Component diagrams

Components

- Components are model elements that represent **independent, interchangeable parts** of a system.
  - Components are more abstract than classes and can be considered to be **stand-alone service providers**
- They conform to and realize one or more **provided and required interfaces**, which determine the behavior of components.
- Components make a system more **reusable**, **scalable**, and **flexible**.

2

Components

- 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
- Components **can range in size** from simple functions to entire application systems

[Sommerville, 2000]

3

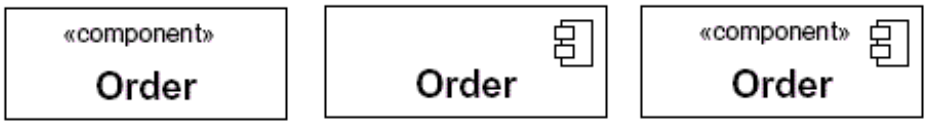
Components

- The **internal structure** of the component must be **hidden and independent**.
  - No dependencies can exist between the contents of the component and external objects (i.e., internal objects do not know external objects).
- Components must **provide interfaces** so that external objects can interact it with them.
- Components must specify their **required interfaces** so that they have access to external objects.

4

Components

- Have a **name** (or path name)
- Have **interfaces**
  - Their interface is published and **all interactions are through the published interface**
- Can have **stereotypes**
  - Executable, library, table, file, document
- Are **substitutable**: can be replaced at design time or run-time by another that offers equivalent functionality based on compatibility of its interfaces



All they mean the same: a component Order
UML version 2.0

5

Interfaces (in general)

- An interface is a **collection of operations that specify a service offered** by a class or a component
- Classes realize an interface and can contain additional operations
- Each interface represents a role played by a class
- Interfaces allow different views of a class used by different clients
- Interfaces are used as “glue” in component-based software

6

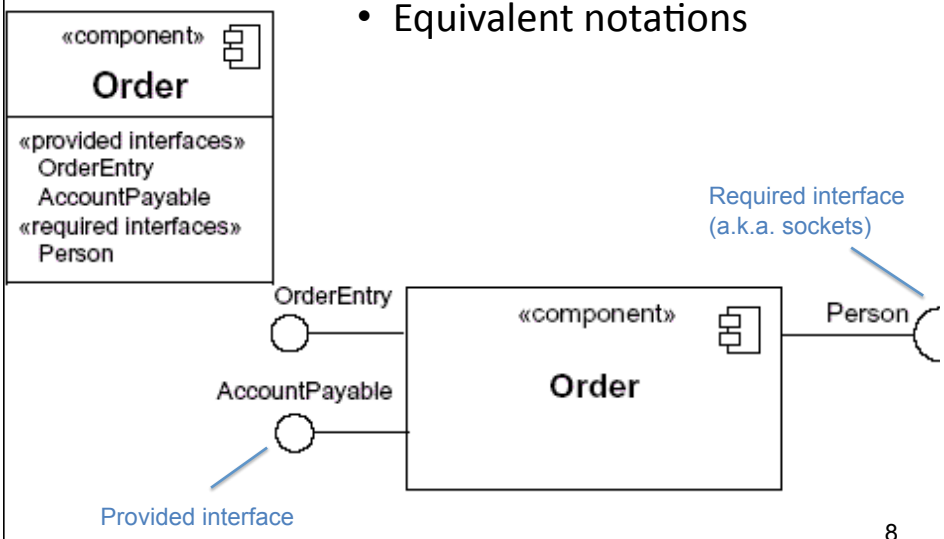
Component's Interfaces

- Interface = classifier with operations, but no attributes
  - defining a cohesive set of behaviors
- Provided Interfaces
  - Defines the services that are provided by the component to other components
- Required interfaces
  - Specifies what services must be made available for the component to execute as specified
- Interface's name is placed near the interface symbol

7

Component's Interfaces

- Equivalent notations



8

Components and 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**'
- An interface realized by a component is called **exporting interface**, i.e., an interface that the component offers as a service to other components
- An interface used by the component is **the importing interface**
- A component may import and export several interfaces
- An interface offered by a component is realized by classes that the component implements

