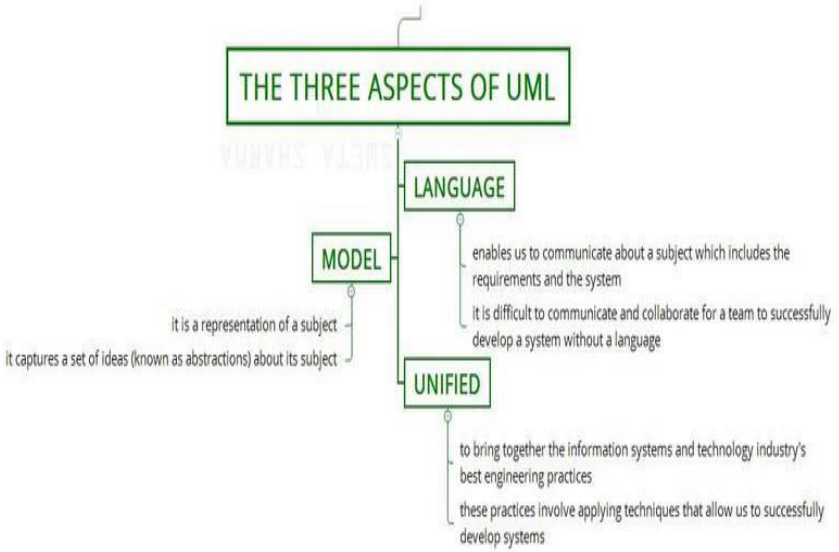
CS6001 – OOAD
Module 7



Implementation Diagram-UML package diagram -Component and Deployment Diagrams

Dr.S.Neelavathy Pari
Assistant Professor (Sr. Grade)
Department of Computer Technology
Anna University, MIT Campus

Instances & Object Diagrams

- "instance" and "object" are largely synonymous; used interchangeably.

- difference:
  - instances of a **class** are called objects or instances; but
  - instances of other abstractions (components, nodes, use cases, and associations) are not called objects but only instances.

What is an instance of an association called?

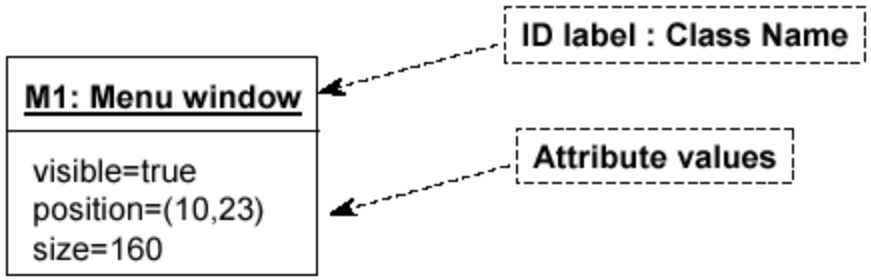
Object Diagrams

- very useful in debugging process.
  - walk through a scenario (e.g., according to use case flows).
  - Identify the set of objects that collaborate in that scenario (e.g., from use case flows).
  - Expose these object's states, attribute values and links among these objects.

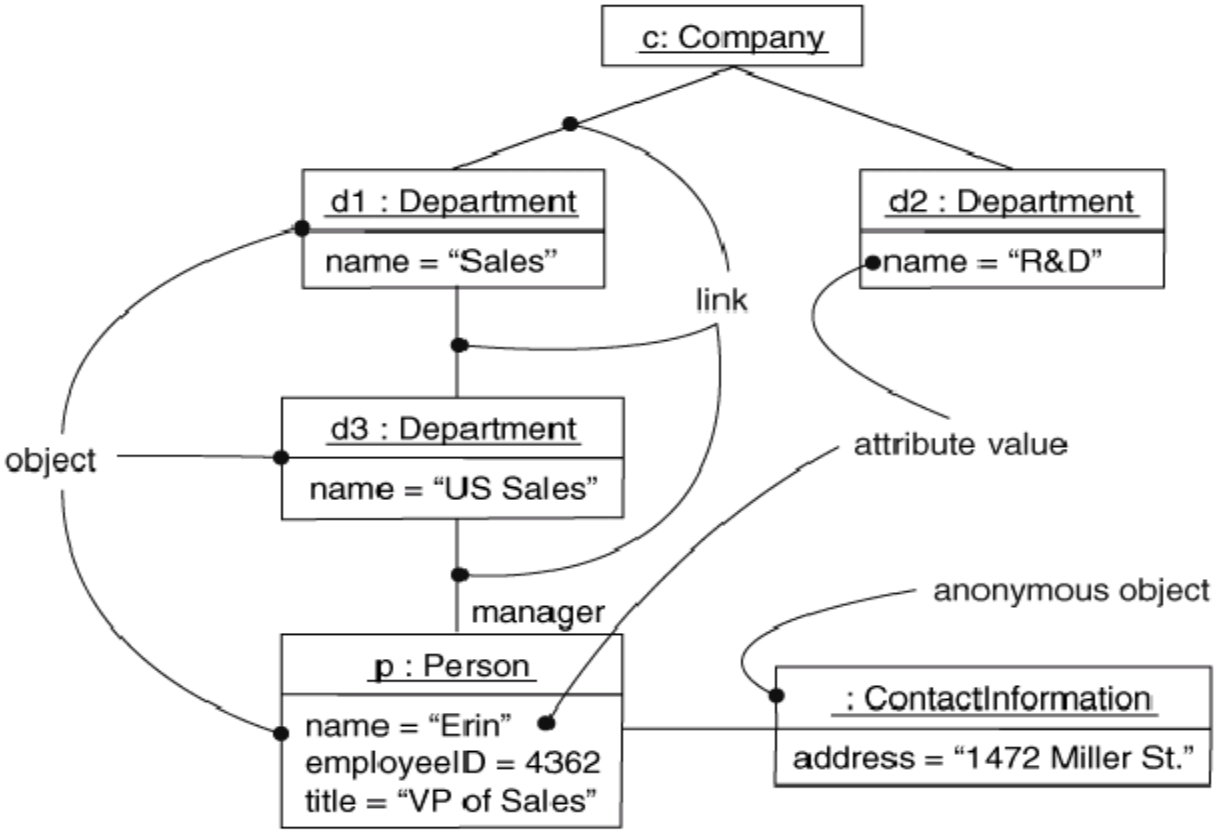
Object Diagrams

- Format is
  - Instance name : Class name
  - Attributes and Values

  - Example:

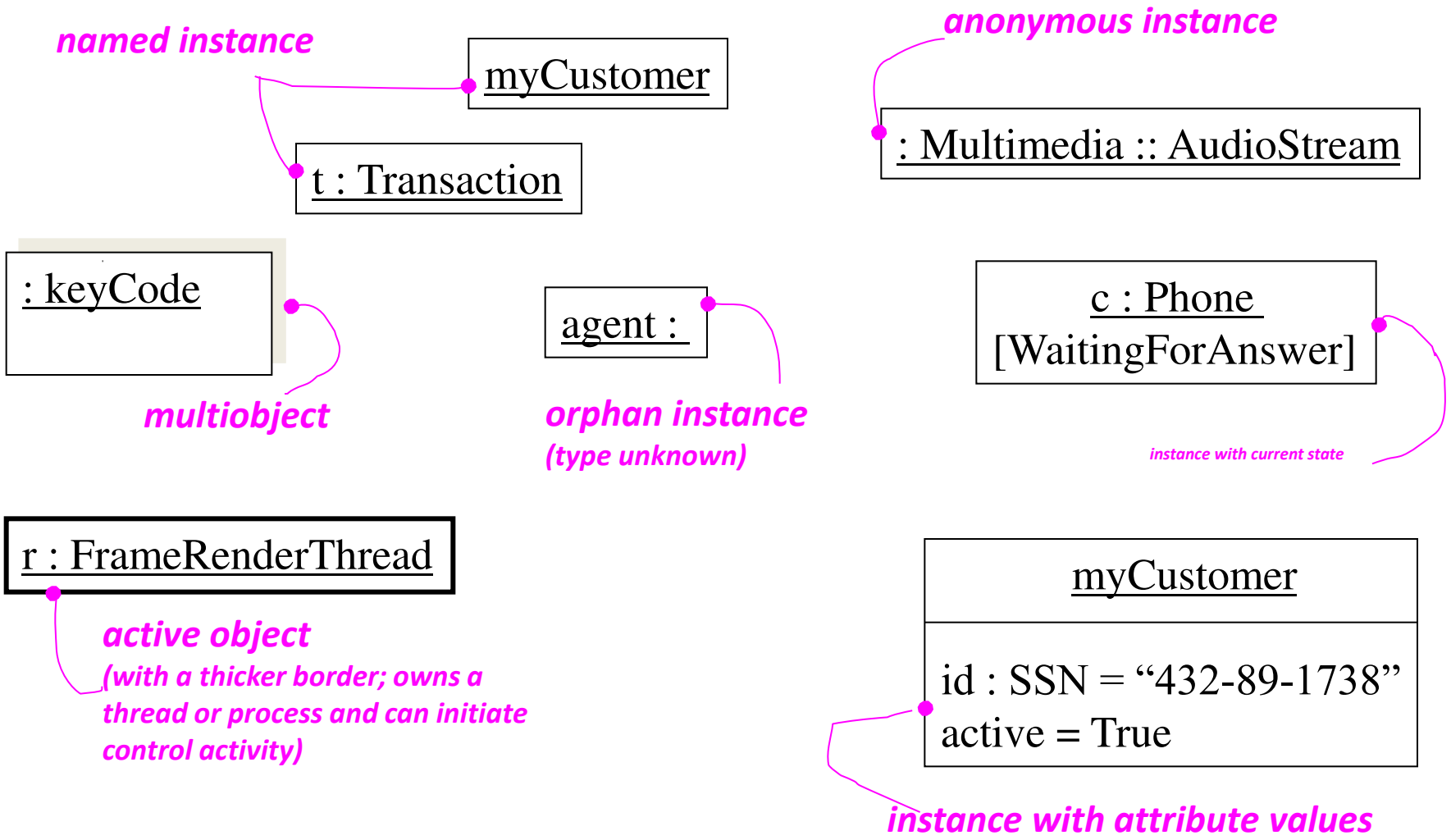


Objects and Links



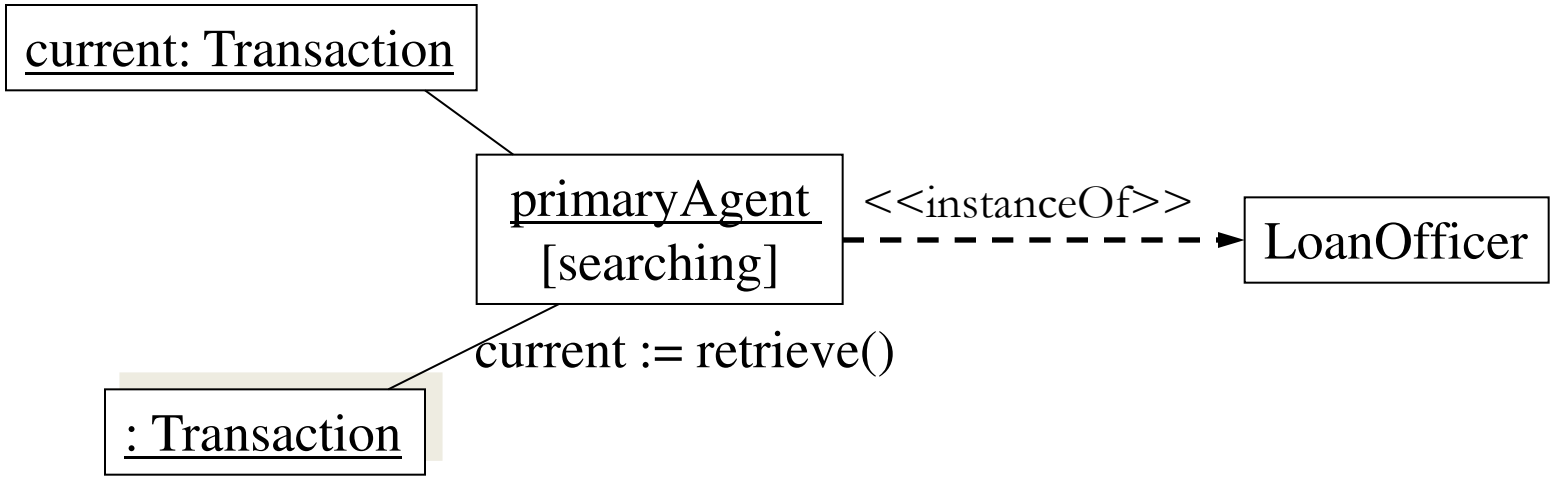
Can add association type and also message type

Instances & Objects - Visual Representation



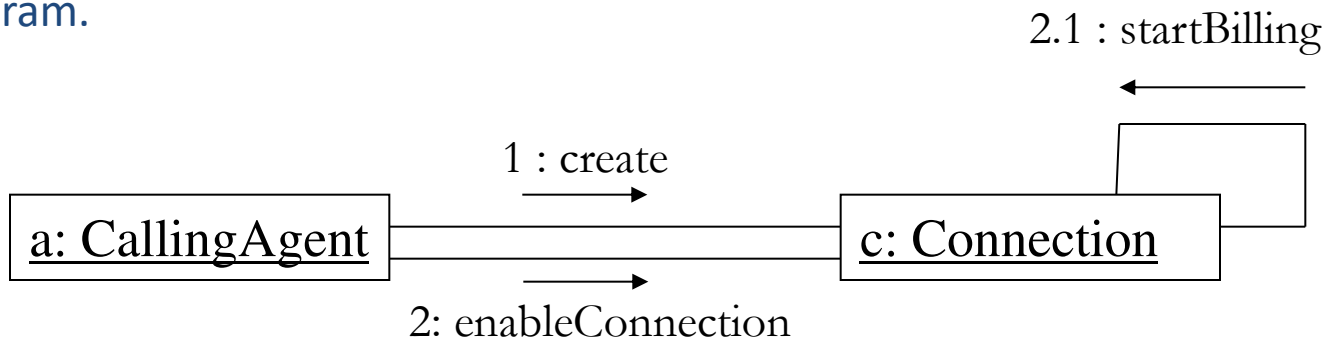
Instances & Objects - Modeling Concrete Instances

- Expose the stereotypes, tagged values, and attributes.
- Show these instances and their relationships in an object diagram.

