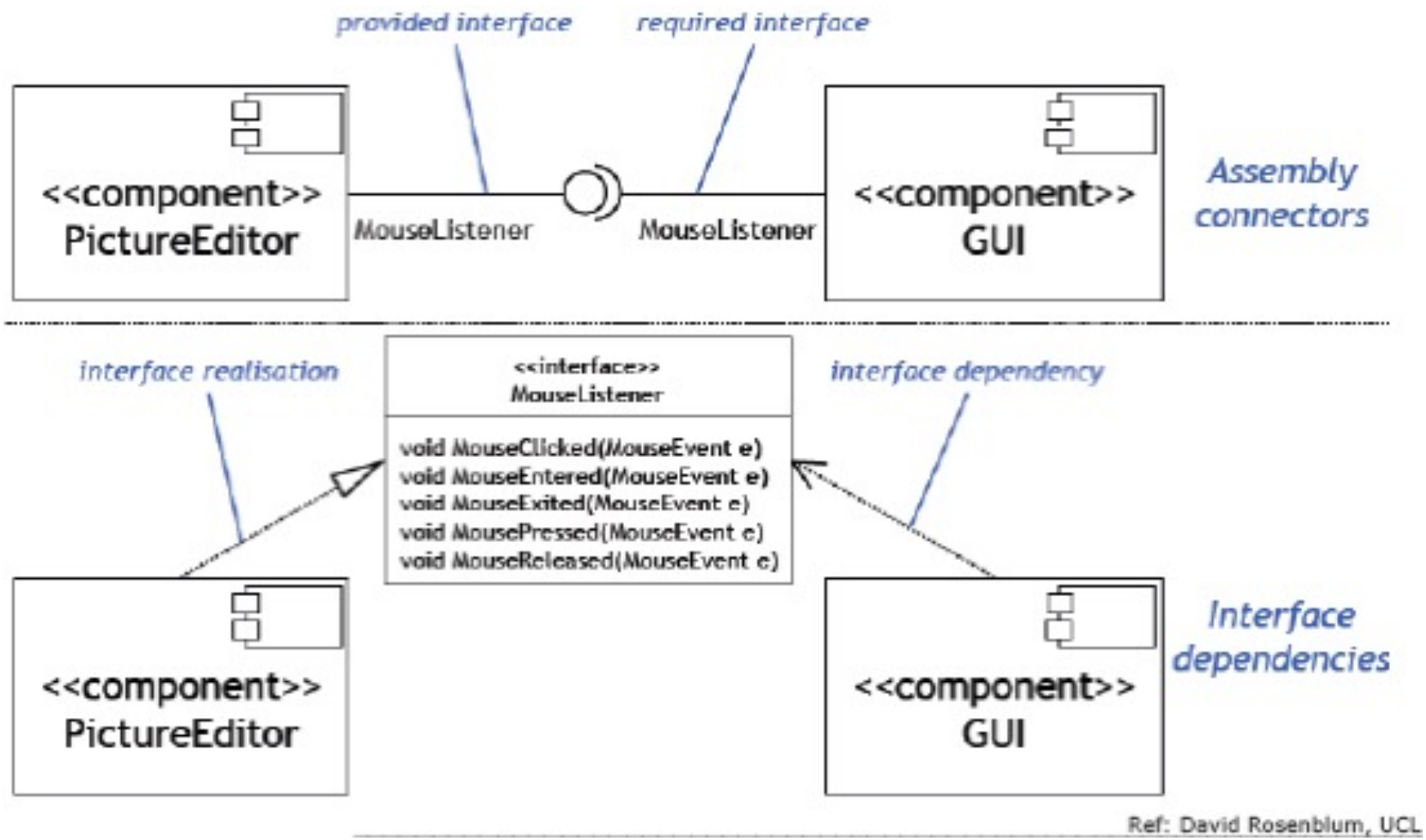


Component Diagrams

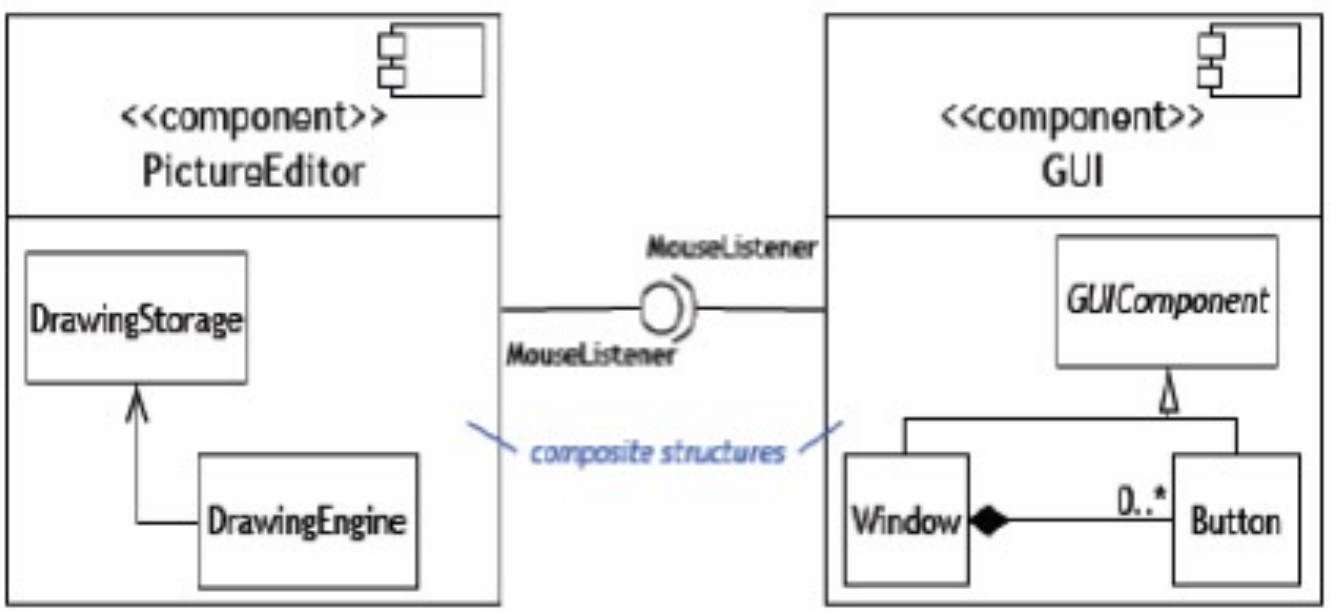


showing a component's relationship with other components, the lollipop and socket notation must also include a dependency arrow (as used in the class diagram). On a component diagram with lollipops and sockets, note that the dependency arrow comes out of the consuming (requiring) socket and its arrow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