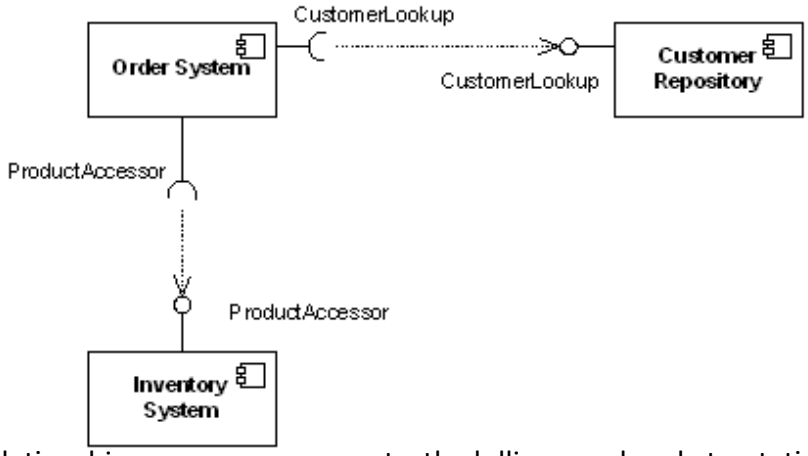
9

What is a Component Diagram?

- **Models the software architecture** of the system, also providing a view of their physical software components, their interfaces, and their dependencies.
- Their main purpose is to **show the structural relationships between the components** of a system. [IBM Rational Libraries]
- They are **composed of components, interfaces and relationships** among them (dependency, generalization, association, realization)

10

Example

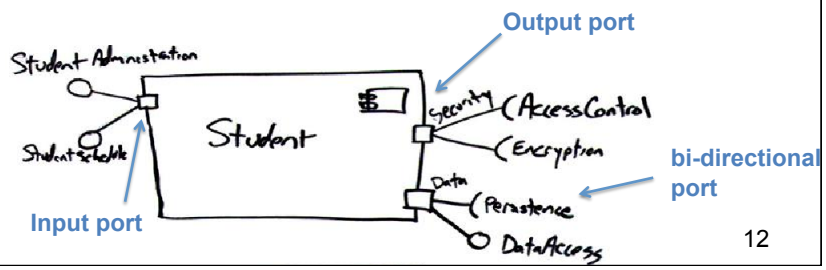


- Relationships among components: the lollipop and socket notation can also include a **dependency arrow** (as used in the class diagram). The **dependency arrow comes out of the consuming (requiring)** socket and its arrow head connects with the provider's lollipop

11

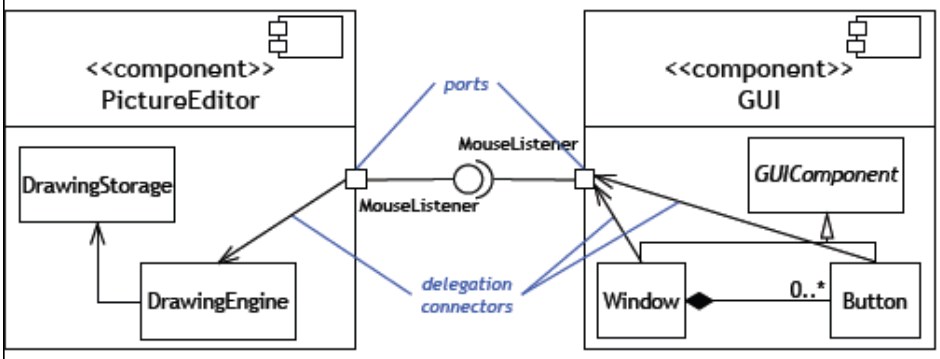
Ports

- A port (feature of a classifier): specifies a distinct interaction point between the classifier and its environment
- Ports:
  - depicted as small squares on the sides of classifiers
  - can be **named**
  - can support **unidirectional** and **bi-directional communication**. (Student component implements three ports: two unidirectional and one bi-directional)



12

Ports (showing connectors)

