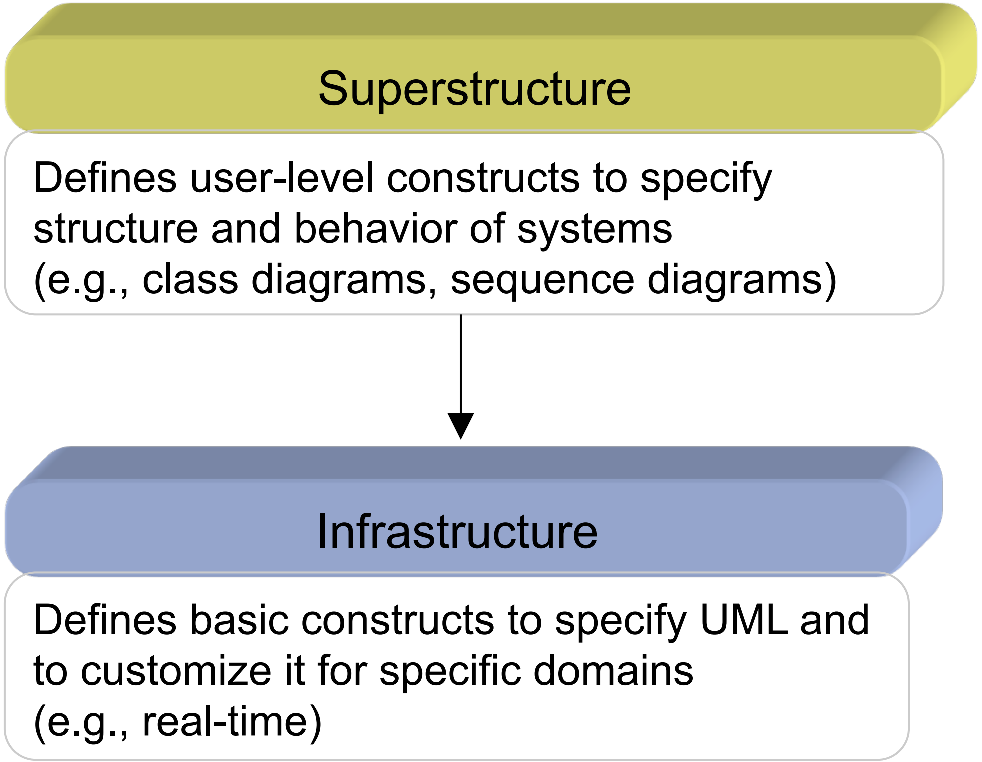
UML advanced

Paolo Ciancarini

Agenda

- Advanced diagrams in UML
- Extension rules of UML
- Stereotypes and profiles
- Modeling sw architectures with UML

UML 2.0



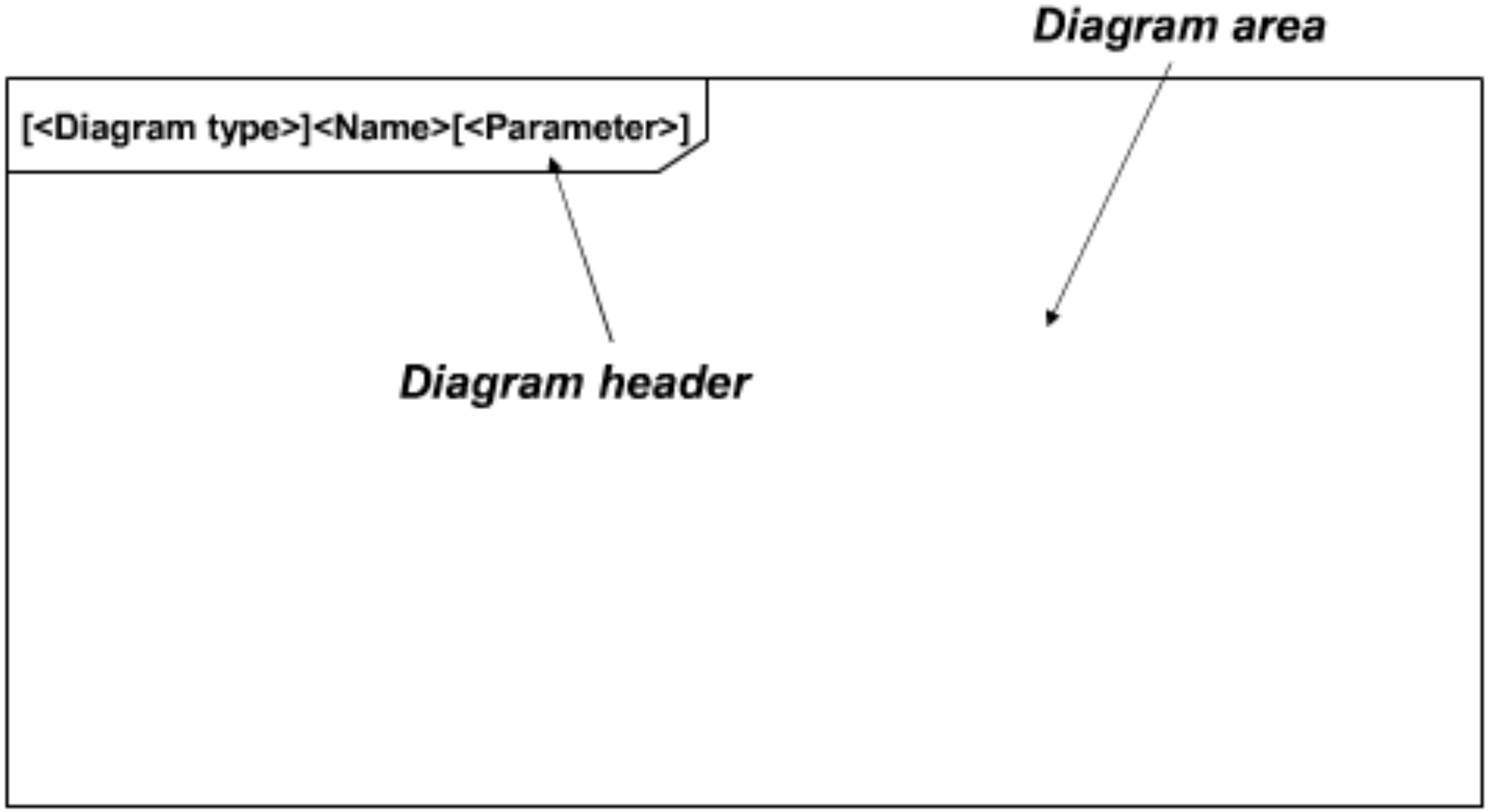
Superstructure

Defines user-level constructs to specify structure and behavior of systems
(e.g., class diagrams, sequence diagrams)

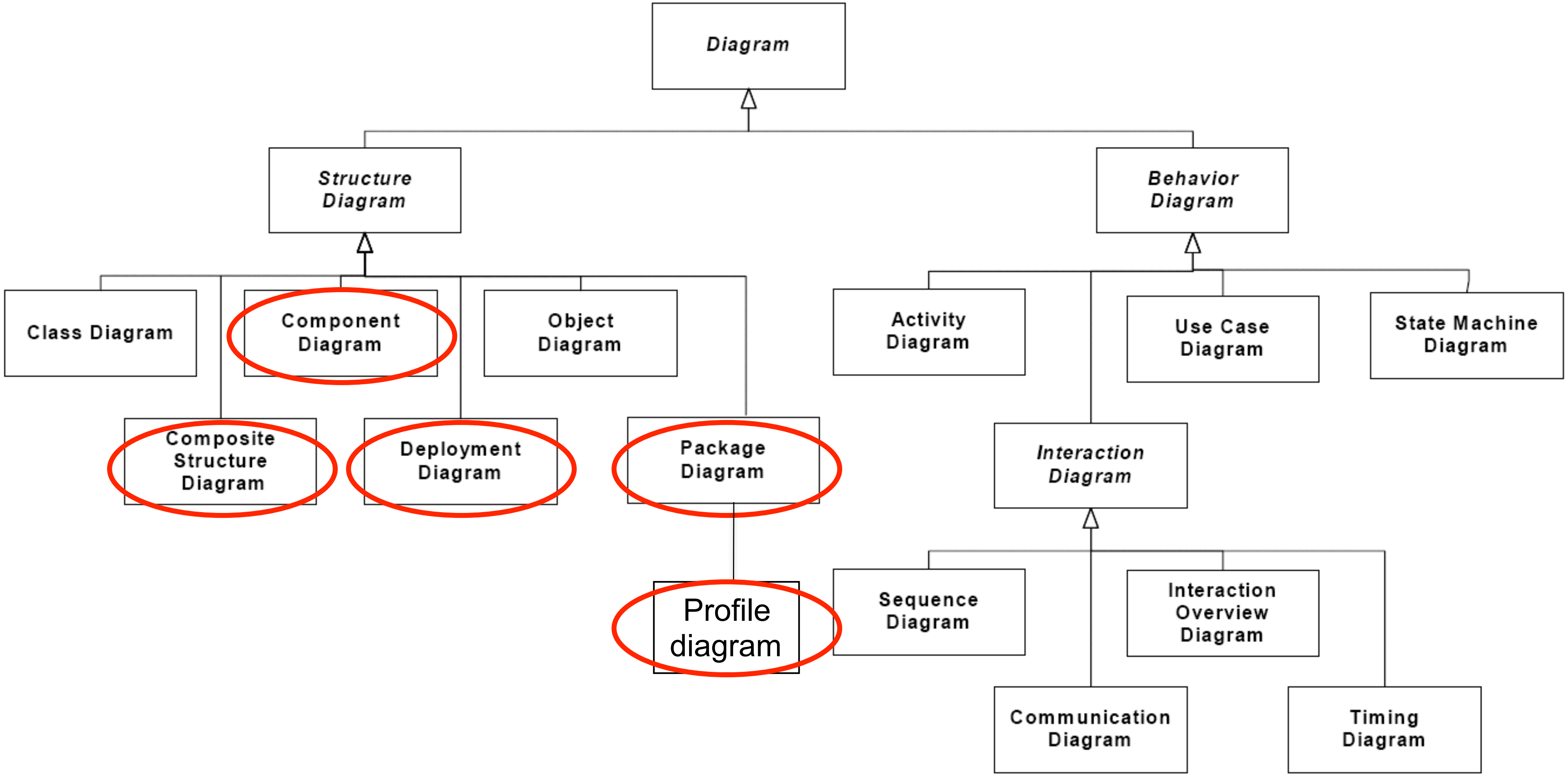
Infrastructure

Defines basic constructs to specify UML and to customize it for specific domains
(e.g., real-time)

UML2: new syntax for diagrams

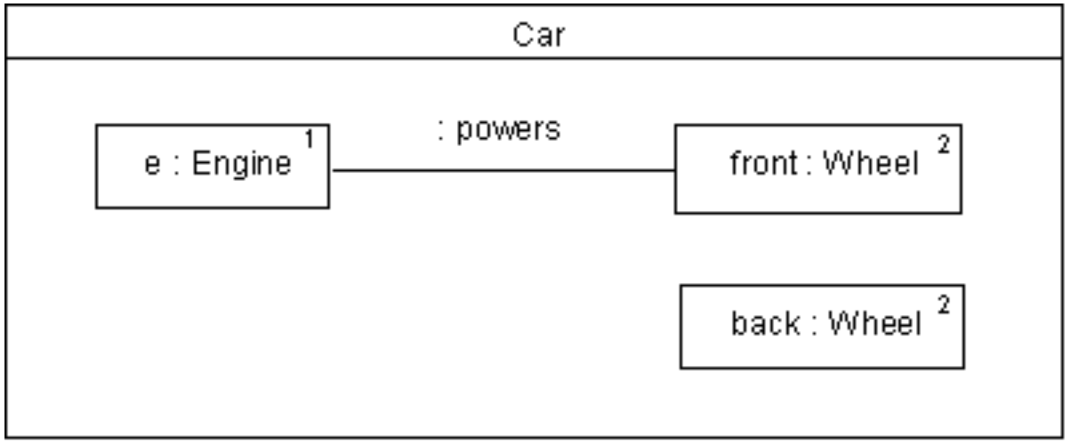


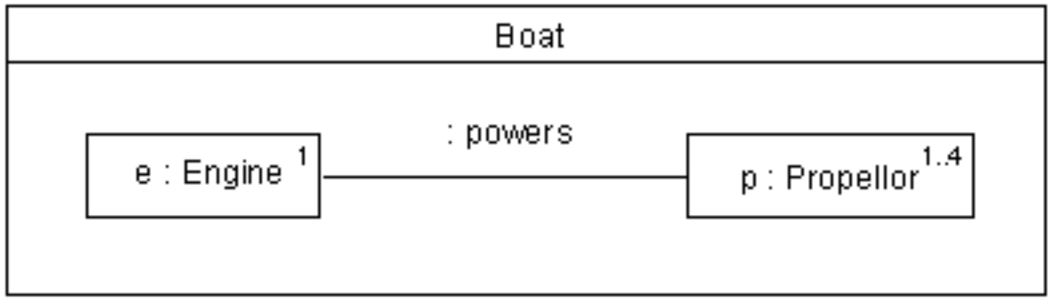
UML2: advanced structure diagrams



Composite structure

- Shows the internal structure of a structured classifier or collaboration





Example

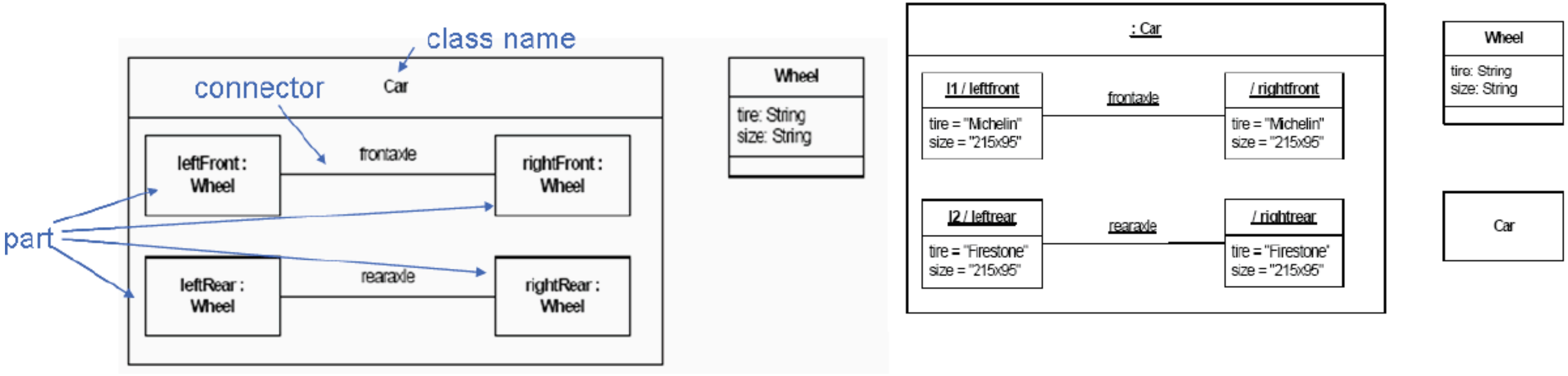
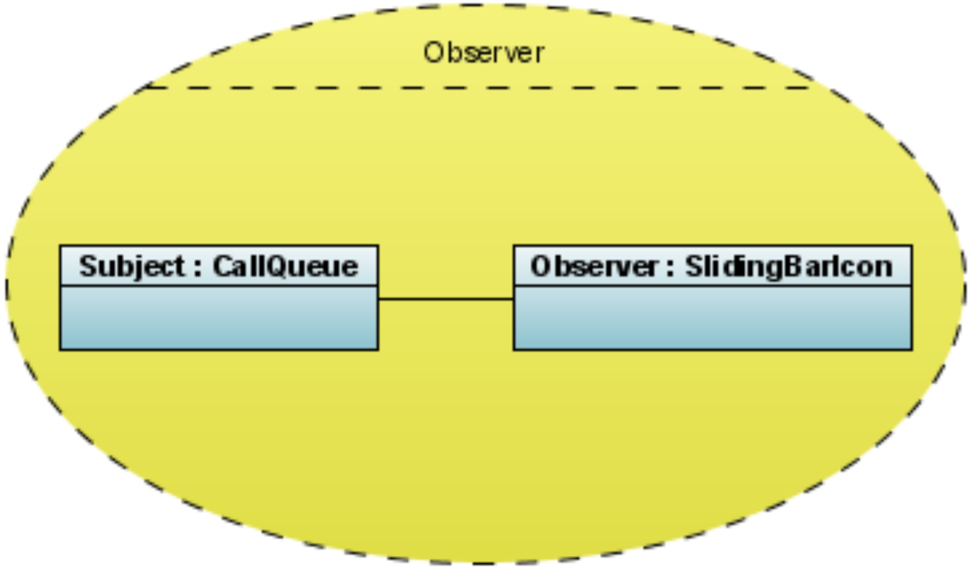


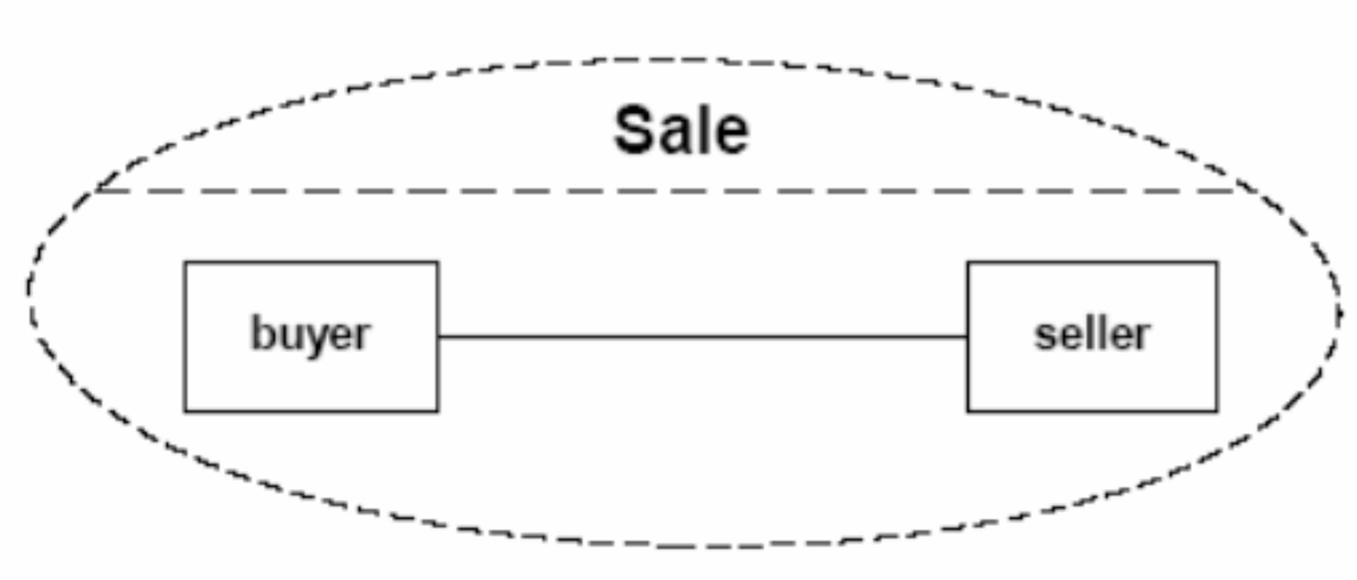
Figure 3 Composite Structure Diagram and its Instantiation

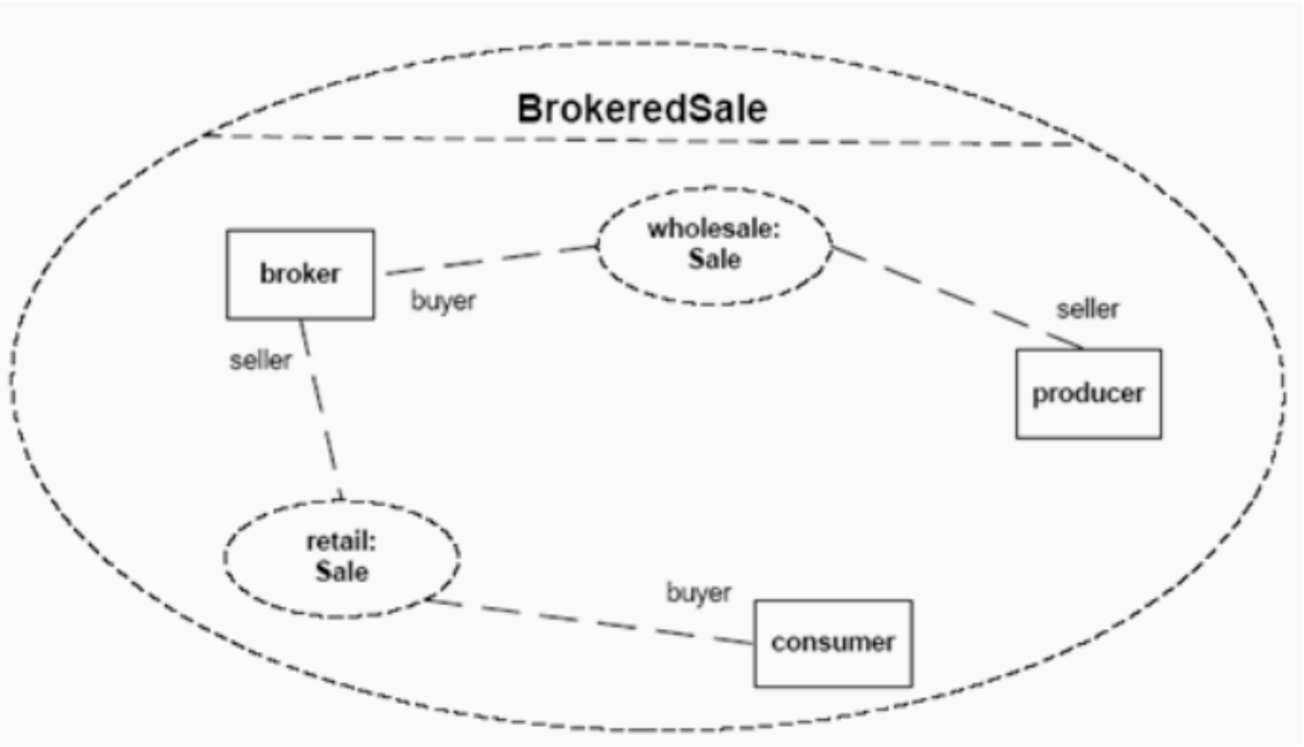
Composite structure

- Shows the internal structure of a structured classifier or collaboration
- Often used for highlighting design patterns



Collaborations





Modularity

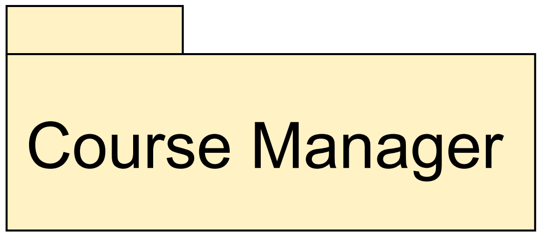
- A complex software system should be designed starting from simpler subsystems, or modules, or components
- **Module**: unit of compilation; eg. a Java package
- **Component**: (functional, stateless) unit of configuration; eg. a new peer connected to a p2p system

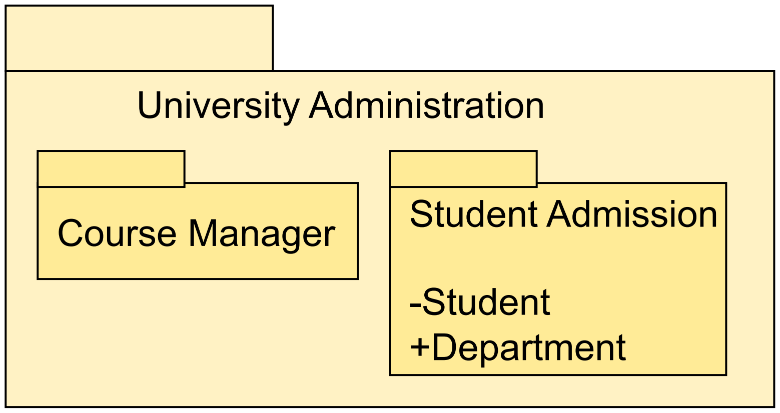
Discuss

- The programming languages you know which modularity mechanisms do they offer?

Grouping Things: ***Packages***

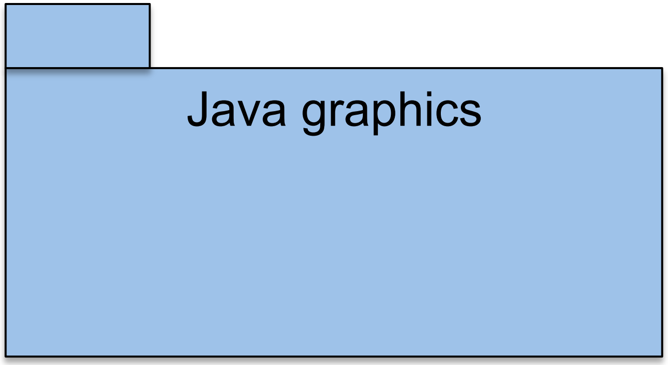
- Useful to organize elements into groups
- Purely conceptual; only exists at development time
- Can be nested
- Variations of packages are: frameworks, models, & subsystems



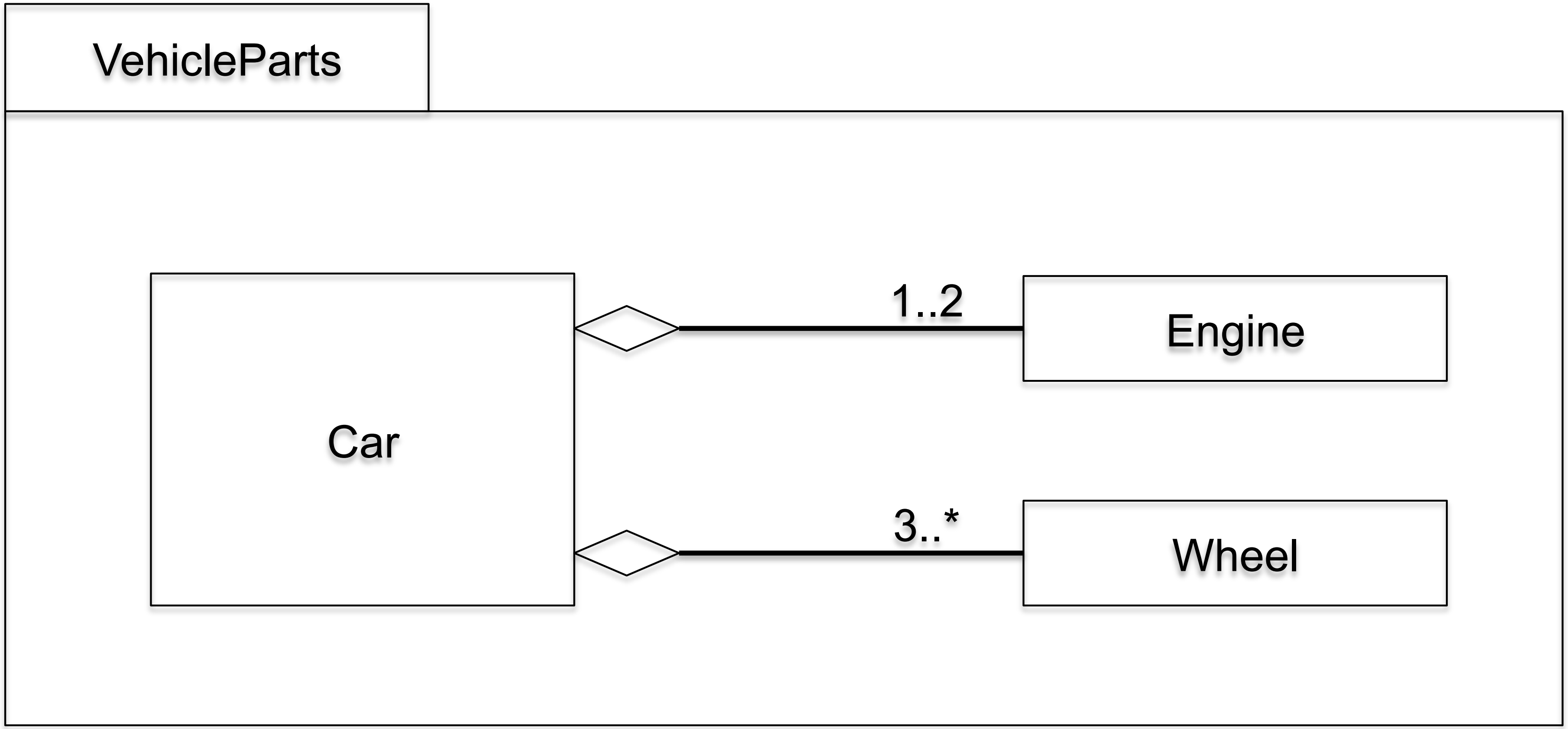


Package

- Module; collects logically coherent elements
- Defines a *namespace* where each element has a unique name