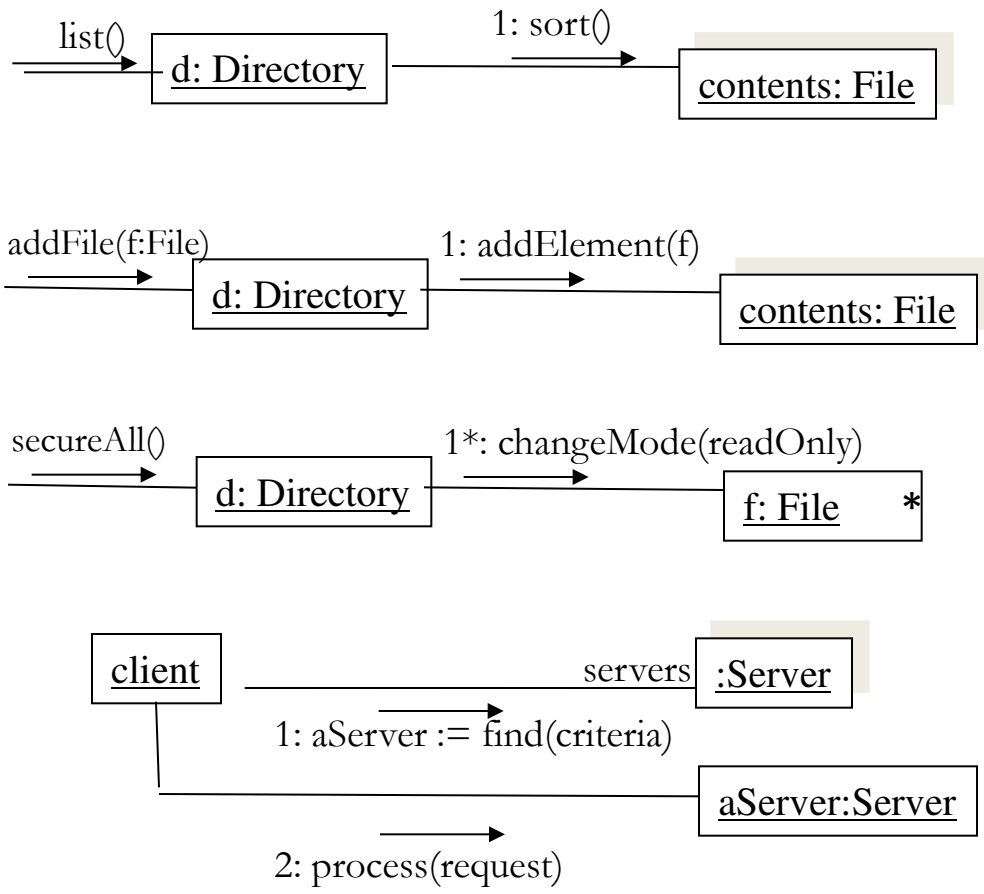


Instances & Objects - Modeling Prototypical Instances

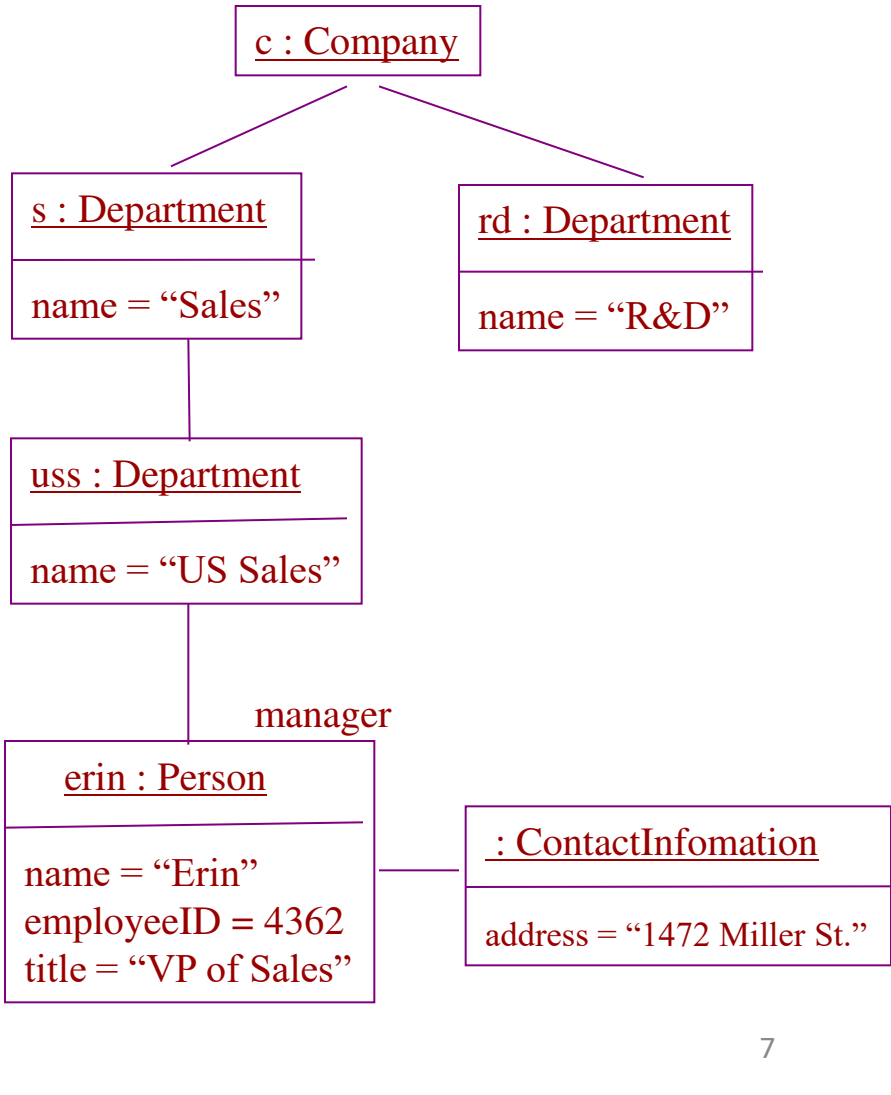
- Show these instances and their relationships in an interaction diagram or an activity diagram.



Instances & Objects – More Examples



call ::= label [guard] ["*"] [return-val-list ":="] msg-name "(" arg-list ")"

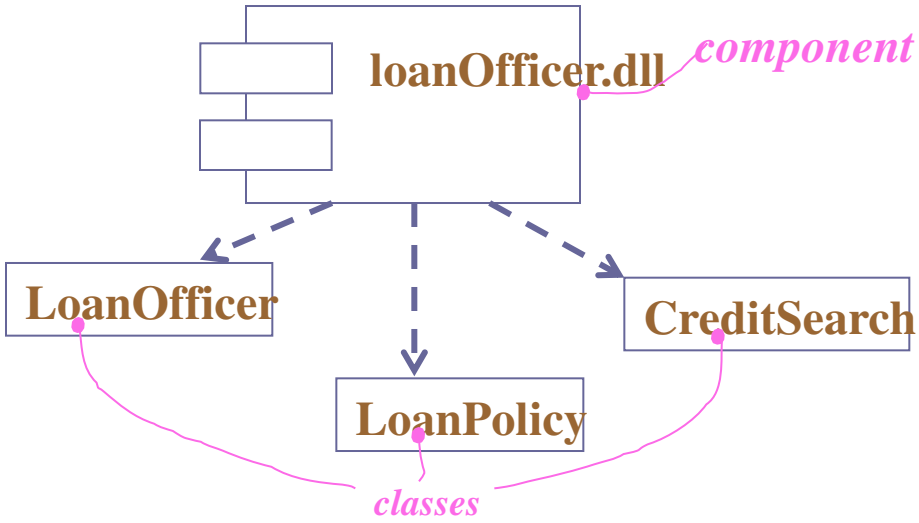


Component Diagrams

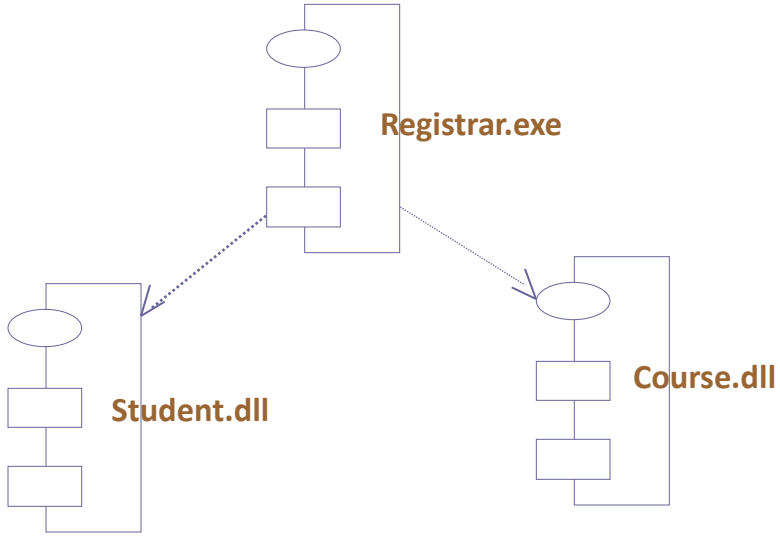
Component Diagram

- Shows a set of components and their relationships.
- Represents the static implementation view of a system.
- Components map to one or more classes, interfaces, or collaborations.

Mapping of Components into Classes



Components and their Relationships



Component Diagram

UML2.0 – architectural view

Big demand, hmm…

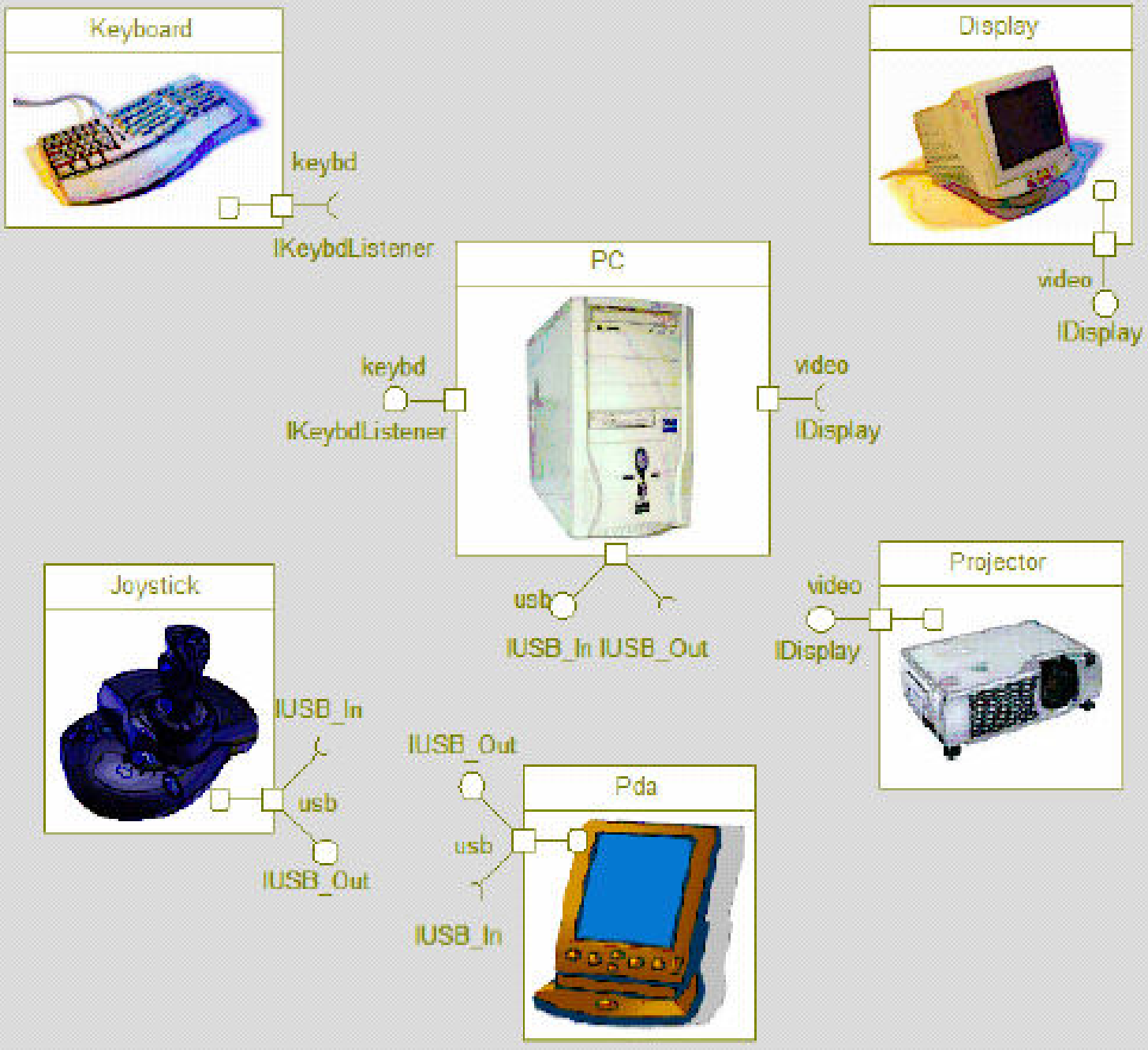
- Short history behind architecture
- Architecture still an emerging discipline
- Challenges, a bumpy road ahead