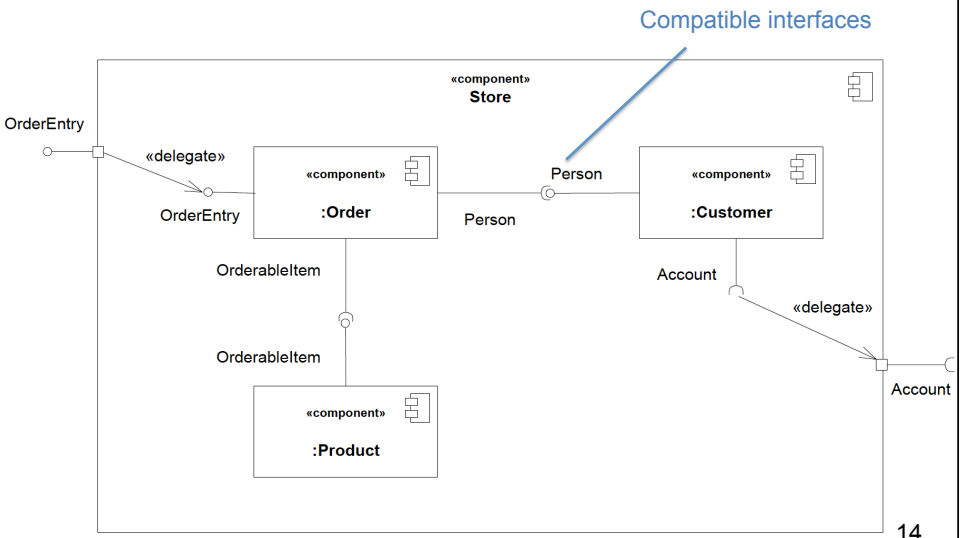


- The **ports and connectors specify how component interfaces are mapped to internal functionality**
- Note that these 'connectors' are rather limited (special cases of those considered in software architectures)

[David Rosenblum, UCL] 13

Components of components

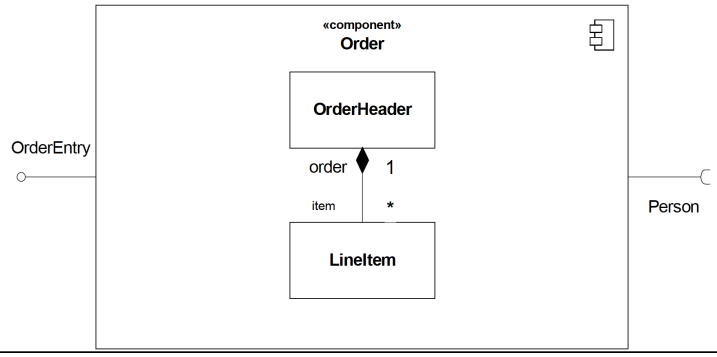
- A component can be composed of other components.



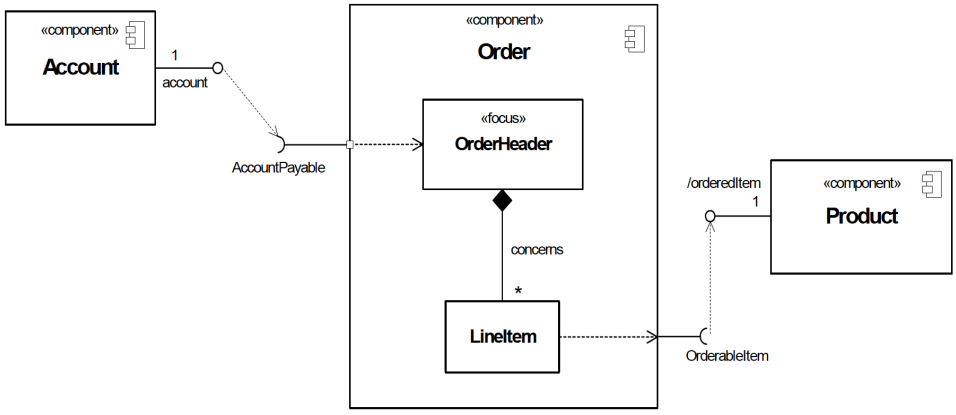
14

Components vs Classes

- Classes represent logic abstractions
- Components implement a set of logic elements (e.g. Classes)
  - Classes can be mapped into components
- Classes may have attributes and operations
- Components have (public) operations that can only be accessed through their interfaces



15



16

Building a component diagram

- Top-down
  - Nice to give an early "landscape" of the project
    - Helps to support team distribution work (from beginning)
  - Dangerous as it "promotes" over-architecting, over-designing
- Bottom-up
  - Nice when we have a collection of classes and decide to "componentize" our design
  - Nice to rescue reusable functionality out of an existing application
  - Nice to distribute work between subteams.
  - (Guidelines next)

17

Build components: guidelines

1. Keep components cohesive
2. Assign interface/boundary classes to application components
3. Assign technical classes to infrastructure components
4. Define class contracts
5. Assign hierarchies to the same component
6. Identify (business) domain components
7. Identify the "collaboration type" of business classes
   a) Server classes belong in their own component
   b) Merge a component into its only client
   c) Pure client classes don't belong in domain components
8. Highly coupled classes belong in the same component
9. Minimize the size of the message flow between components
10. Define component contracts