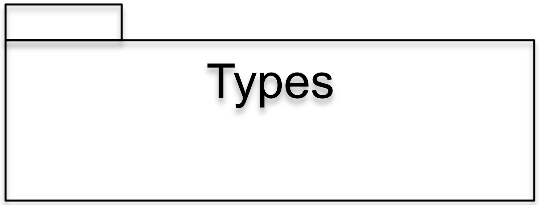


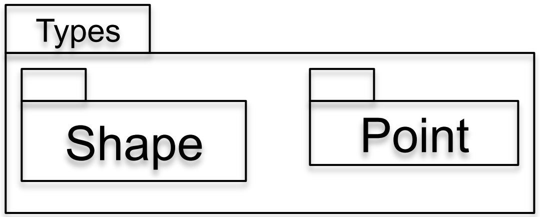
Package containing classes

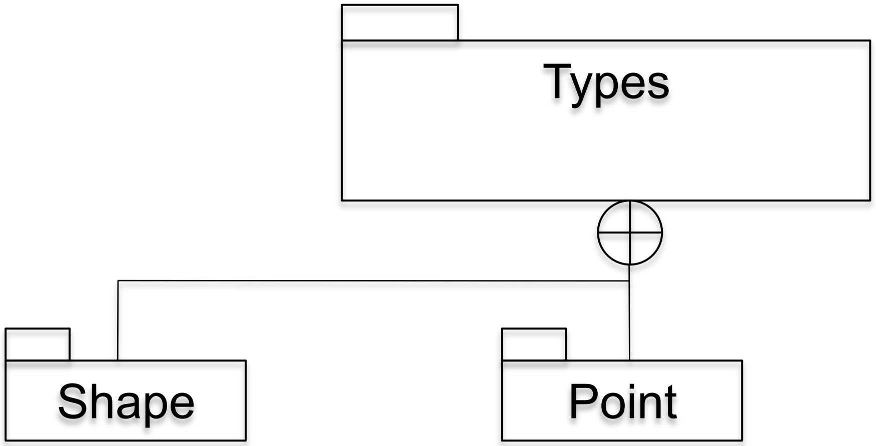


The classes Engine and Wheel are included in the package VehicleParts

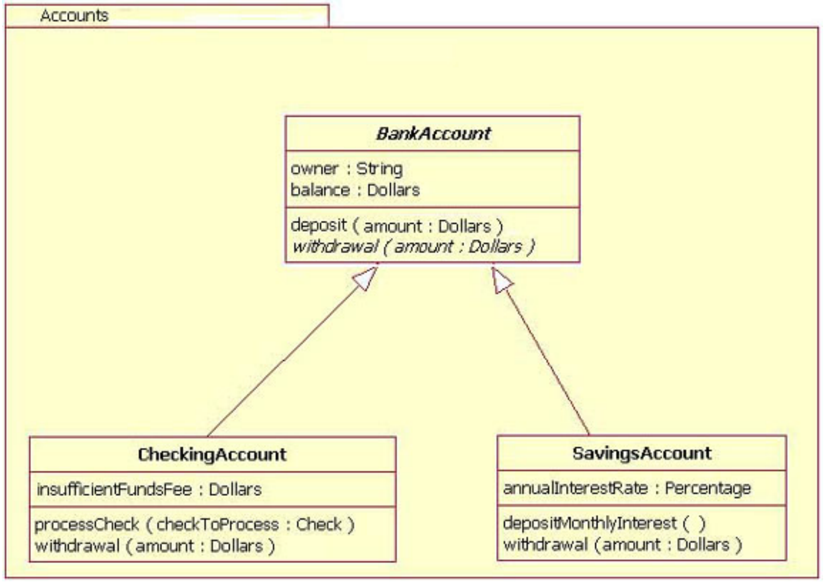
Package notations

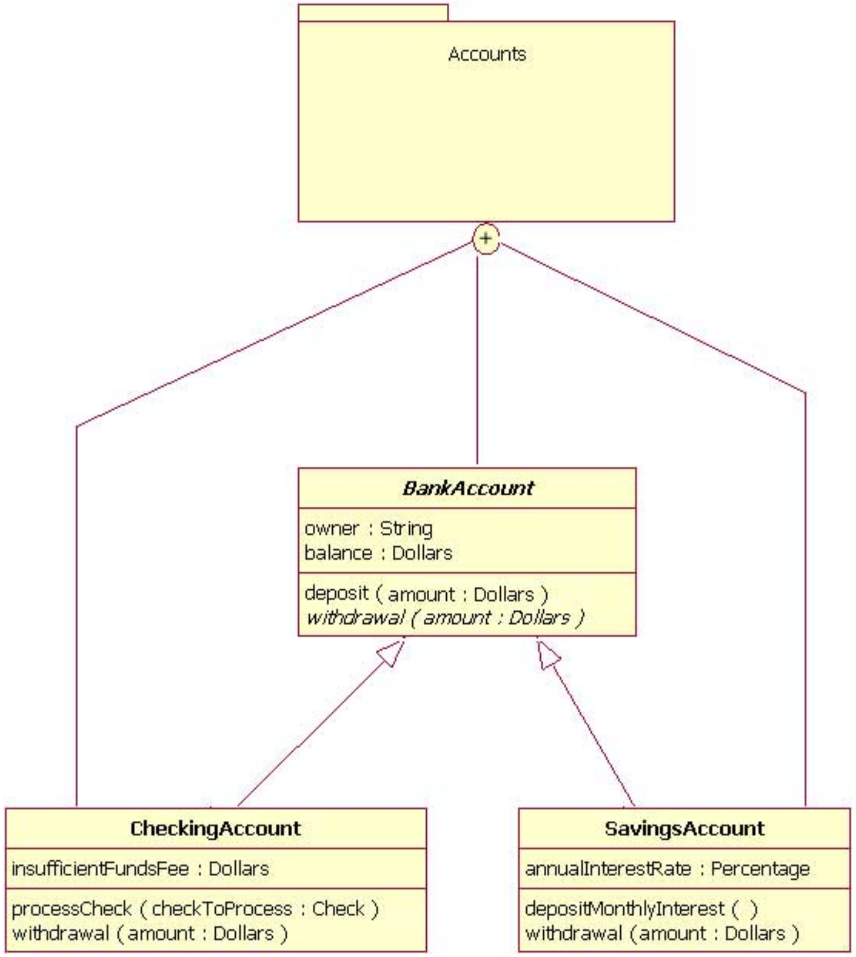






Equivalent representations

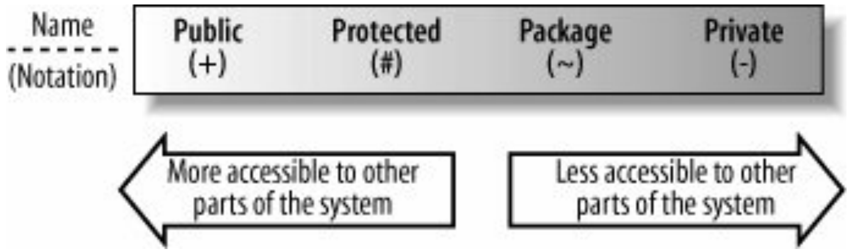

Accounts
BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)


Accounts
+
BankAccount
owner : String
balance : Dollars
deposit (amount : Dollars)
withdrawal (amount : Dollars)
CheckingAccount
insufficientFundsFee : Dollars
processCheck (checkToProcess : Check)
withdrawal (amount : Dollars)
SavingsAccount
annualInterestRate : Percentage
depositMonthlyInterest ( )
withdrawal (amount : Dollars)

Visibility declarations

- Field declarations
  - `Birthday:Date`
  - `+duration:int = 100`
  - `-students[1..MAX_SIZE]: Student`
- Method declarations
  - `+move(dx:int, dy:int): void`
  - `+getSize():int`

Visibility	Notation
public	+
protected	#
package	~
private	-



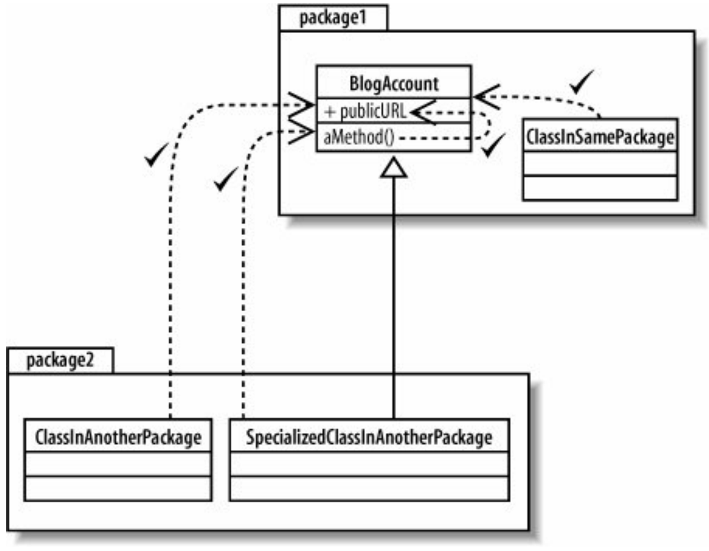
Name
(Notation)

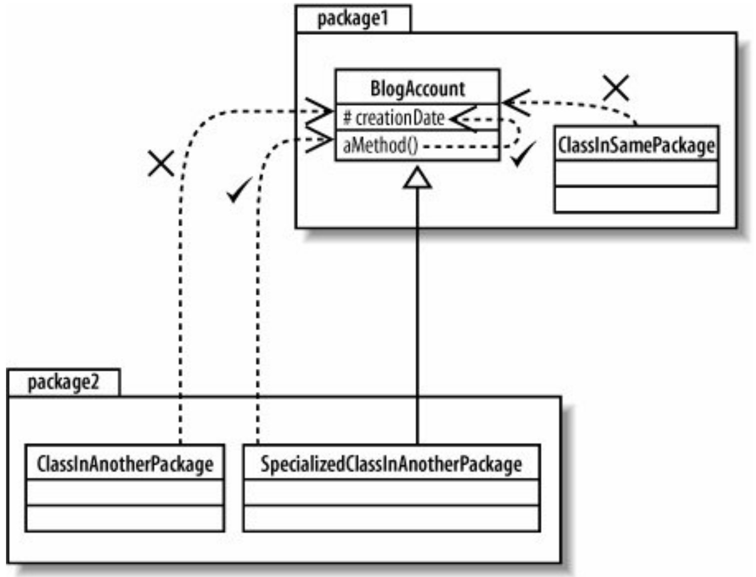
Public (+) Protected (#) Package (~) Private (-)

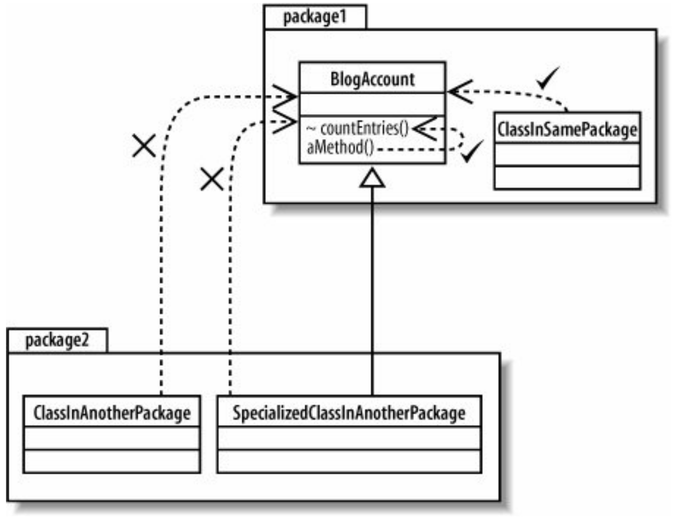
More accessible to other parts of the system

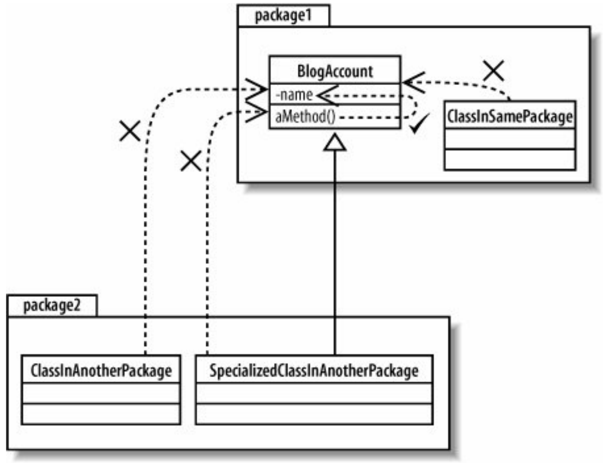
Less accessible to other parts of the system

Visibility

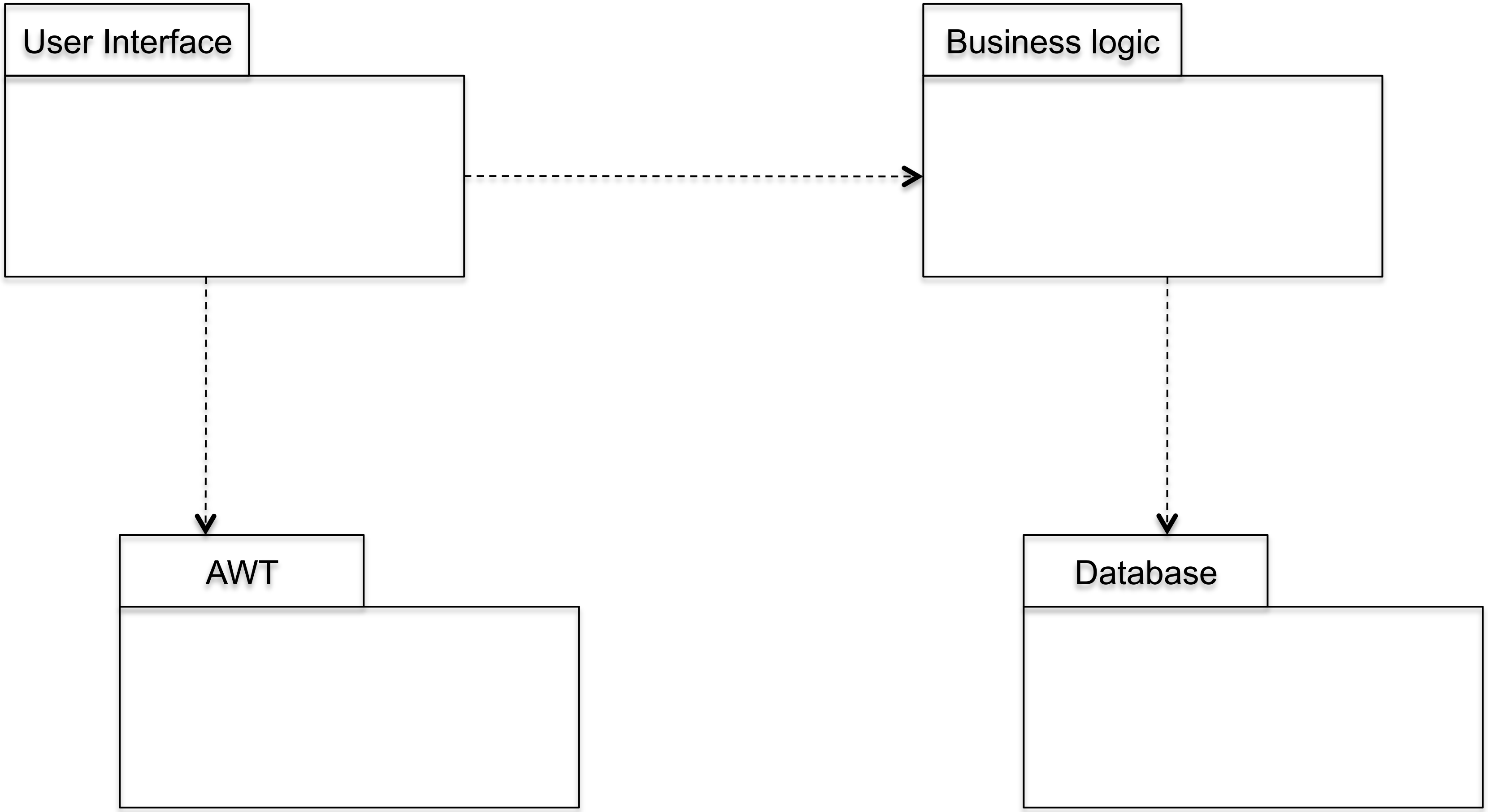

package1
BlogAccount
+ publicURL
aMethod()
✓
ClassInSamePackage
✓
✓
✓
package2
ClassInAnotherPackage
SpecializedClassInAnotherPackage


package1
BlogAccount
# creationDate
aMethod()
✕
ClassInSamePackage
✓
✕
✓
package2
ClassInAnotherPackage
SpecializedClassInAnotherPackage


package1
BlogAccount
~ countEntries()
aMethod()
✓
ClassInSamePackage
✓
✕
✕
package2
ClassInAnotherPackage
SpecializedClassInAnotherPackage


package1
BlogAccount
-name
aMethod()
✕
ClassInSamePackage
✓
✕
✕
package2
ClassInAnotherPackage
SpecializedClassInAnotherPackage

A package diagram



Metamodel packages (from 2.1.1)

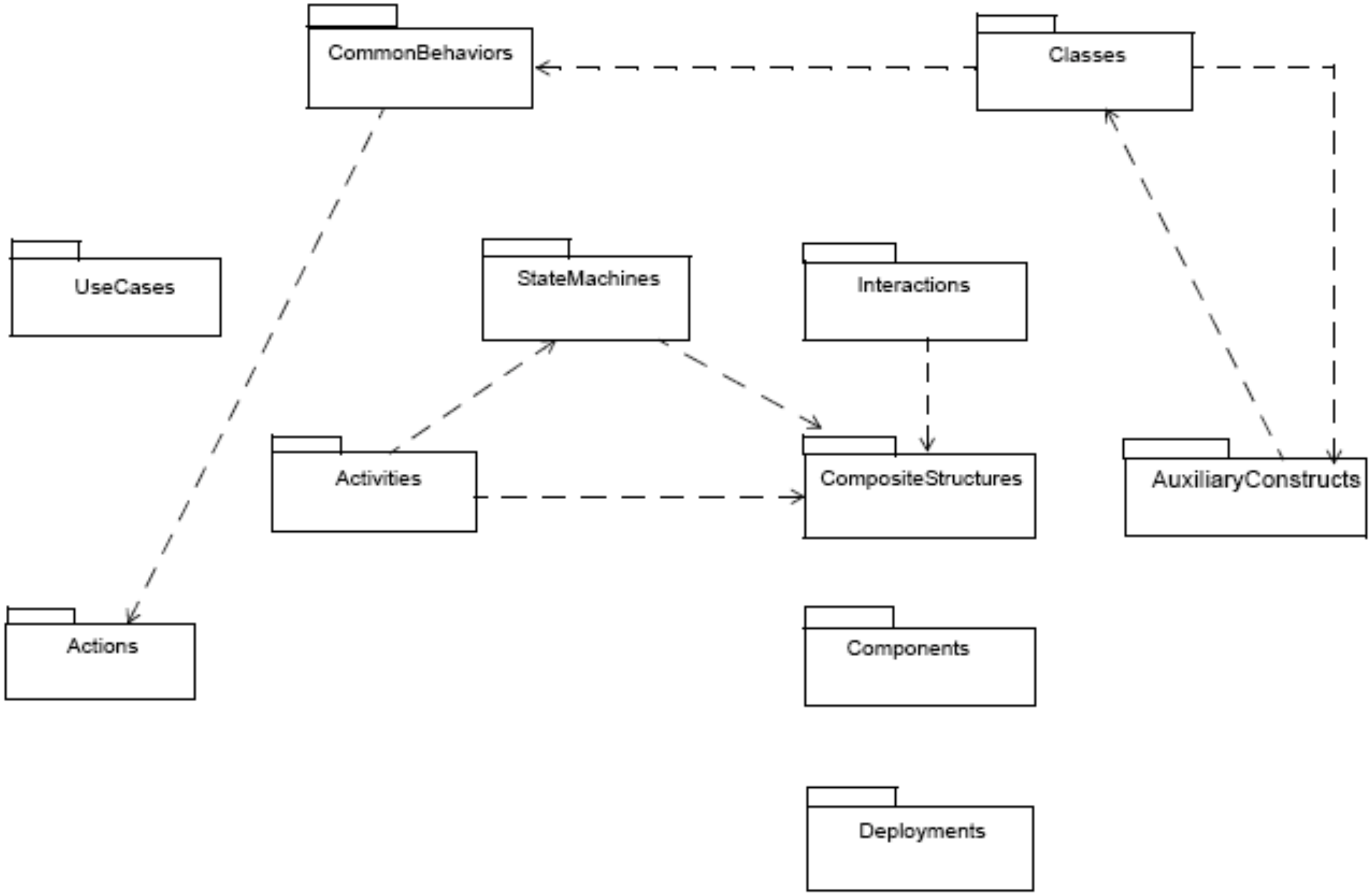


Figure 7.5 - The top-level package structure of the UML 2.1.1 Superstructure

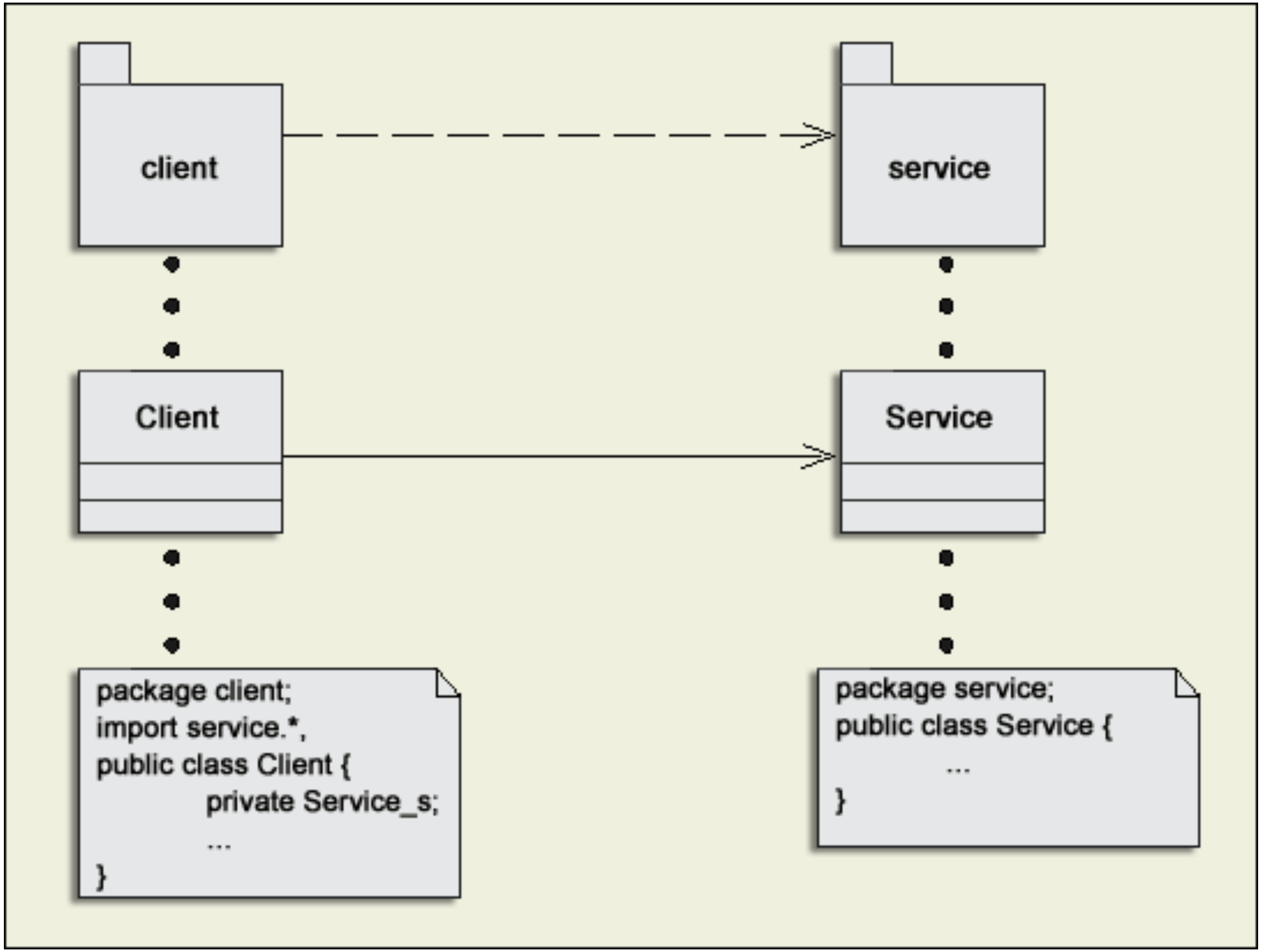
Dependencies

“A dependency exists between two elements if changes to the definition of one element may cause changes to the other.”

“A dependency between two packages exists if any dependency exists between any two classes in the packages.”

Martin Fowler

Dependencies among packages and classes

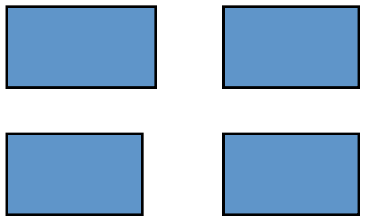


```
package client;
import service.*,
public class Client {
          private Service_s;
          ...
}
```

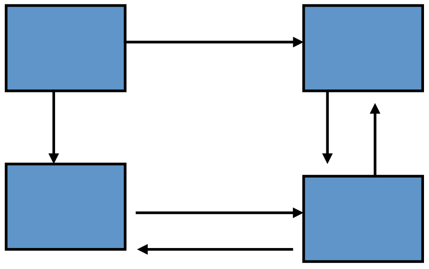
```
package service;
public class Service {
          ...
}
```

Minimizing coupling

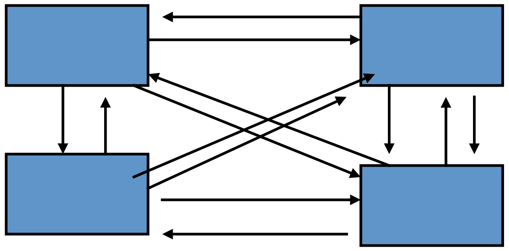
- When a class changes, all classes which depend upon it generally need to be changed or recompiled
- More independent, more *robust* to changes



Uncoupled



Loosely Coupled



Highly Coupled

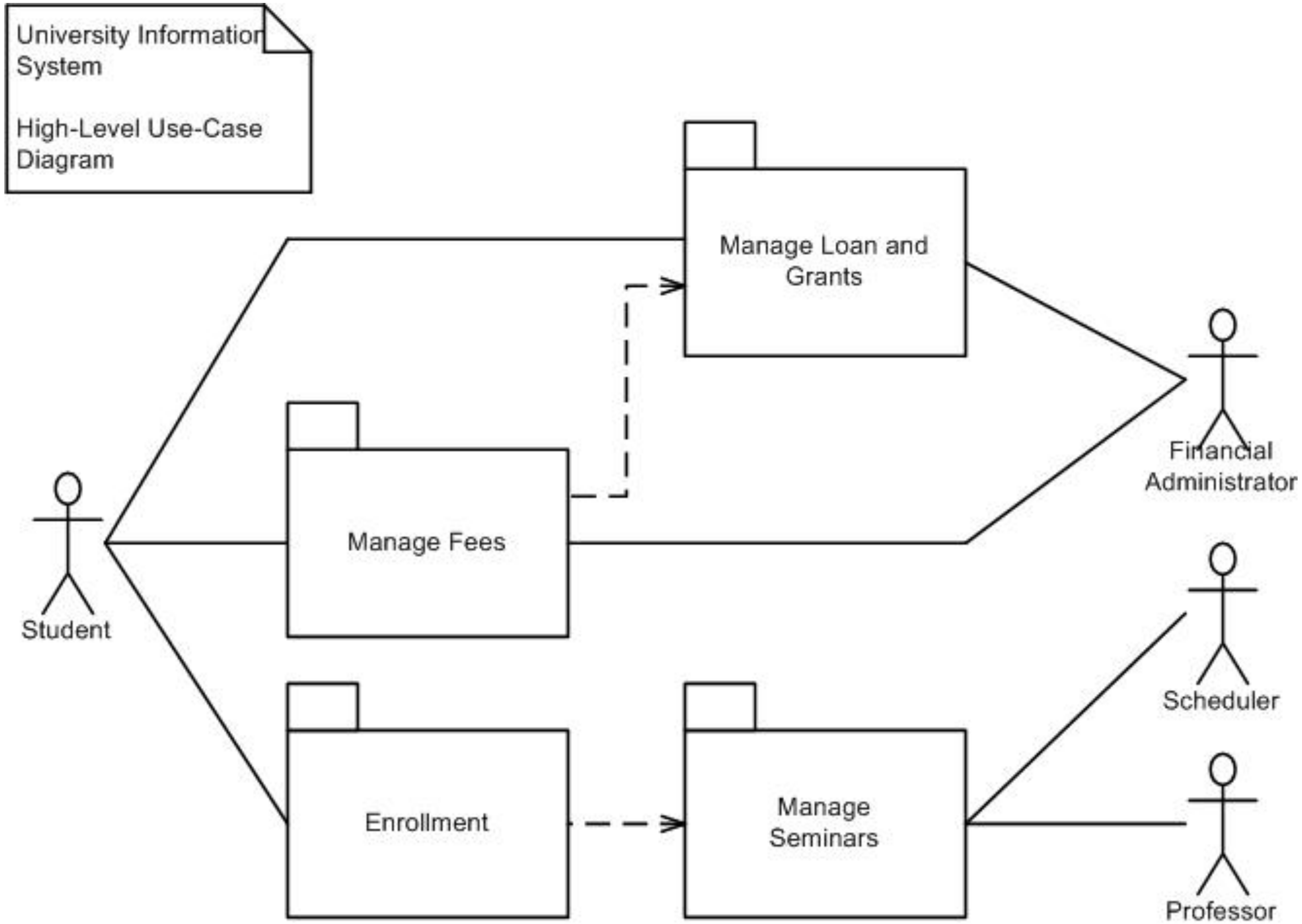
Encouraging cohesion

- Group types (interface and classes) which fulfill a similar purpose, service or function
- Internal package coupling, or *relational cohesion*, can be quantified
  - RC = NumberOfInternalRelations/ NumberOfTypes
  - The bigger of RC, the higher cohesion
  - Most useful for packages containing some implementation classes

Package diagram

- Packages can be nested inside other packages
- Packages can be used to add details to Use cases
- Package diagrams are often used to show the contents of components, which are often packages in the Java sense

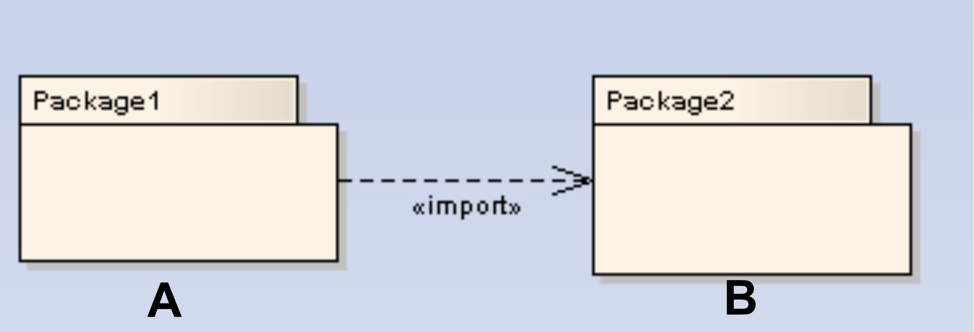
Packages inside Use Cases



Relations between packages

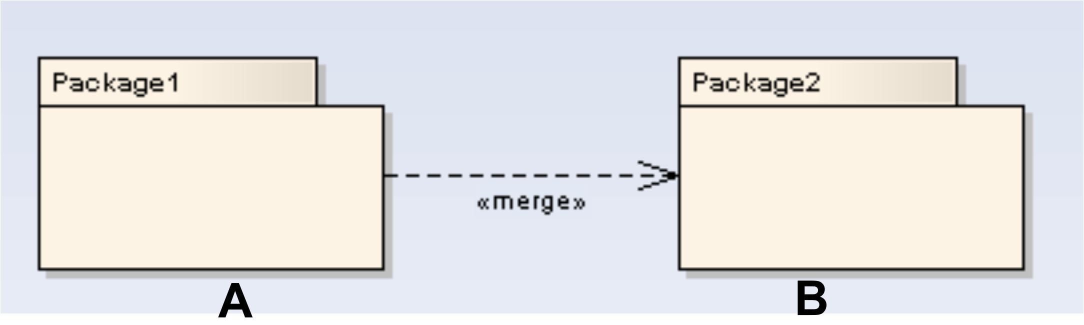
- Package dependency
- Package import
- Package merge
- Profile application

Package A imports package B



the importing namespace A adds the names of the members of the other package B to its own namespace

Package A merges with package B



It is similar to Generalization: the source package A adds the characteristics of the target B to its own characteristics resulting in a new A' package that combines the characteristics of both A and B

This mechanism should be used when elements defined in different packages have the same name and represent the same concept