- UML and architecture evolving in parallel
- Component diagram in need of better formalization and experimentation

Component Diagram – another example

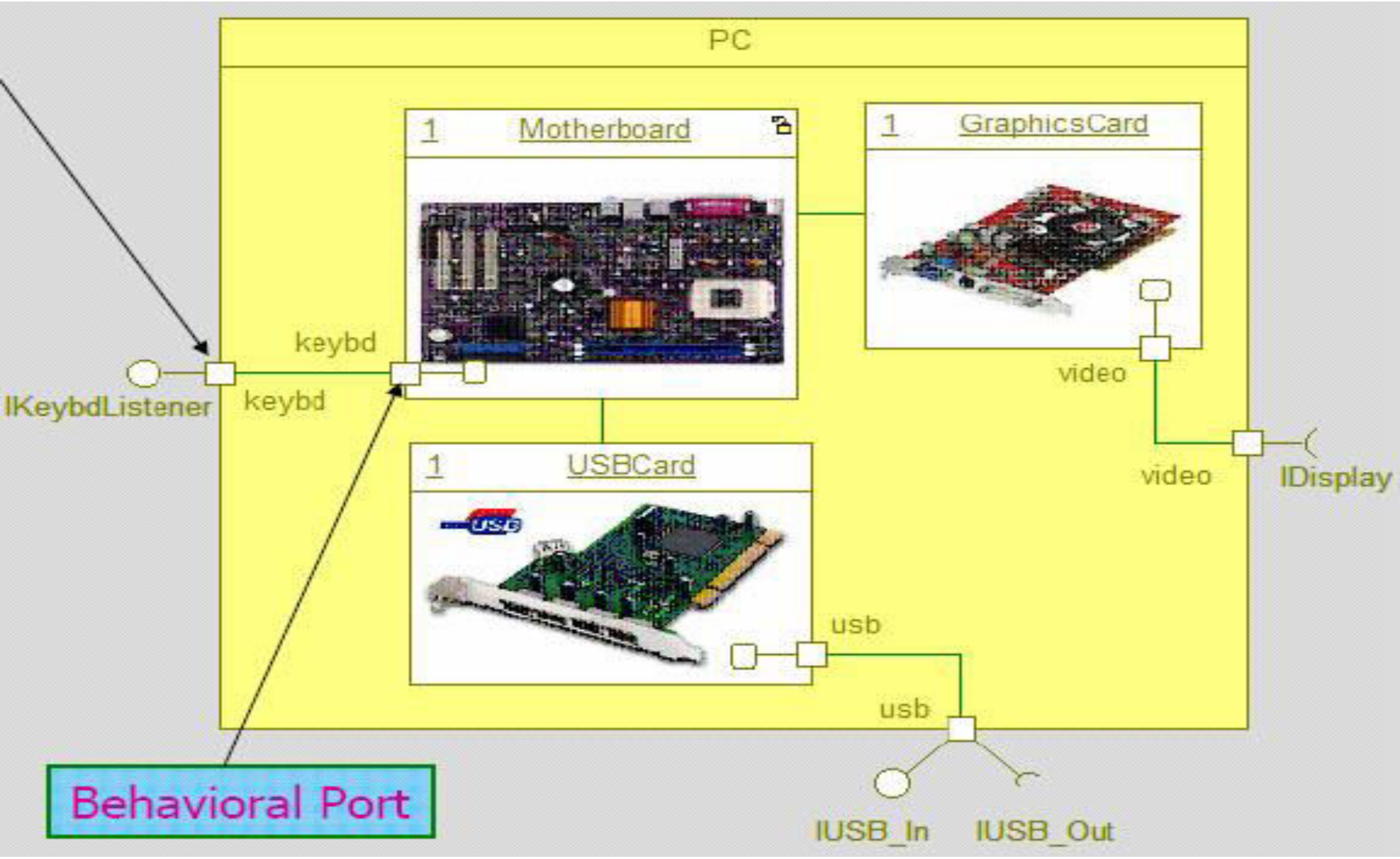
(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)

Component Diagram – another example



Component Diagram – another example

(www.cs.tut.fi/tapahtumat/olio2004/richardson.pdf)

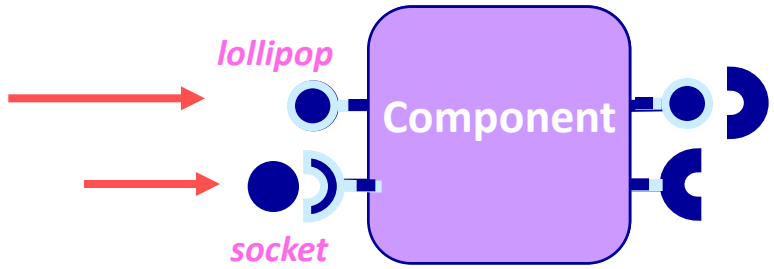


Component Diagram

UML2.0 – architectural view

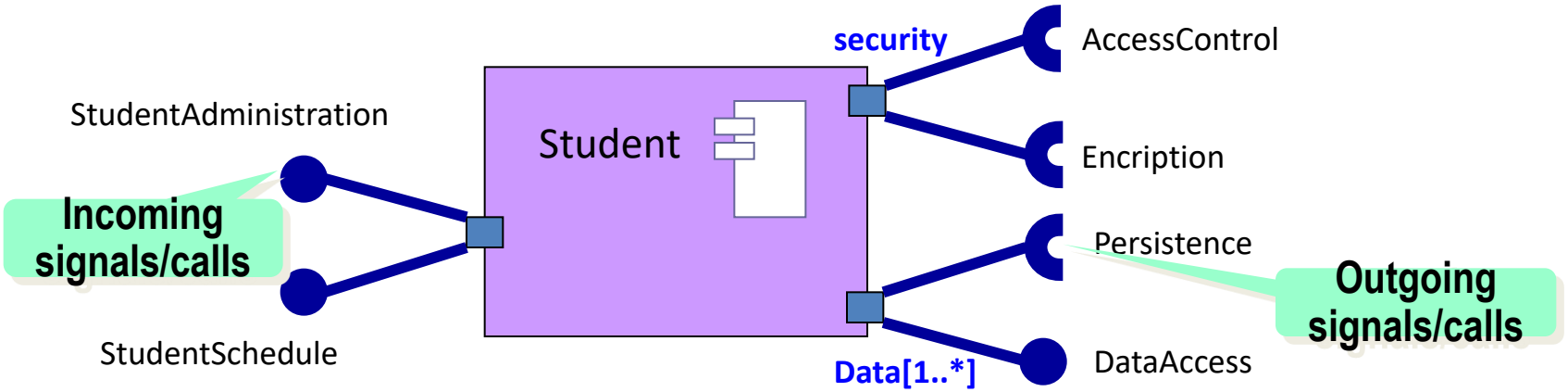
Explicit description of *interfaces*:

- provided services to other components
- requested services from other components



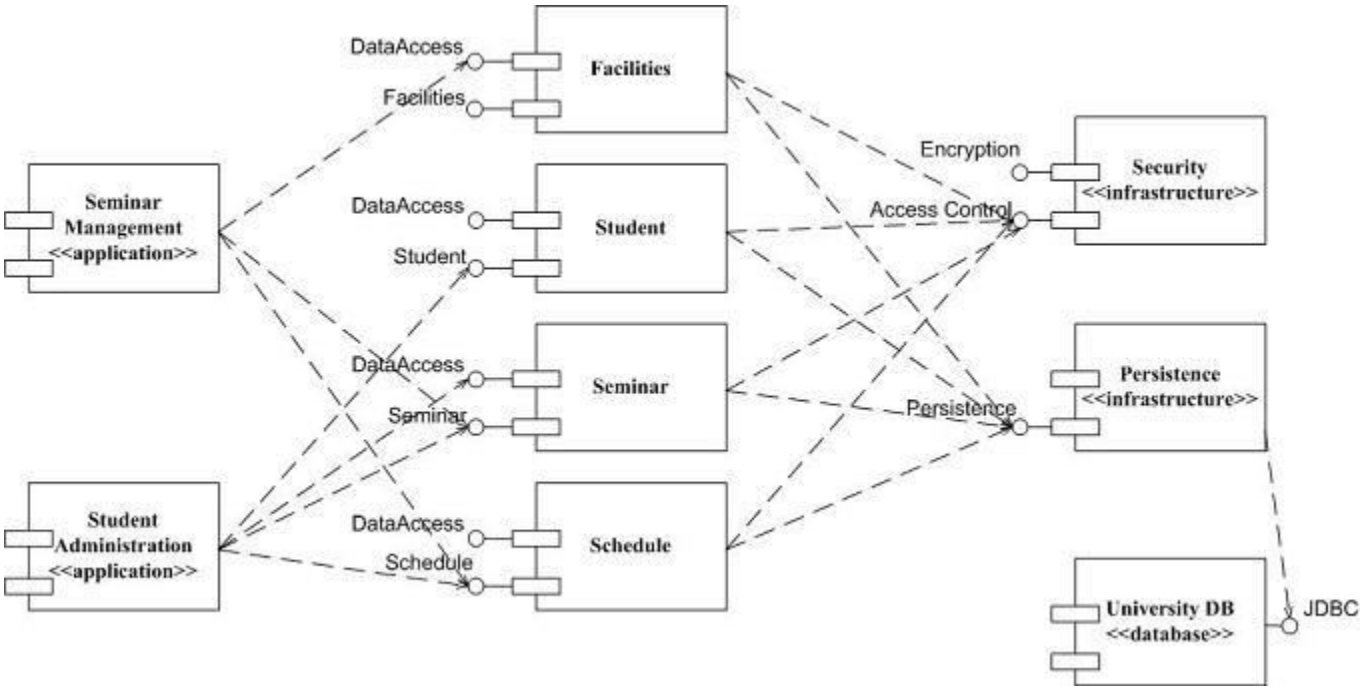
- An interface is a collection of 1..* methods, and 0..* attributes
- Interfaces can consist of synchronous and / or asynchronous operations
- A *port* (*square*) is an interaction point between the component and its environment.
- Can be named; Can support uni-directional (either provide or require) or bi-directional (both provide and require) communication; Can support multiple interfaces.
- possibly concurrent interactions
- fully isolate an object's internals from its environment

caller or callee?



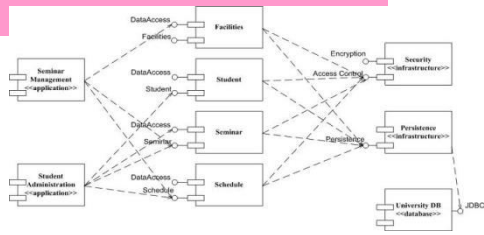
Component Diagram: UML 1.x and UML 2.0

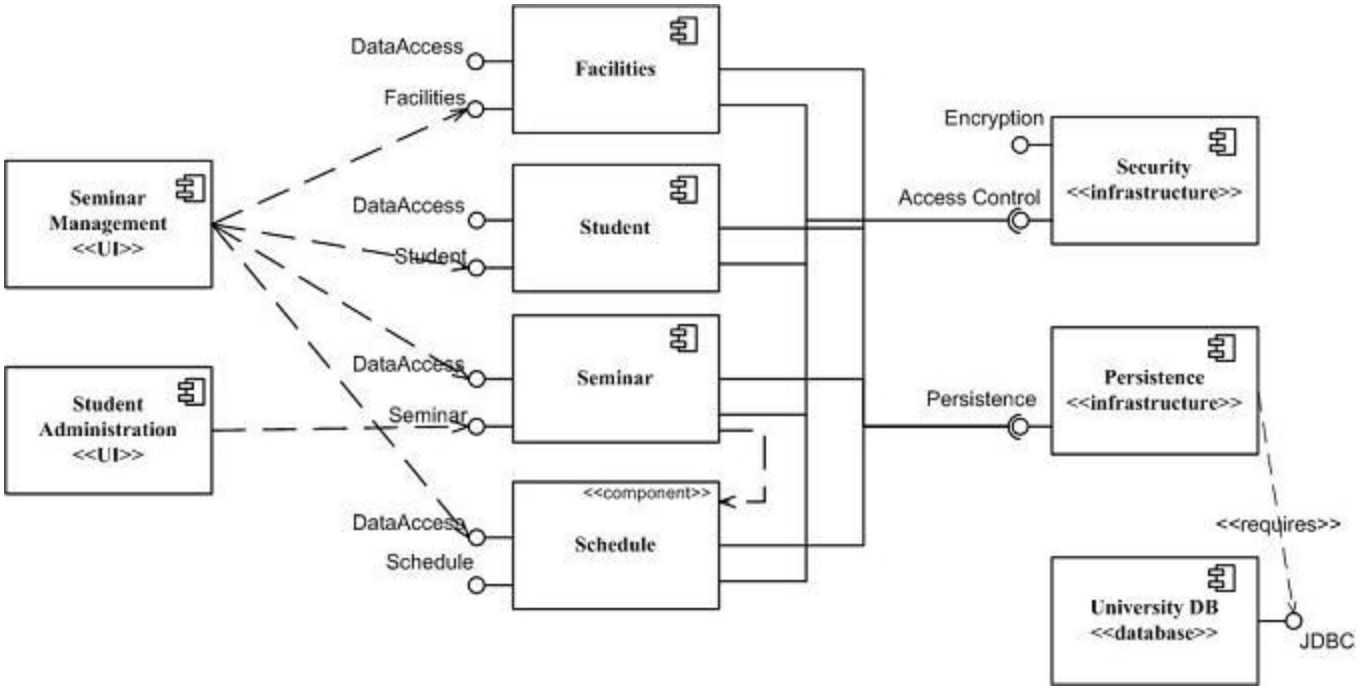
(http://www.agilemodeling.com/artifacts/componentDiagram.htm)