[The object primer: agile model-driven development with UML 2.0, Scott W. Ambler]

18

Keep components cohesive

- 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

19

Assign user interface classes to application components

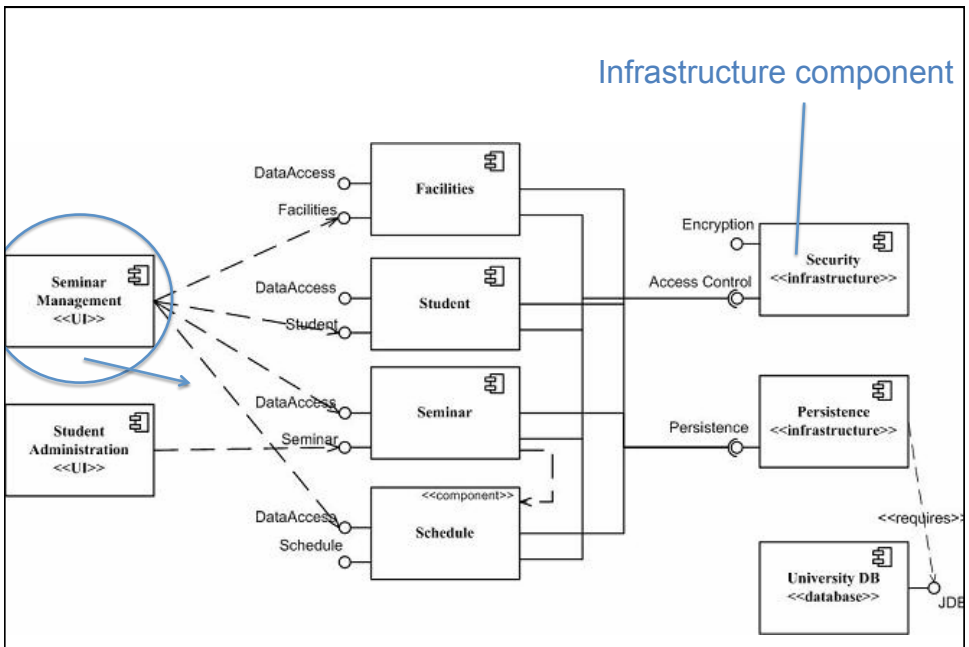
- User interface/boundary classes should be placed in components with the **application stereotype**.
  - These classes implement screens, pages, or reports, as well as those that implement "glue logic" such as identifying which screen/page/… to display
- In Java these types of classes would include Java Server Pages (JSPs), servlets, and screen classes implemented via user interface class libraries such as Swing

20

Assign technical classes to infrastructure components

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

21



User interfaces assigned to application components

22

Define class contracts

- A class contract is **any method that directly responds to a message sent from other objects**.
  - For example, the contracts of the *Seminar* class likely include operations such as *enrollStudent()* and *dropStudent()*.
- To identify components, **all the operations that aren't class contracts can be ignored**
  - As they don't contribute to communication between objects distributed in different components

23

Assign hierarchies to the same component

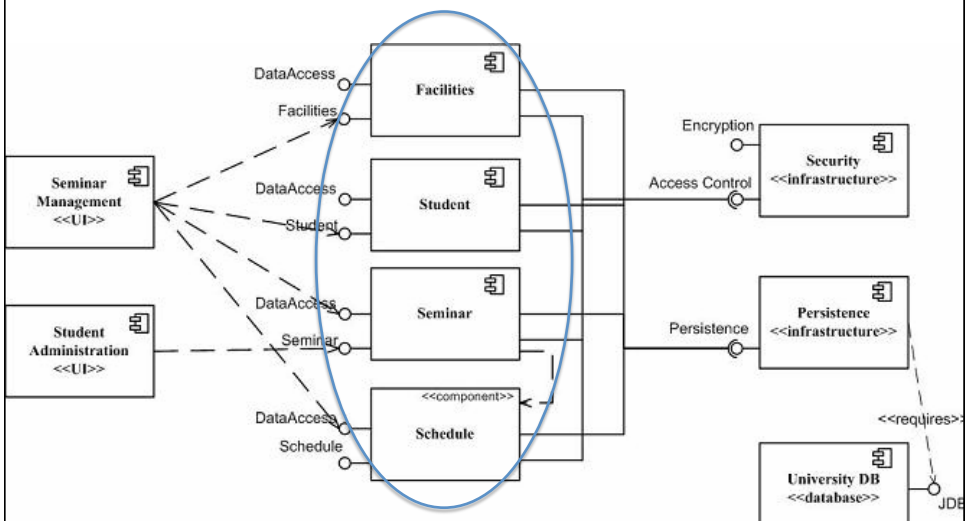
- 99.9% of the time it makes sense to assign all of the classes of a hierarchy (inheritance hierarchy or composition) to the same component

24

Identify (business) domain components

- (Business) domain component is a set of **classes that collaborate among themselves to support a cohesive set of contracts**.
- Because we **want to minimize the network traffic to reduce response time** of our application we want to design our components so that most of the information flow occurs within the components and not between them.