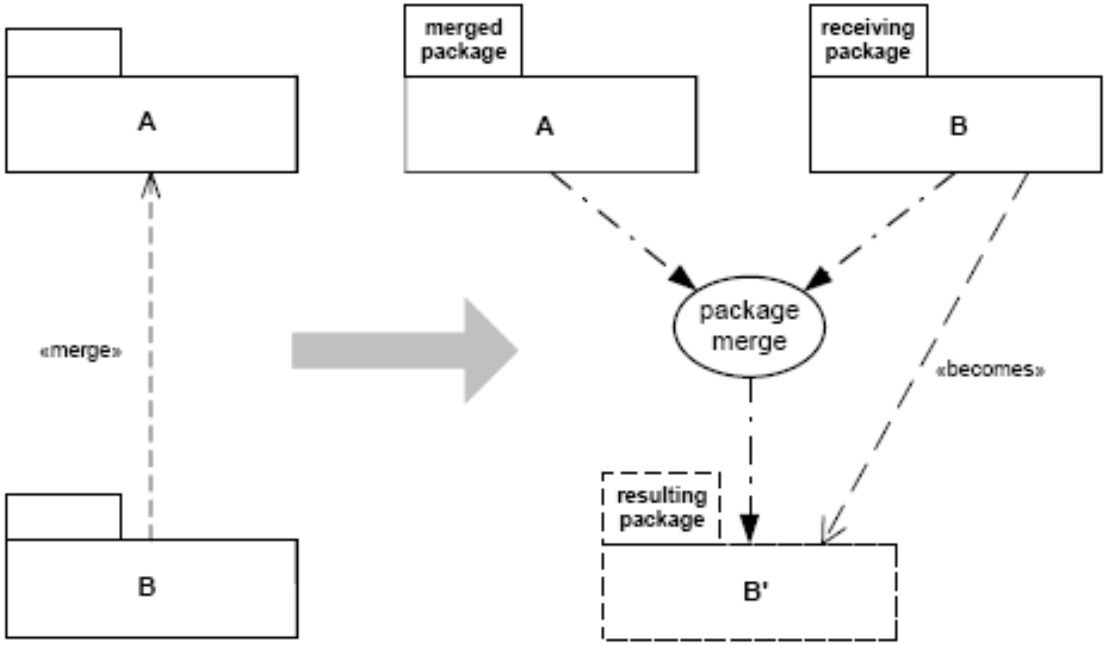
It is used to provide different definitions of a given concept for different purposes, starting from a common base definition

Package *merge*

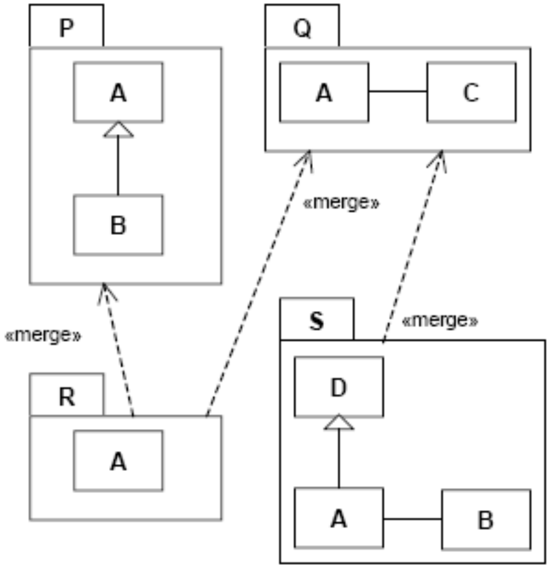
- **Merge**: directed relationship between two packages indicating that their contents must be combined
- Package merge allows modeling concepts defined in one package to be extended with new features
- UML itself is a merge of a large number of packages
- By selecting which increments to merge, it is possible to obtain a custom definition of a concept

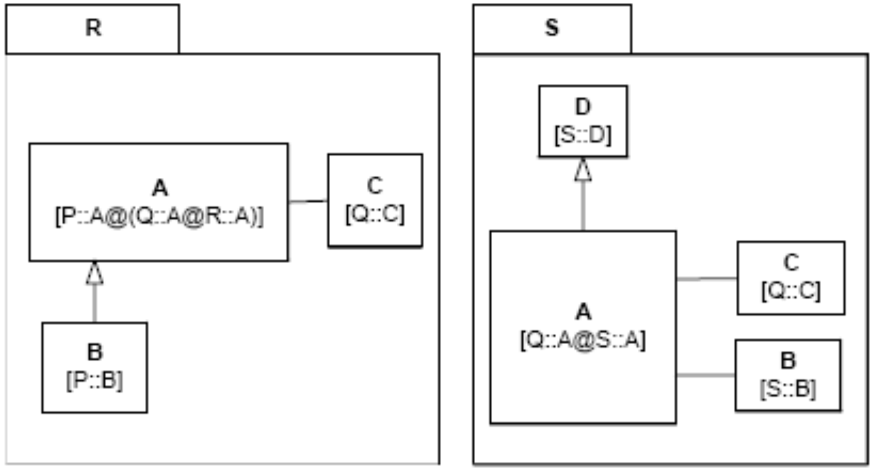
Package Merge

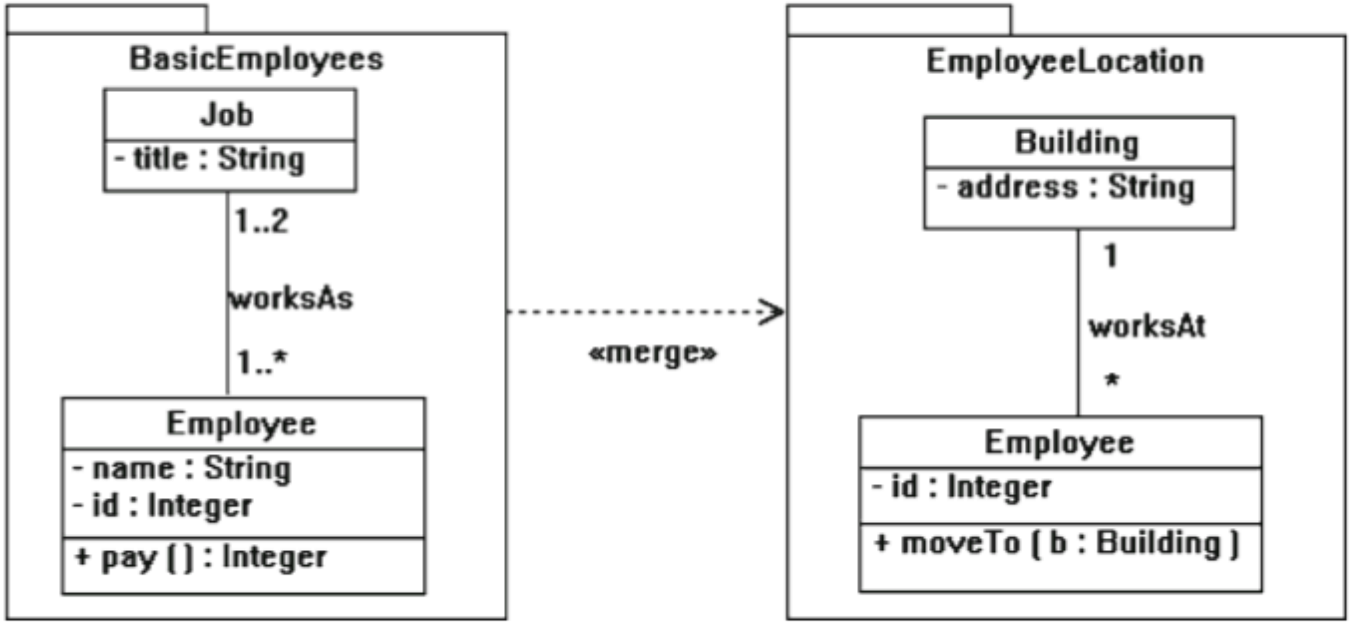
- Package *merge* allows modeling concepts defined at one level to be extended with new features

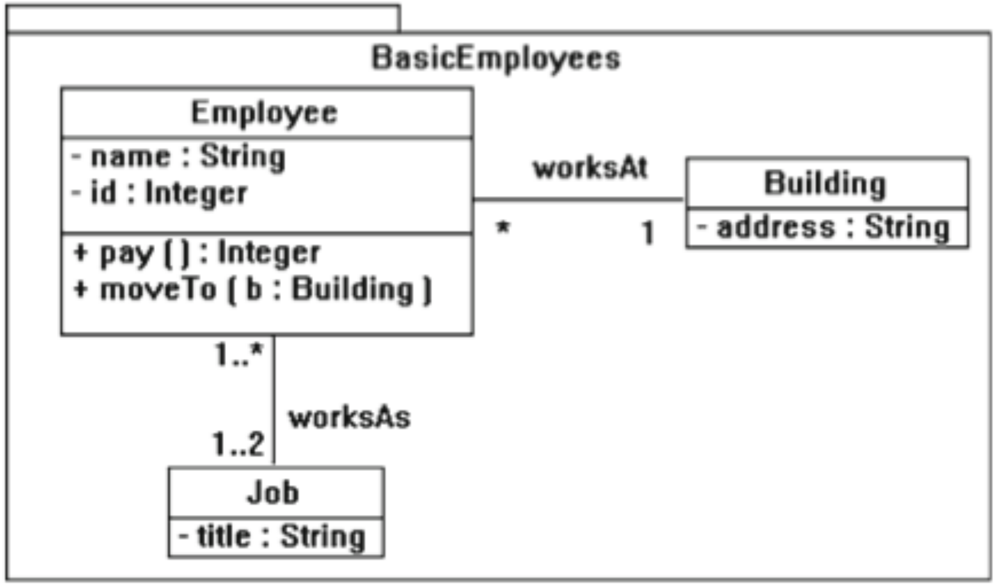


Package Merge - Example

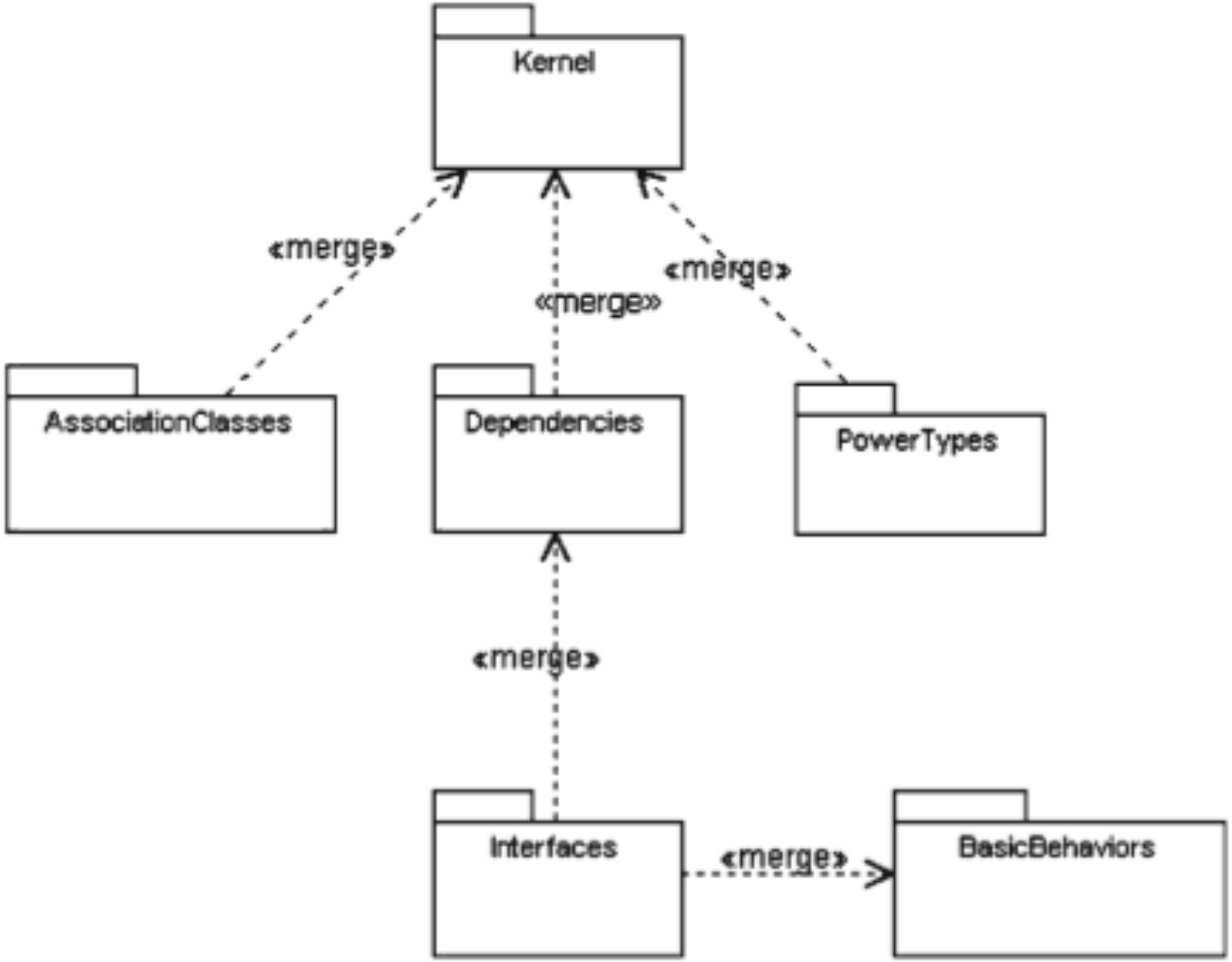




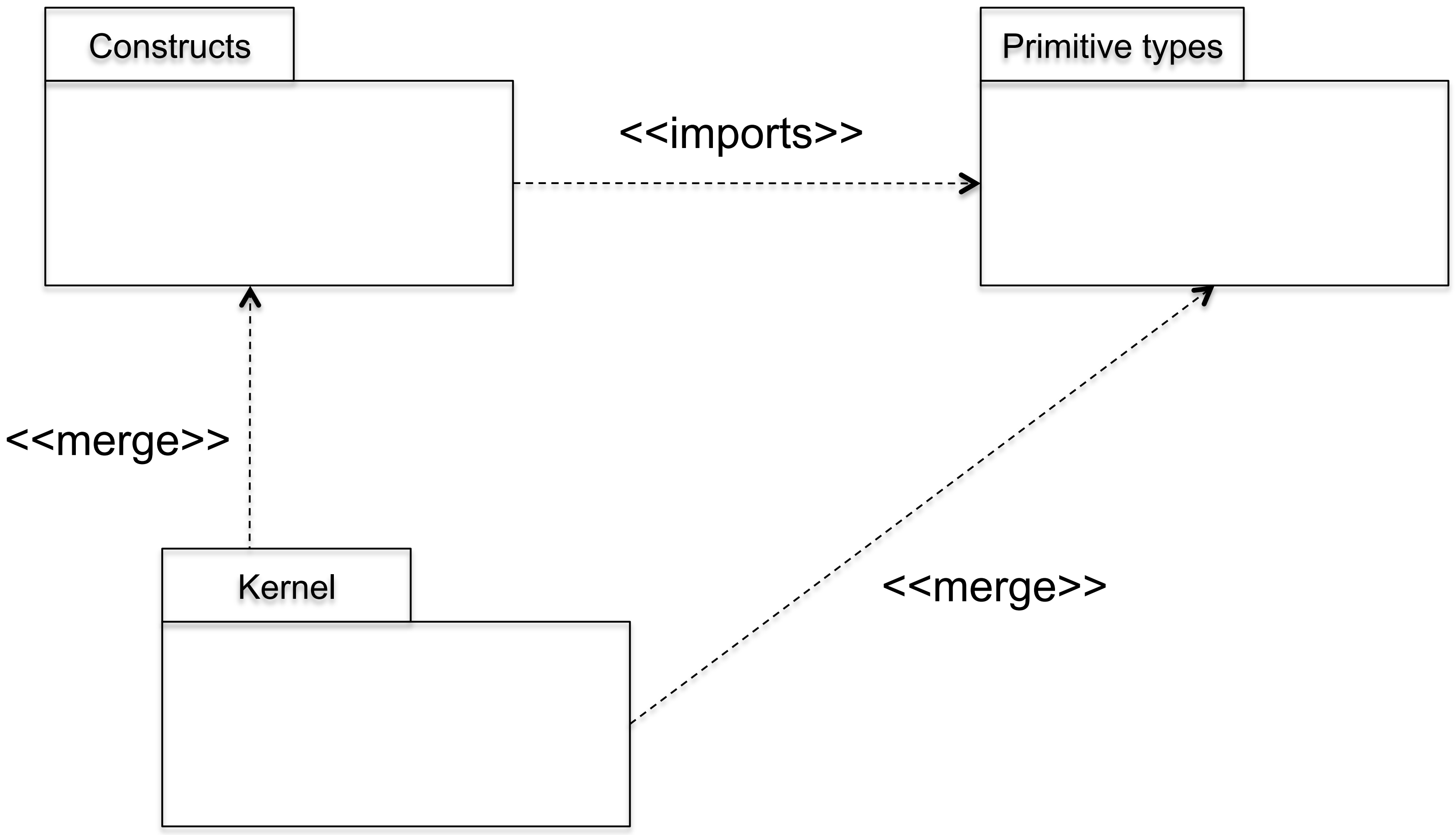

BasicEmployees
Job
- title : String
1..2
worksAs
1..*
Employee
- name : String
- id : Integer
+ pay ( ) : Integer
«merge»
EmployeeLocation
Building
- address : String
1
worksAt
*
Employee
- id : Integer
+ moveTo (b : Building)


BasicEmployees
Employee
- name : String
- id : Integer
+ pay ( ) : Integer
+ moveTo (b : Building)
worksAt
*
1
Building
- address : String
1..*
worksAs
1..2
Job
- title : String

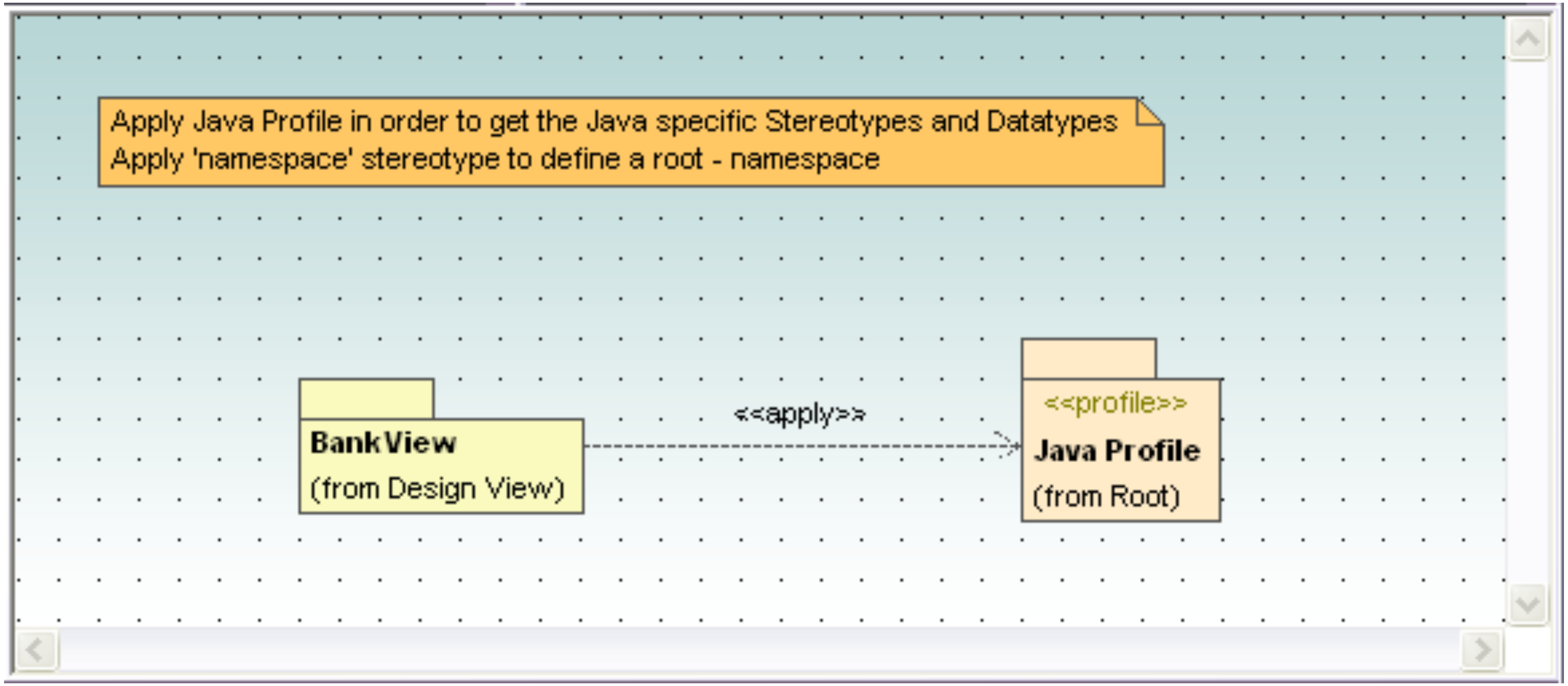
Merge in the UML metamodel



Example (from UML Superstructure, sect 7.1)



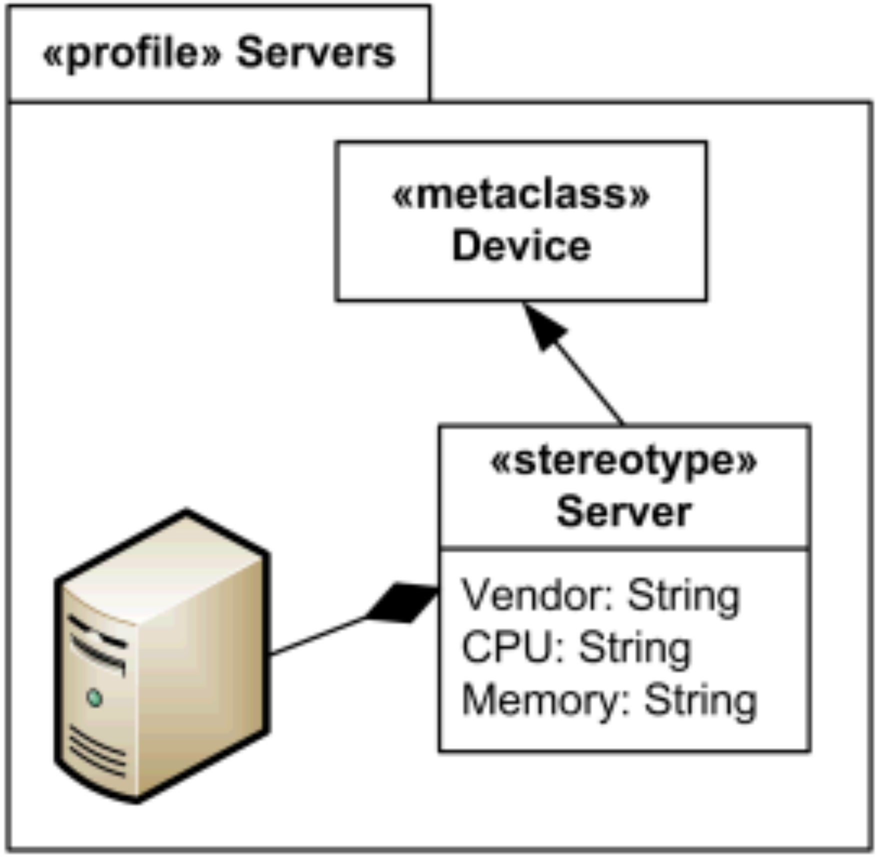
Profile application



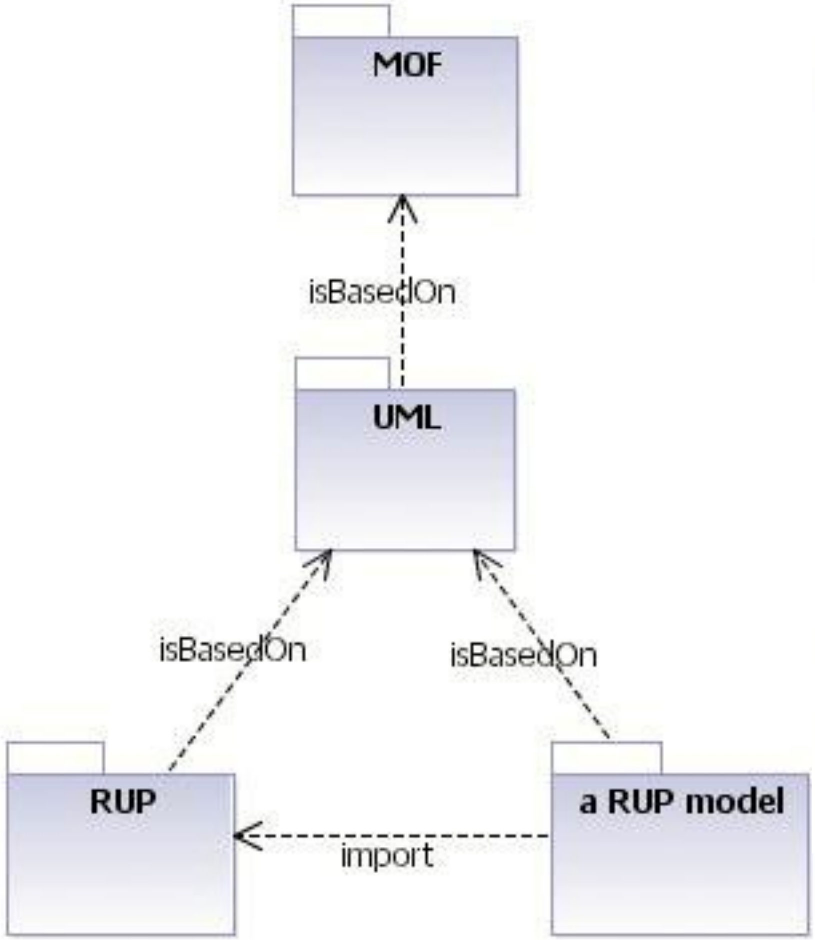
Apply Java Profile in order to get the Java specific Stereotypes and Datatypes
Apply 'namespace' stereotype to define a root - namespace

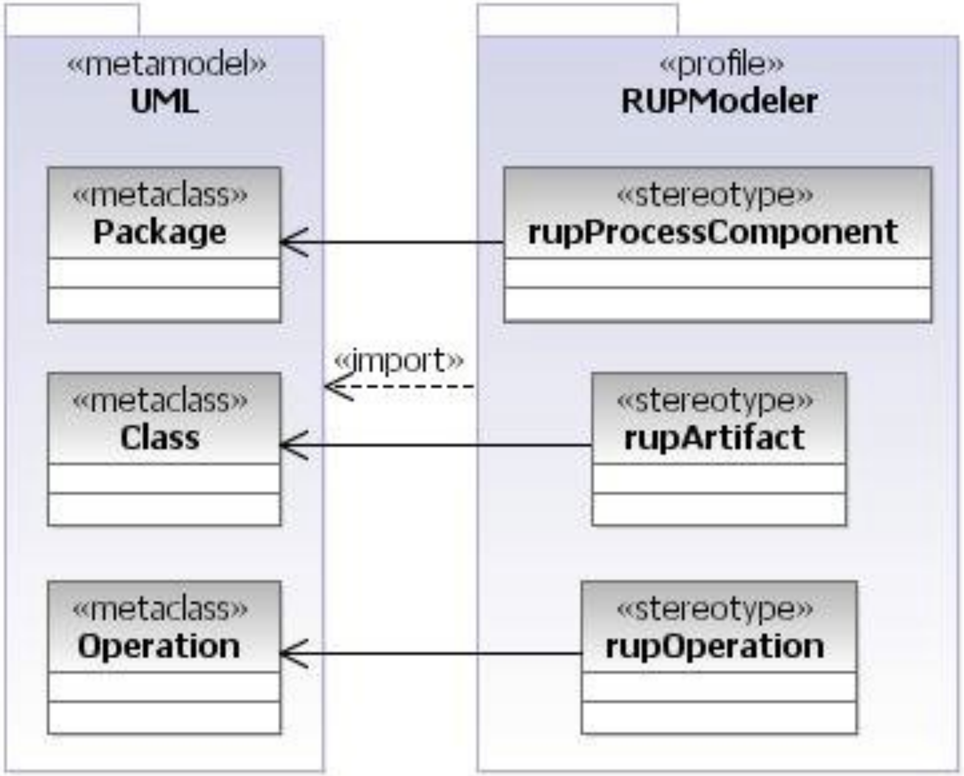
Profile diagram

- A profile diagram is a special package diagram including stereotypes and tagged values
- Used to describe a domain or a reference architecture



Example: RUP as a profile

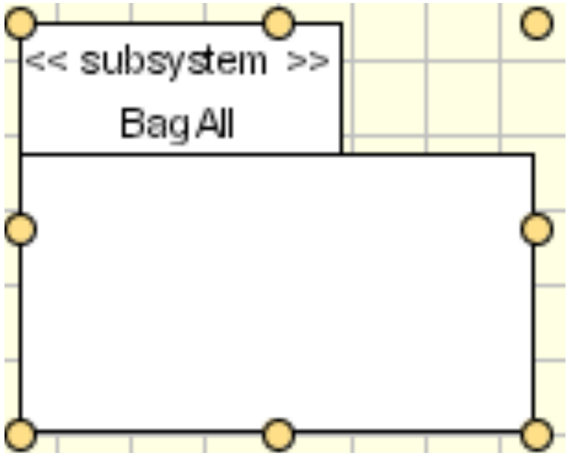




A fragment of the descriptive stereotypes

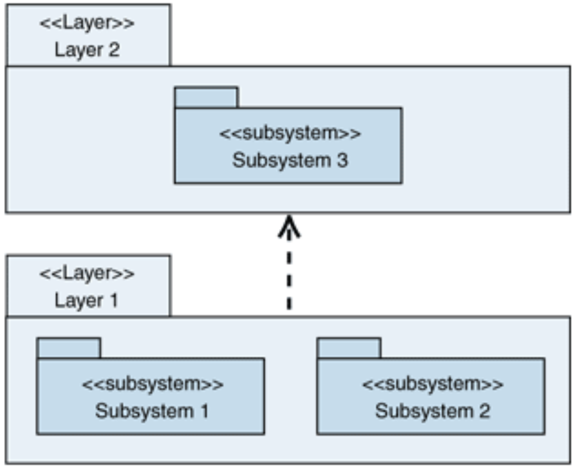
Note on sub-systems

- A subsystem is labeled with a <<subsystem>> stereotype
- In UML1 a subsystem is both a classifier and a package: a unit of behavior
- In UML2 a subsystem is a component



Package vs sub-system

- Packages are a general purpose mechanism for *grouping* elements
  - E.g. java.util package is not a subsystem
- Subsystems have clearly defined responsibilities, implement requirements, exhibit behavior, and expose interfaces that do work