Component Diagram: UML 1.x and UML 2.0

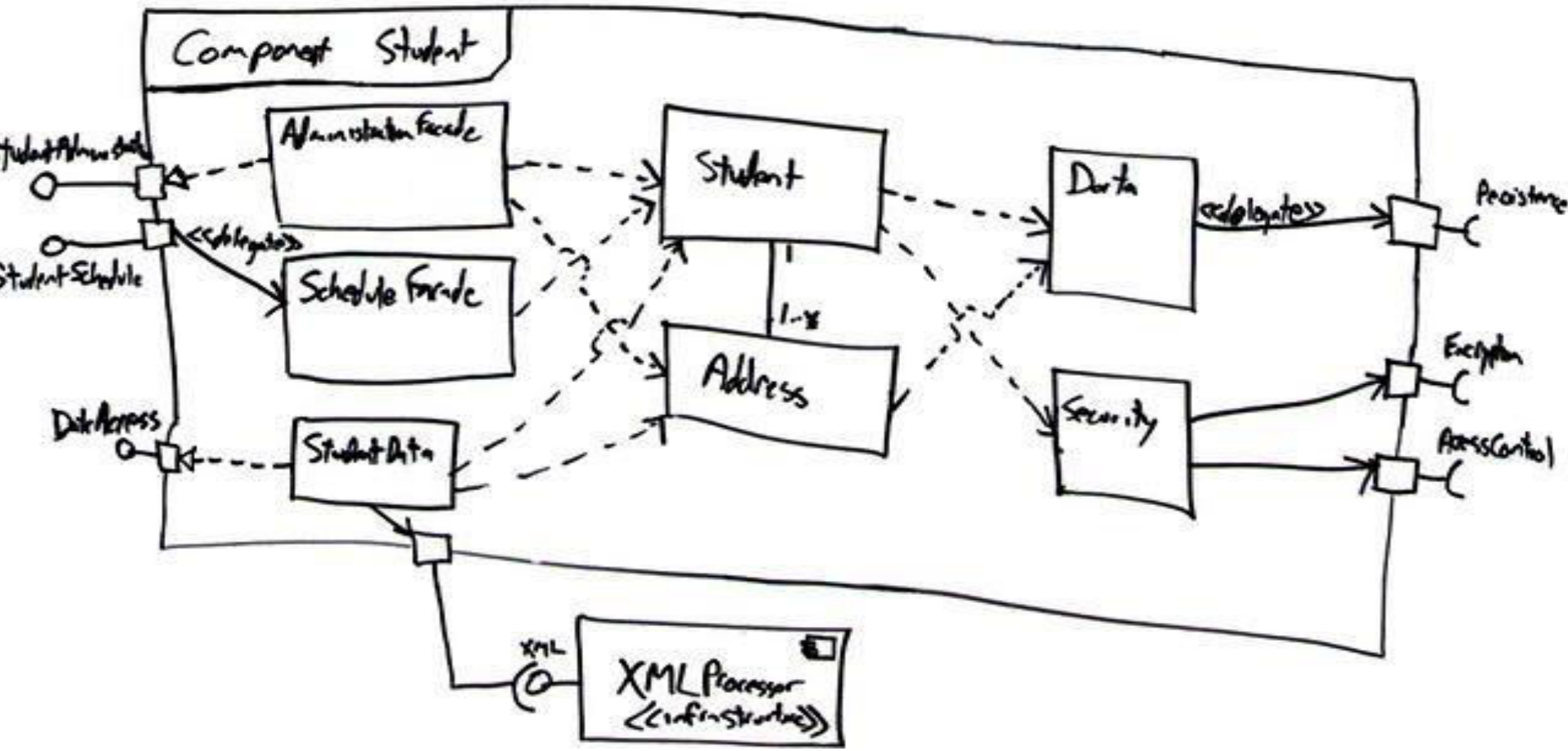
(http://www.agilemodeling.com/artifacts/componentDiagram.htm)





So, how many different conventions for components in UML2.0?

Building a Component

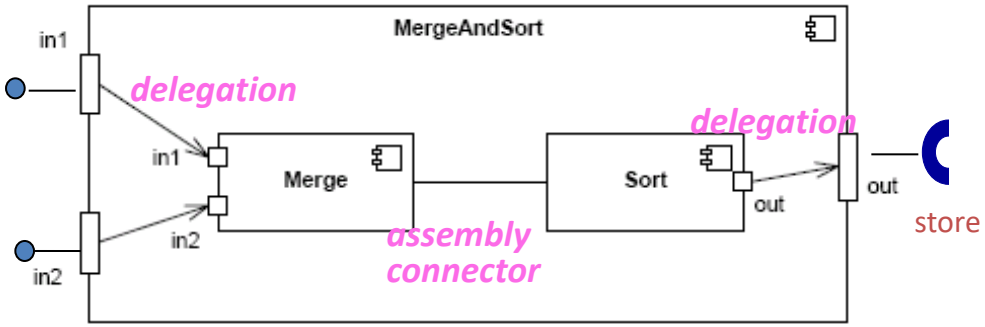


- simplified the ports to either provide or require a single interface
- relationships between ports and internal classes in three different ways:
  i) as *stereotyped delegates* (flow), as *delegates*, and as *realize*s (logical->physical) relationships
- Cohesive reuse and change of classes; acyclic component dependency ***???***

Component Diagram – Connector & Another Example

- a connector: just a link between two or more connectable elements (e.g., ports or interfaces)
- 2 kinds of connectors: *assembly* and *delegation*. For ***"wiring"***
  - An ***assembly*** connector: a binding between a provided interface and a required interface (or ports) that indicates that one component provides the services required by another; ***simple line/ball-and-socket/lollipop-socket notation***
  - A ***delegation*** connector binds a component's external behavior (as specified at a port) to an internal realization of that behavior by one of its parts *(provide-provide, request-request).*

External View of a Component with Ports



Internal View of a Component with Ports

Left delegation: direction of arrowhead indicates "provides"

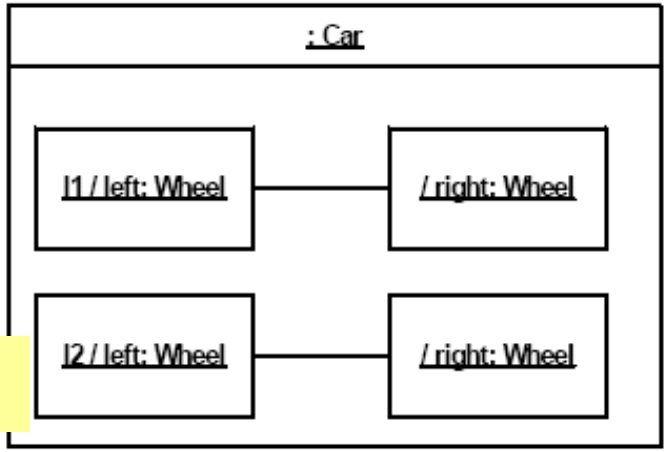
Right delegation: direction of arrowhead indicates "requests"

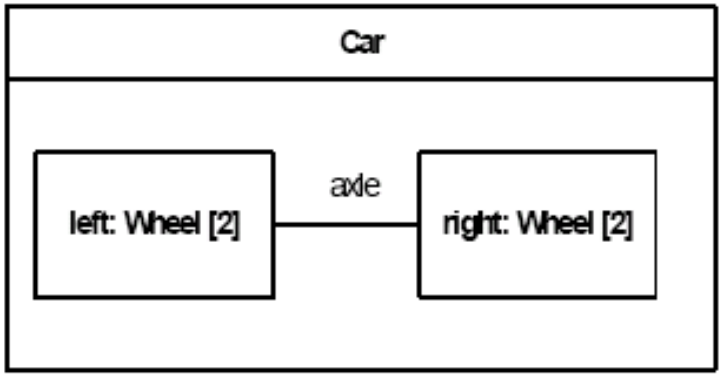
So, what levels of abstractions for connections?

Structured Class

- A *structured class(ifier)* is defined, in whole or in part, in terms of a number of *parts* - contained instances owned or referenced by the structured class(ifier).
- With a similar meaning to a **composition** relation
- A structured classifier's parts are created within the containing classifier (either when the structured classifier is created or later) and are de is destroyed.
- Like classes and components, combine the descrip ers with ***ports and interfaces***

Any difference?





component or class?

connector

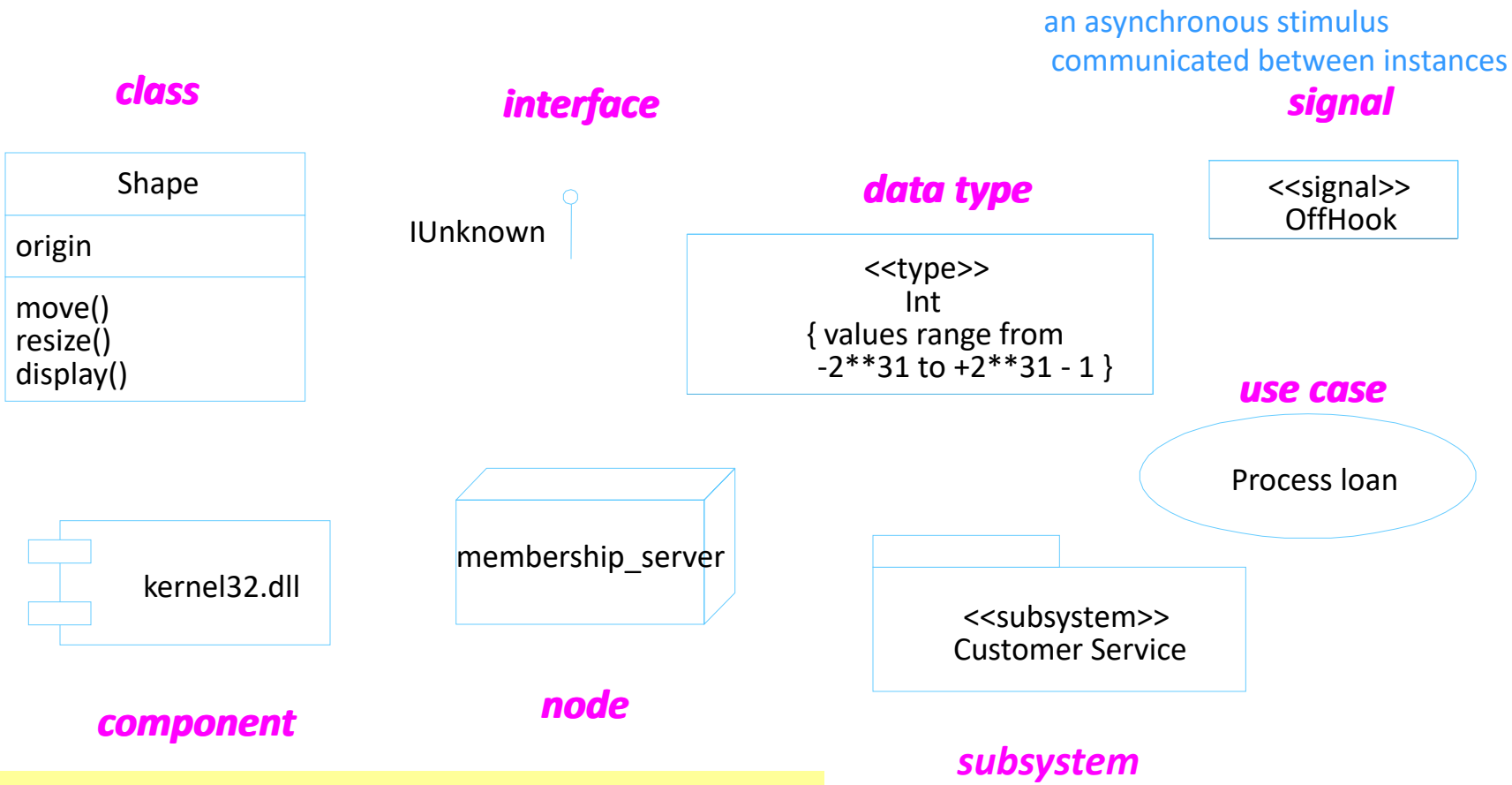
label /roleName : type

Components extend classes with additional features such as

- the ability to ***own more types of elements*** than classes can; e.g., packages, constraints, use cases, and artifacts
- deployment specifications that define the execution parameters of a component deployed to a node

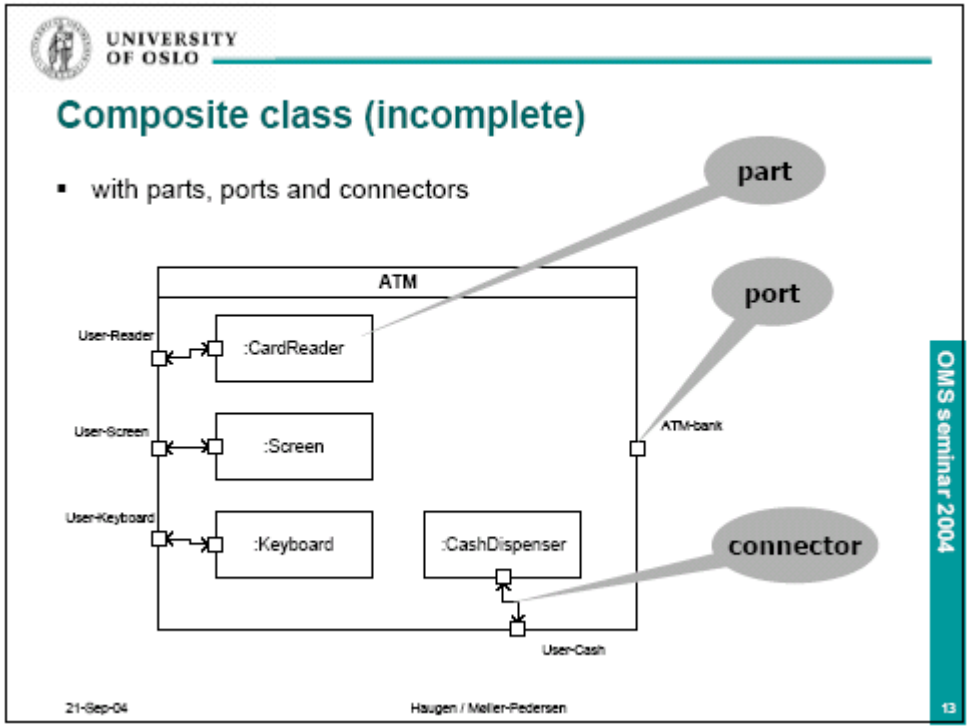
Classifiers

- Classifier—mechanism that describes structural (e.g. class attributes) and behavioral (e.g. class operations) features. In general, those *modeling elements* that can have *instances* are called classifiers.
- *cf. Packages and generalization relationships do not have instances.*



Generalizable Element, Classifier, Class, Component?