25



Students, Facilities, Seminar, Schedule are Business Domain Components

26

Identify the "collaboration type" of business classes

- To determine which domain component a business class belongs to identify its distribution type:
  - **Server class**: receives, but doesn't send, messages
  - **Client class**: sends, but doesn't receive, messages
  - **Client/server class**: both sends and receives messages
- After identified the distribution type of each class, you are in a position to start identifying potential (business) domain components.

27

Server classes belong in their own component

- Pure server classes belong in a domain component and often form their own domain components
  - they are the "last stop" for message flow (use case execution) within an application

28

Merge a component into its only client

- If you have a **domain component that is a server to only one other domain component**, you may decide to combine the two components

29

Pure client classes don't belong in domain components

- Client classes do not belong in a domain component as they only generate messages
  - as the purpose of a domain component is to respond to messages.
- Client classes have nothing to add to the functionality offered by a domain component and very likely belong in an application component instead

30

Highly coupled classes belong in the same component

- If two or more classes **collaborate frequently, they should probably be in the same** domain component to reduce the network traffic between the two classes.
  - **Especially when the interaction involves large object**s (as parameters or received as return values).
- The basic idea is that **highly coupled classes belong together**.

31

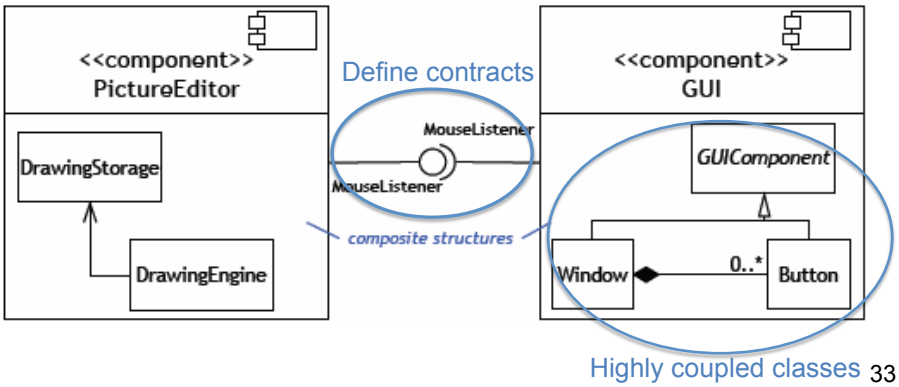
Minimize the size of the message flow between components

- **Client/server classes belong in a domain component**, but there may be a choice as to which domain component they belong to.
- Choose so that communication between components will be low
  - Merge a component into its only client

32

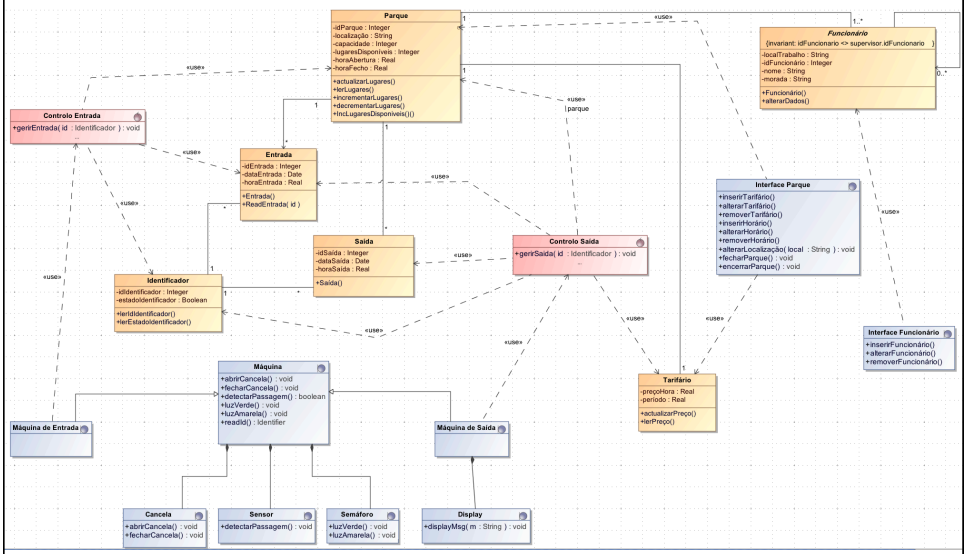
Define component contracts

- Each component will offer services to its clients, each such service is a component contract



33

Do you feel strong?