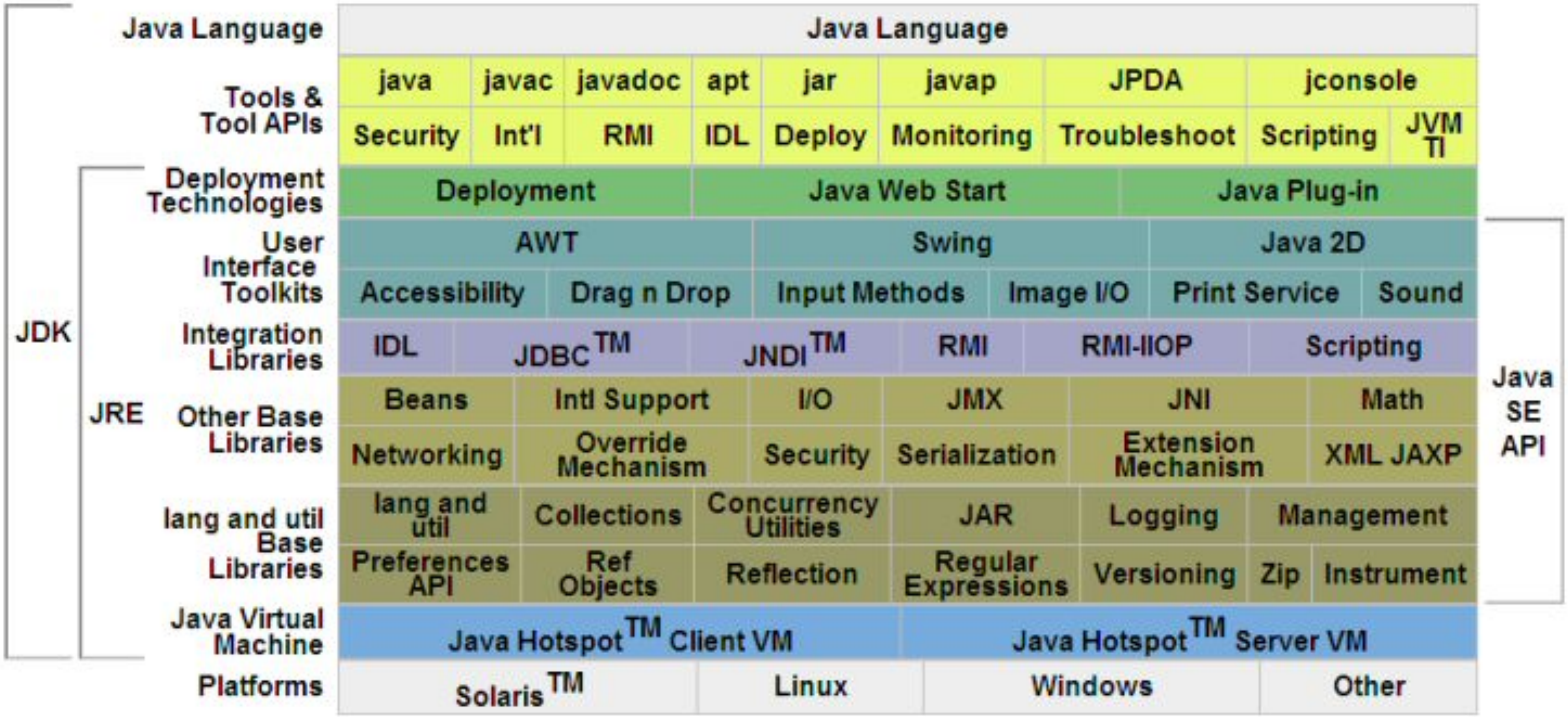
Exercise

Describe using UML packages the structure and dependencies of some Java libraries you know

Example: Java packages

Packages include:

- java.applet
- java.awt
- java.io
- java.math



JDK

JRE

Java SE API

Java Language: Java Language

Tools & Tool APIs: java, javac, javadoc, apt, jar, javap, JPDA, jconsole

Security, Int'l, RMI, IDL, Deploy, Monitoring, Troubleshoot, Scripting, JVM TI

Deployment Technologies: Deployment, Java Web Start, Java Plug-in

User Interface Toolkits: AWT, Swing, Java 2D

Accessibility, Drag n Drop, Input Methods, Image I/O, Print Service, Sound

Integration Libraries: IDL, JDBC™, JNDI™, RMI, RMI-IIOP, Scripting

Other Base Libraries: Beans, Intl Support, I/O, JMX, JNI, Math

Networking, Override Mechanism, Security, Serialization, Extension Mechanism, XML JAXP

lang and util Base Libraries: lang and util, Collections, Concurrency Utilities, JAR, Logging, Management

Preferences API, Ref Objects, Reflection, Regular Expressions, Versioning, Zip, Instrument

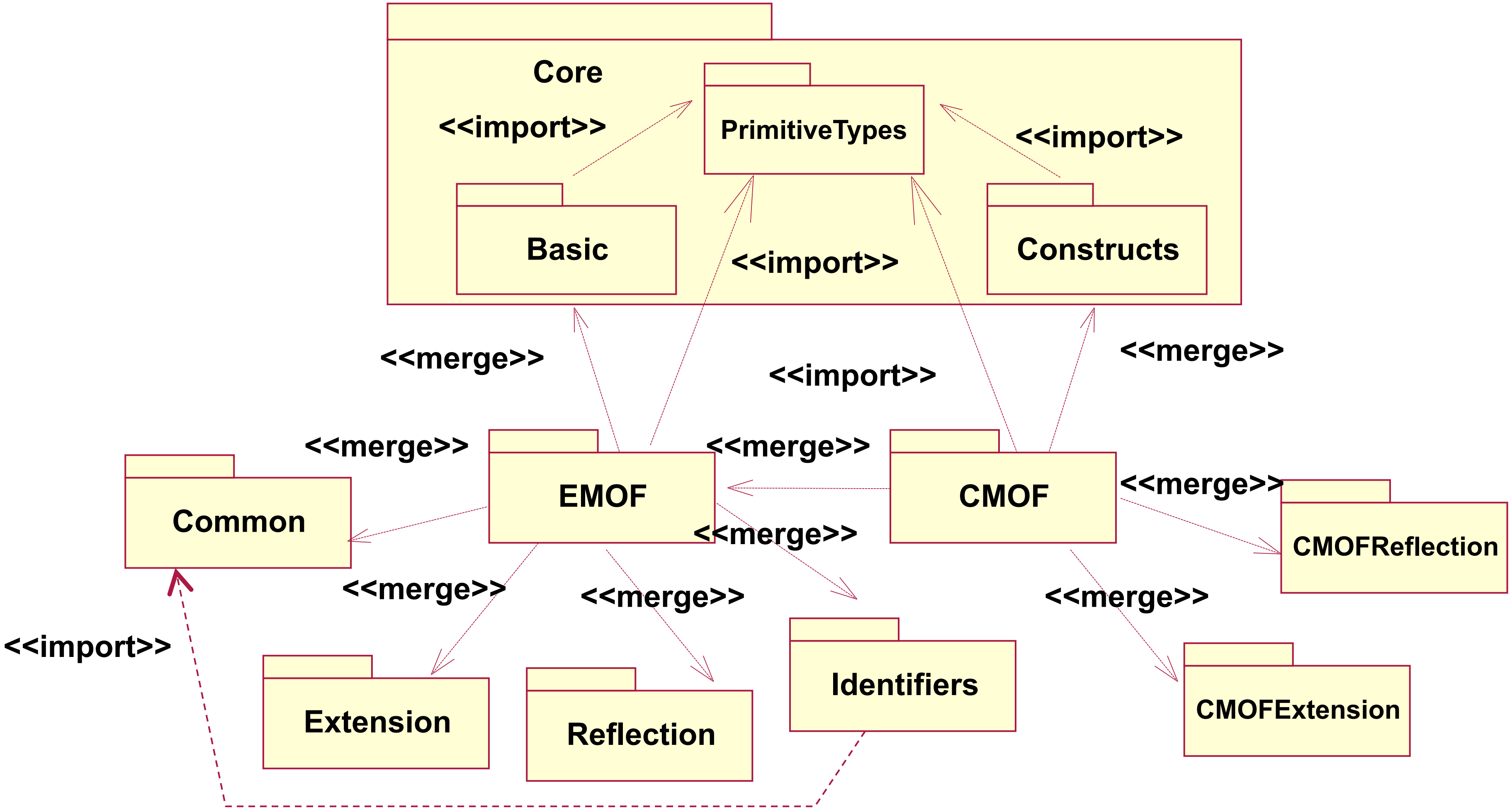
Java Virtual Machine: Java Hotspot™ Client VM, Java Hotspot™ Server VM

Platforms: Solaris™, Linux, Windows, Other

Organizing Classes Into Packages

Package	Purpose	Sample Class
java.lang	Language Support	Math
java.util	Utilities	Random
java.io	Input and Output	PrintScreen
Java.awt	Abstract Windowing Toolkit	Color
Java.applet	Applets	Applet
Java.net	Networking	Socket
Java.sql	Database Access	ResultSet
Java.swing	Swing user interface	JButton
Org.omg.CORBA	Common Object Request Broker Architecture	IntHolder

Example: MOF 2.0 structure



Logical architecture

- Logical architecture is the large-scale organization of the software classes into packages (or namespaces), subsystems, and layers.
- Logical - because there is no decision about how these elements are deployed across different operating system processes or across physical computers in a network (deployment architecture)

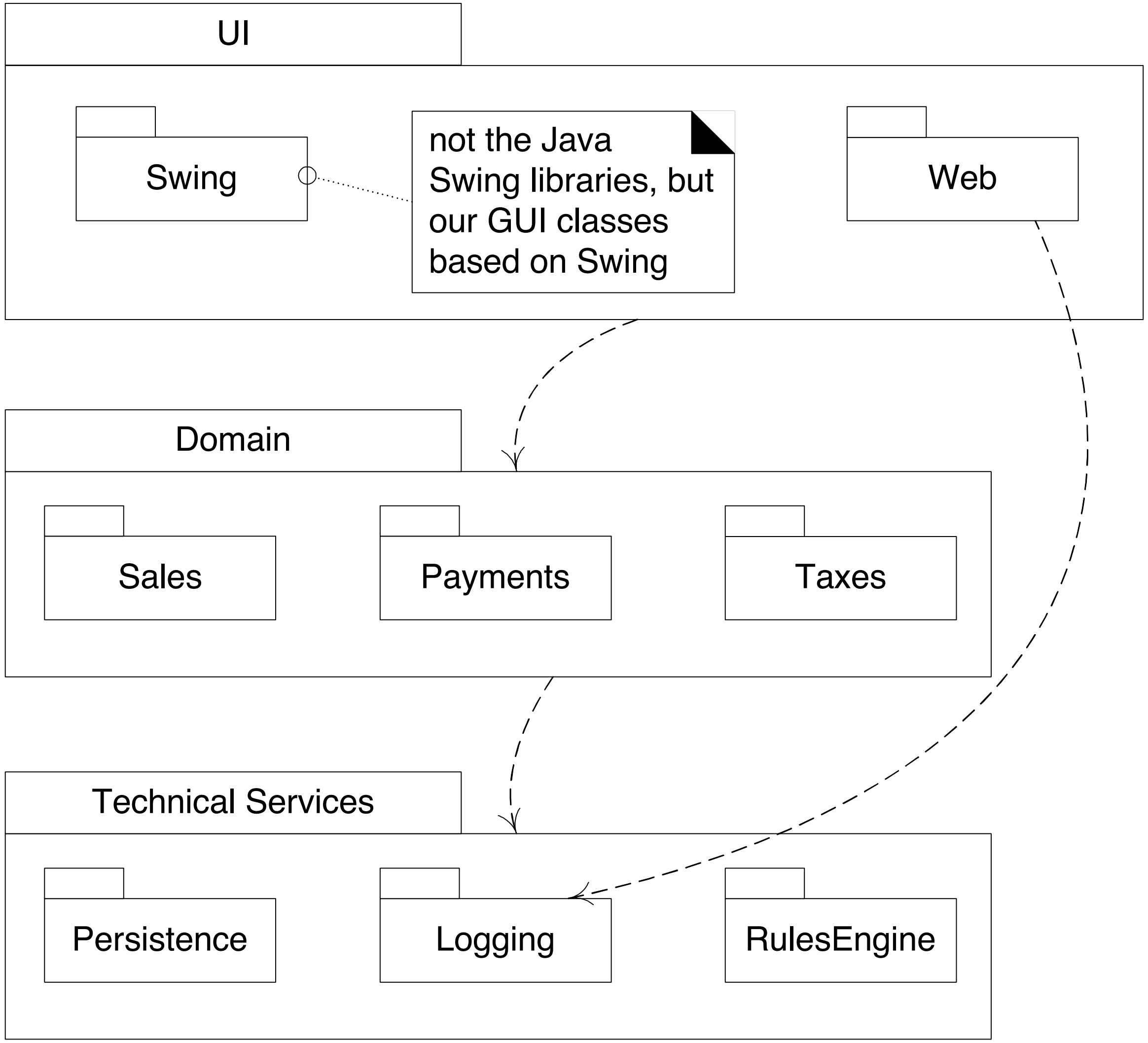
Logical Architecture and Layers

- Logical architecture: the large-scale organization of software classes into packages, subsystems, and layers.
  - “Logical” because no decisions about deployment are implied
- Layer: a very coarse-grained grouping of classes, packages, or subsystems that has cohesive responsibility for a major aspect of the system

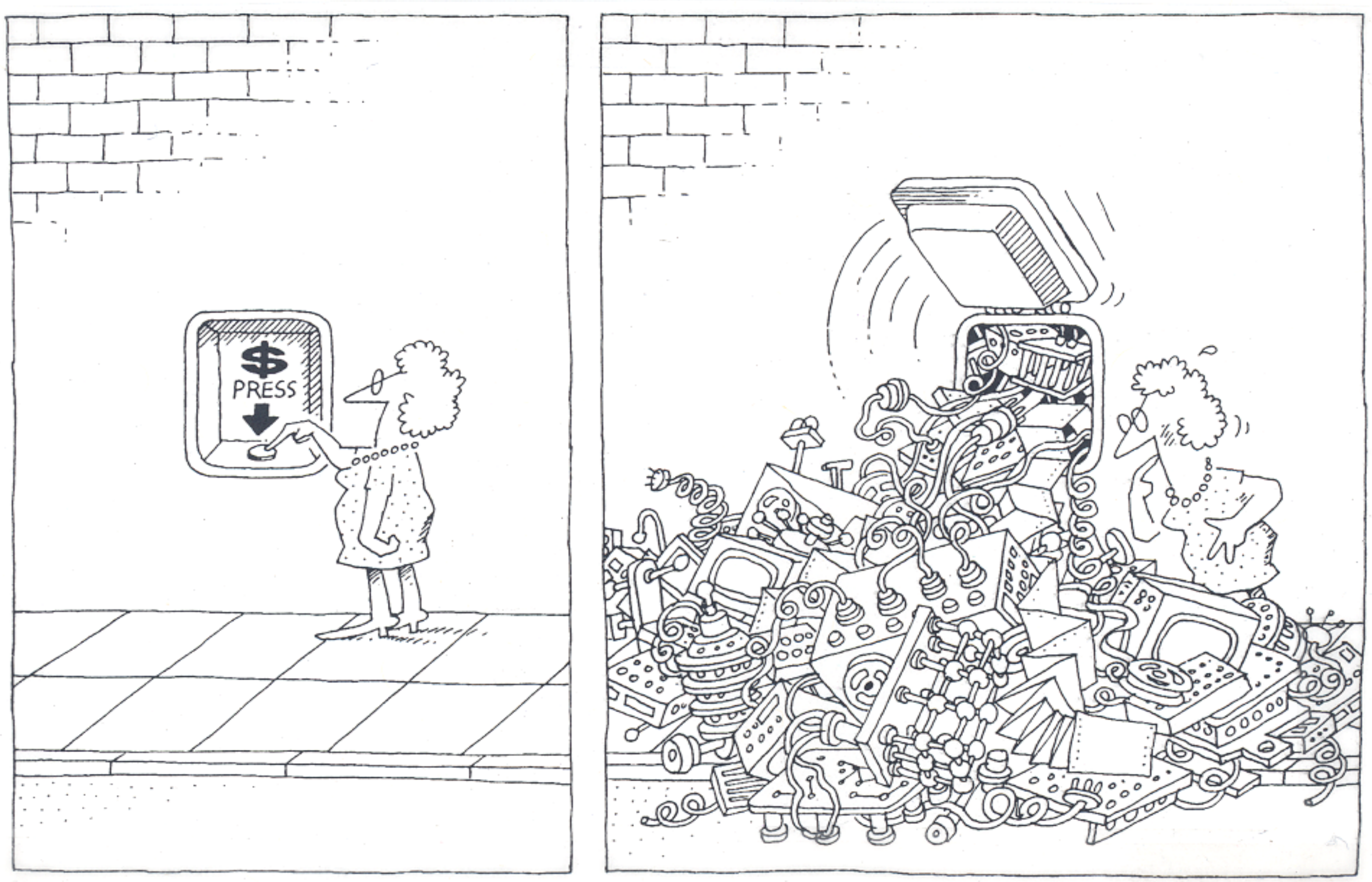
Layered Architectures

- Typical layers in an OO system:
  - User Interface
  - Application Logic and Domain Objects
  - Technical Services
    - Application-independent, reusable across systems.
- Relationships between layers:
  - Strict layered architecture: a layer only calls upon services of the layer directly below it.
  - Relaxed layered architecture: a higher layer calls upon several lower layers.

Layers shown with package diagram



Components

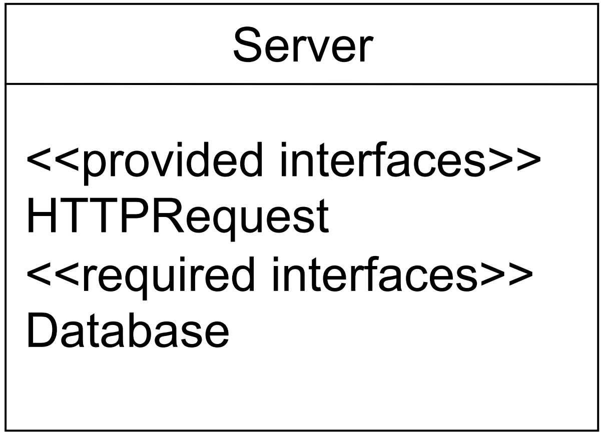


The task of the software development team is to engineer the illusion of simplicity *[Booch]*

Sw architectures are made of components

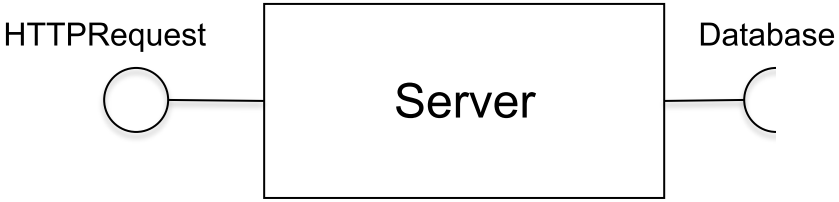
- The main building blocks of software architectures are components
- Example: in a client-server system, there are two kind of components: servers and clients
- Example: in a service oriented architecture each service is a component

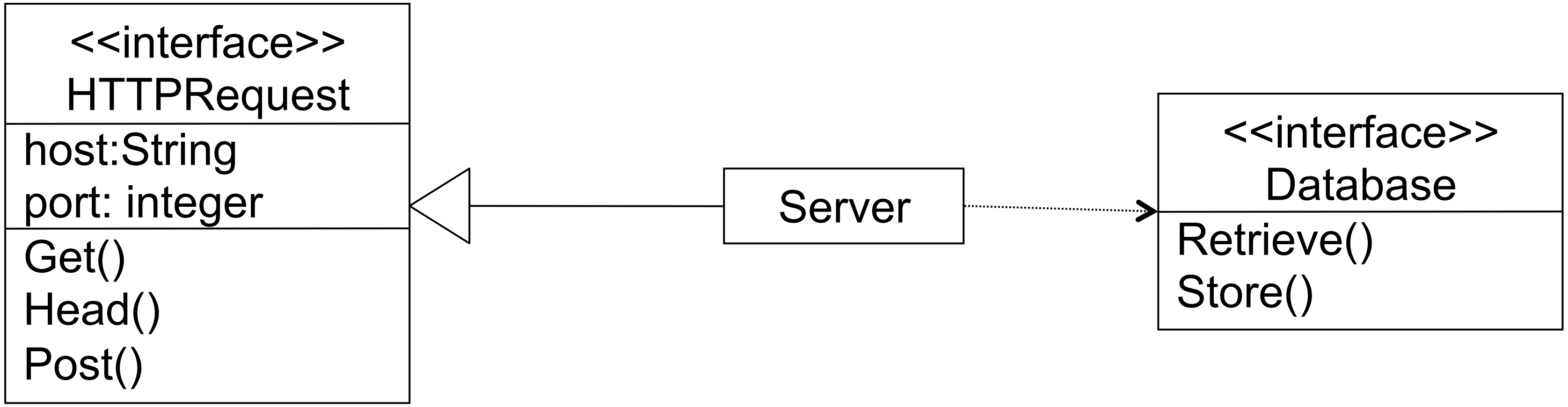
Components are classifiers - example



Server

<<provided interfaces>>
HTTPRequest
<<required interfaces>>
Database





<<interface>>
HTTPRequest

host:String
port: integer

Get()
Head()
Post()

<<interface>>
Database

Retrieve()
Store()

Components in sw architectures

- A component is a binary unit of independent deployment
  - well separated from other components
  - can be deployed independently
- A component is a unit of third-party composition
  - is composable
  - comes with clear specifications of what it requires and provides
  - interacts with its environment through well-defined interfaces
- A component has no persistent state
  - temporary state set only through well-defined interfaces
  - substituible by a functionally equivalent component

[Szyperski]

Components in practice

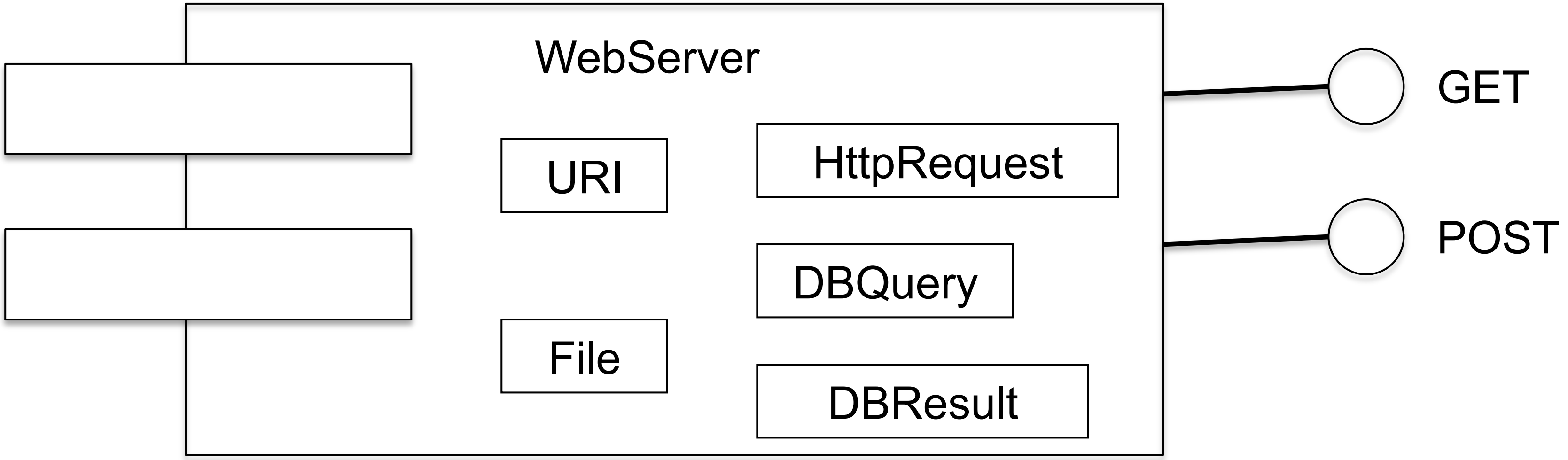
- Plugins - web browsers, audio codecs Winamp, video codecs Quicktime, etc.
- Operating system drivers
- Dynamic link libraries
- A very common usage is to have a core system, often monolithic, and then specify additional modules conforming to the *interface* offered by the core

Problems with components

- Reuse: a component A which provides the same interface as a component B could have been built for a different task (and have a different behavior), and A and B so would not be interchangeable
- Runtime reconfiguration: how to change some component A of a system S while S is running? If we deactivate A, what happens to the components which depend from it?

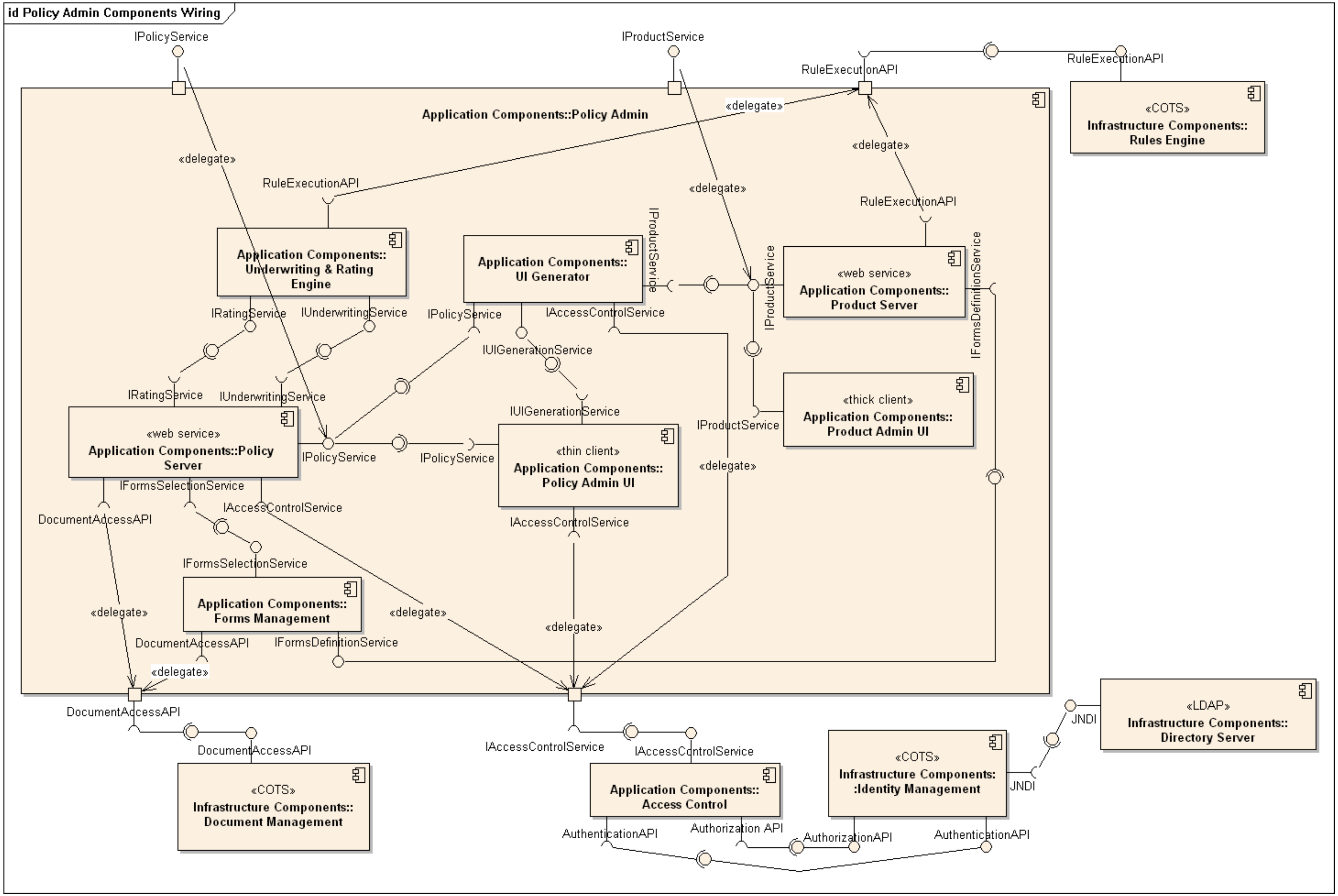
Example

Refined view of a WebServer component

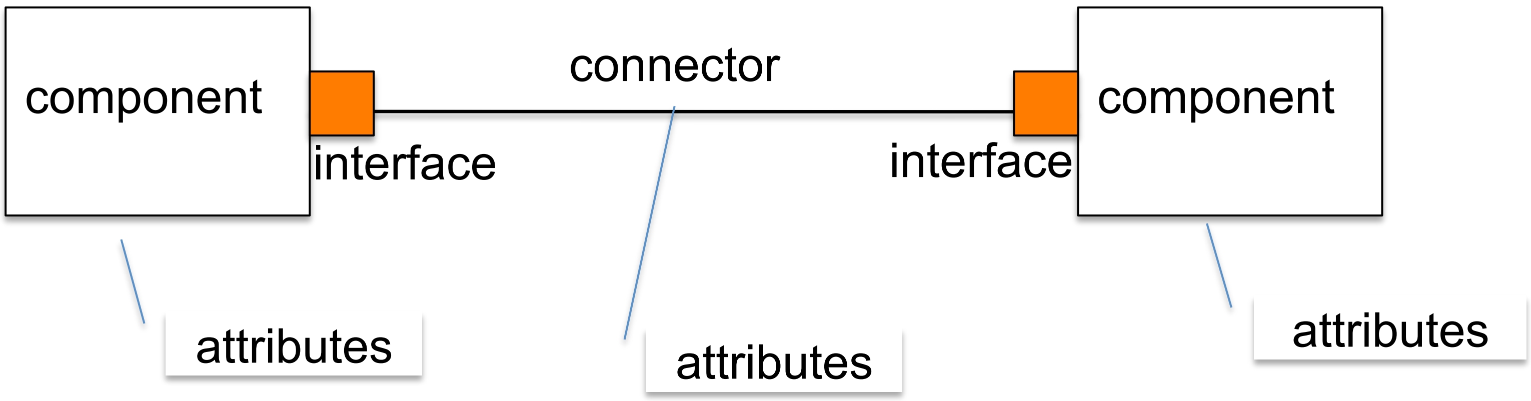


Components in UML2

- A component is a modular unit with well-defined interfaces that is replaceable within its environment
- It has autonomous behavior unit within a system
  - Has one or more *provided* or *required* interfaces
  - A component is encapsulated: its internals are hidden and inaccessible; possibly cohesive
  - Its dependencies are designed such that it can be treated as independently as possible


id Policy Admin Components Wiring
IPolicyService
IProductService
RuleExecutionAPI
RuleExecutionAPI
«COTS»
Infrastructure Components::
Rules Engine
«delegate»
Application Components::Policy Admin
«delegate»
«delegate»
«delegate»
RuleExecutionAPI
RuleExecutionAPI
Application Components::
Underwriting & Rating
Engine
Application Components::
UI Generator
IProductService
IProductService
«web service»
Application Components::
Product Server
IFormsDefinitionService
IRatingService
IUnderwritingService
IPolicyService
IAccessControlService
IUIGenerationService
IRatingService
IUnderwritingService
IUIGenerationService
«thick client»
Application Components::
Product Admin UI
IProductService
«web service»
Application Components::Policy
Server
IPolicyService
IPolicyService
«thin client»
Application Components::
Policy Admin UI
«delegate»
IFormsSelectionService
IAccessControlService
DocumentAccessAPI
IAccessControlService
IFormsSelectionService
Application Components::
Forms Management
«delegate»
«delegate»
«delegate»
DocumentAccessAPI
IFormsDefinitionService
«delegate»
DocumentAccessAPI
DocumentAccessAPI
IAccessControlService
IAccessControlService
«COTS»
Infrastructure Components::
Document Management
Application Components::
Access Control
«COTS»
Infrastructure Components:
:Identity Management
JNDI
JNDI
«LDAP»
Infrastructure Components::
Directory Server
AuthenticationAPI
Authorization API
AuthorizationAPI
AuthenticationAPI