Structured Class – Another Example

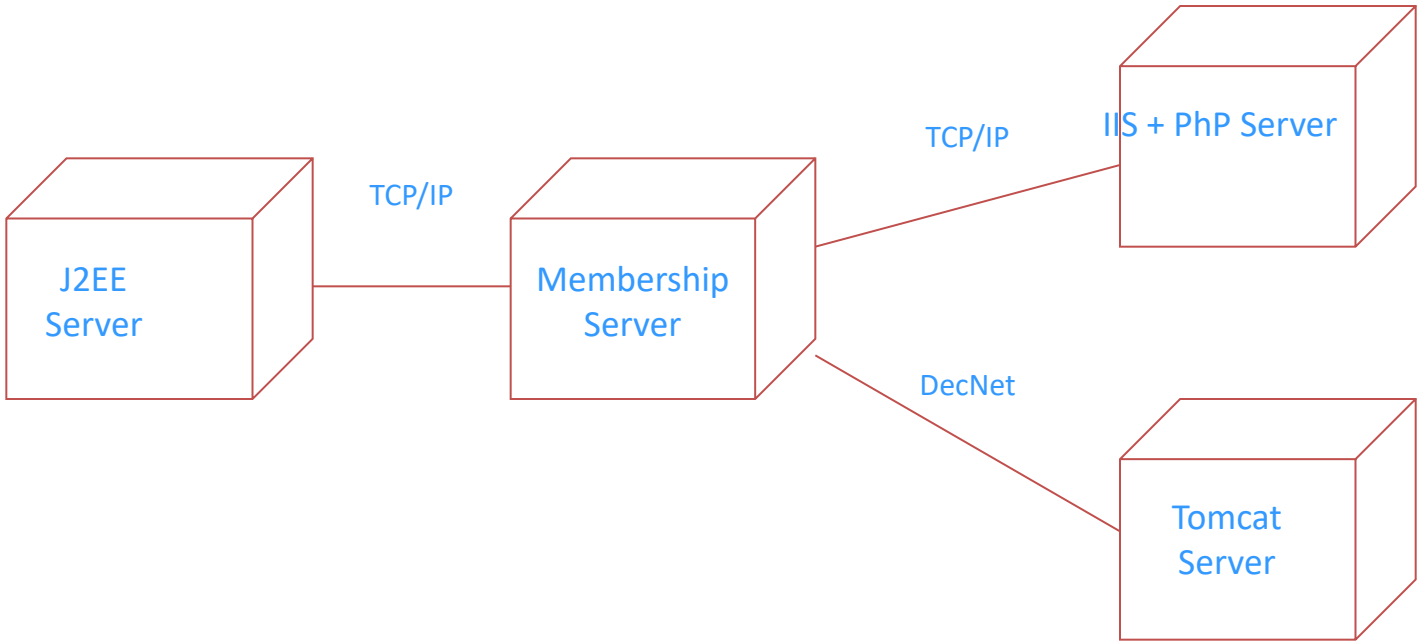


what kind?

Deployment Diagrams

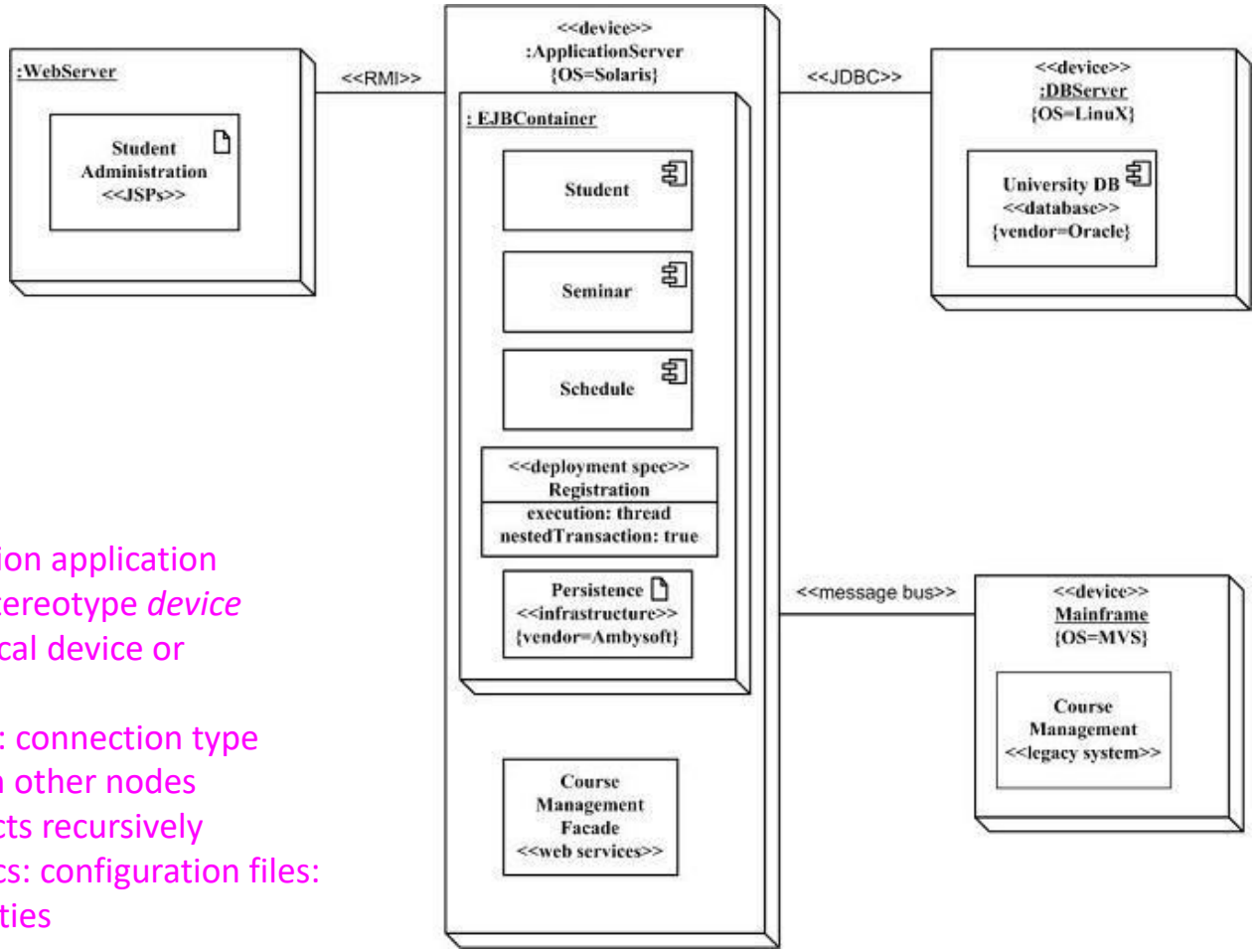
Deployment Diagram

- Shows a set of *processing* nodes and their relationships.
- Represents the static deployment view of an *architecture*.
- Nodes typically enclose one or more components.



Structural Diagrams - Deployment Diagram

http://www.agilemodeling.com/artifacts/deploymentDiagram.htm)



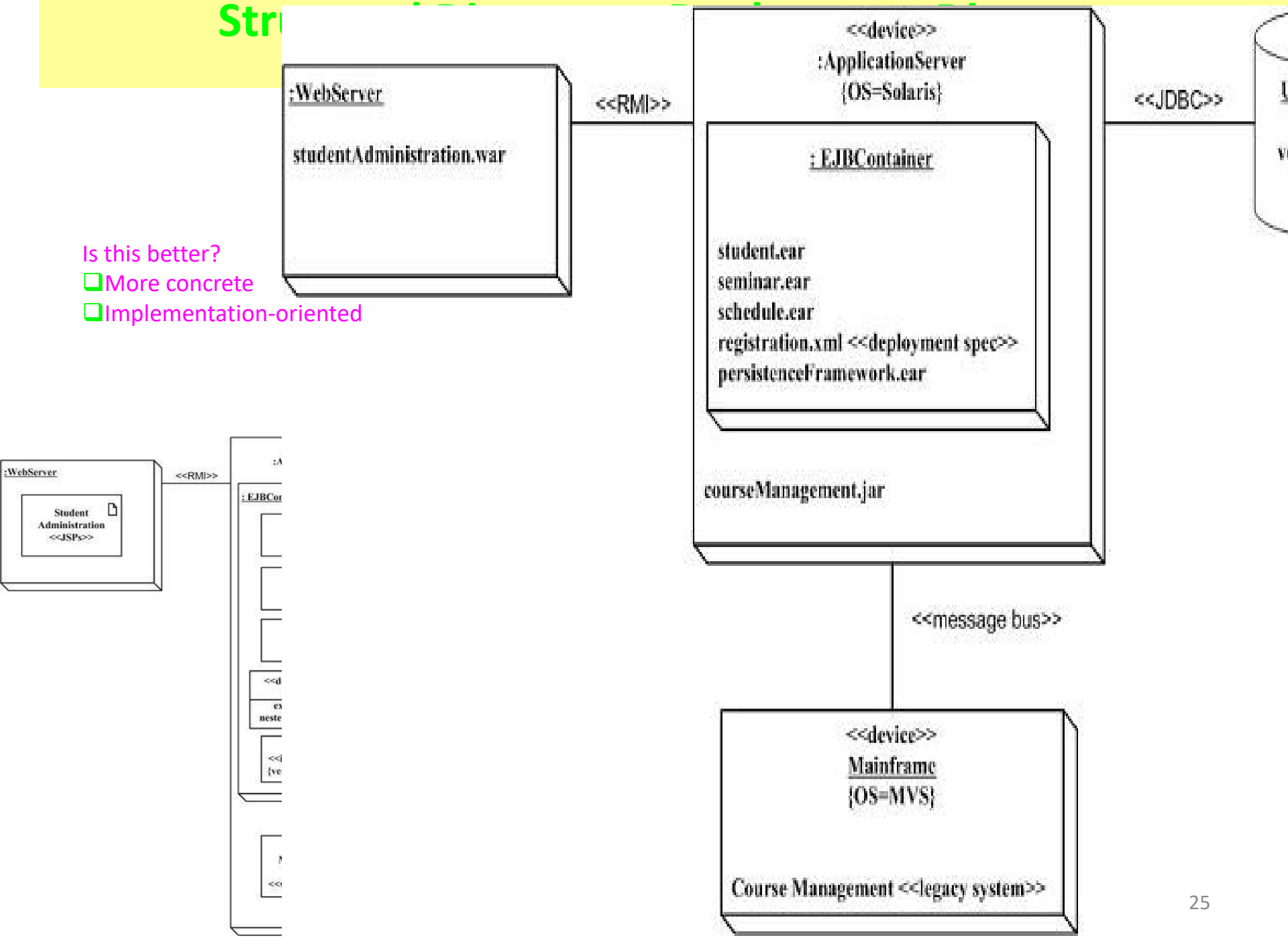
Student administration application

- Physical nodes - stereotype *device*
- *WebServer* - physical device or software artifact
- *RMI*/*message bus*: connection type
- Nodes can contain other nodes or software artifacts recursively
- Deployment specs: configuration files: name and properties

Str

Is this better?