Component Diagrams



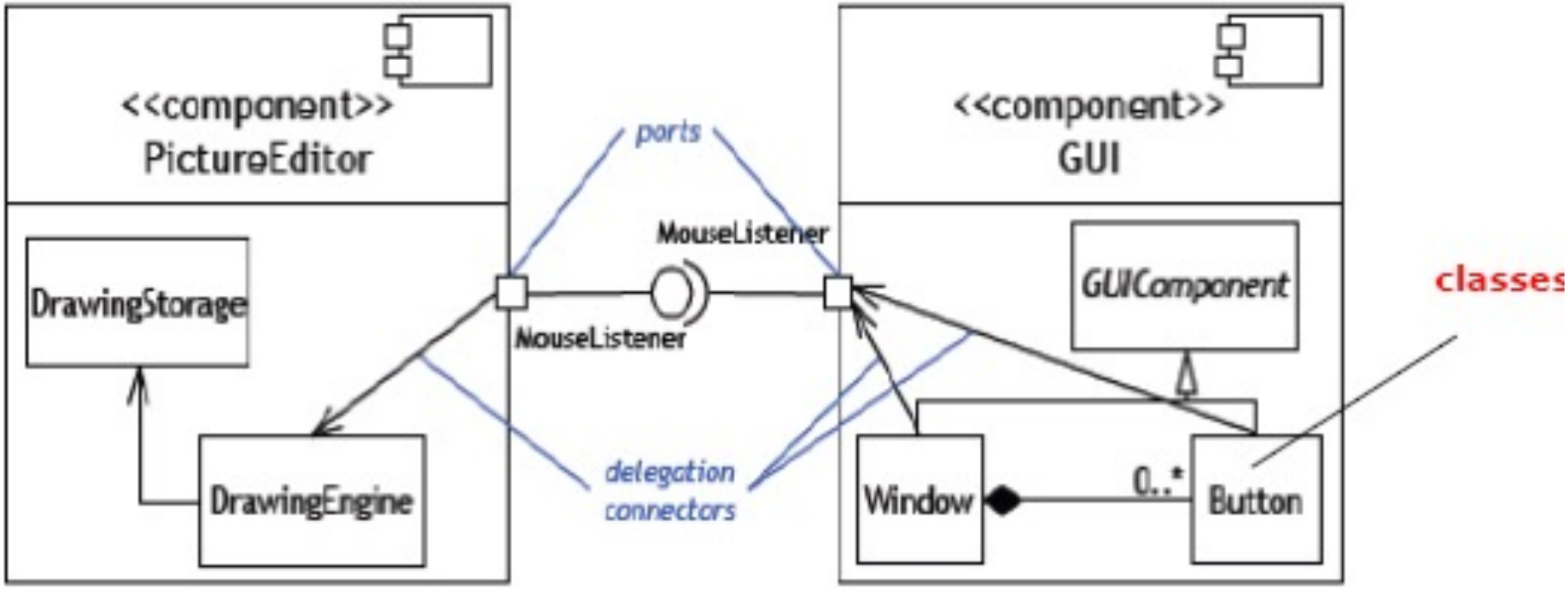
Composite Structure in Component Diagrams



Ref: David Rosenblum, UCL

A composite structure depicts the internal realisation of component functionality