Components and connectors



component

connector

component

interface

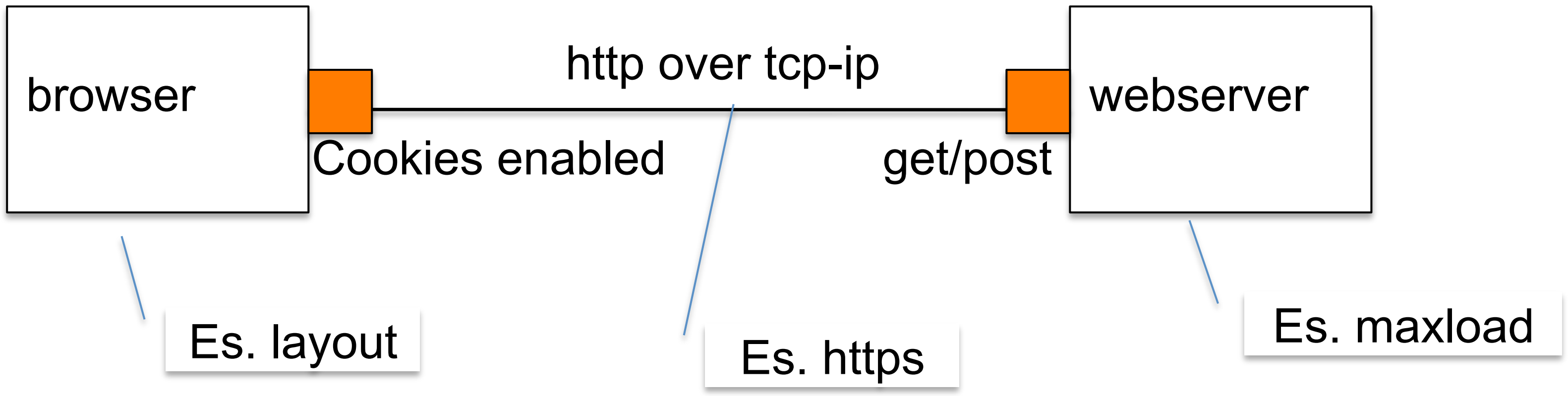
interface

attributes

attributes

attributes

Example:



browser

http over tcp-ip

webserver

Cookies enabled

get/post

Es. layout

Es. https

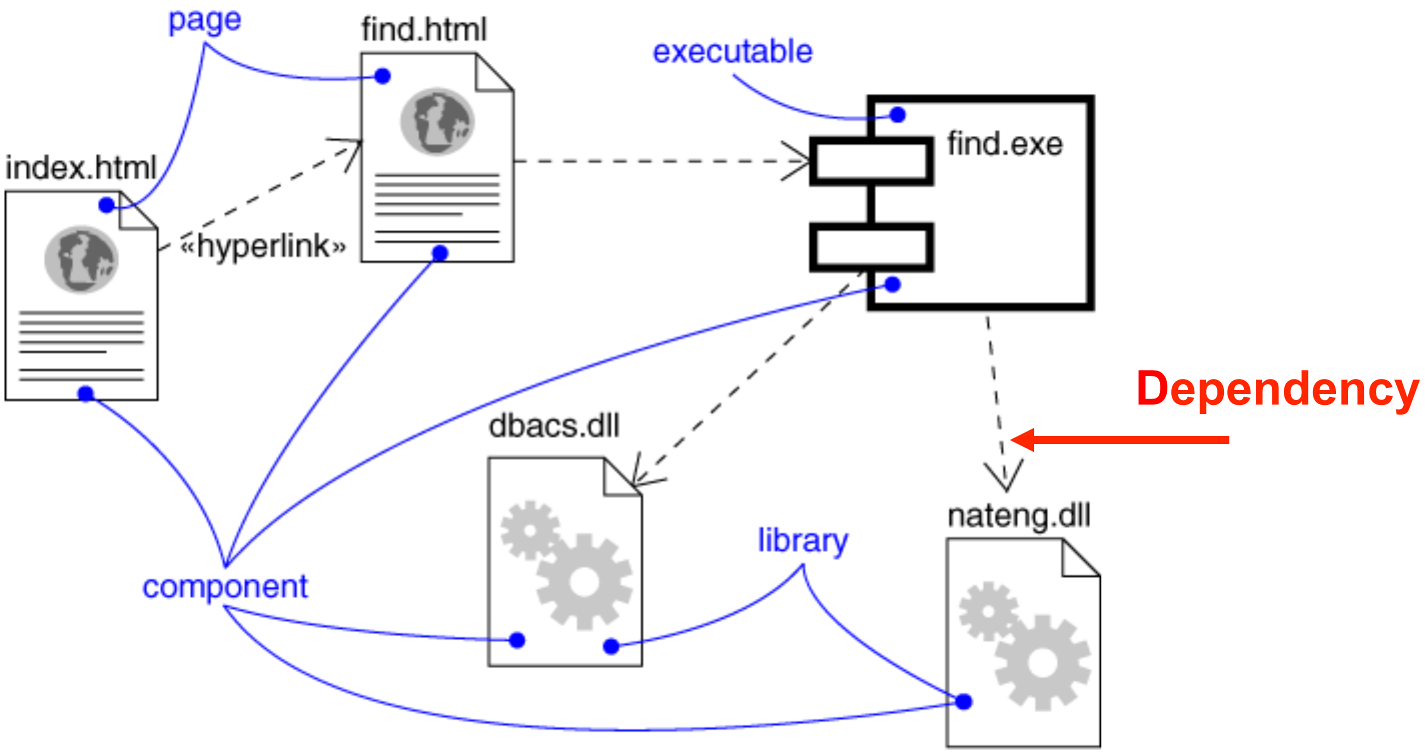
Es. maxload

Component diagram

- *Component diagrams* show the organization and dependencies among software components
- Component types:
  - Documents (eg. source code)
  - Object code (binary file, static or dynamic libraries)
  - Executable program

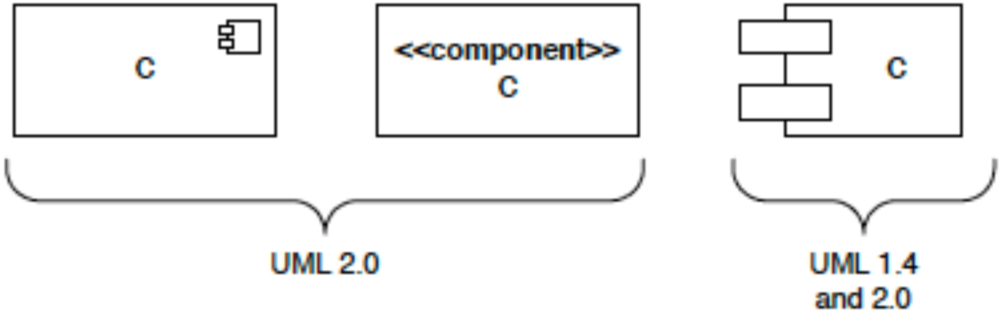
Component diagram

- Captures the physical structure of the implementation
- Shows dependencies

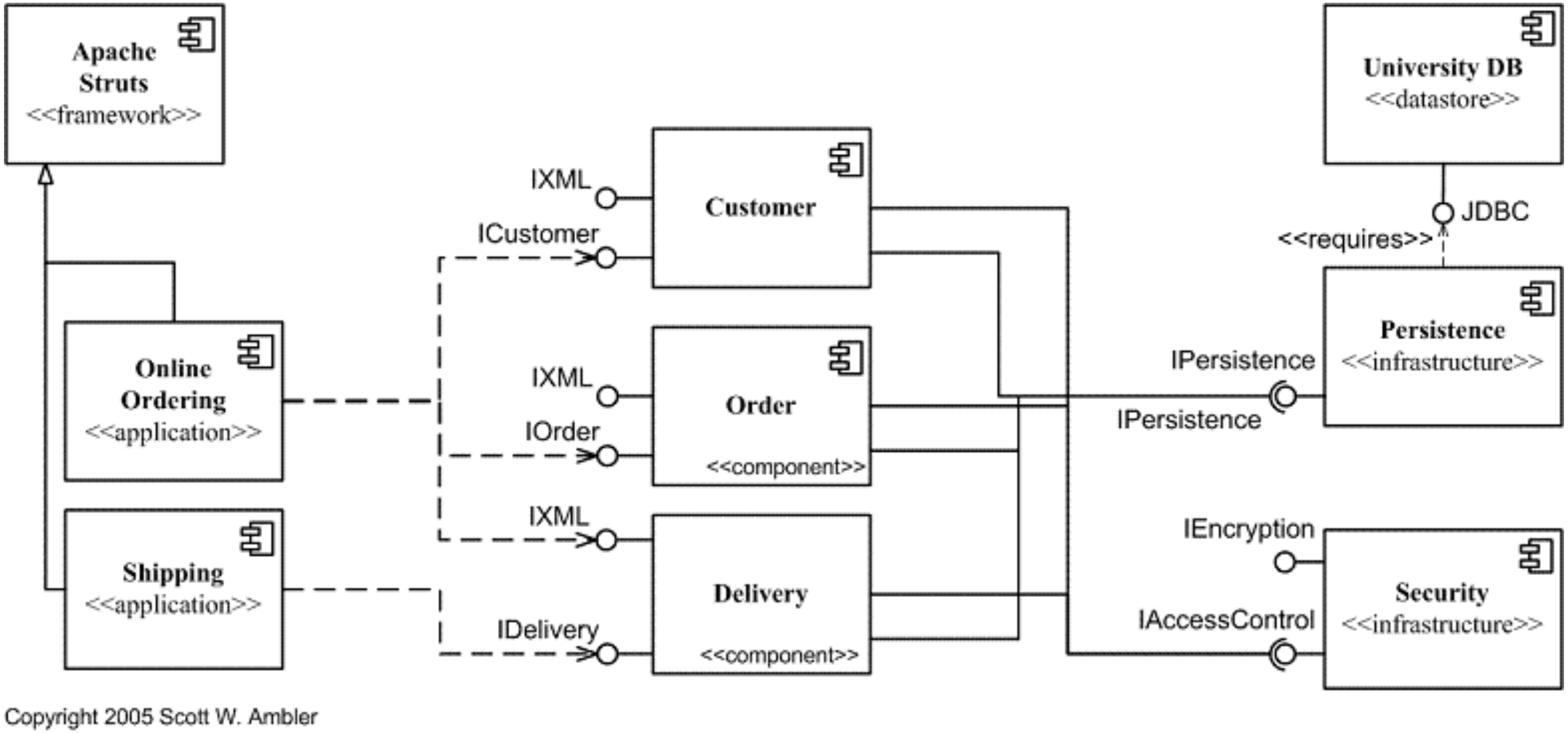


Component diagrams

- Drawing a component is very similar to drawing a class. In fact, a component is a specialized version of the class concept
- A component is drawn as a rectangle with optional compartments stacked vertically
- The component stereotype icon is a rectangle with two smaller rectangles protruding on its left side (was the UML 1.4 notation for a component)

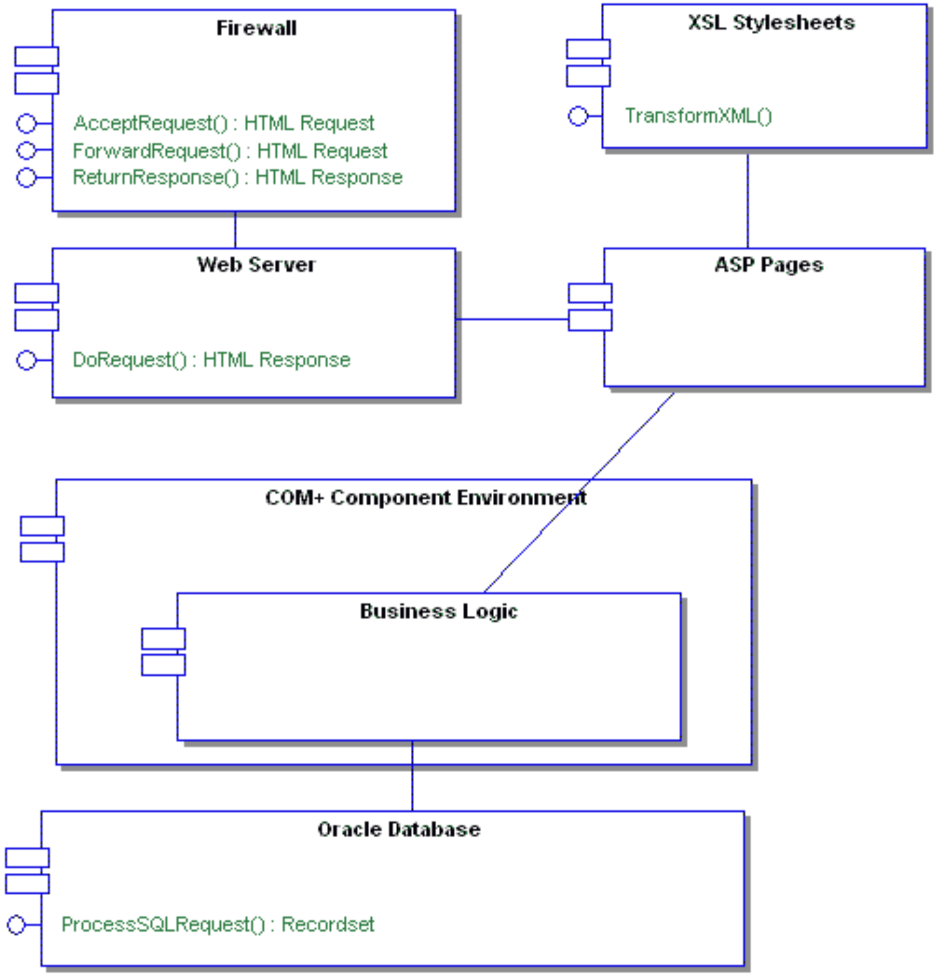


Component diagram: example

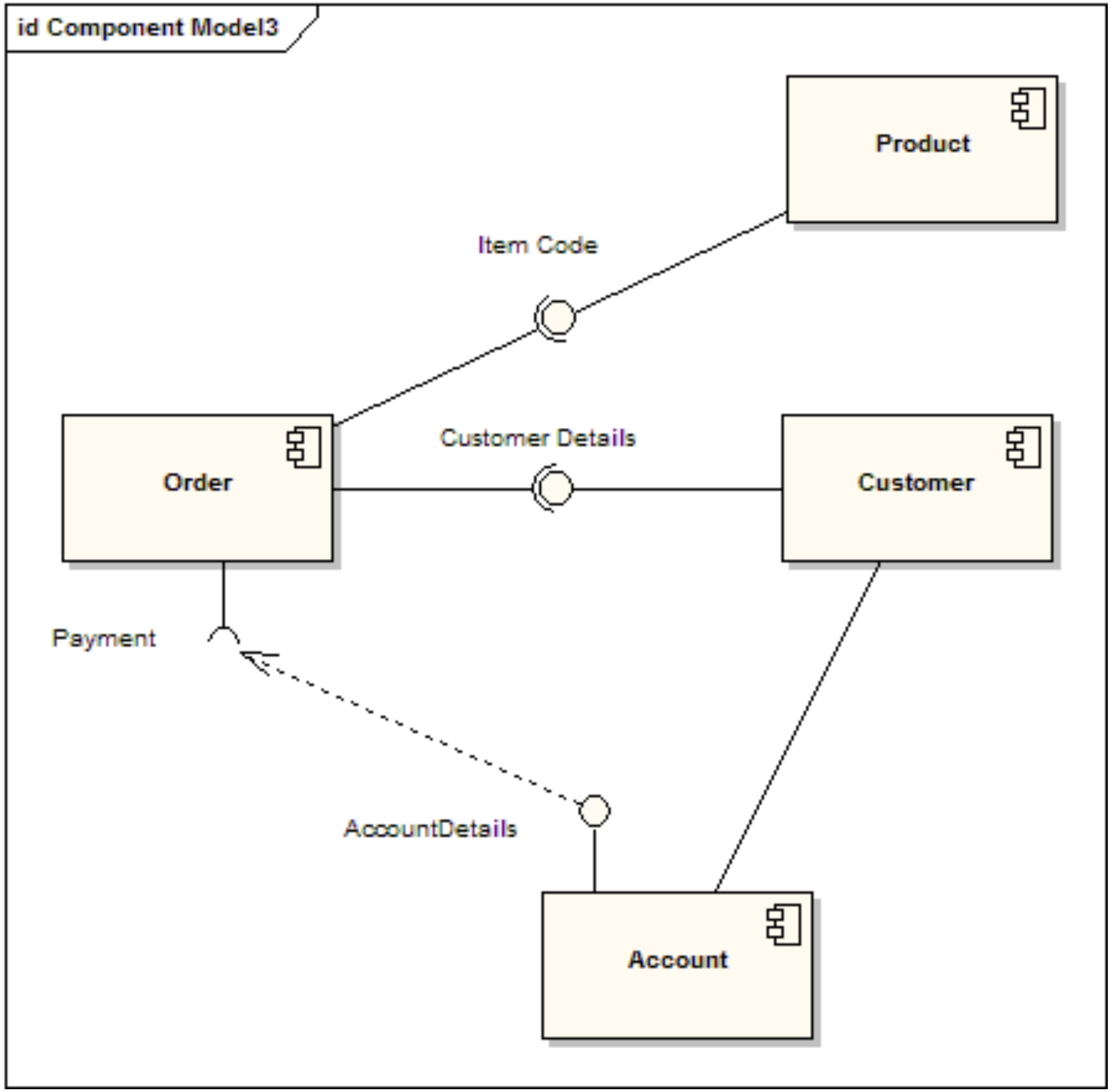


Copyright 2005 Scott W. Ambler

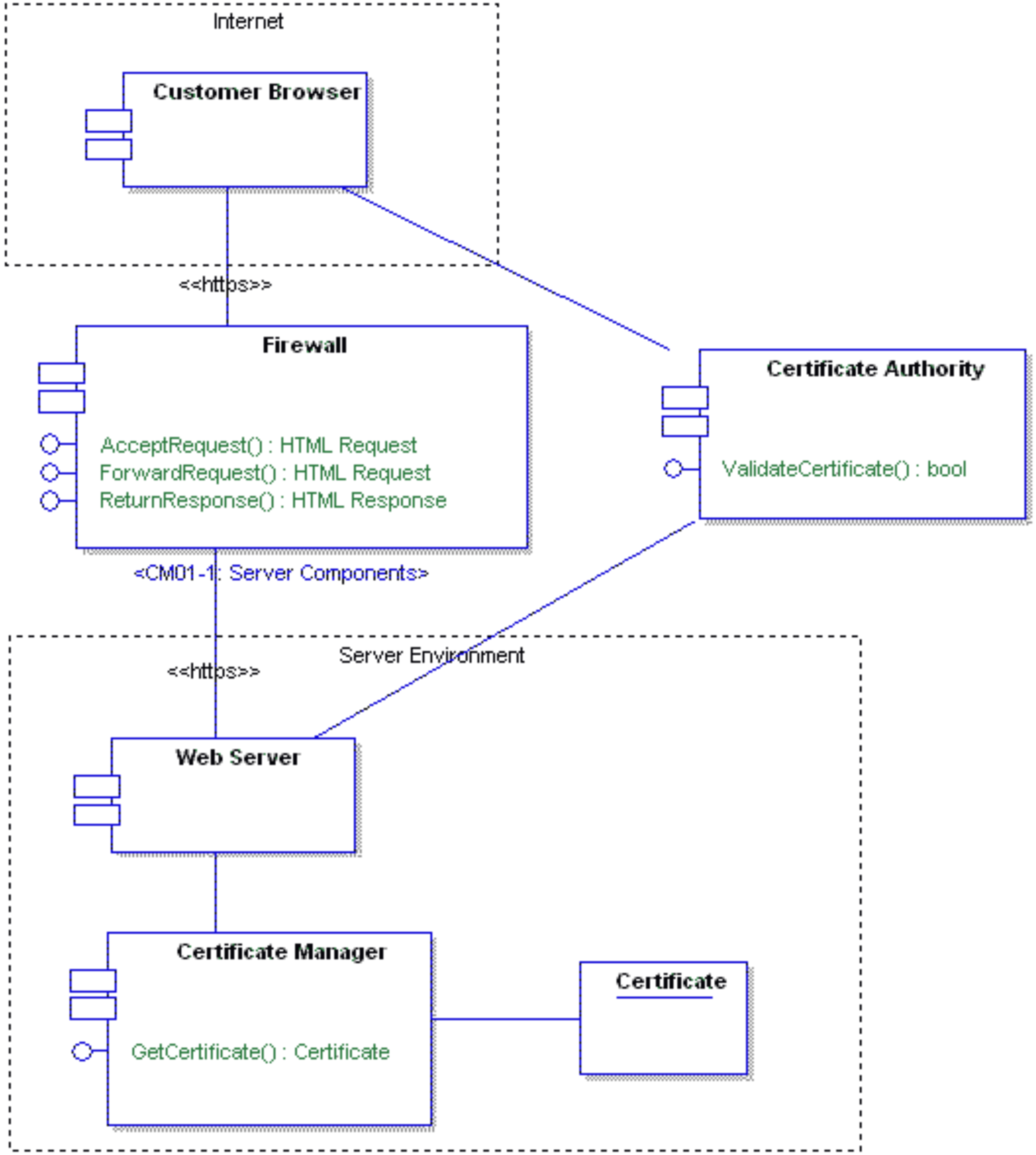
Component diagram: example


Firewall
AcceptRequest() : HTML Request
ForwardRequest() : HTML Request
ReturnResponse() : HTML Response
XSL Stylesheets
TransformXML()
Web Server
DoRequest() : HTML Response
ASP Pages
COM+ Component Environment
Business Logic
Oracle Database
ProcessSQLRequest() : Recordset

Component diagram: example



Example


Internet
Customer Browser
<<https>>
Firewall
AcceptRequest() : HTML Request
ForwardRequest() : HTML Request
ReturnResponse() : HTML Response
Certificate Authority
ValidateCertificate() : bool
<CM01-1: Server Components>
Server Environment
<<https>>
Web Server
Certificate Manager
GetCertificate() : Certificate
Certificate

Component elements

- A component can have
  - Interfaces

    An interface represents a declaration of a set of operations and obligations
  - Usage dependencies

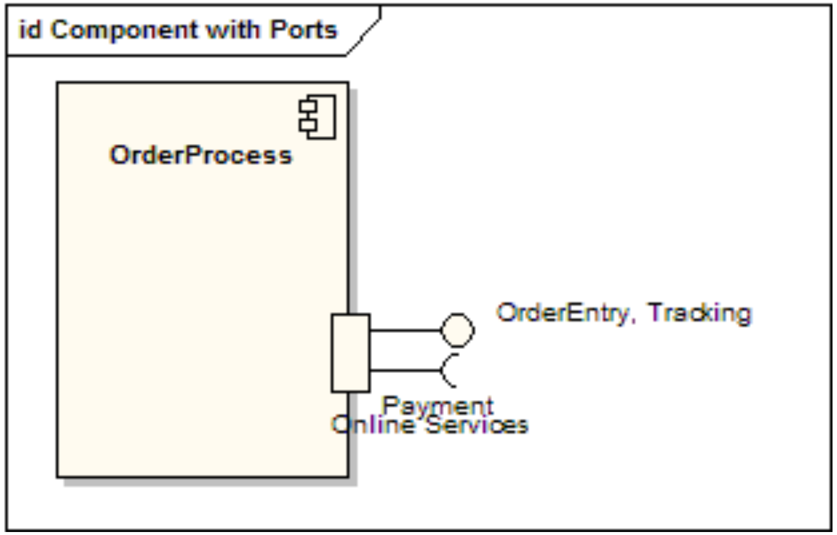
    A usage dependency is relationship which one element requires another element for its full implementation
  - Ports

    A port represents an interaction point between a component and its environment
  - Connectors (beware: these are UML mechanisms)
    - A connector connects two components
    - Can also connect the external contract of a component to its internal structure

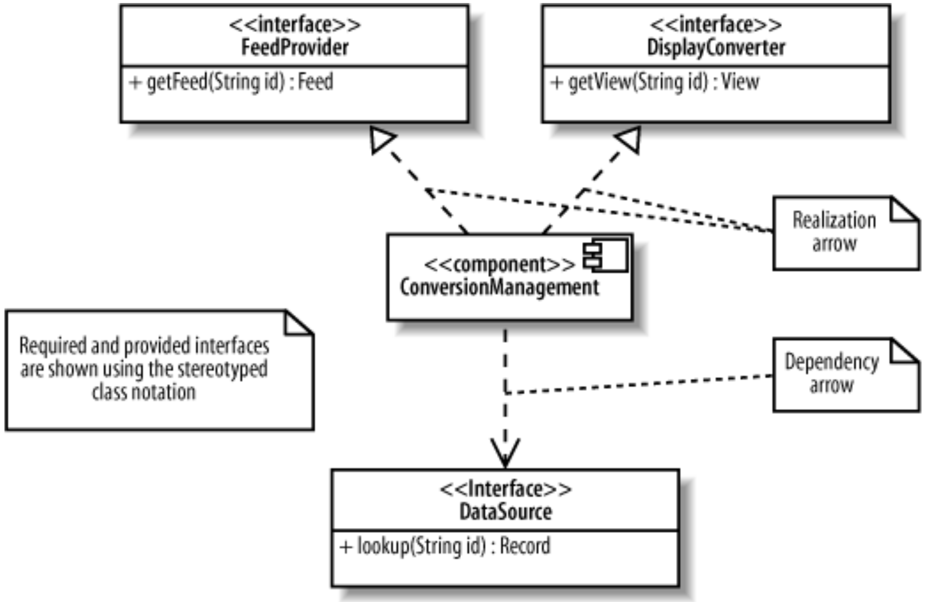
Ports

- Using ports in component diagrams allows for a service or behavior to be specified to its environment as well as a service or behavior that a component requires.
- Ports may specify inputs and outputs as they can operate bi-directionally

The example details a component with a port for online services with composes two *provided* interfaces "order entry" and "tracking" and a *required* interface "payment"



Equivalent representations



<<interface>>
FeedProvider
+ getFeed(String id) : Feed

<<interface>>
DisplayConverter
+ getView(String id) : View

<<component>>
ConversionManagement

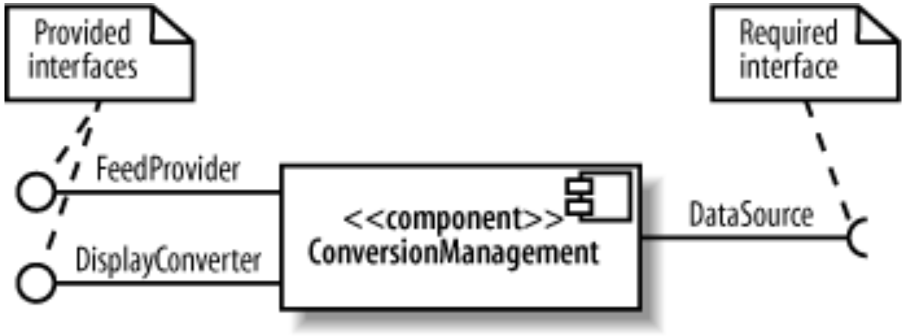
Realization arrow

Dependency arrow

Required and provided interfaces are shown using the stereotyped class notation

<<Interface>>
DataSource
+ lookup(String id) : Record

Using class stereotype



Provided interfaces

Required interface

FeedProvider

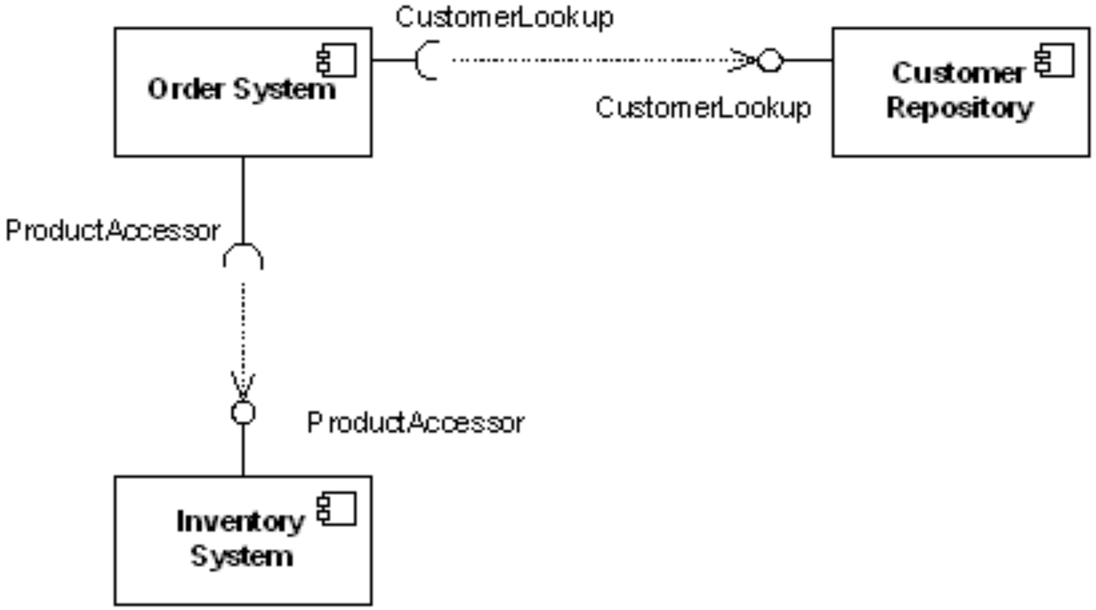
DisplayConverter

<<component>>
ConversionManagement

DataSource

Using lollipop stereotype

Dependencies and ports



- When showing a component's relationship with other components, the lollipop and socket notation must also include a dependency arrow
- Note that the dependency arrow comes out of the consuming (requiring) socket and its arrow head connects with the provider's lollipop

Connectors

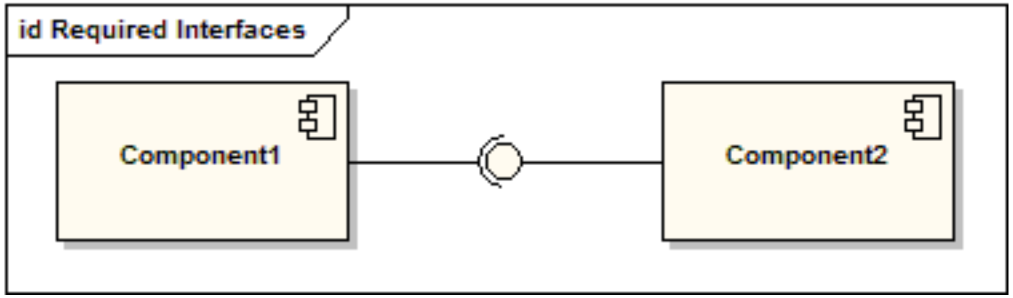
UML2 introduced connectors for ports

Typical UML connectors are:

- Assembly
- Delegate
- Associate
- Generalize
- Realize

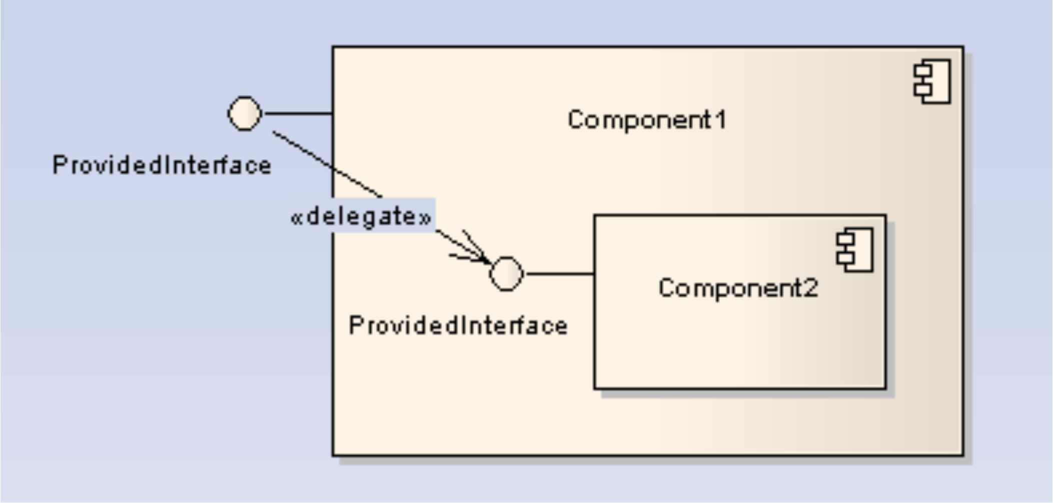
Assembly connector

- The **assembly** connector bridges a component’s required interface (Component1) with the provided interface of another component (Component2)
- This allows a component to provide the services that another component requires