- More concrete
- Implementation-oriented

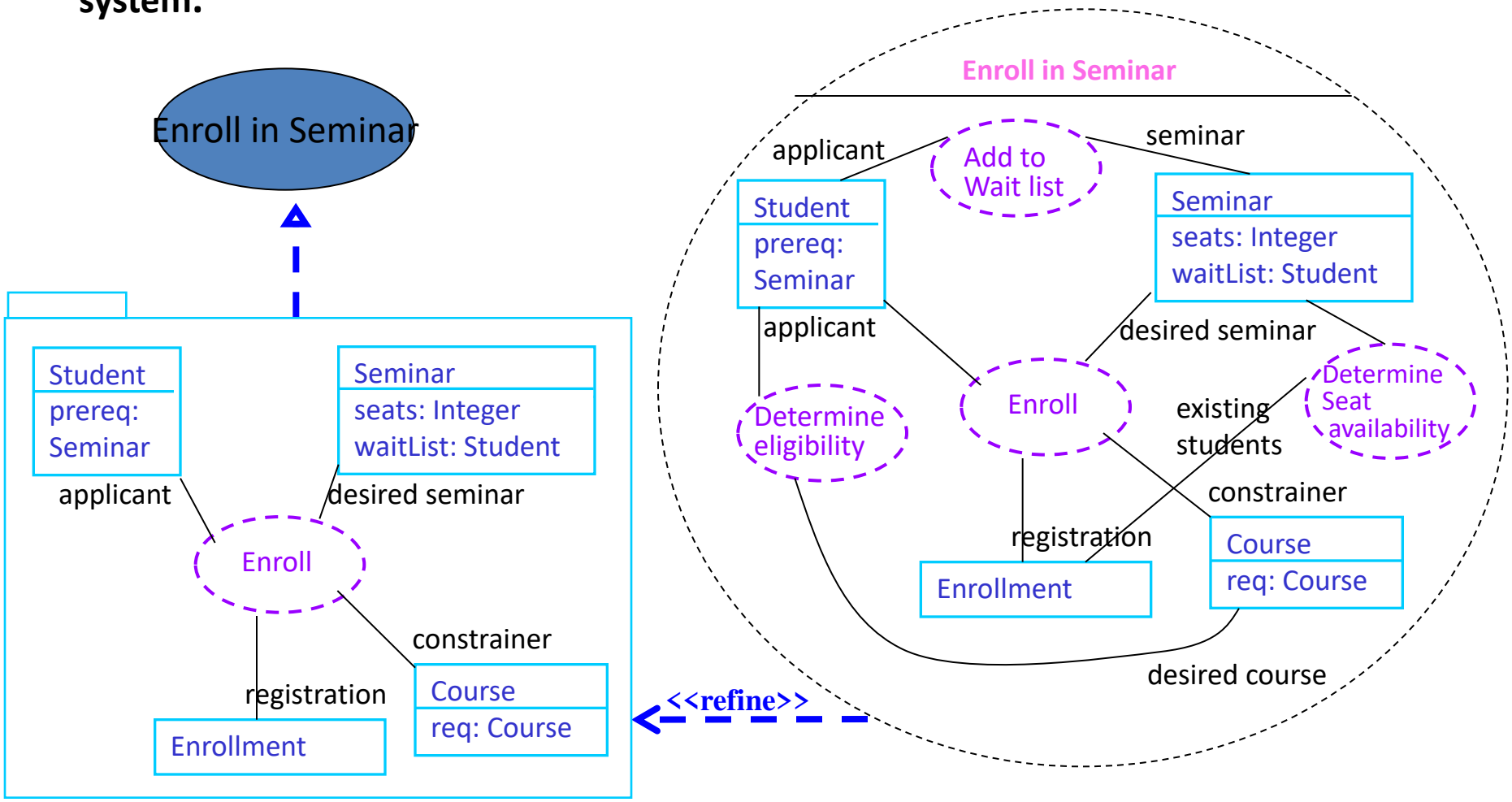


Composite Structure Diagrams

Composite Structure Diagrams

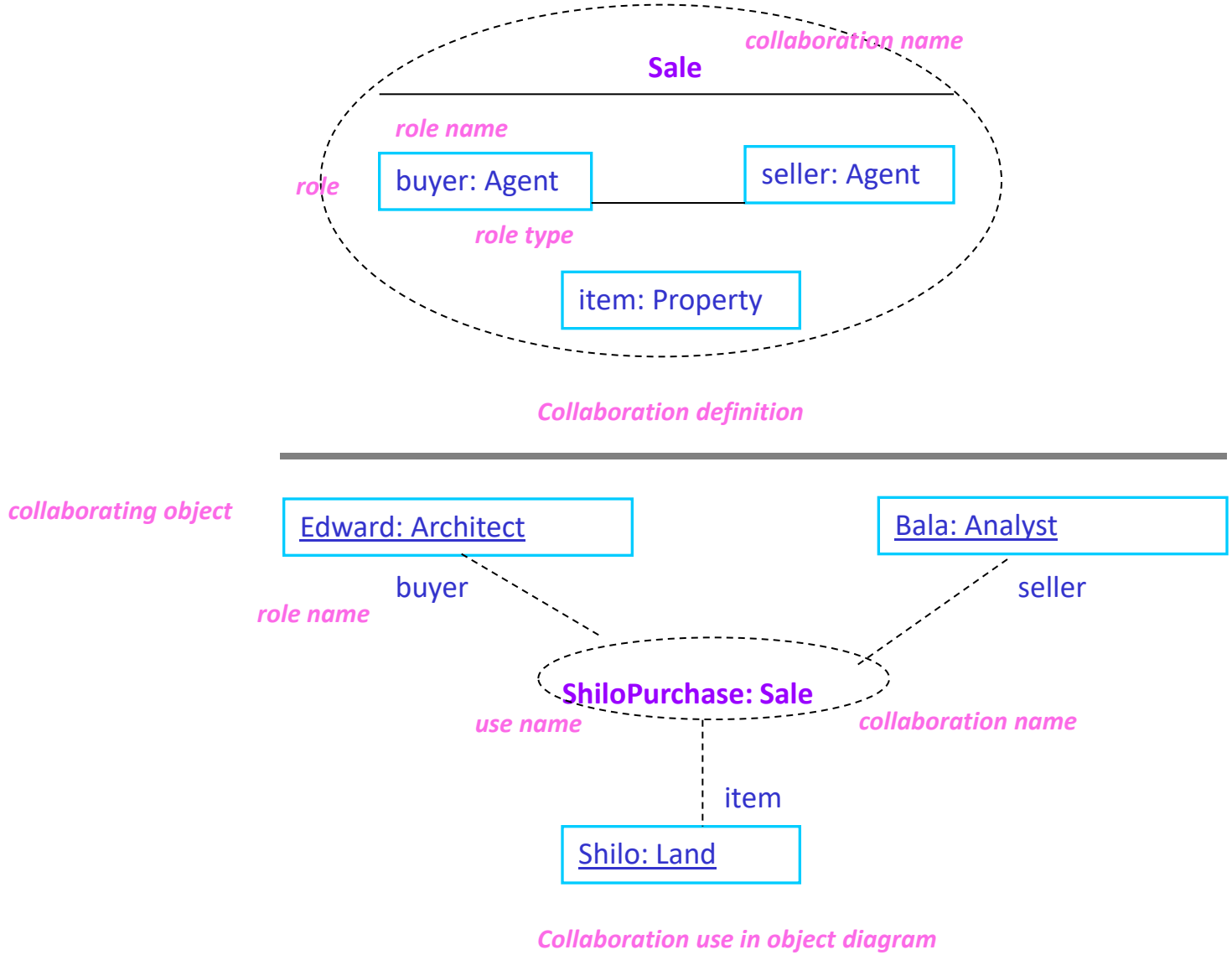
(http://www.agilemodeling.com/artifacts/compositeStructureDiagram.htm)

- **Depicts the internal structure of a classifier (such as a class, component, or collaboration), including the interaction points of the classifier to other parts of the system.**



structured class, structured component, structured use case, structured node, structured interface, …

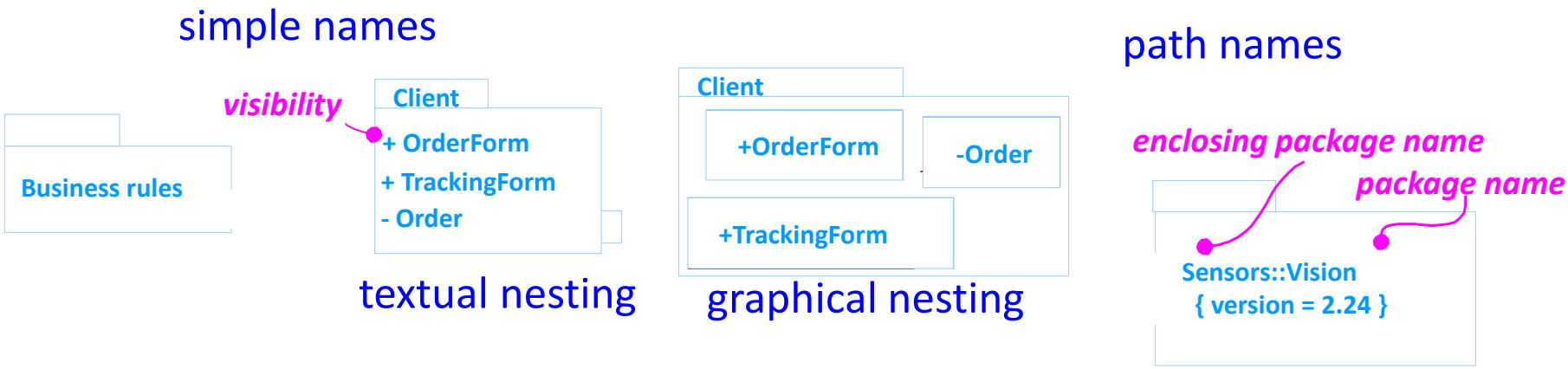
Variations [Rumbaugh – UML 2.0 Reference: p234]



Packages

Packages

- Package — general-purpose mechanism for organizing elements into groups.
- Nested Elements: Composite relationship (When the whole dies, its parts die as well, but not necessarily vice versa)
- (C++ namespace; specialization means “derived”)

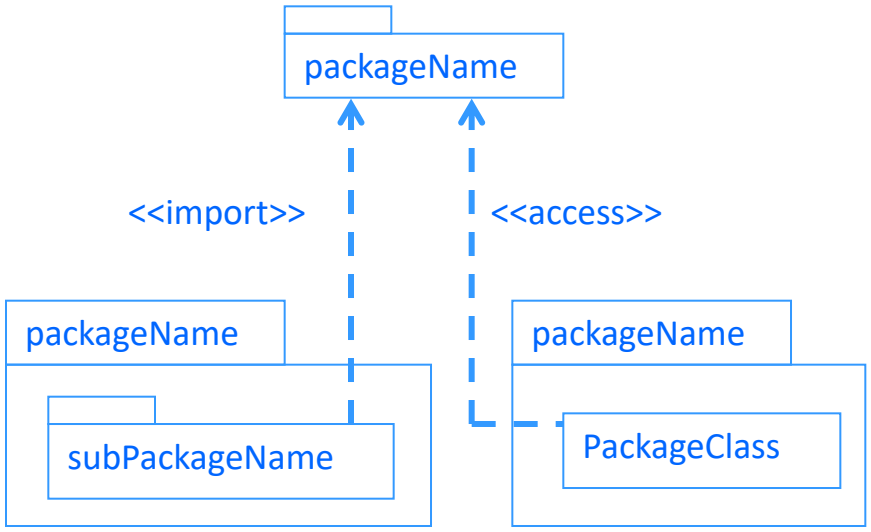


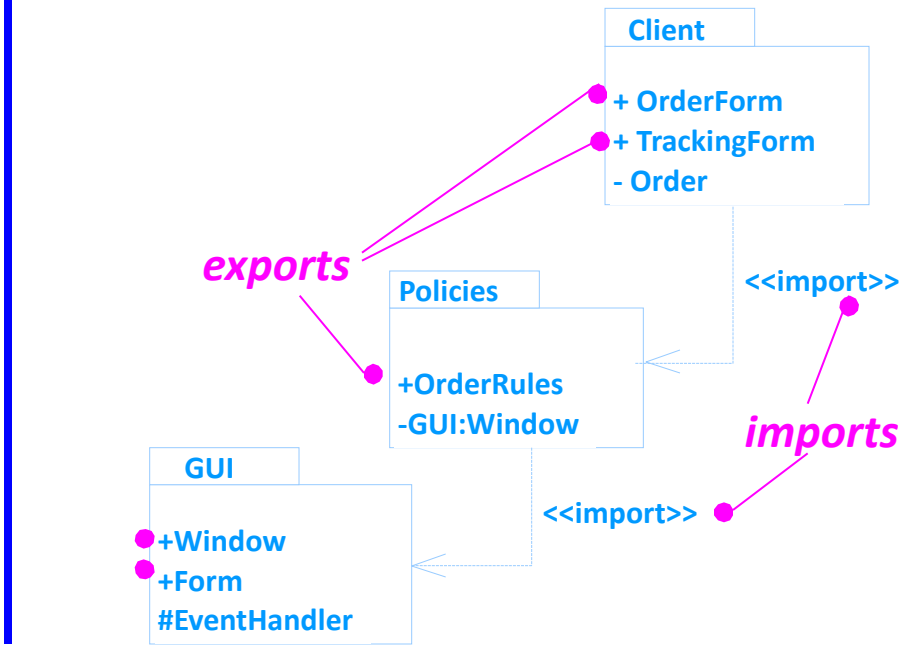
Visibility

- Packages that are friends to another may see all the elements of that package, no matter what their visibility.
- If an element is visible within a package, it is visible within all packages nested inside the package.