Ports

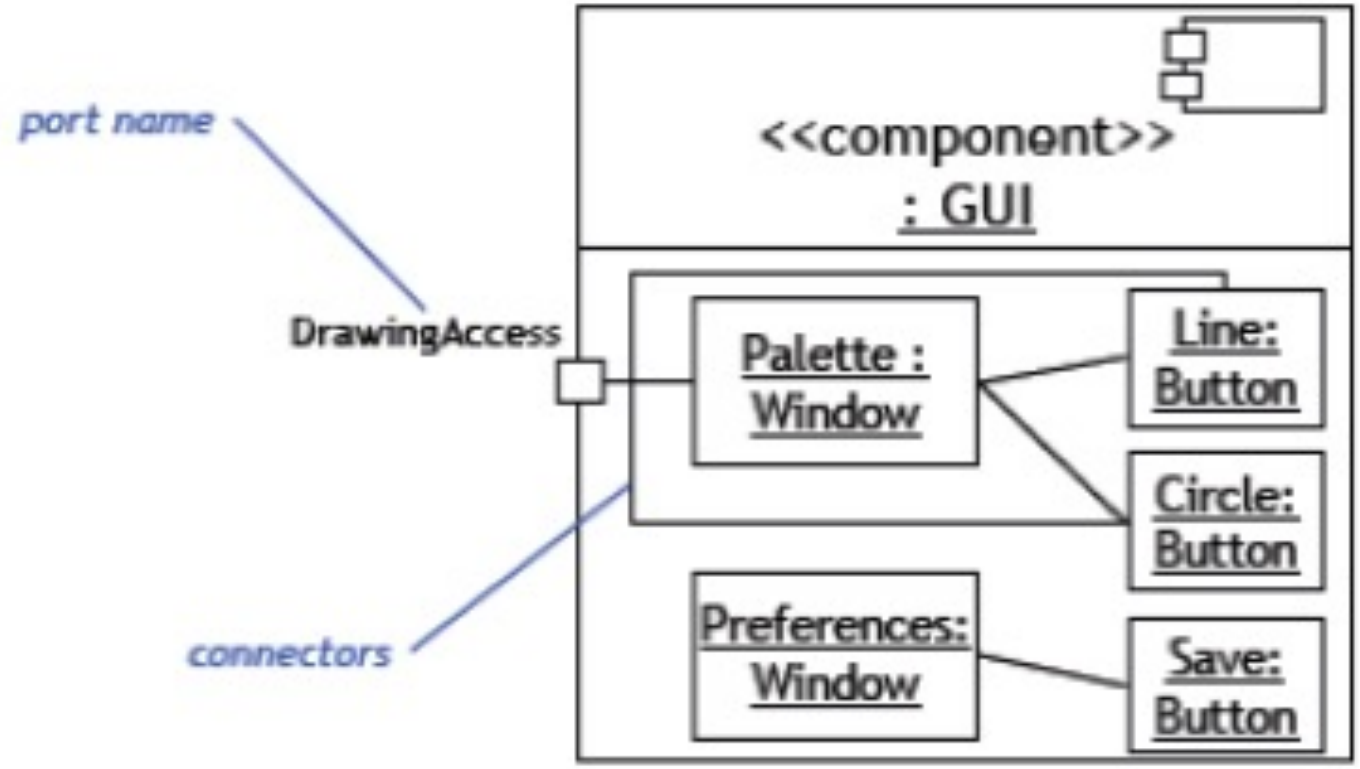


Ref: David Rosenblum, UCL

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