Delegate connector

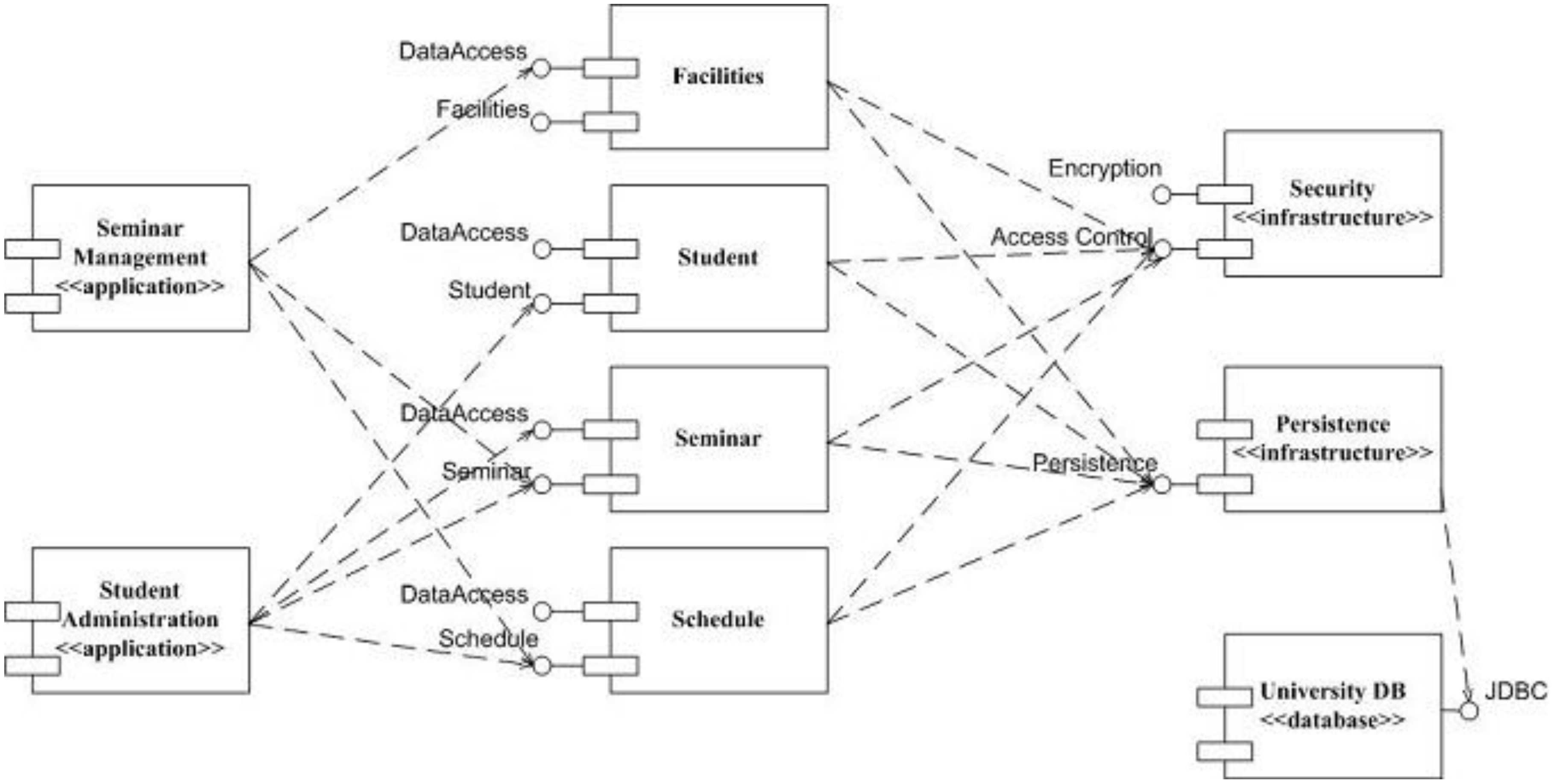
- The **delegate** connector defines the internal assembly of a component's external ports and interfaces
- It links the external contract of a component (specified by its ports) to the internal realization of that behavior by the component's parts
- It represents the forwarding of signals (operation requests and events): a signal that arrives at a port that has a delegation connector to a part or to another port will be passed on to that target for handling



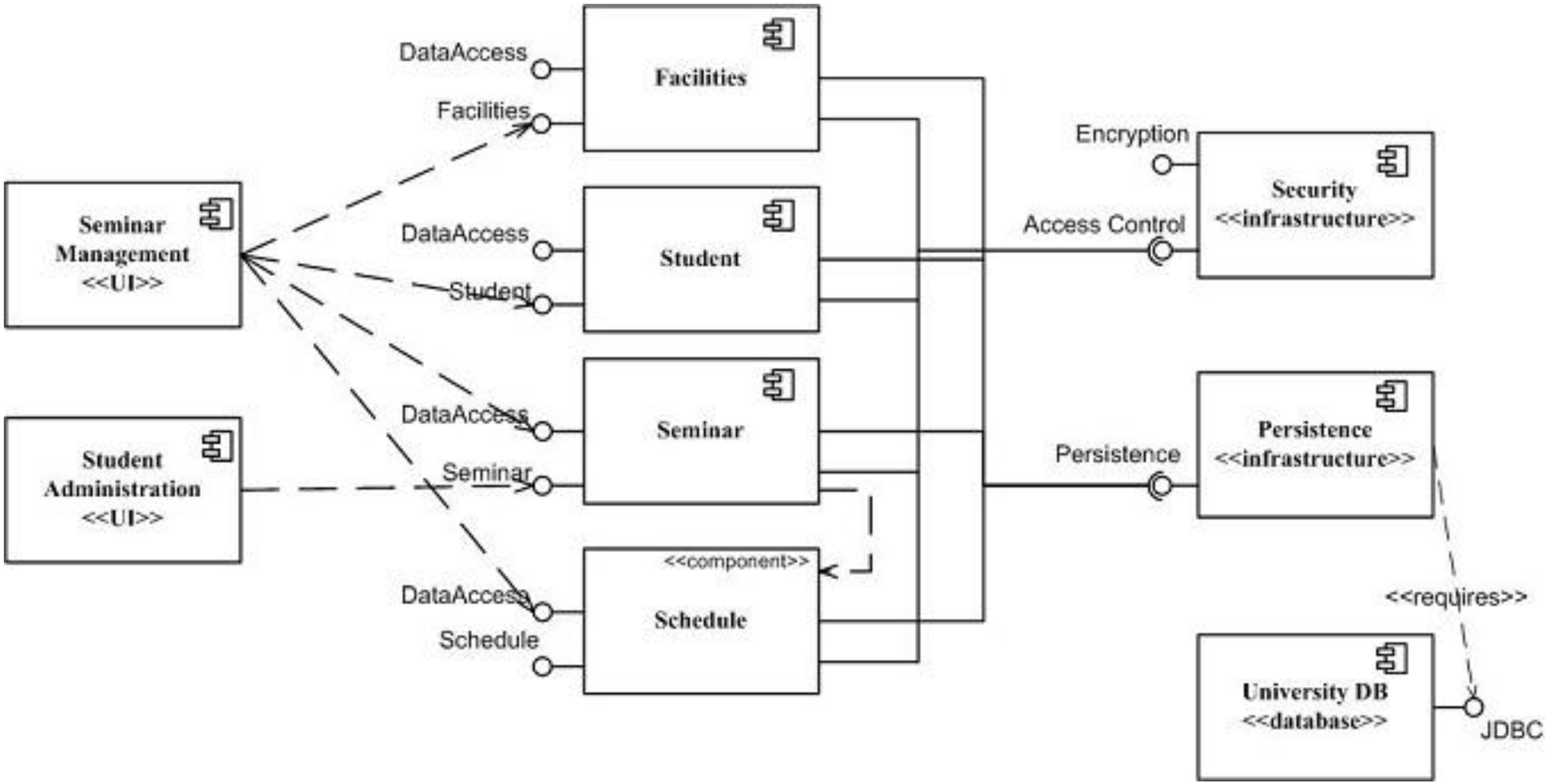
Components in UML1

- In UML 1.1, a component represented an implementation item, such as a file or an executable
- This conflicted with the more common use of the term “component”, which refers to things such as COM components
- Over time and across successive versions, the original UML meaning of components has been (mostly) lost

Components in UML1


DataAccess
Facilities
Facilities
Seminar Management <<application>>
DataAccess
Student
Student
DataAccess
Seminar
Seminar
Student Administration <<application>>
DataAccess
Schedule
Schedule
Encryption
Security <<infrastructure>>
Access Control
Persistence
Persistence <<infrastructure>>
University DB <<database>>
JDBC

Components in UML2



Components in UML2

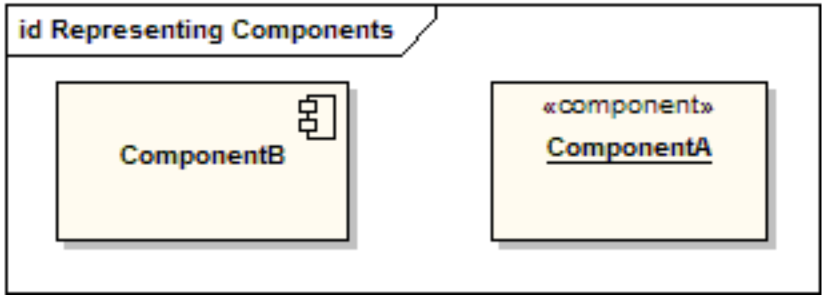
- In UML 2, components are autonomous, encapsulated units within a system or subsystem that provide one or more interfaces
- Components are design units that represent things that will typically be implemented using replaceable modules
- Unlike UML 1.x, components are in UML2 strictly logical, design-time constructs
- The idea is that we can easily reuse and/or substitute a different component implementation in our designs because a component encapsulates behavior and implements specified interfaces

Artifacts

- The physical items called components in UML1.x are called "artifacts" in UML 2
- An artifact is a physical unit, such as a file, executable, script, database, etc.
- Only artifacts live on physical nodes; classes and components do not have "location"
- However, an artifact may “manifest” components and other classifiers (i.e., classes)
- A single component could be “*manifested*” by multiple artifacts, which could be on the same or different nodes, so a single component could indirectly be implemented on multiple nodes

Difference between package and component diagrams

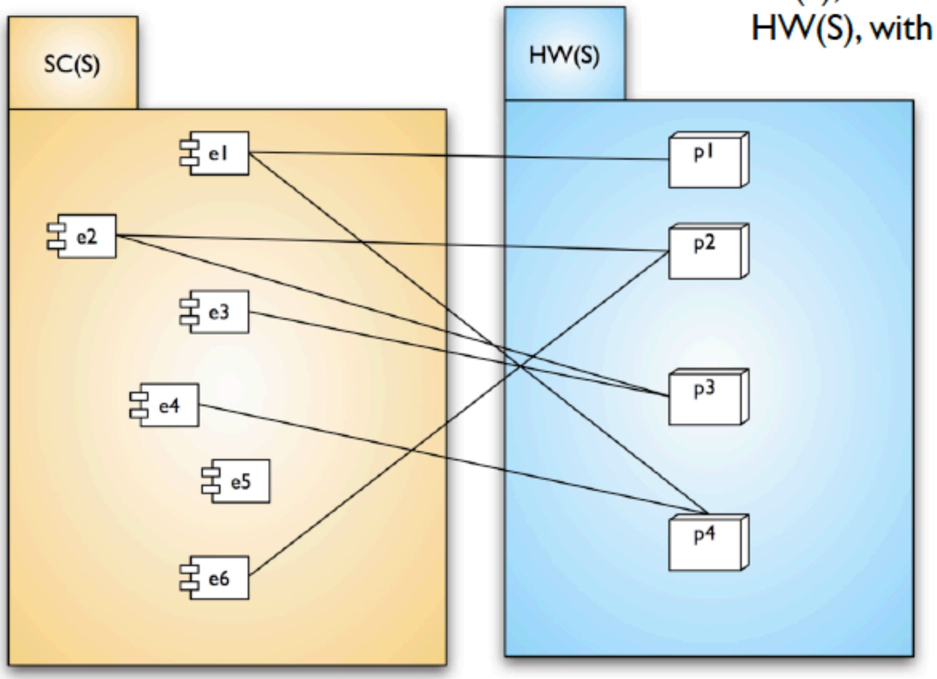
- Components (or *subsystems*, in UML2) are similar to package diagrams, as they define boundaries used to group elements into logical structures
- The difference between package and component diagrams is that the latter offer a semantically richer grouping mechanism
- In a component diagram all model elements are private, whereas a package diagram only displays public items



Remember

- A Package Diagram is used to organize the model
- A Component Diagram is used to organize the solution
- Packages are a tool for version control (and configuration management)
- Components are a tool for representing the solution logic in terms of a configuration

Configuration



Consider two views of a system, S, a software component view, SC(S), with software elements, e1, ... e6, and a hardware view, HW(S), with hardware platforms, p1, ... p4

A view correspondence expressing which software elements execute on which platforms might be:
ExecutesOn = { (e1, p1), (e1, p4), (e2, p2), (e2, p3), (e3, p3), (e4, p4), e6, p2) }

Composite structure vs component diagrams

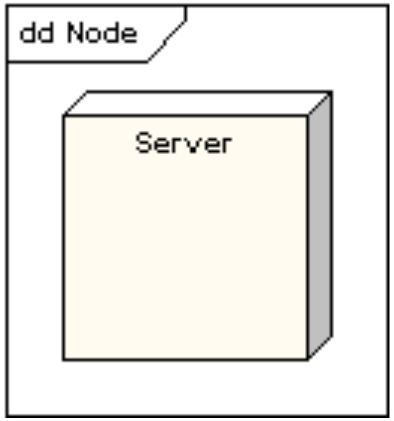
- A component diagram shows the internal parts, connectors and ports that implement a component; when the component is instantiated, copies of its internal parts are also instantiated
- A composite structure diagram shows the internal structure of a class or a collaboration

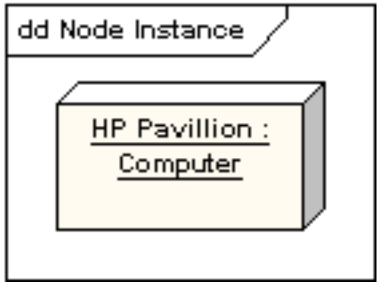
Exercise

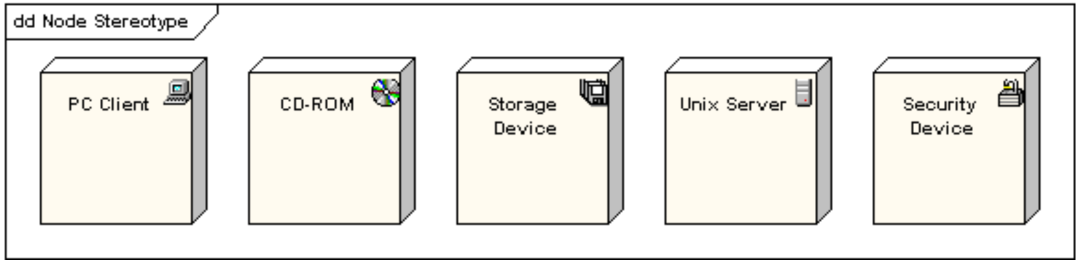
Describe the runtime structure of components of a system you know, and compare it with its package structure

Deployment diagram

- A deployment diagram models the run-time architecture of a system. It shows the configuration of the hardware elements (nodes) and shows how software components are mapped onto those nodes
- A **node** is shown as a three-dimensional box shape