Dependency –Among Packages

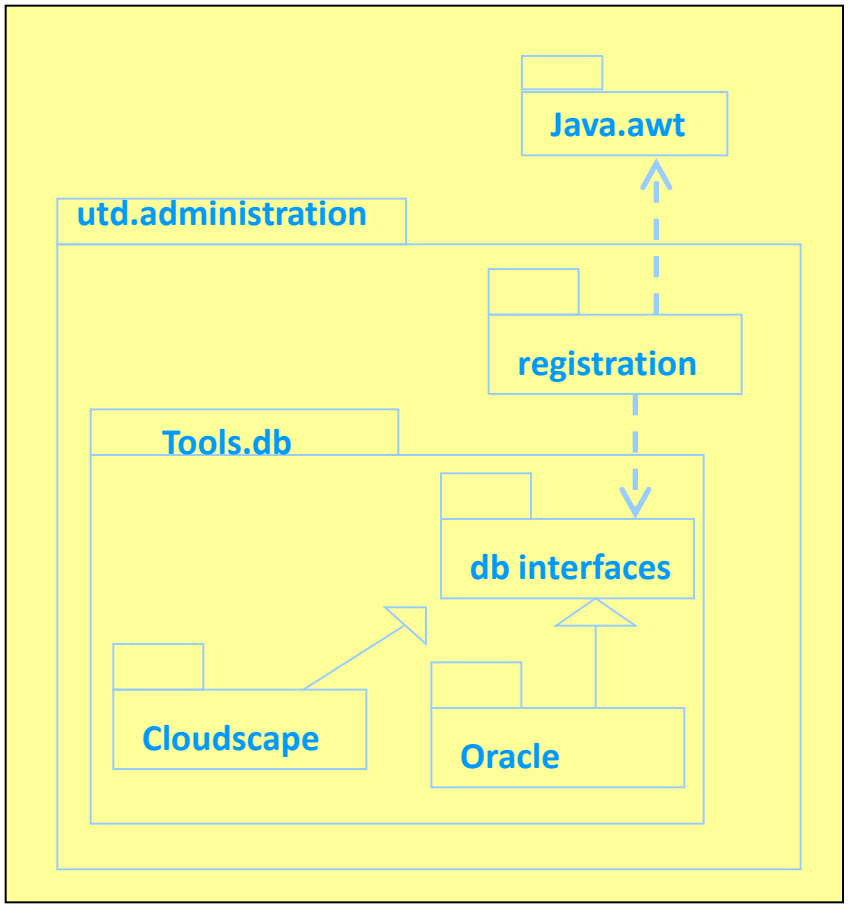
- Two Stereotypes of Dependency Among Packages:
  - *access*: the source package is granted the right to reference the elements of the target package *(:: convention)*
  - *import*: a kind of access; the ***public*** contents of the target package enter the flat namespace of the source as if they had been declared in the source





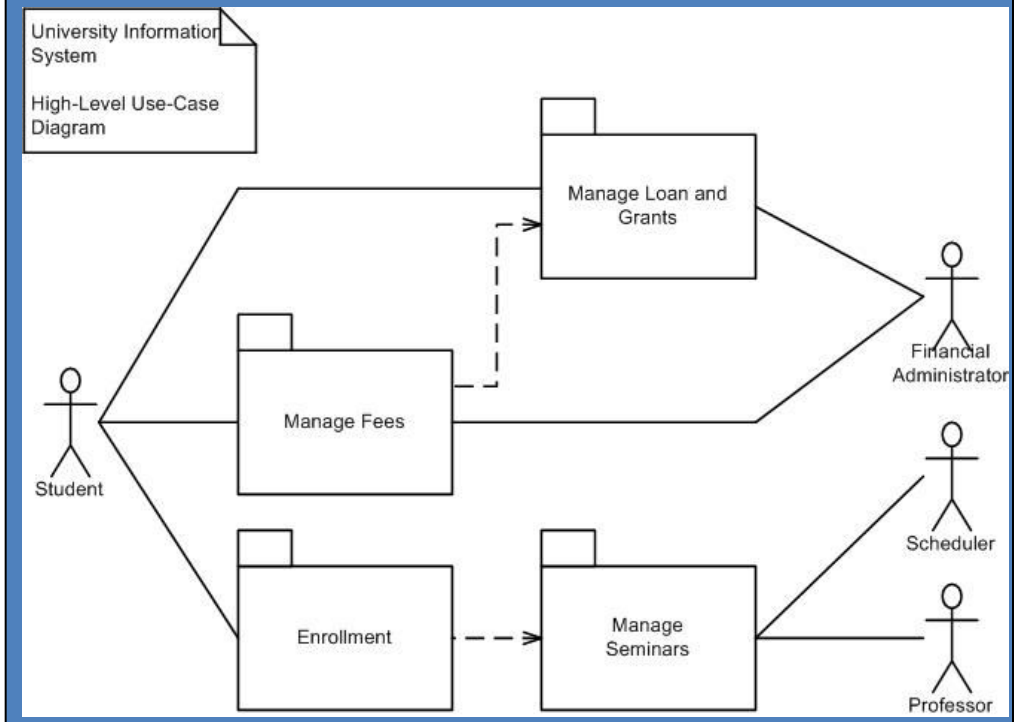
Modeling Groups of Elements

- Look for “clumps” of elements that are semantically close to one another.
- Surround “clumps” with a package.
- Identify public elements of each package.
- Identify import dependencies.



Use Case package Diagram

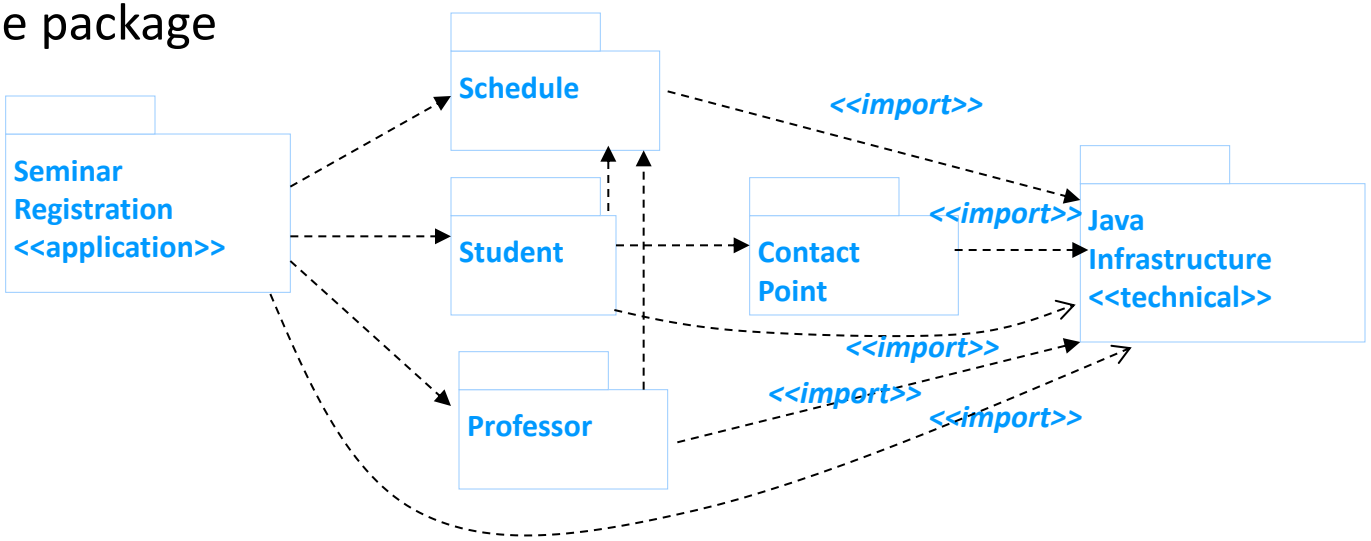
- Included and extending use cases belong in the same package as the parent/base use case
- Cohesive, and goal-oriented packaging
- Actors could be inside or outside each package



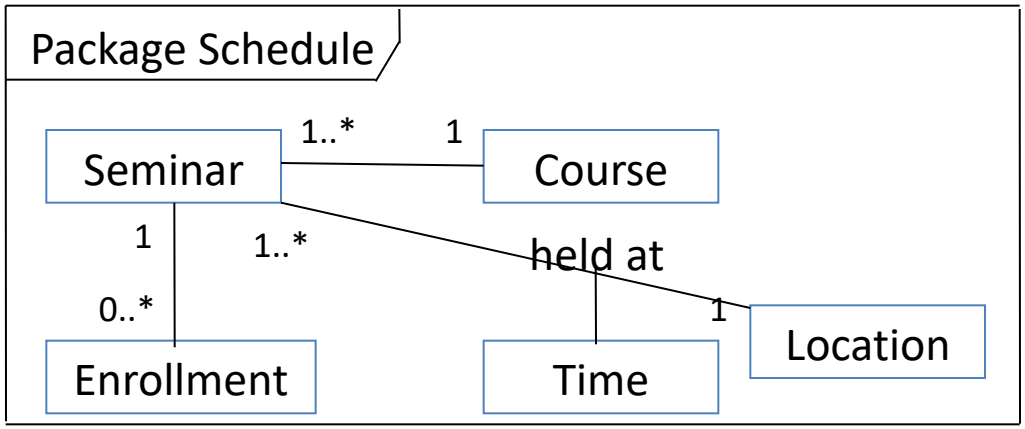
Class Package Diagrams

(http://www.agilemodeling.com/artifacts/packageDiagram.htm)

- Classes related through inheritance, composition or communication often belong in the same package



- A ***frame*** depicts the contents of a package (or components, classes, operations, etc.)
- Heading: rectangle with a cut-off bottom-right corner, [kind] name [parameter]



A frame encapsulates
a collection of collaborating instances or
refers to another representation of such

Common Mechanisms

- Adornments

  Notes & Compartments

- Extensibility Mechanisms
  - –Stereotypes - Extension of the UML metaclasses.
  - –Tagged Values - Extension of the properties of a UML element.
  - –Constraints - Extension of the semantics of a UML element.

Adornments

- Textual or graphical items added to an element's basic notation.
- Notes - Graphical symbol for rendering constraints or comments attached to an element or collection of elements; No Semantic Impact

Rendered as a rectangle with a dog-eared corner.

See smartCard.doc for details about this routine.

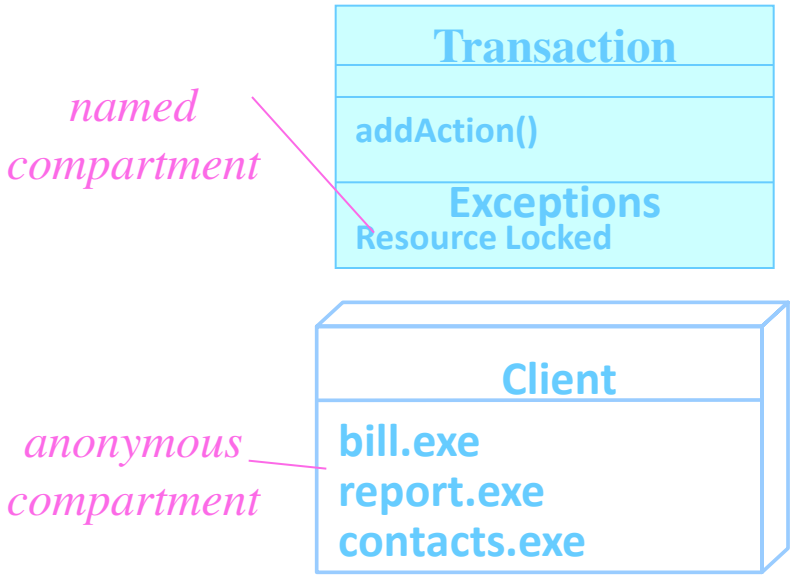
May contain combination of text and graphics.

See http://www.rational.com for related info.

May contain URLs linking to external documents.

Additional Adornments

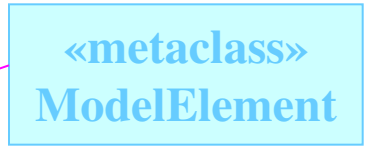
- Placed near the element as
  - Text
  - Graphic
- Special compartments for adornments in
  - Classes
  - Components
  - Nodes



Stereotypes

- Mechanisms for extending the UML vocabulary.
- Allows for new modeling building blocks or parts.