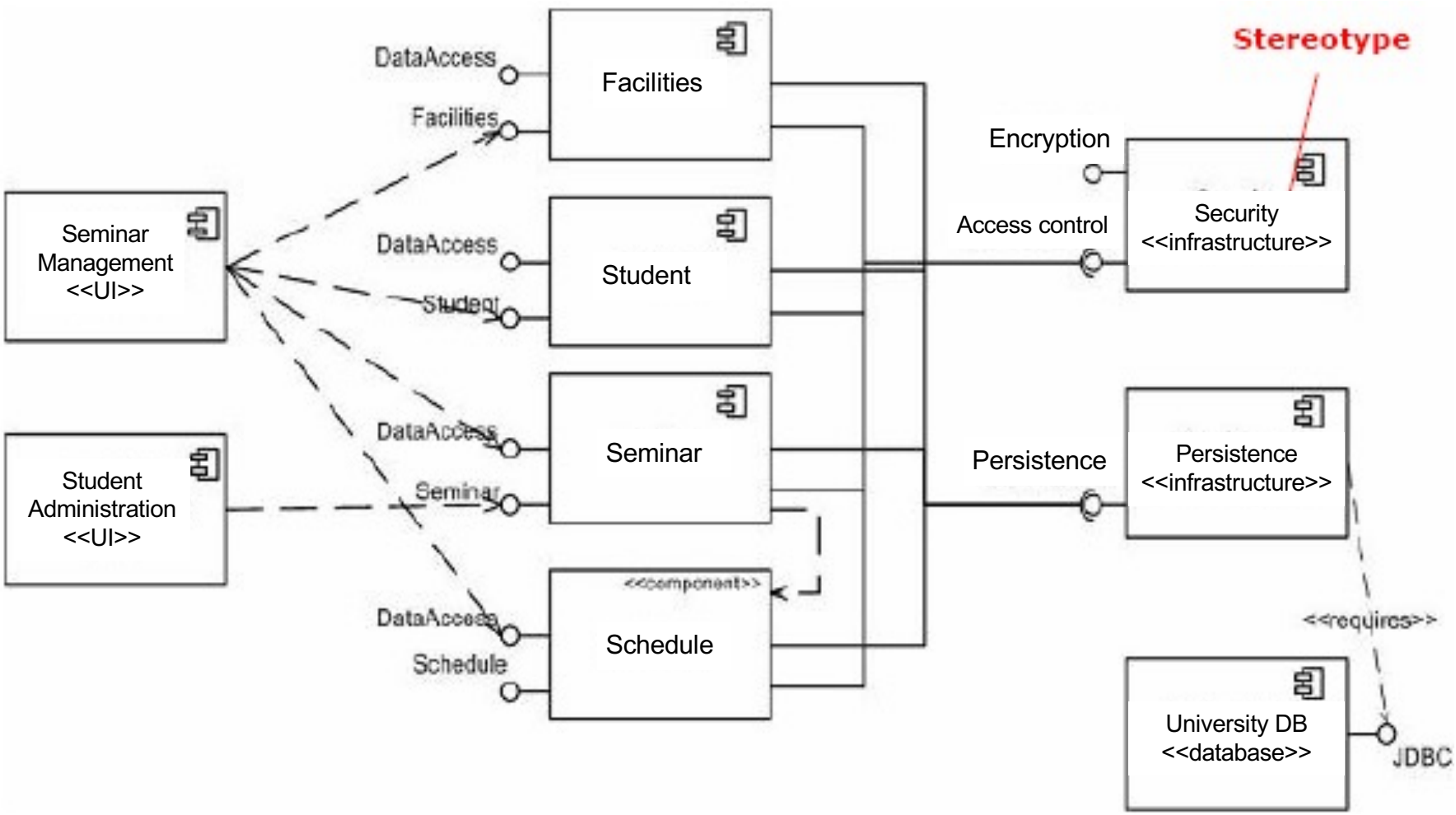
Ports



Ref: David Rosenblum, UCL

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

Example