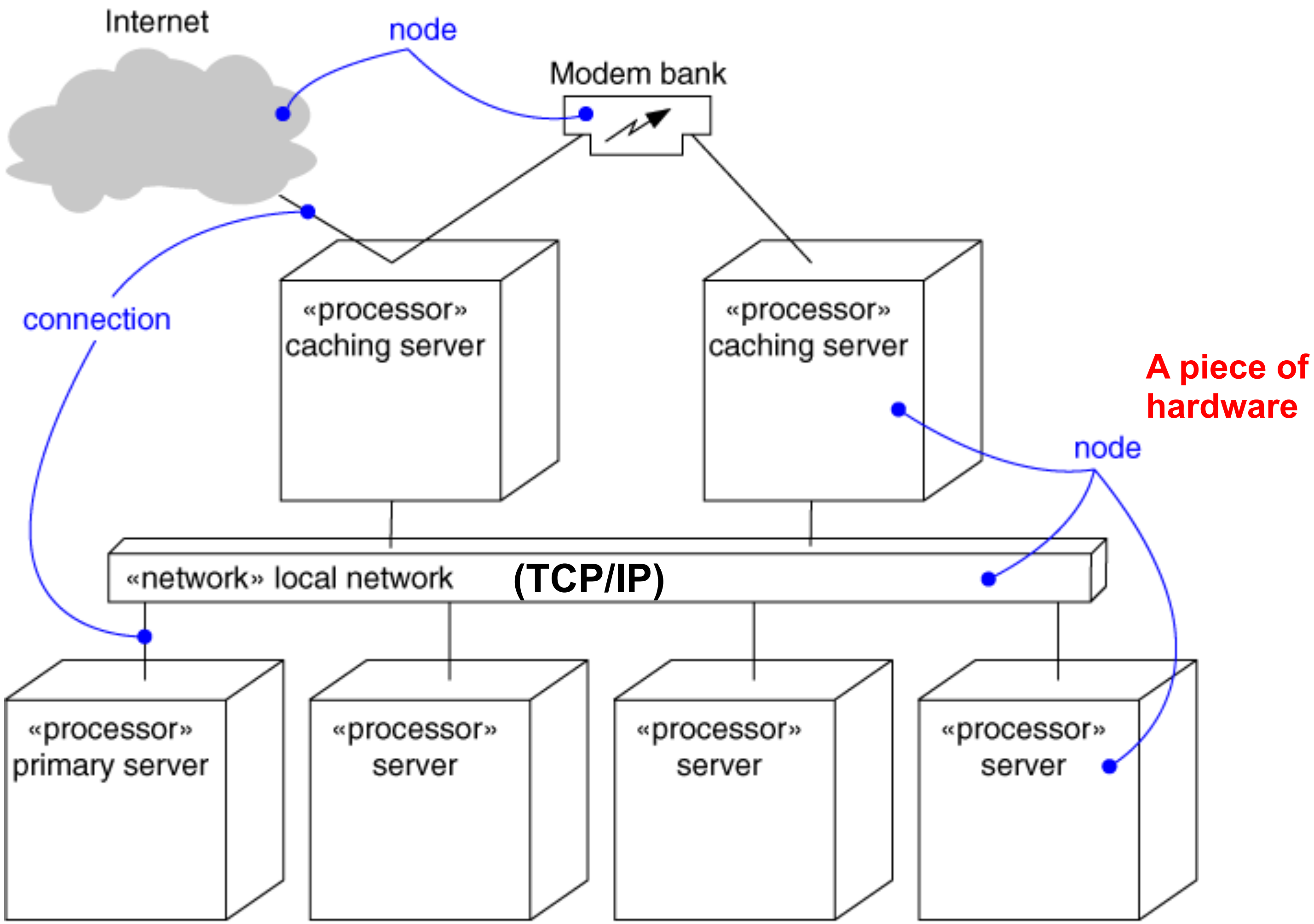




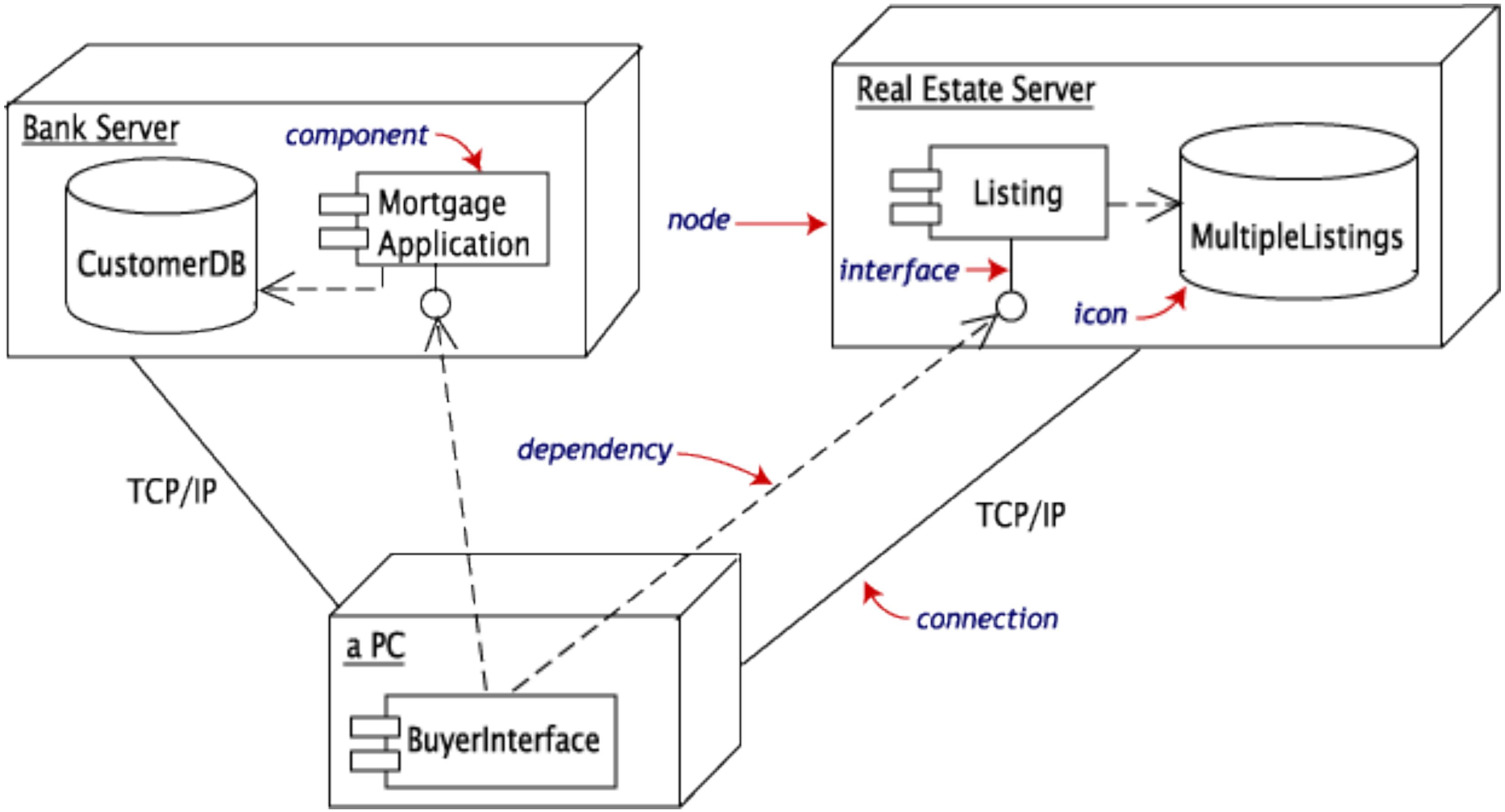


Deployment diagram

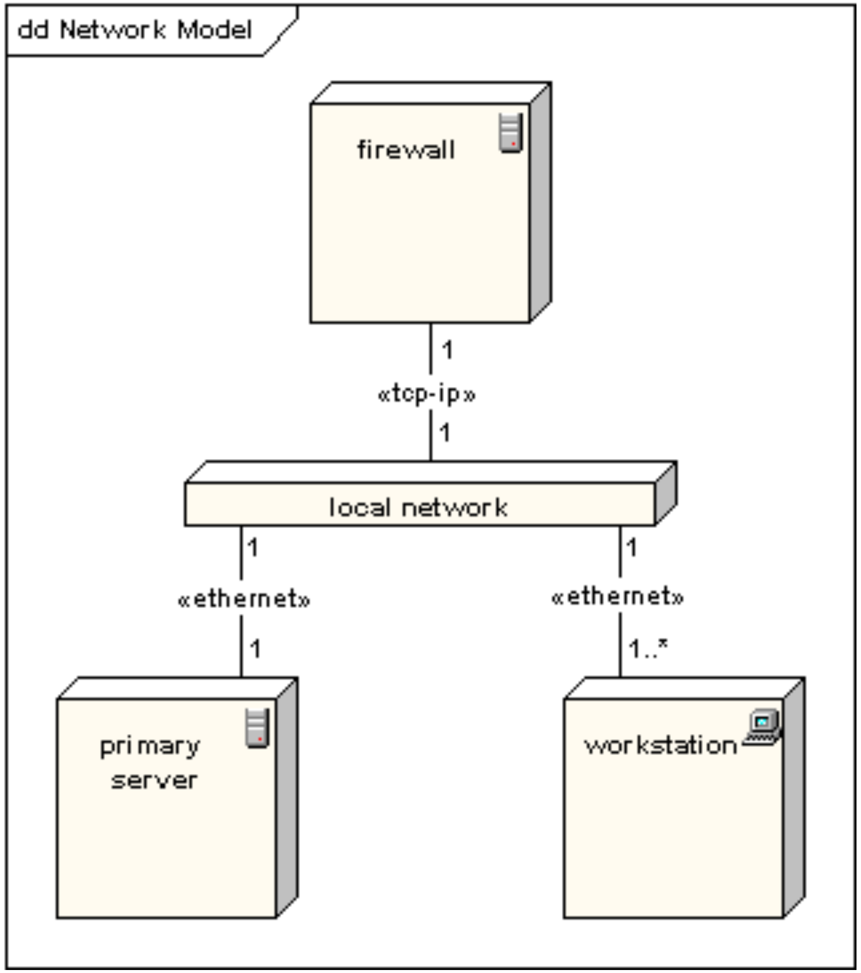
- Captures the topology of a system

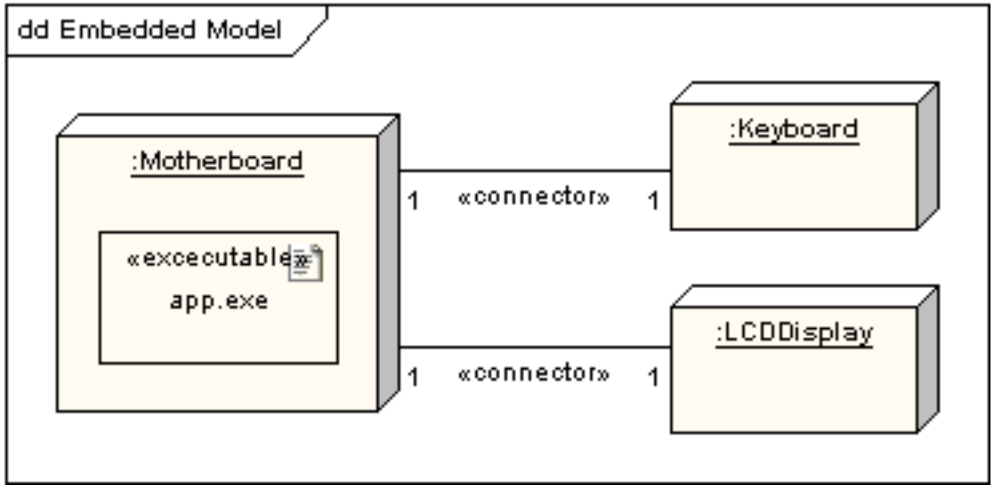


Deployment diagram with components

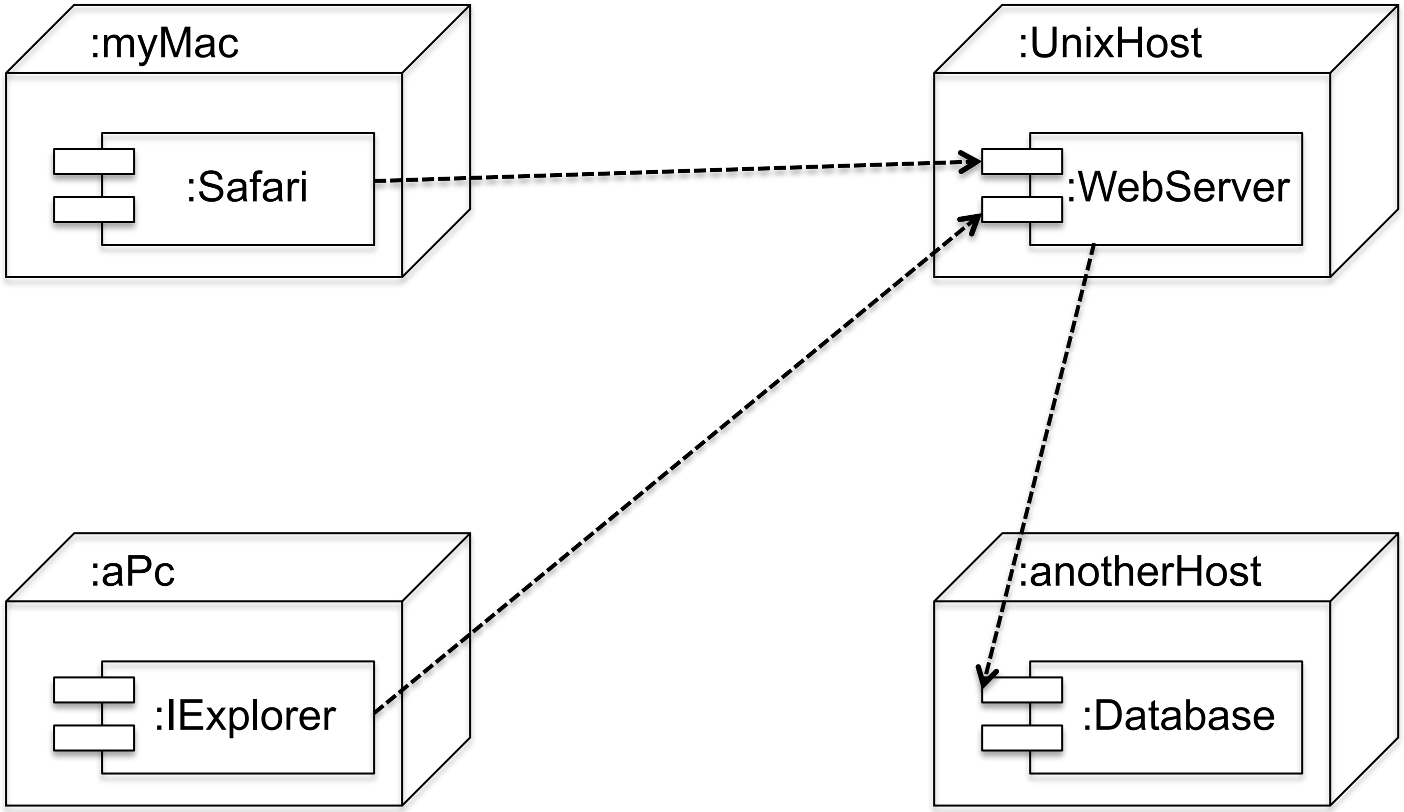


Deployment diagrams

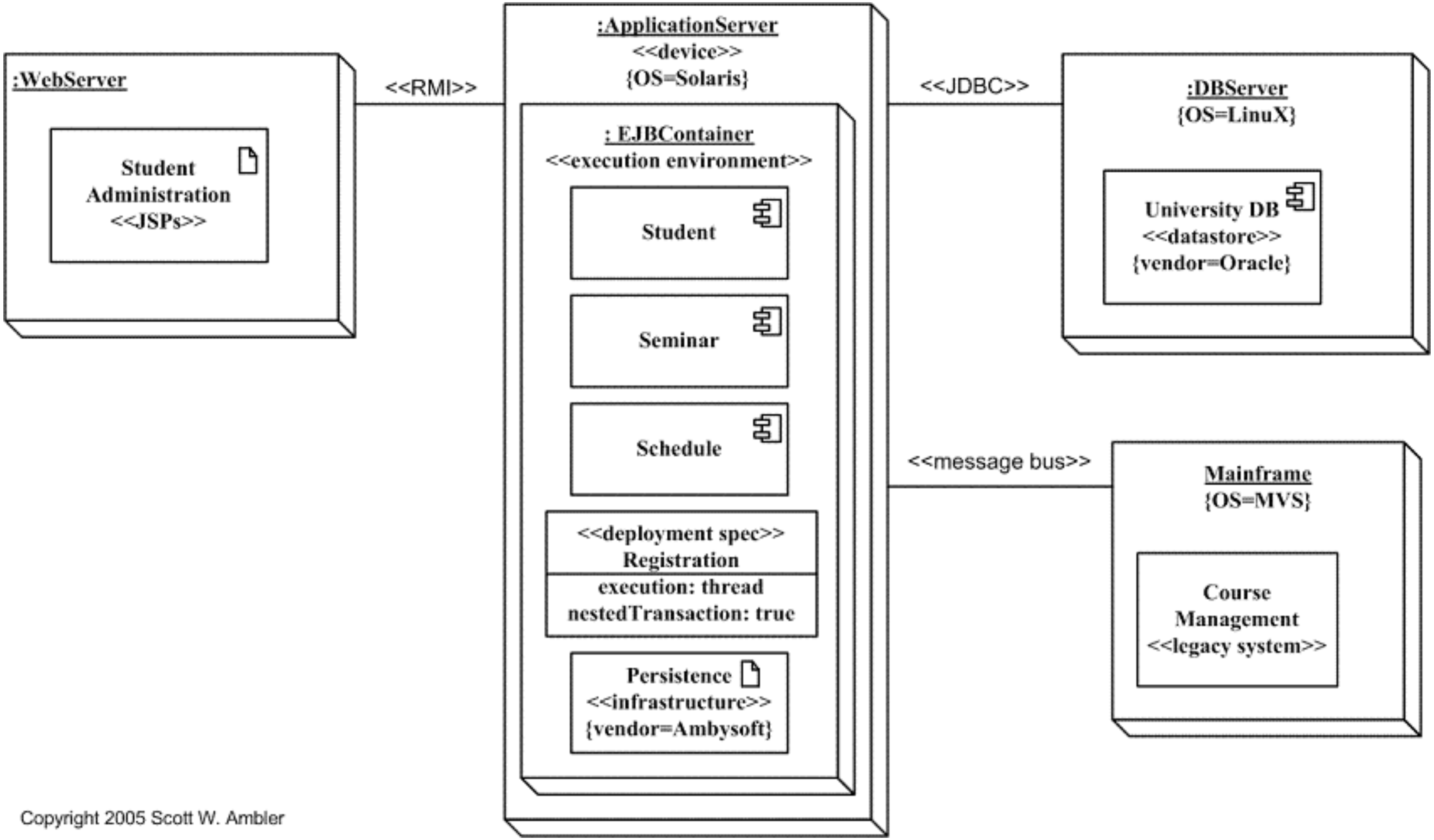




Example


:myMac
:Safari
:UnixHost
:WebServer
:aPc
:IExplorer
:anotherHost
:Database

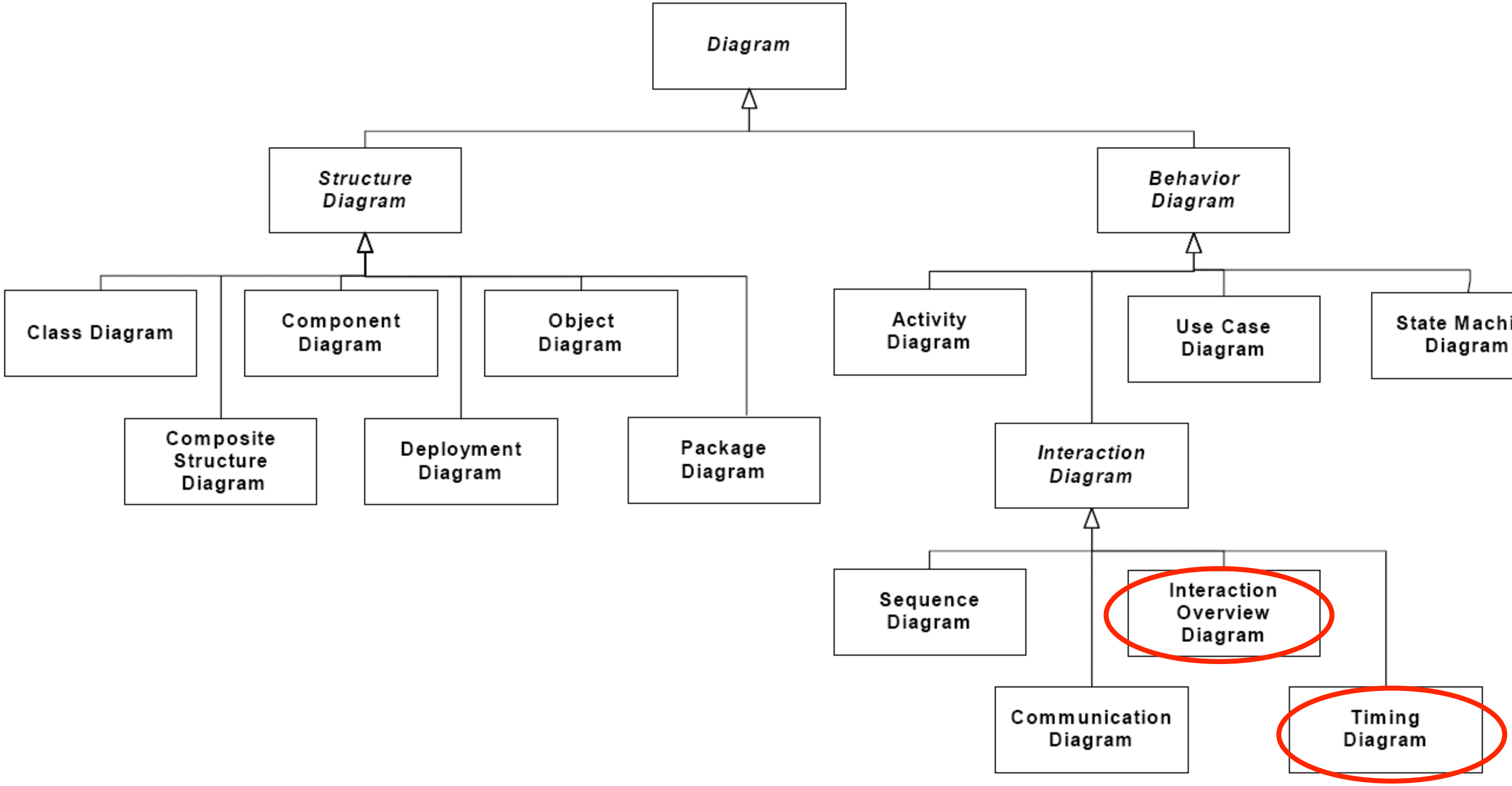
Example


:WebServer
Student
Administration
<<JSPs>>
<<RMI>>
:ApplicationServer
<<device>>
{OS=Solaris}
: EJBContainer
<<execution environment>>
Student
Seminar
Schedule
<<deployment spec>>
Registration
execution: thread
nestedTransaction: true
Persistence
<<infrastructure>>
{vendor=Ambysoft}
<<JDBC>>
:DBServer
{OS=LinuX}
University DB
<<datastore>>
{vendor=Oracle}
<<message bus>>
Mainframe
{OS=MVS}
Course
Management
<<legacy system>>

Exercise

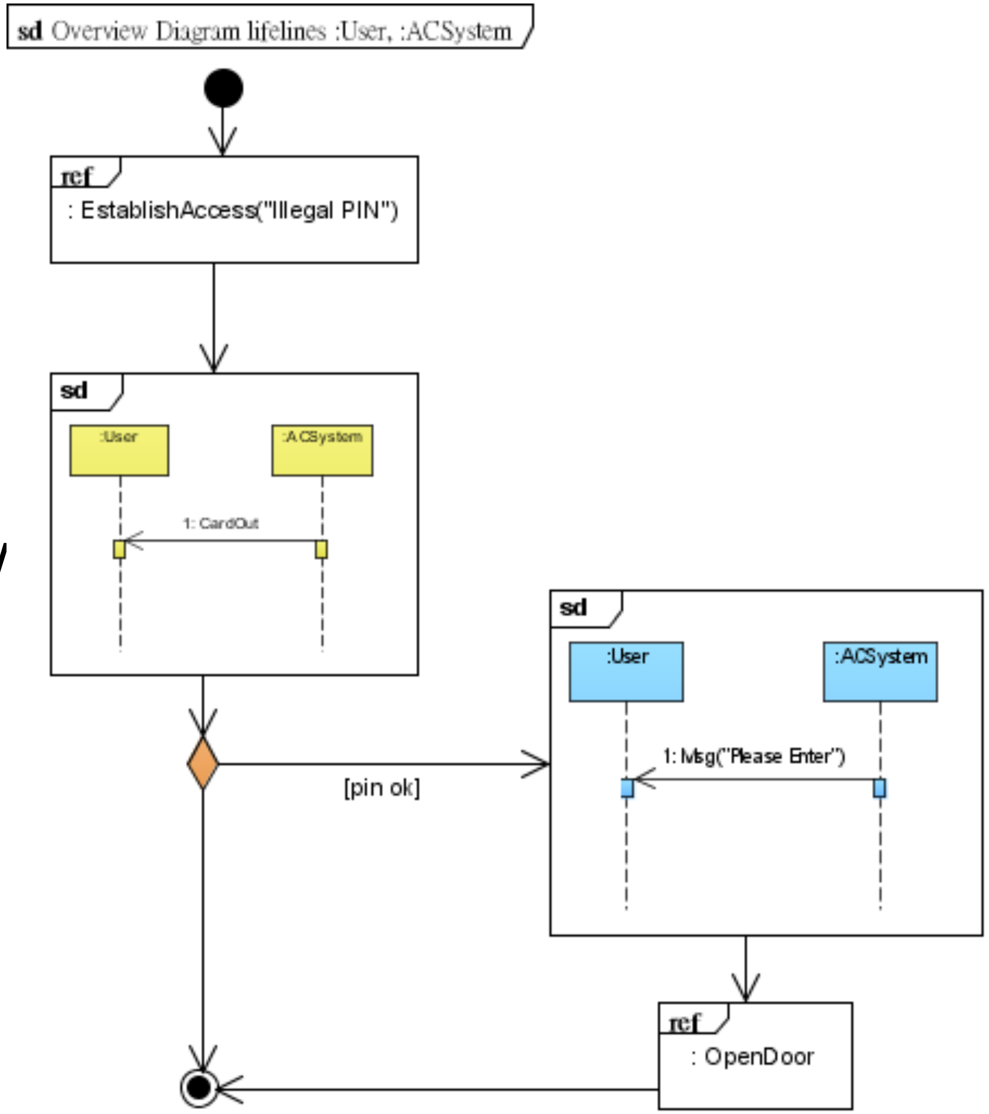
Describe the deployment structure of a system you know

UML2: novel behavior diagrams



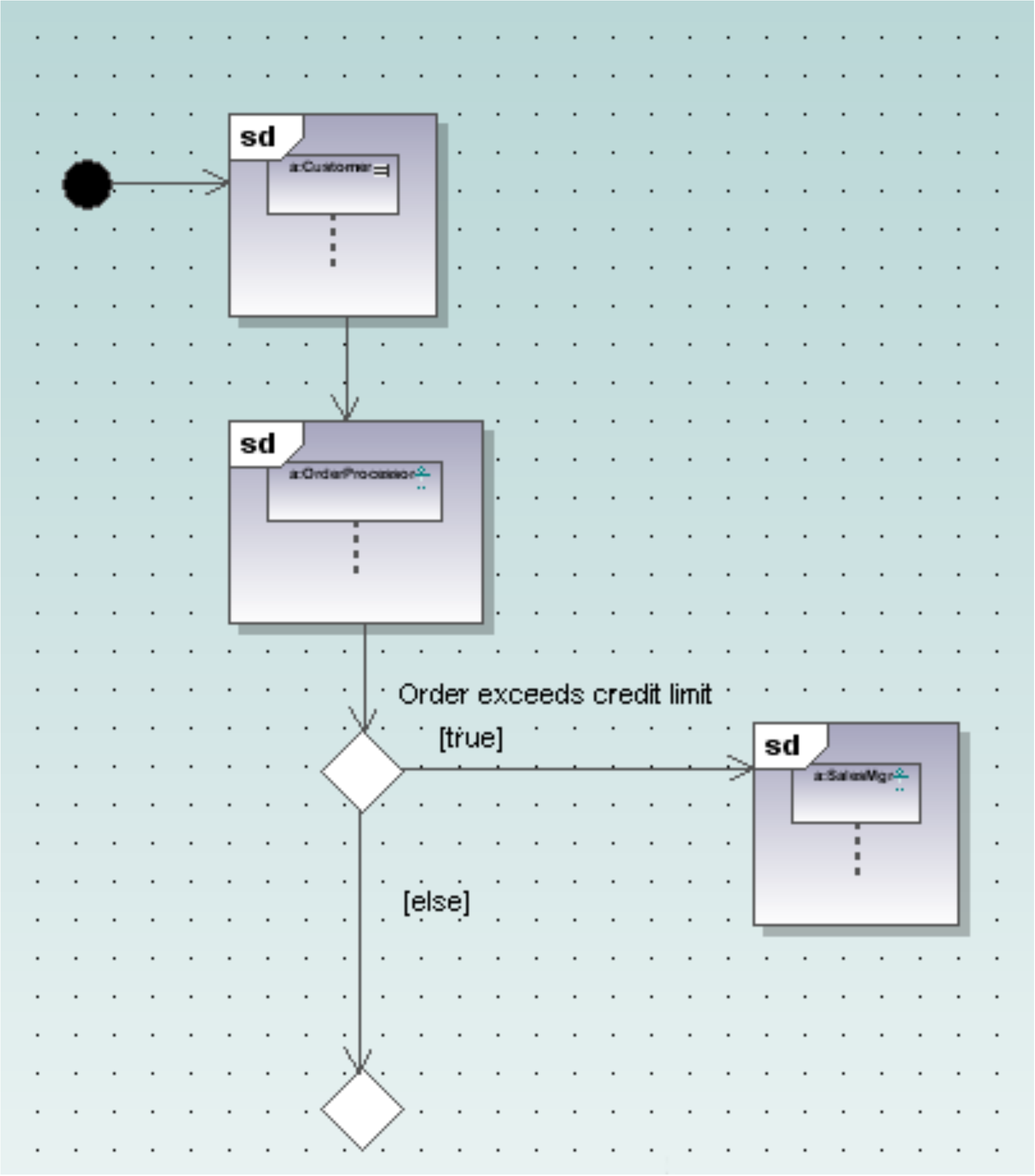
Interaction overview diagram

- A variant of activity diagrams
- The Interaction Overview diagram focuses on the flow of control of the interactions
- Visualizes the cooperation among other interaction diagrams to illustrate a control flow serving some purpose

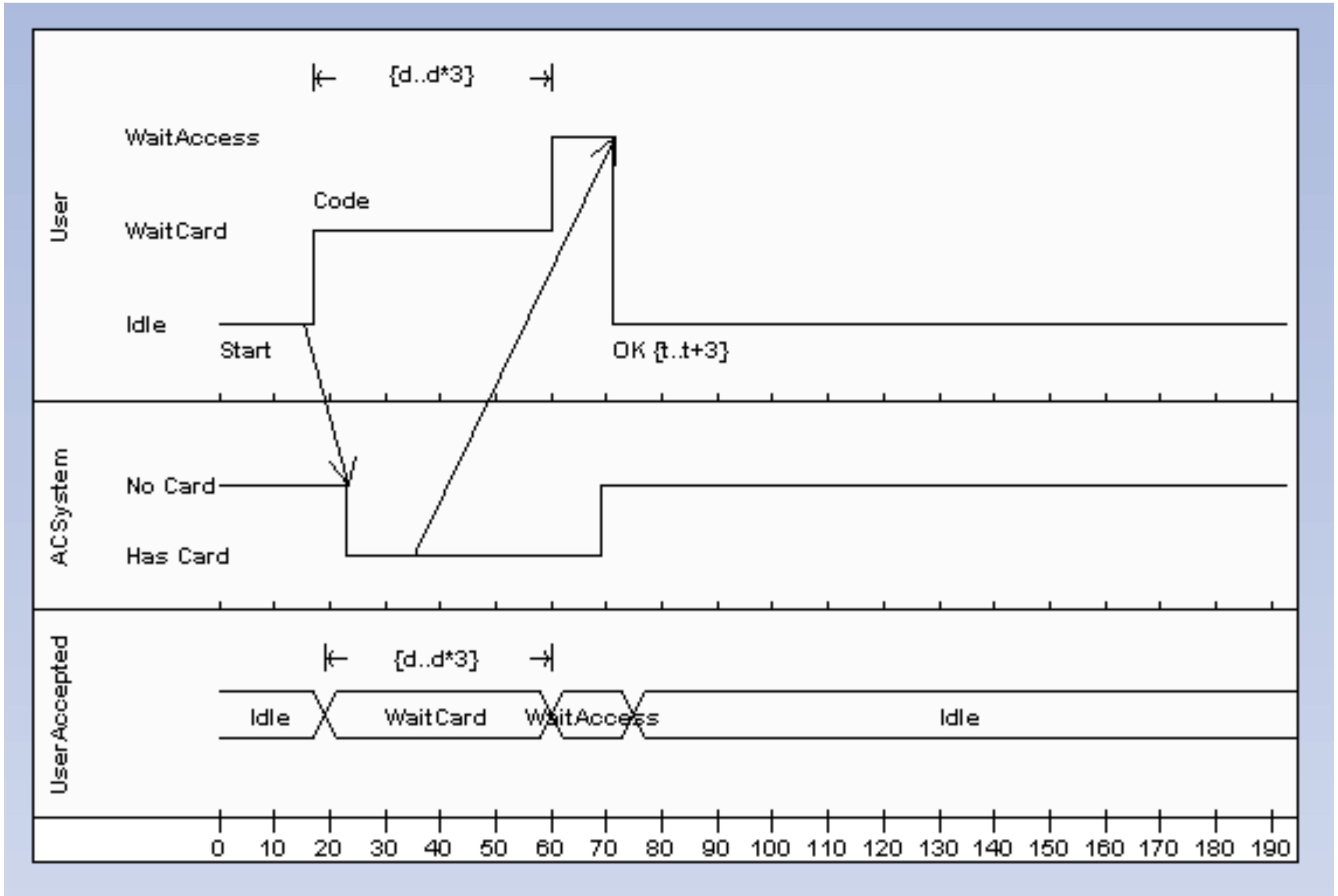


Interaction overview

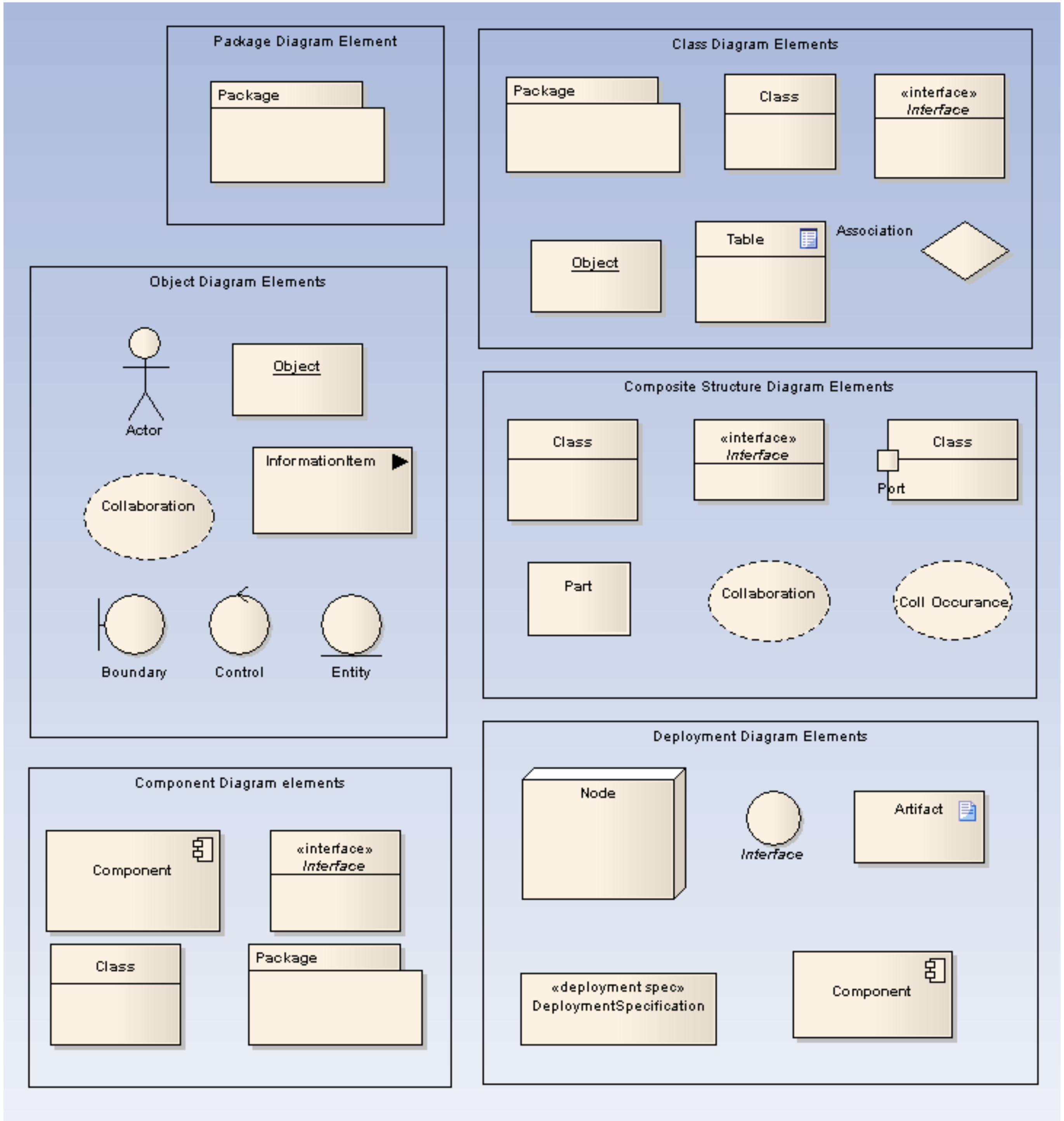
Use an interaction overview diagram to deconstruct a complex scenario that would otherwise require multiple if-then-else paths to be illustrated as a single sequence diagram



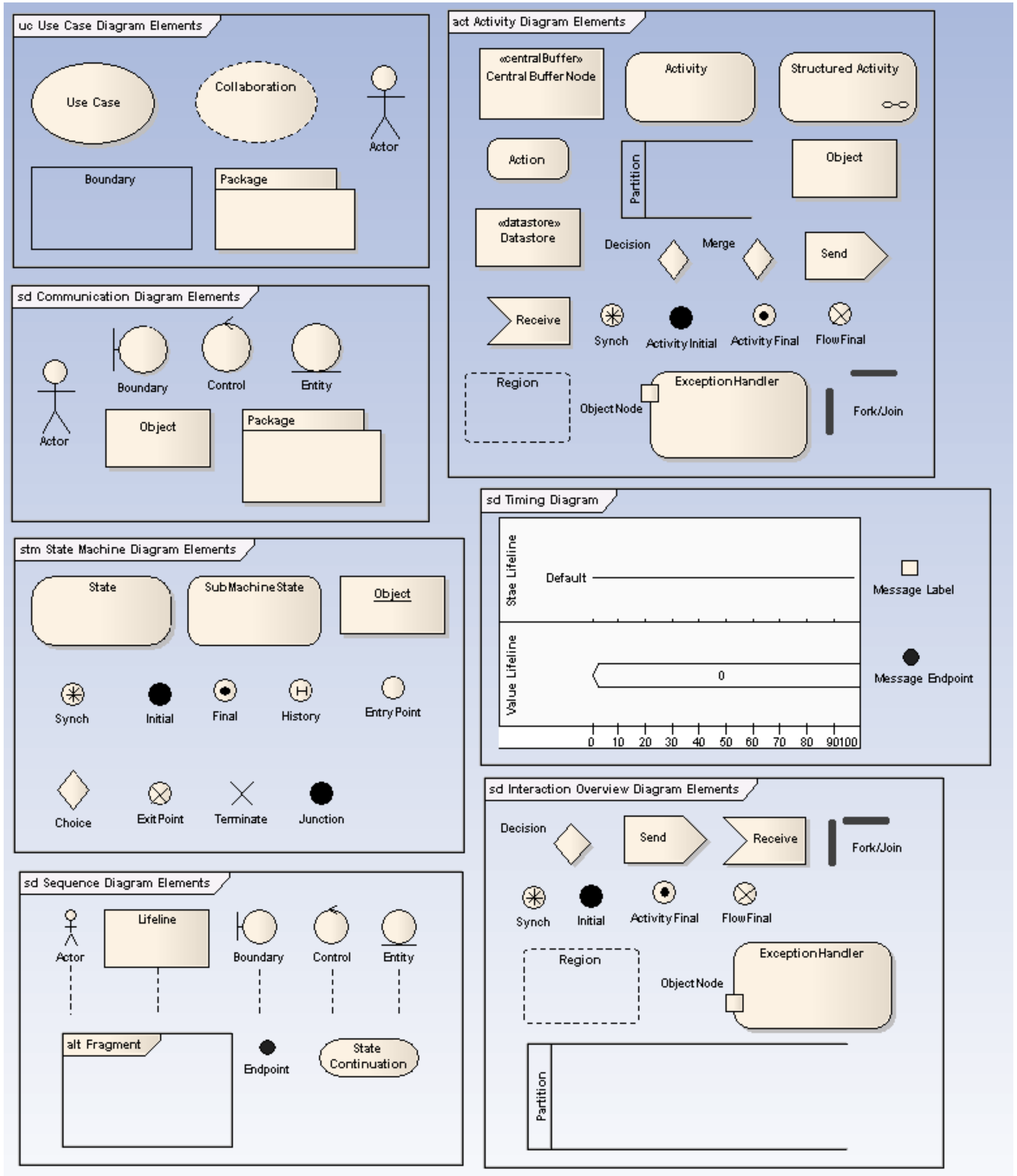
Timing diagram



Structural diagrams: summary

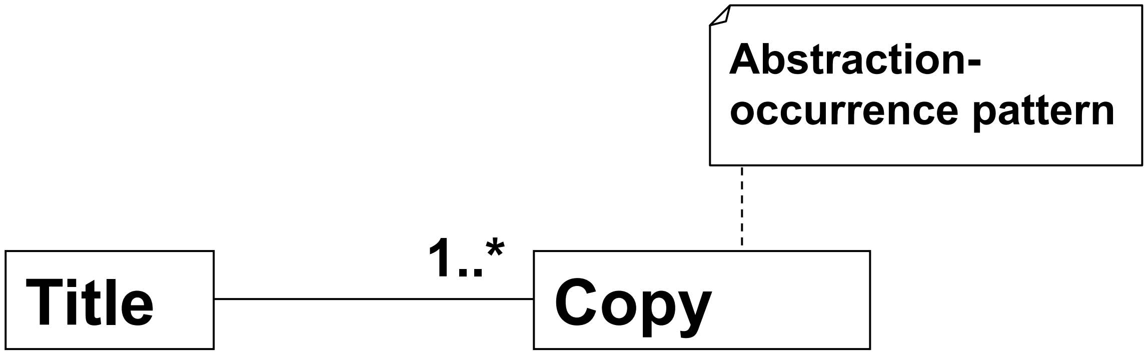


Behavioral diagrams: summary



Comments

- Comments are used to clarify the models
  - e.g. comments may be used for explaining the rationale behind some design decisions
- A comment is shown as a text string within a note icon
- (A note icon can also contain an OCL expression)

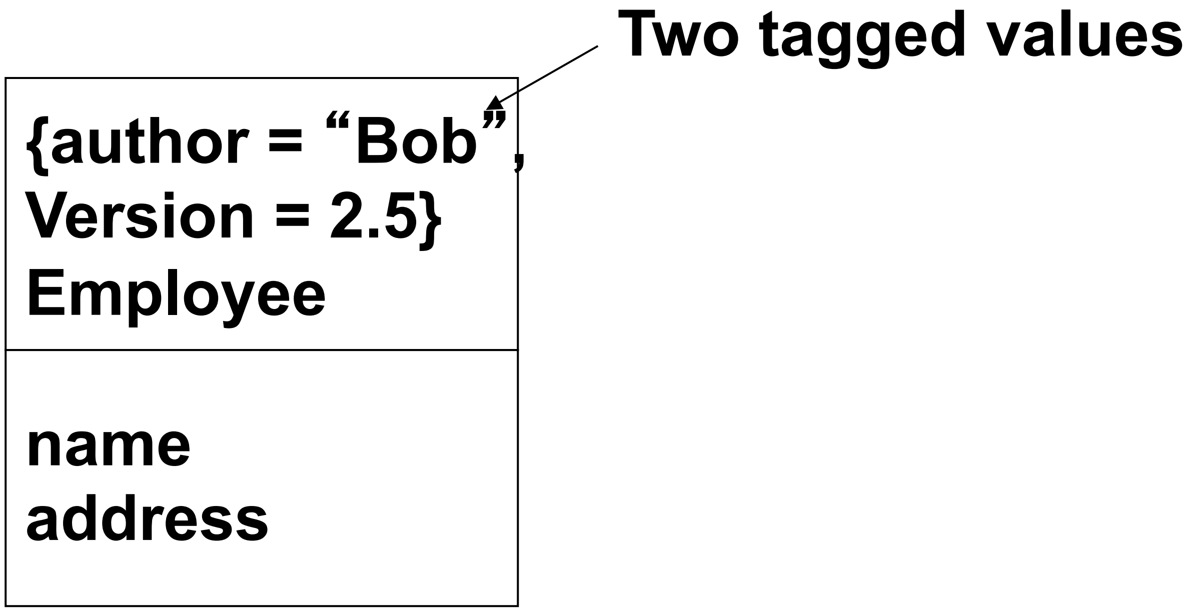


Tagged Values

- Tagged values
  - Define additional properties for any kind of model element
  - Can be defined for existing model elements and for stereotypes
  - Are shown as a tag-value pair, where the tag represents the property and the value represents the value of the property
- Tagged values can be useful for adding properties about
  - code generation
  - version control
  - configuration management
  - authorship
  - etc.