34

Draw a component diagram

35

Deployment diagrams

36

Deployment diagram

- Models the **run-time architecture** of a system
- Depicts a **static view of the run-time configuration of processing nodes**, visualizing the distribution of the components running on those nodes
  - Ex. nodes: server, client, modem, printer, etc.
- Deployment diagrams show: the **hardware**, the **software** that is installed on that hardware, and the **middleware** used to connect the disparate machines to one another!

37

Nodes and connections

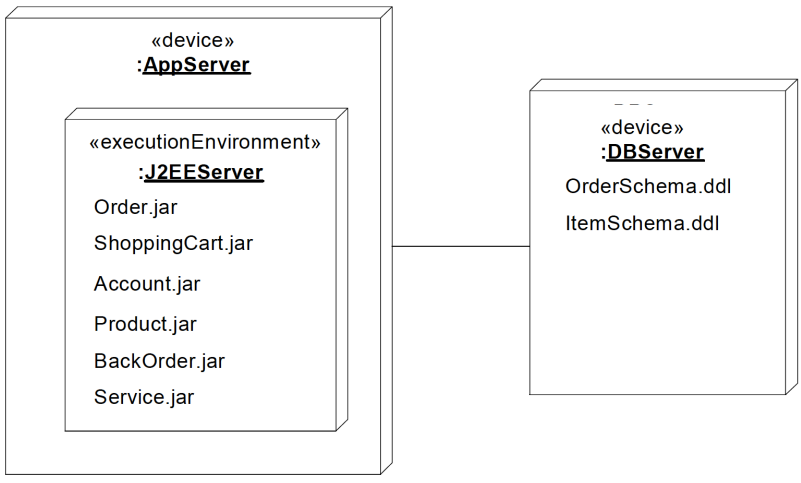
- Deployment diagrams include notation elements used in a component diagram, plus
- **nodes** which represent either a **physical machine or a virtual machine** (e.g., a mainframe node)
  - they are represented as 3-D boxes and can be processors (e.g. server) or devices (e.g. modem)
- and **connections** (dependencies and associations)
  - are represented with simple lines, and are assigned stereotypes to indicate the type of connection

38

Nodes

- **Node**
  - a physical object that represents a **processing resource**
  - generally, having at **least a memory and often processing capability** as well

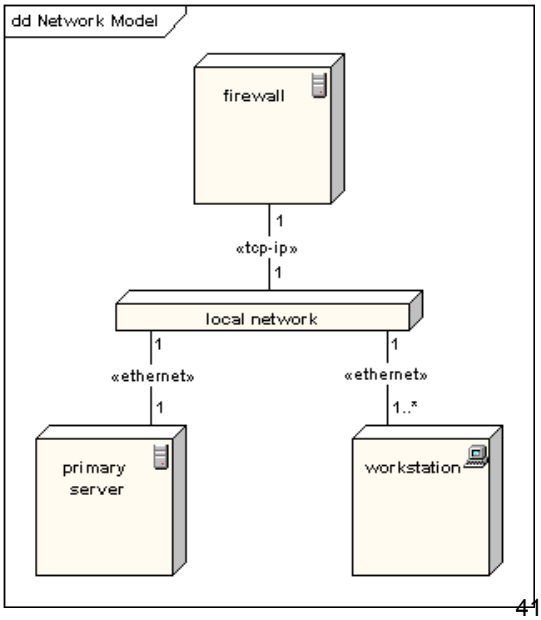
39



40

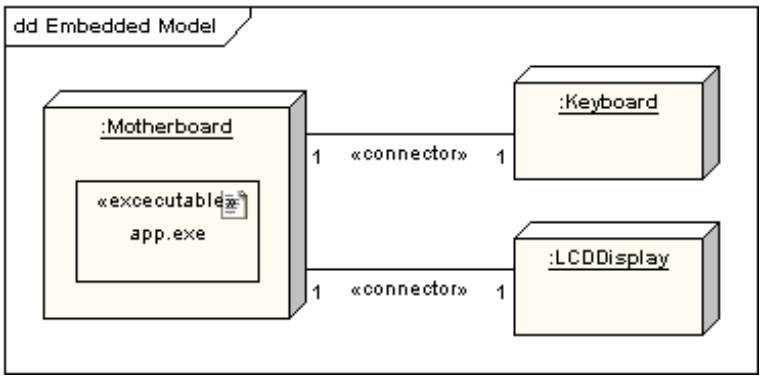
Deployment diagram

Deployment diagram for a network, depicting **network protocols as stereotypes**, and **multiplicities at the association ends**