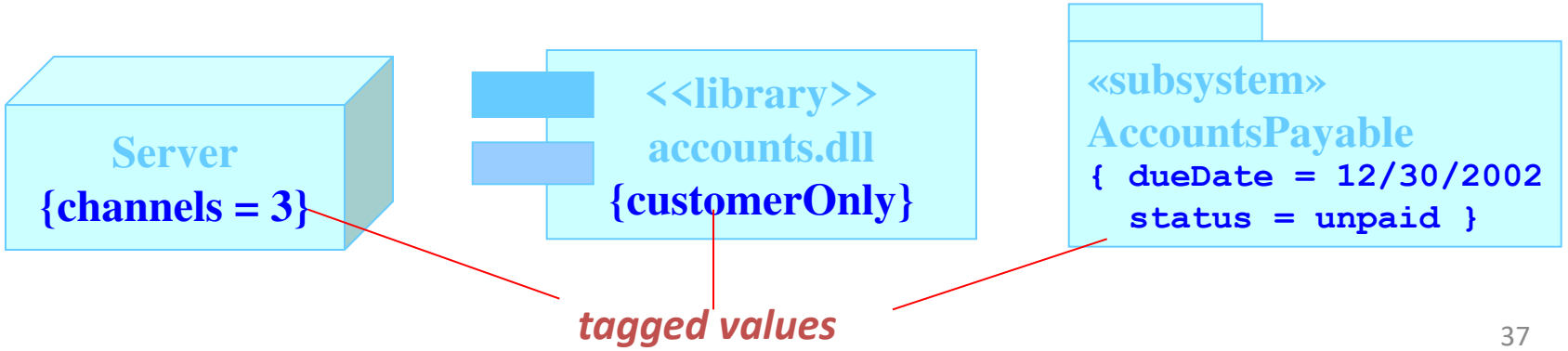
- Allow controlled extension of metamodel classes.
  [UML11_Metamodel_Diagrams.pdf]
- Graphically rendered as
  - Name enclosed in guillemets (<< >>)
    - <<stereotype>>
  - New icon



- The new building block can have
  - its own special properties through a set of tagged values
  - its own semantics through constraints

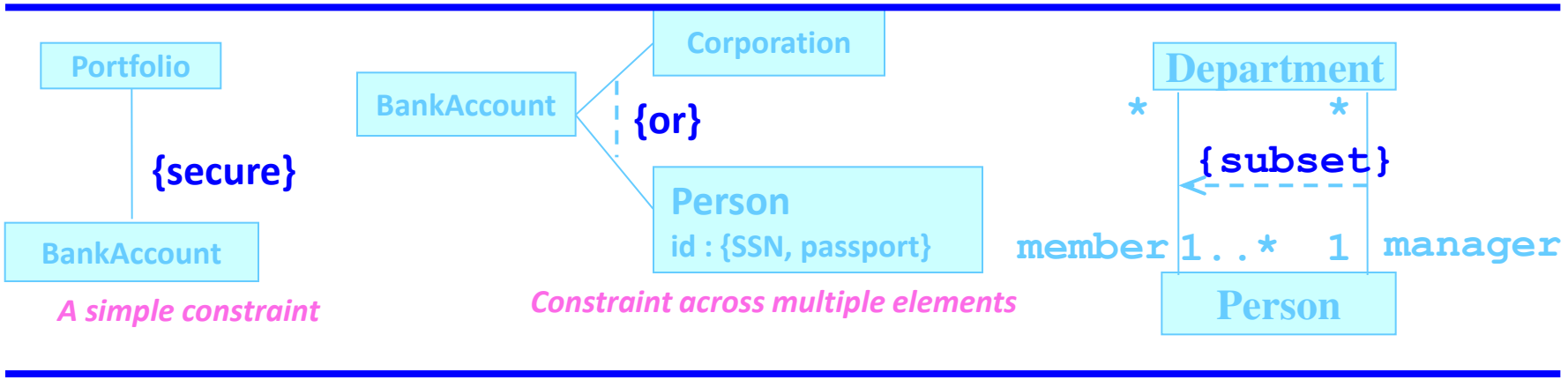
Tagged Values

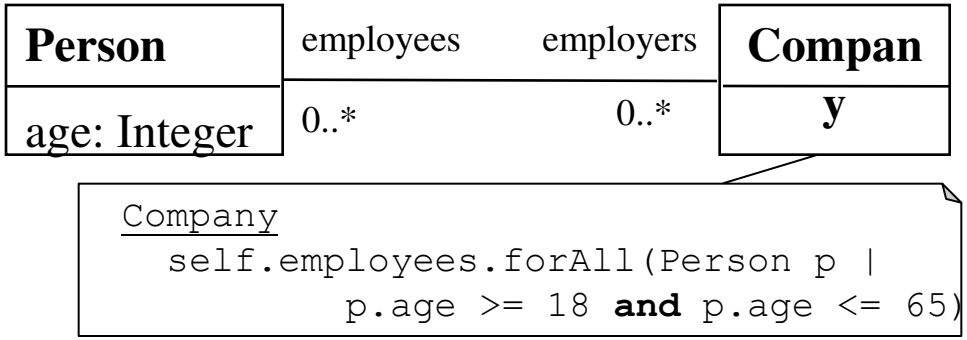
- a (name, value) pair describes a property of a model element.
- Properties allow the extension of "***metamodel***" element attributes.
- modifies the semantics of the element to which it relates.
- Rendered as a text string enclosed in braces { }
- Placed below the name of another element.



Constraints

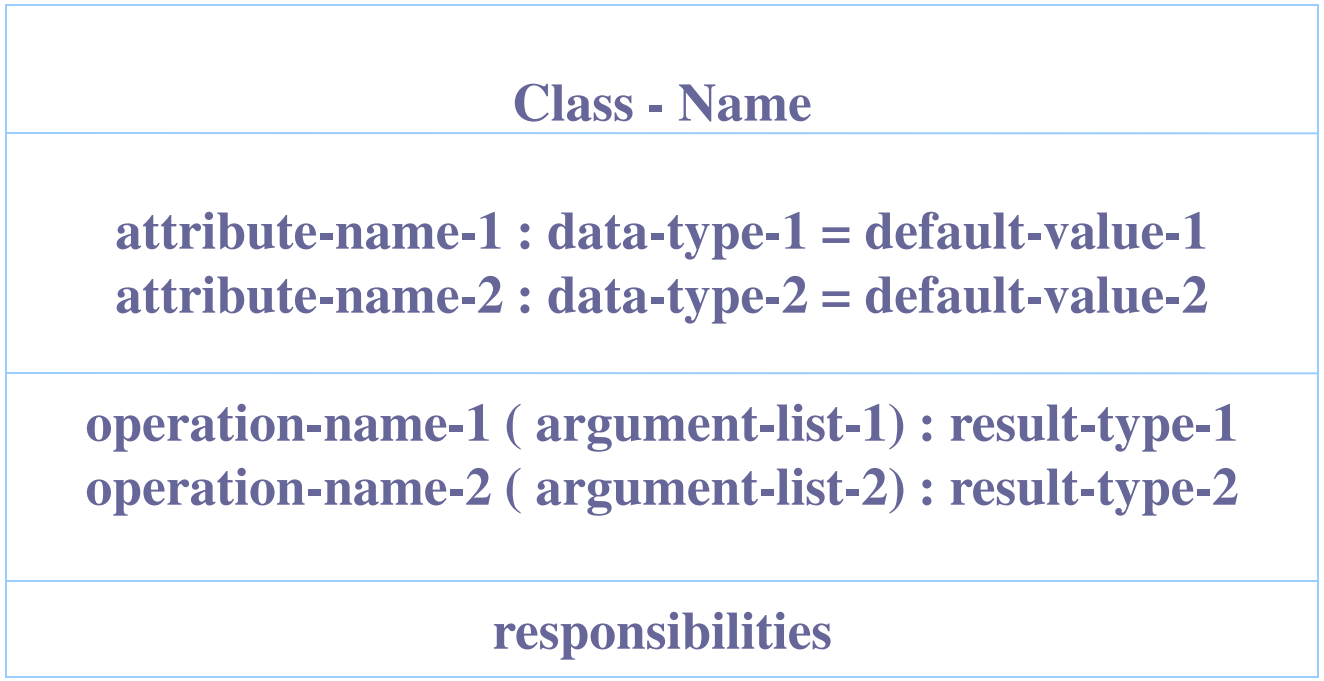
- Extension of the semantics of a UML element.
- Allows new or modified rules
- Rendered in braces {}.
  - Informally as free-form text, or
  - Formally in UML's Object Constraint Language (OCL):
    E.g., {self.wife.gender = female and self.husband.gender = male}





Appendix
Some Additional Material

Classes: Notation and Semantics



To model the <<semantics>> (meaning) of a class:

- Specify the body of each method (pre-/post-conditions and invariants)
- Specify the state machine for the class
- Specify the collaboration for the class
- Specify the responsibilities (contract)

Attributes

- Syntax
  [visibility] name [multiplicity] [: type] [= initial-value] [{property-string }]
- Visibility

  \+ public; - private; # protected; {default = +}
- type
  - There are several defined in Rational Rose.
  - You can define your own.

  Or you can define your own: e.g. {leaf}
- property-string

  Built-in property-strings:
  - *changeable*—no restrictions (default)
  - *addOnly*—values may not be removed or altered, but may be added
  - *frozen*—may not be changed after initialization

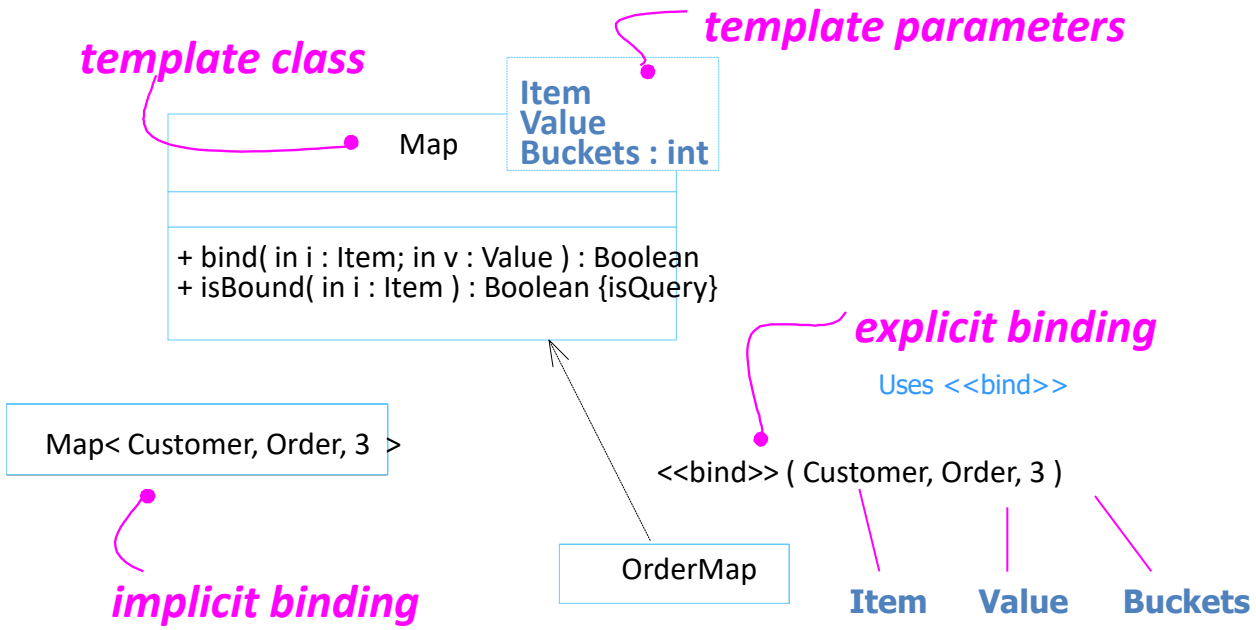
origin	Name only
+ origin	Visibility and name
origin : Point	Name and type
head : *Item	Name and complex type
name [0..1] : String	Name, multiplicity, and type
origin : Point = { 0, 0 }	Name, type, and initial value
id : Integer { frozen }	Name and property

Operations

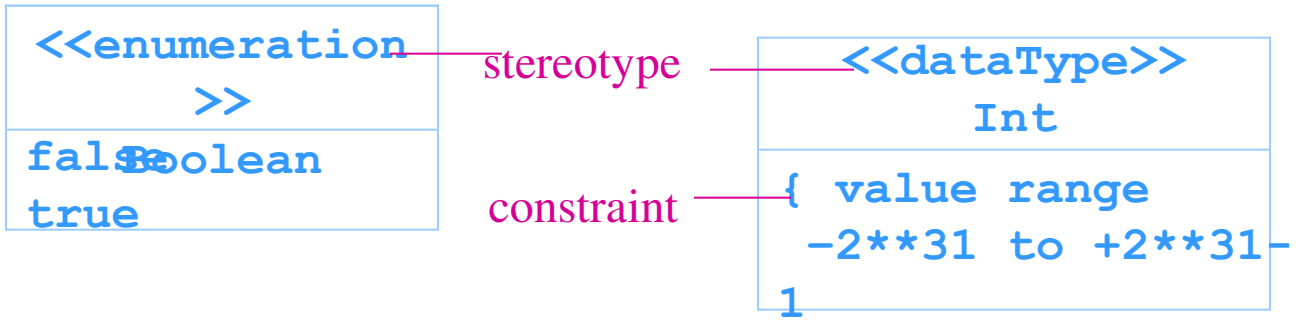
- Syntax
  [visibility] name [(parameter-list)] [: return-type] [(property-string)]
- Visibility
  + public; - private; # protected; {default = +}
- parameter-list syntax
  [direction] name : type [= default-value]
- direction
  - *in*—input parameter; may not be modified
  - *out*—output parameter; may be modified
  - *inout*—input parameter; may be modified
- property-string
  - *leaf*
  - *isQuery*—state is not affected
  - *sequential*—not thread safe
  - *guarded*—thread safe (Java synchronized)
  - *concurrent*—typically atomic; safe for multiple flows of control

Template Classes; Primitive Types

- A template class is a parameterized element and defines a family of classes
- In order to use a template class, it has to be instantiated
- Instantiation involves binding formal template parameters to actual ones, resulting in a concrete class



Primitive Types using a class notation



Interface: A Java Example

```
public interface SoundFromSpaceListener extends EventListener {
    void handleSoundFromSpace(SoundFromSpaceEventObject sfseo);
}

public class SpaceObservatory implements SoundFromSpaceListener
    public void handleSoundFromSpace(SoundFromSpaceEventObject sfseo) {
            soundDetected = true;
            callForPressConference();
    }
```

Can you draw a UML diagram corresponding to this?

Package Diagrams: Standard Elements

- *Façade* — only a view on some other package.
- *Framework* — package consisting mainly of patterns.
- *Stub* — a package that serves as a proxy for the public contents of another package.
- *Subsystem* — a package representing an independent part of the system being modeled.
- *System* — a package representing the entire system being modeled.

Is <<import>> transitive?
Is visibility transitive?
Does <<friend>> apply to all types of visibility: +, -, #?

Dependency –Among Objects

- 3 Stereotypes of Dependency in Interactions among Objects:
  - *become*: the target is the same object as the source but at a later point in time and with possibly different values, state, or roles
  - *call*: the source operation invokes the target operation
  - *copy*: the target object is an exact, but independent, copy of the source