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.

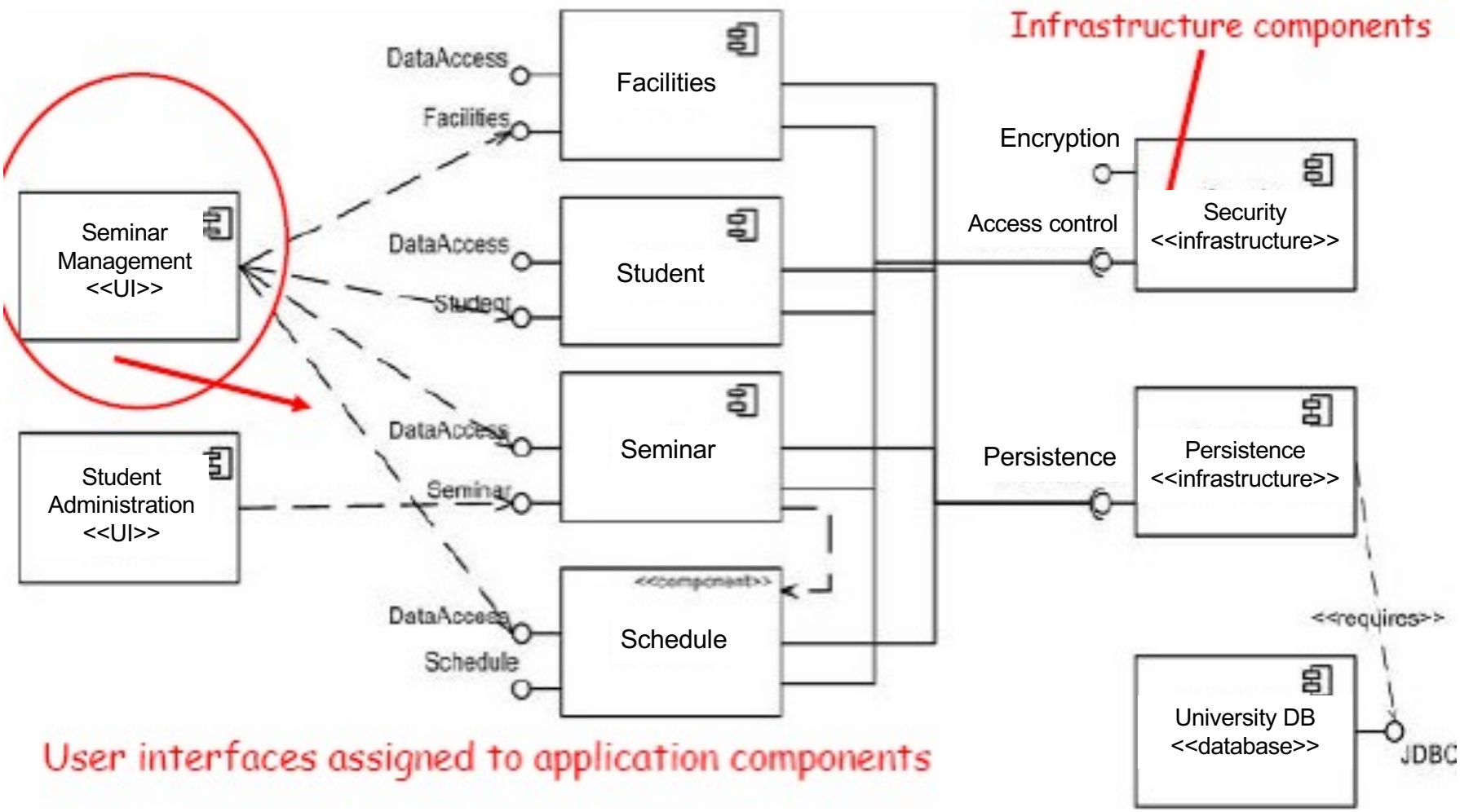
User *interface* classes assigned as application components.