41

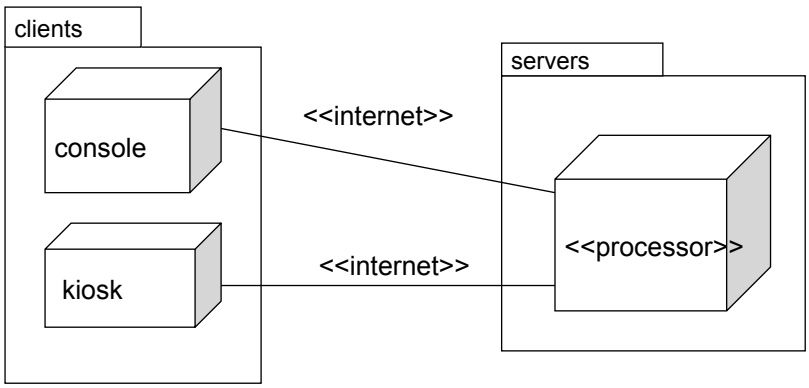
Deployment diagram



Deployment diagram for part of an embedded system, depicting an **executable artifact as being contained by the motherboard node**

42

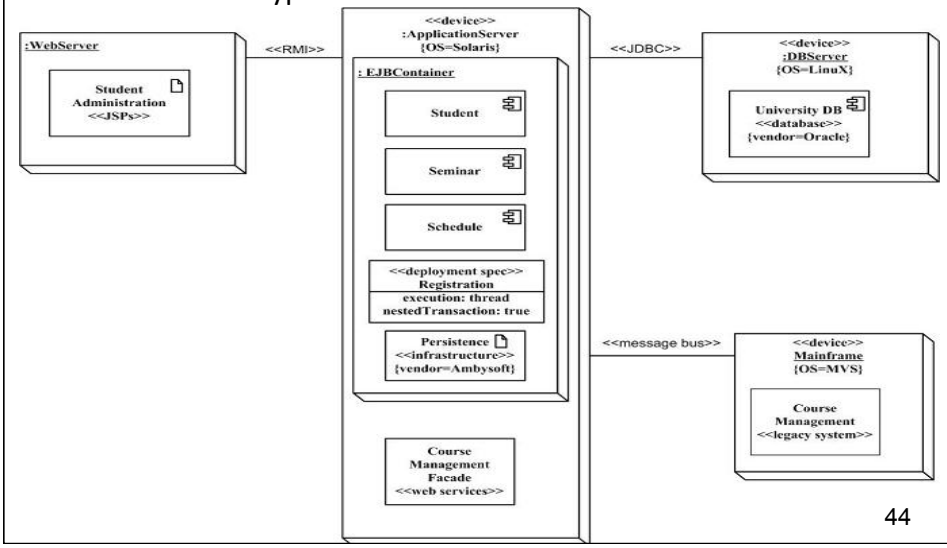
Deployment diagram



Packages being used to structure two types of nodes

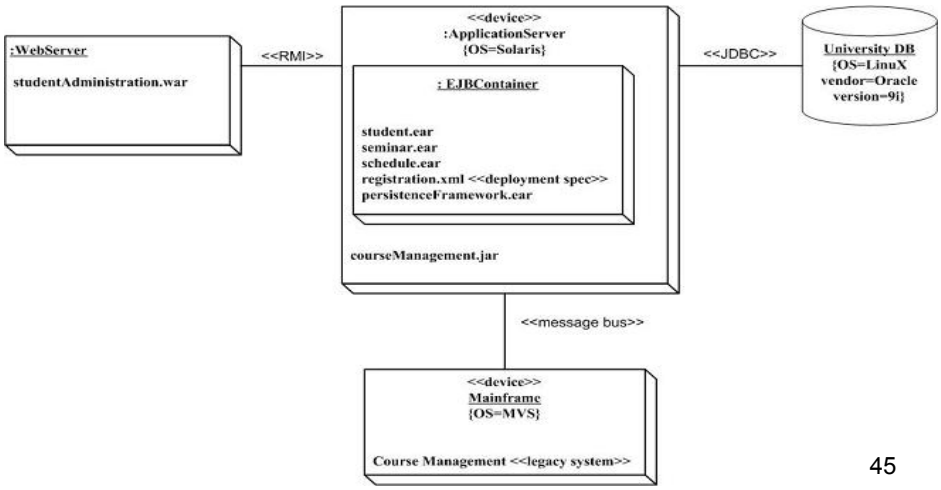
43

- Physical nodes should be labeled with the stereotype <<*device*>>.
- Connections between nodes are represented with simple lines, and are assigned stereotypes such as <<*RMI*>> and <<*message bus*>> to indicate the type of connection.



44

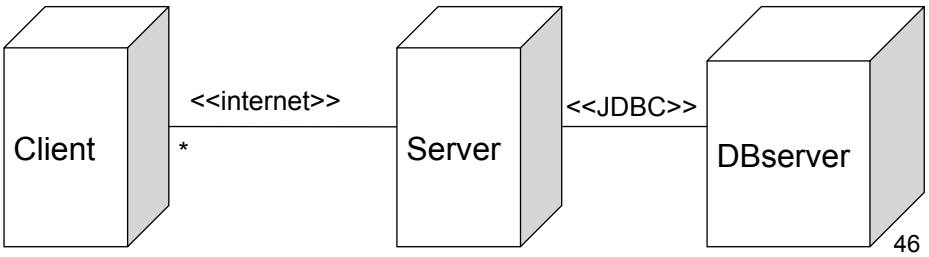
- Better, more concise example. Software elements are listed by their physical filenames (developers will be interested in this).
- A drum is used as a visual stereotype for the *University DB* database, making it easier to distinguish on the diagram.



45

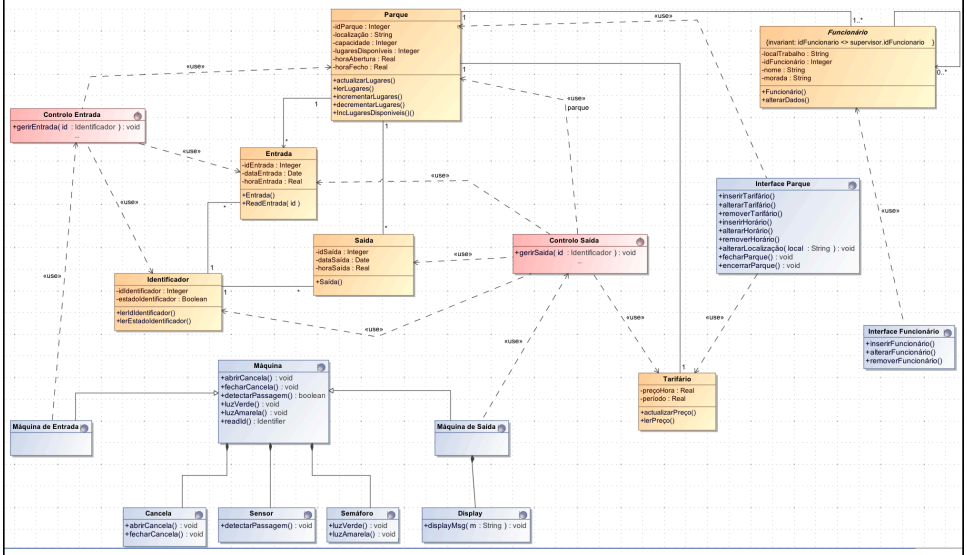
Deployment diagram

- Used to model
  - client-server systems
  - distributed systems
  - *embedded systems*
  - *...*



46

Still feeling strong?



47

Which deployment diagram would you propose?

- Do it for the component diagram of that class diagram

48

How Many Diagrams Needed?

- Depends:
  - We use diagrams to visualize the system from different perspectives.
  - No complex system can be understood in its entirety from one perspective.
  - Diagrams are used for communication
- Model elements will appear on one or more diagrams.
  - For example, a class may appear on one or more class diagrams, be represented in a state machine diagram, and have instances appear on a sequence diagram.
  - Each diagram will provide a different perspective.

49

Sources

- Agile Modeling
- IBM's Rational Library
- *The object primer: agile model-driven development with UML 2.0*, Scott W. Ambler
- UML 2.0 Superstrcture
- UML 2.o Infrastructure

50