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



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 a *composition hierarchy*, to the same component.

Identify business domain components.

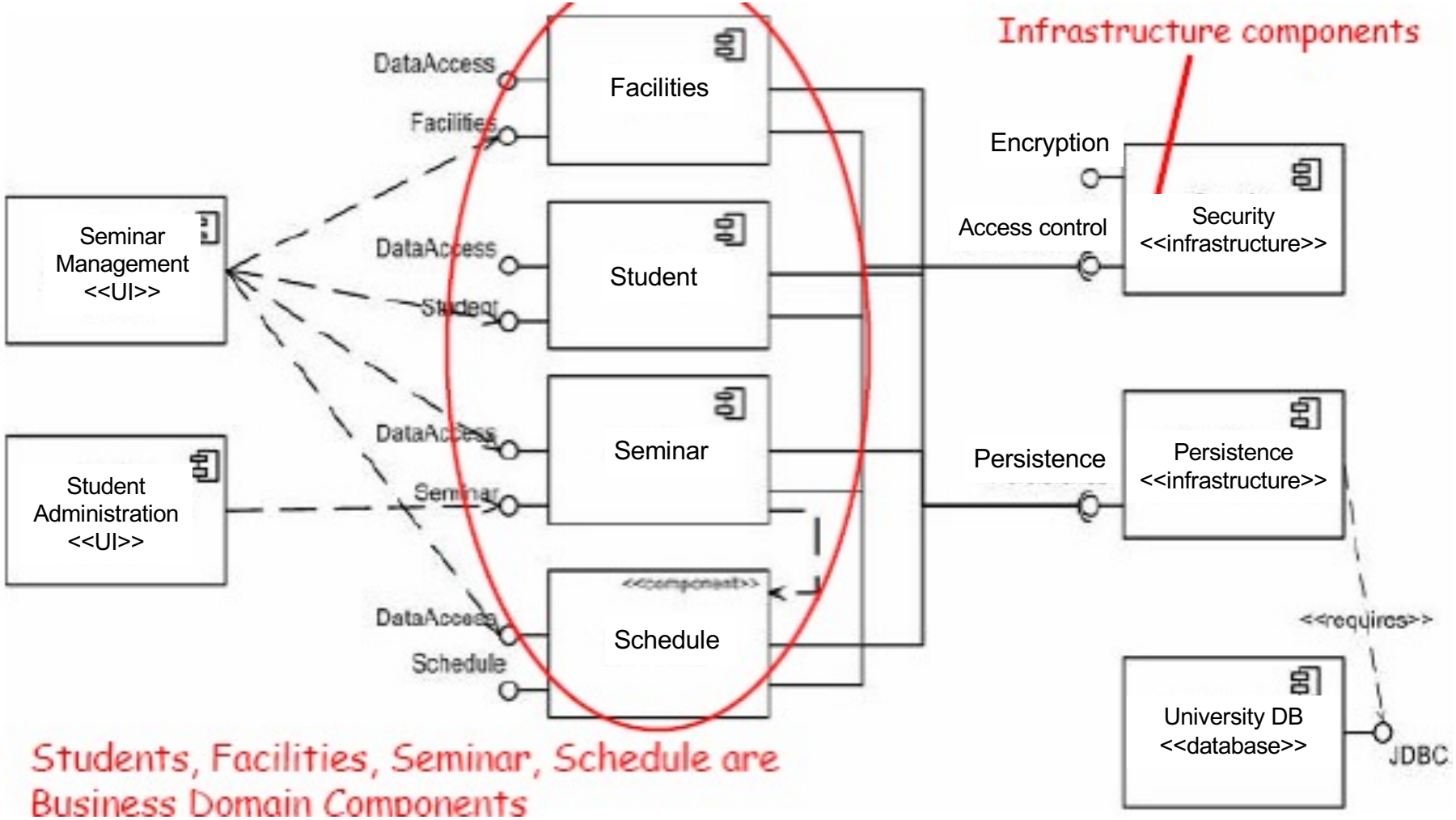
Because you want to minimize network traffic to reduce the response time of your application, you want to design your business domain components in such a way that most of the *information flow* occurs *within* the components and not *between* them.

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



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.

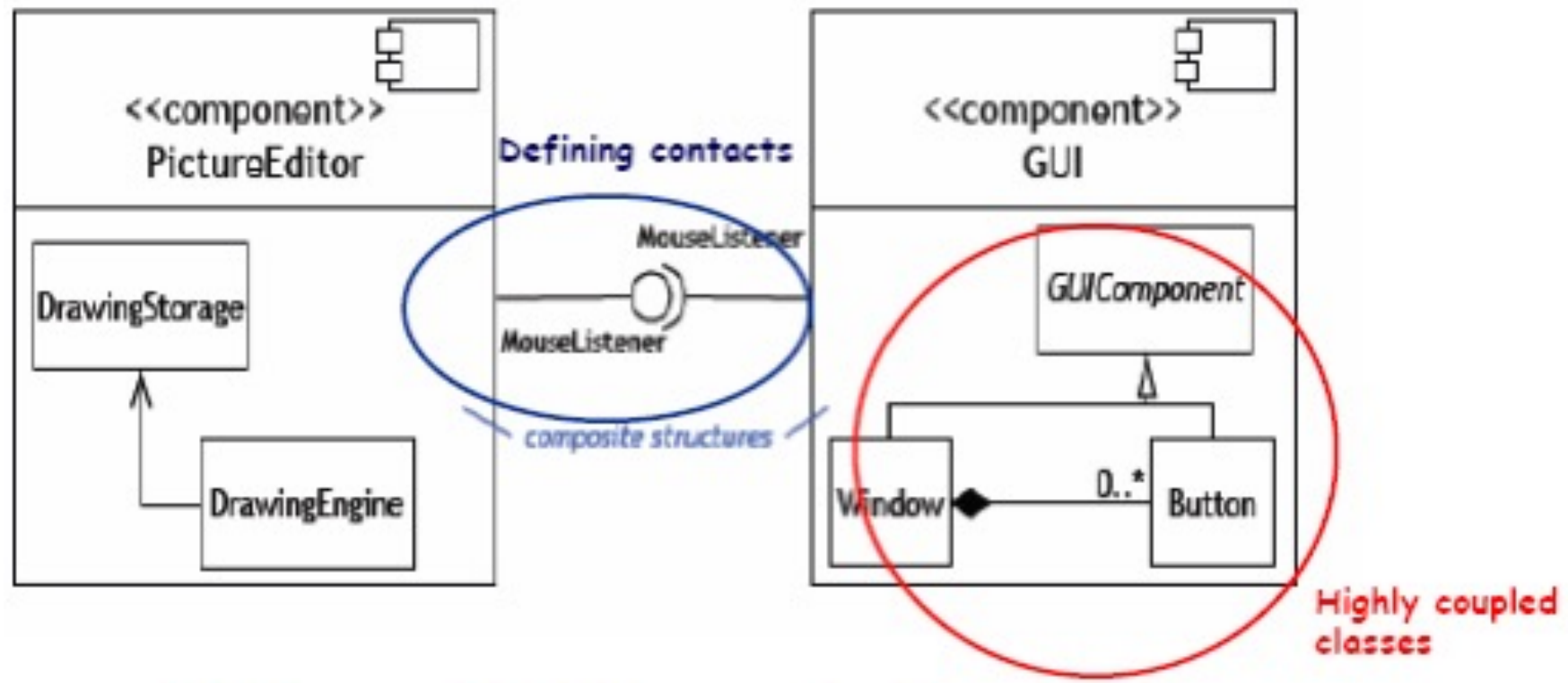
Minimize the size of the *message flow* between components.

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