Tagged Value example

A tagged value is a string enclosed by brackets {} and which consists of the tag, a separator (the symbol =), and a value



Two tagged values

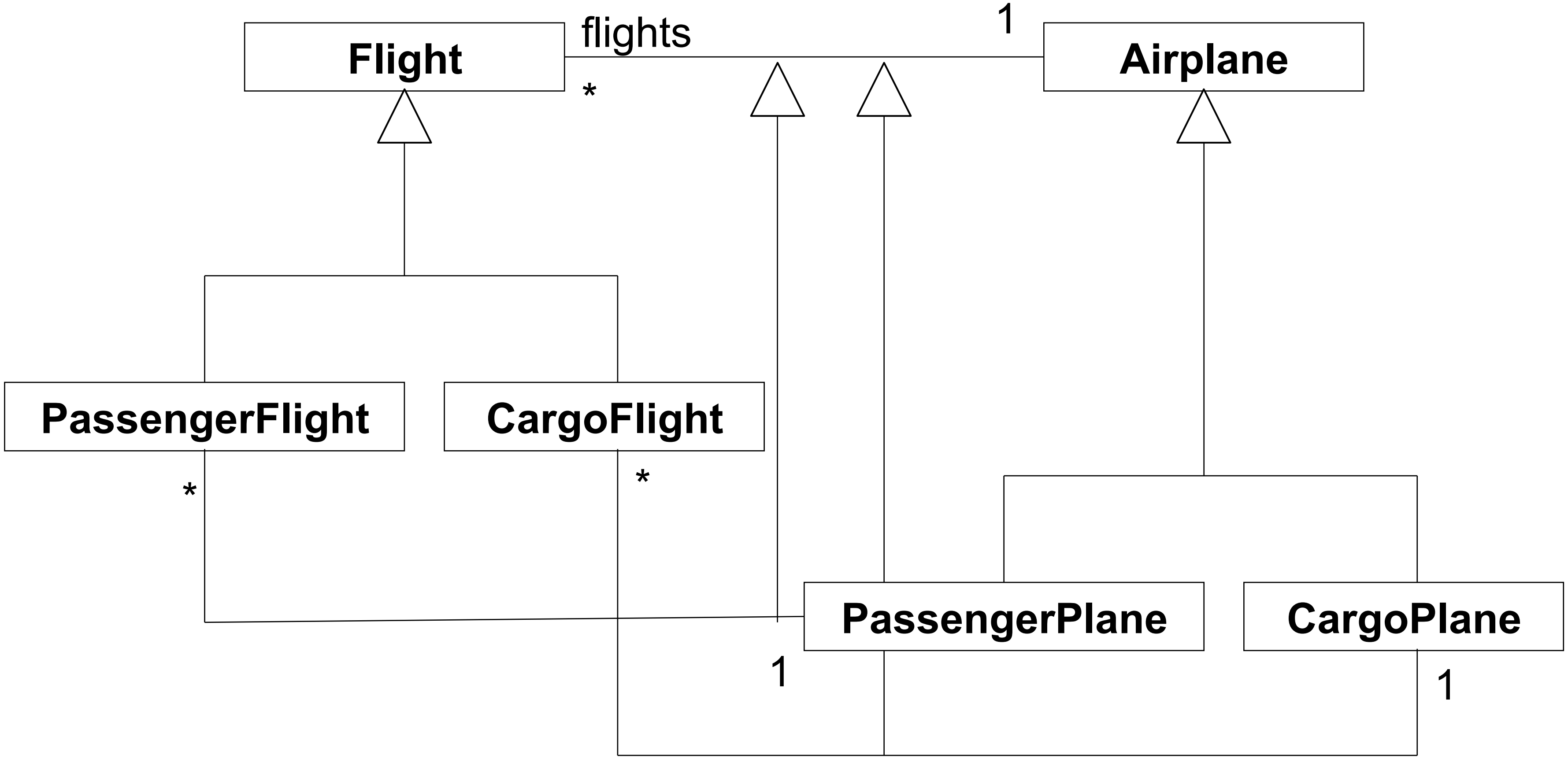
{author = "Bob", Version = 2.5}
Employee

name
address

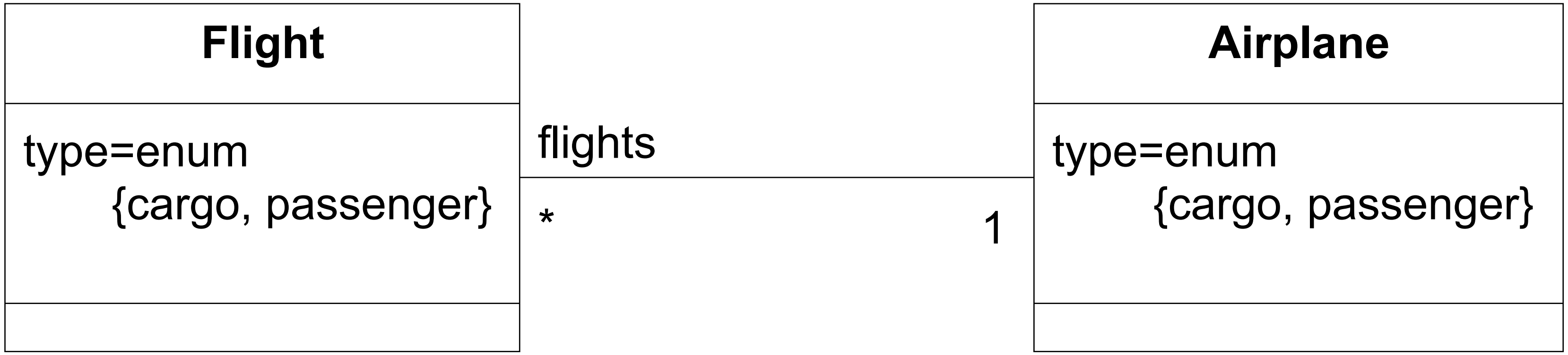
Constraints

- Constraints can be used to specify conditions that must be held true at all times for the elements of a model
- In models that depict software systems, constraints represent conditions or restrictions difficult to model using “normal” constructs
- In models that depict time-critical software systems, constraints provide a statement about the relative or absolute value of time during an interaction
- The constraints in a model can appear in any type of UML diagram

Without constraints



Using constraints



context Flight
inv: type = #cargo implies airplane.type= #cargo
inv: type = #passenger implies airplane.type= #passenger

Object Constraint Language (OCL)

- The OCL is a declarative language to write logical formulas which denote constraints to apply to UML models
- OCL is a key component of the new OMG standard recommendation for transforming models: the **QVT specification**, a key component of MDA

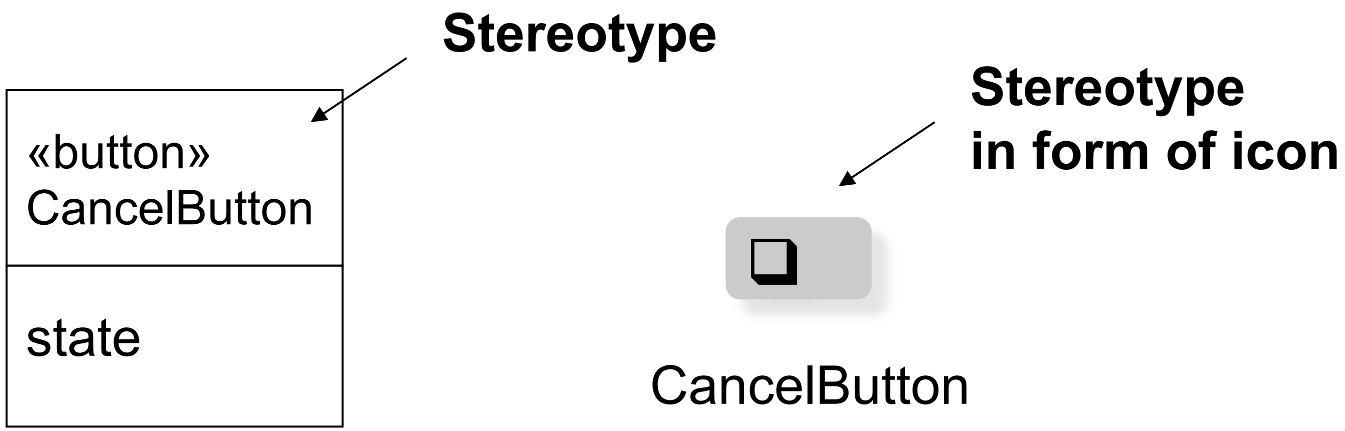
Stereotypes

- Stereotypes are used to extend UML to create new model elements that can be used in specific domains
- E.g. when modeling an elevator control system, we want represent some classes, states etc. as
  - «hardware»
  - «software»

Representing stereotypes

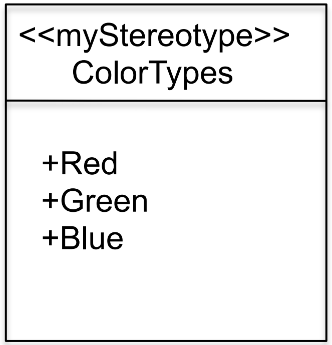
Two ways of representing a stereotype:

- Place the name of the stereotype above the name of an existing UML element using *guillemets* «» (e.g. «node»)
- Create new icons



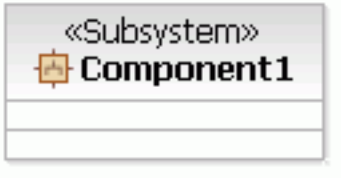
Extensibility of UML

- Stereotypes are used to extend the basic UML notational elements
- Stereotypes may be used to classify and extend associations, inheritance relationships, classes, and components
- Examples:
  - Class stereotypes:
    <<boundary>> and <<exception>>
  - Inheritance stereotypes:
    <<uses>> and <<extends>>
  - Component stereotypes: <<subsystem>>



<<myStereotype>>
ColorTypes

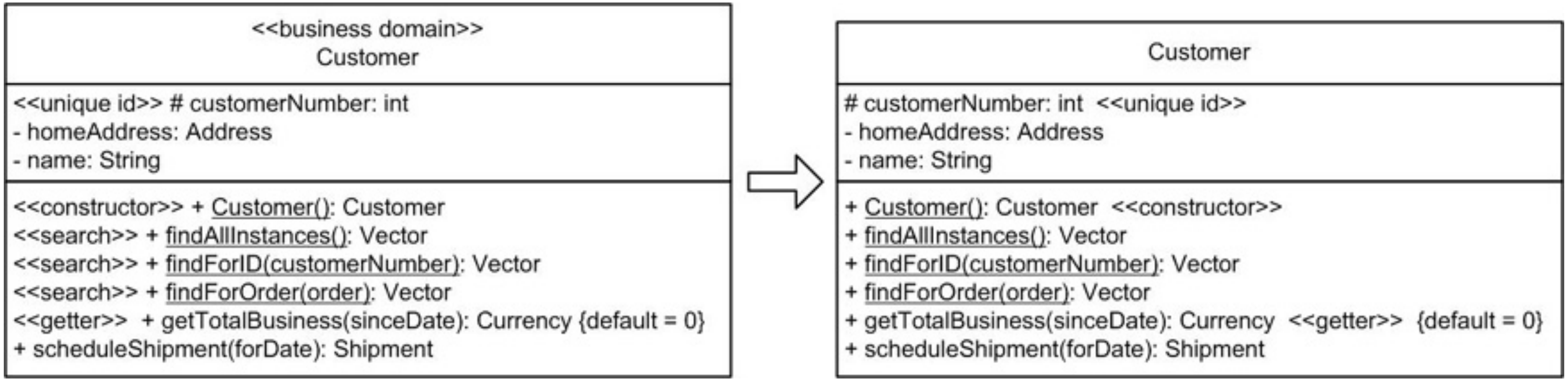
+Red
+Green
+Blue



«Subsystem»
Component1

Stereotype

- A stereotype denotes a variation on an existing modeling element with the same form but with a modified intent



<<business domain>>
Customer

<<unique id>> # customerNumber: int
- homeAddress: Address
- name: String

<<constructor>> + Customer(): Customer
<<search>> + findAllInstances(): Vector
<<search>> + findForID(customerNumber): Vector
<<search>> + findForOrder(order): Vector
<<getter>> + getTotalBusiness(sinceDate): Currency {default = 0}
+ scheduleShipment(forDate): Shipment

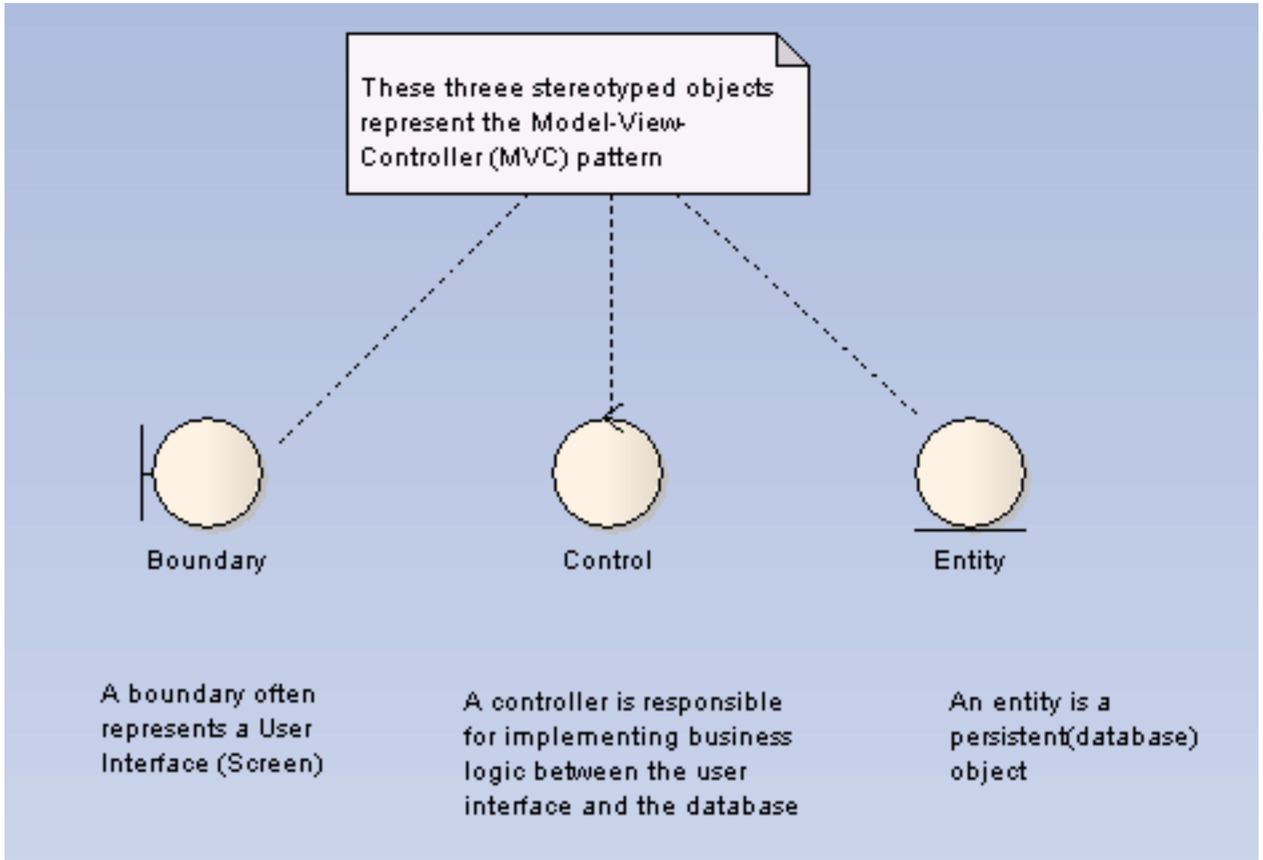
Customer

customerNumber: int <<unique id>>
- homeAddress: Address
- name: String

+ Customer(): Customer <<constructor>>
+ findAllInstances(): Vector
+ findForID(customerNumber): Vector
+ findForOrder(order): Vector
+ getTotalBusiness(sinceDate): Currency <<getter>> {default = 0}
+ scheduleShipment(forDate): Shipment

www.jeckle.de/files/UML1_2/apndxa.pdf

Notational stereotypes



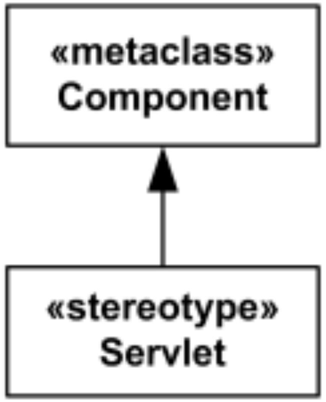
These threee stereotyped objects represent the Model-View-Controller (MVC) pattern

A boundary often represents a User Interface (Screen)

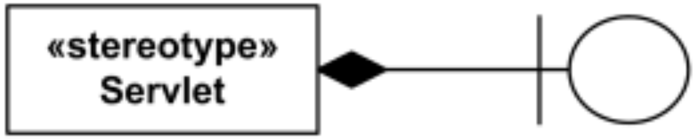
A controller is responsible for implementing business logic between the user interface and the database

An entity is a persistent(database) object

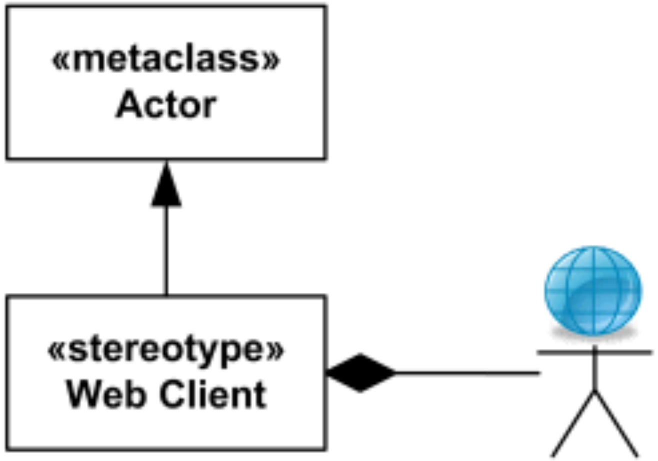
Examples



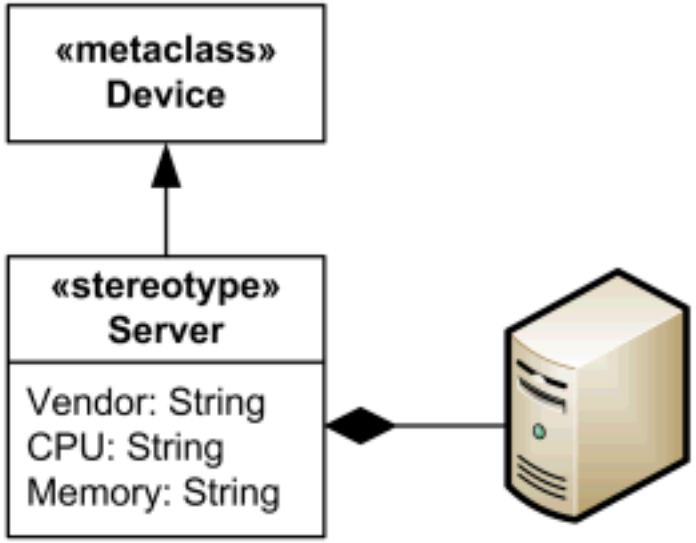
Stereotype Servlet
extends Component



Servlet stereotype with
attached custom icon

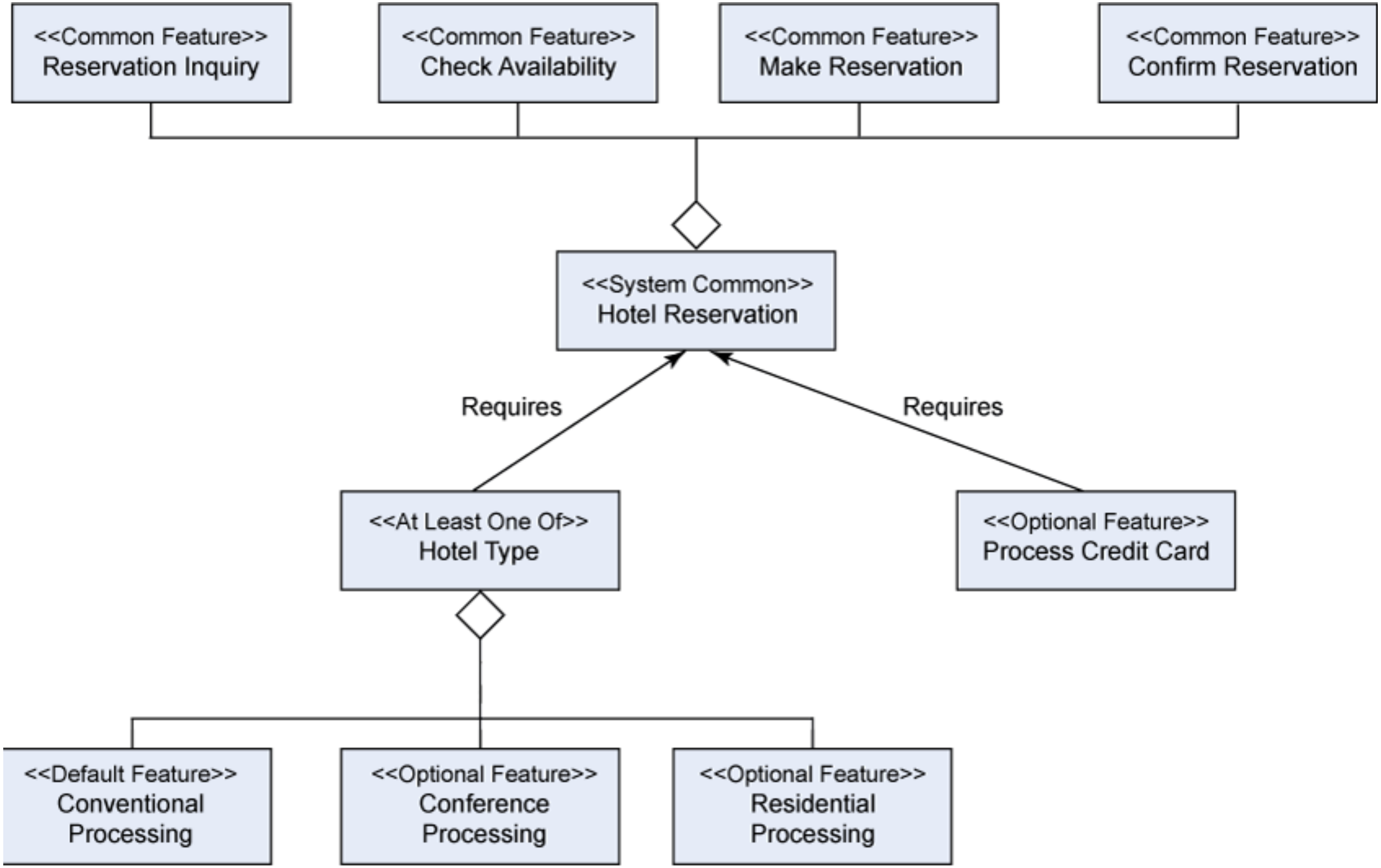


Actor is extended by
stereotype Web Client with
attached custom icon

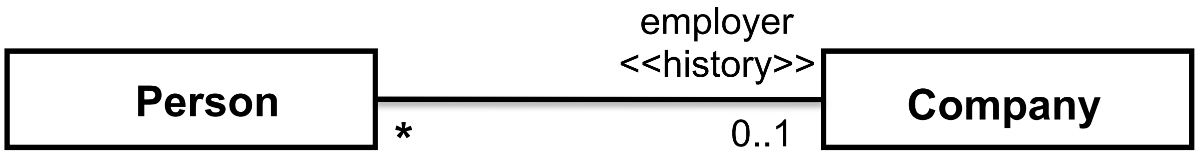


Device extended by Server
stereotype with tag definitions
and custom icon

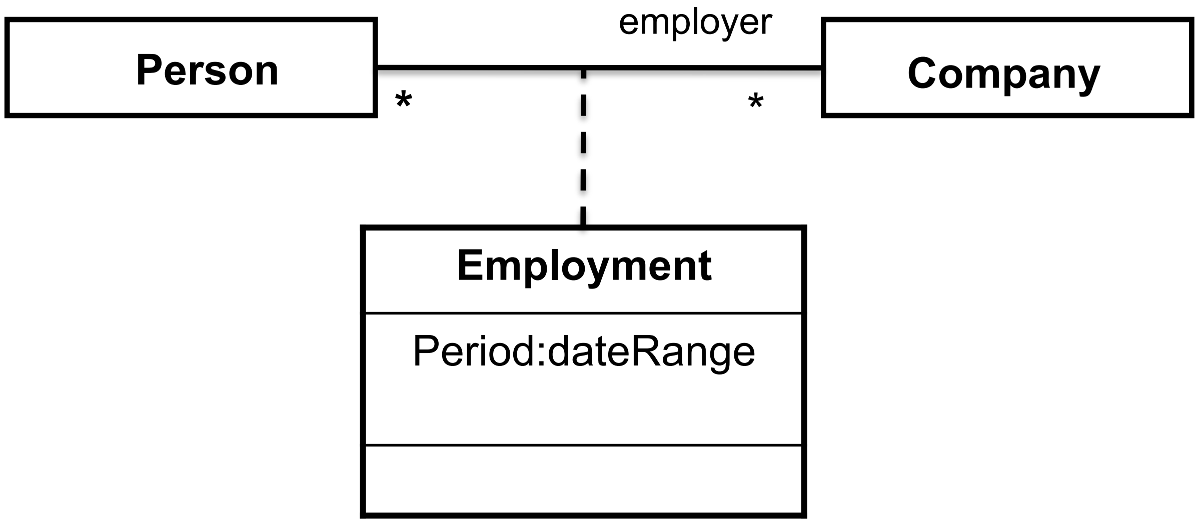
Example


<<Common Feature>>
Reservation Inquiry
<<Common Feature>>
Check Availability
<<Common Feature>>
Make Reservation
<<Common Feature>>
Confirm Reservation
<<System Common>>
Hotel Reservation
Requires
Requires
<<At Least One Of>>
Hotel Type
<<Optional Feature>>
Process Credit Card
<<Default Feature>>
Conventional Processing
<<Optional Feature>>
Conference Processing
<<Optional Feature>>
Residential Processing

Example: <<history>> stereotype [Fowler]



- The model says that a Person may work for only a single Company at one time. Over time, however, a Person may work for several Companies. The stereotype captures the following situation:



Common stereotypes

Stereotype	denotes
<<application>>	A front-end of a system
<<database>>	A database
<<document>>	A printed or digital document
<<executable>>	A component executable on a node
<<file>>	Data file
<<infrastructure>>	Technical component, eg. An audit logger
<<library>>	Object or function library
<<source code>>	Text file to be compiled, like .java or .cpp
<<table>>	Table within a database
<<web service>>	One or more web services
<<XML DTD>>	XML Document Type Definition

Color UML

A special set of stereotypes defines

Color UML

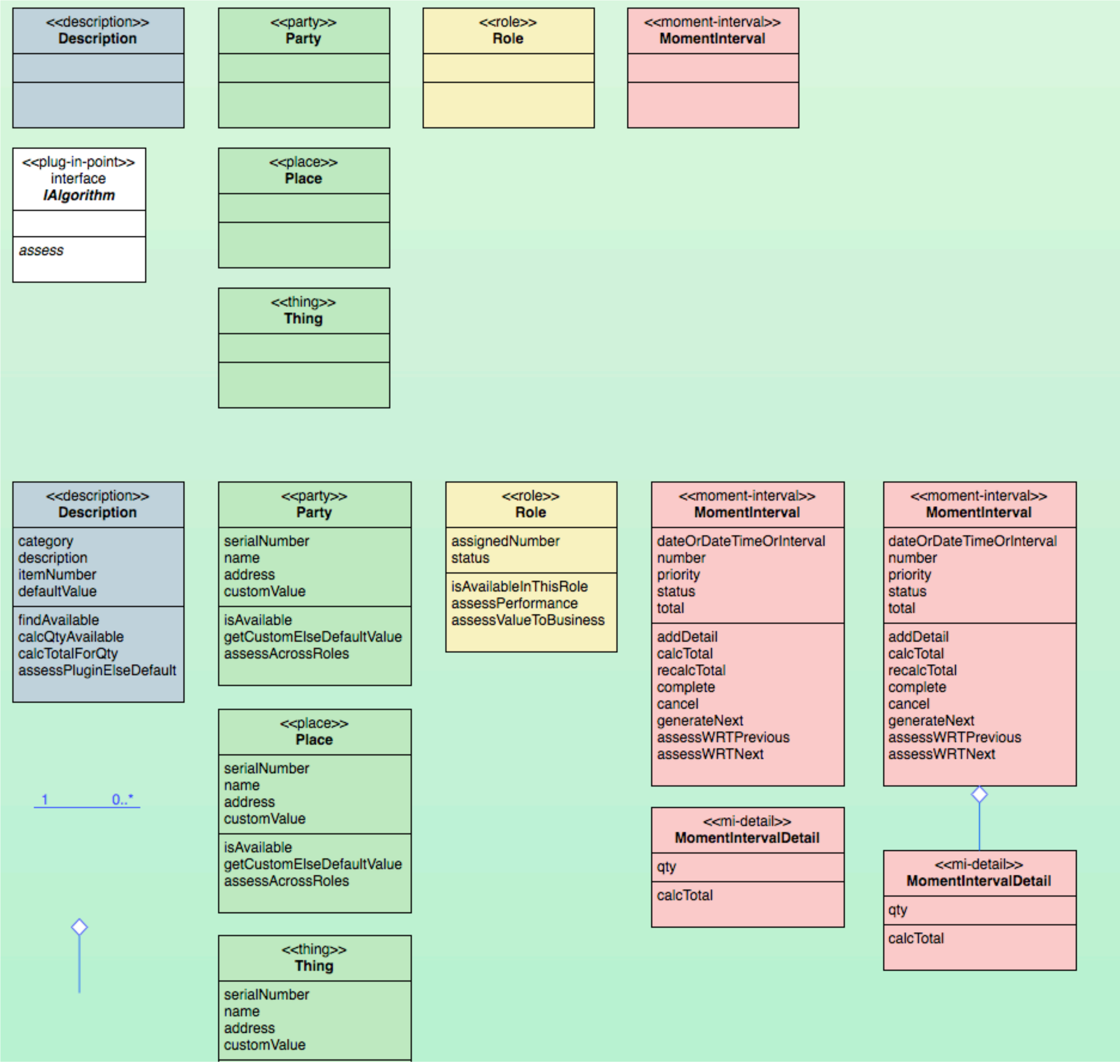
Green: things

Pink: time

Yellow: roles

Blue: description

en.wikipedia.org/wiki/UML_colors



<<description>>
Description

<<party>>
Party

<<role>>
Role

<<moment-interval>>
MomentInterval

<<plug-in-point>>
interface
IAlgorithm

assess

<<place>>
Place

<<thing>>
Thing

<<description>>
Description

category
description
itemNumber
defaultValue

findAvailable
calcQtyAvailable
calcTotalForQty
assessPluginElseDefault

<<party>>
Party

serialNumber
name
address
customValue

isAvailable
getCustomElseDefaultValue
assessAcrossRoles

<<role>>
Role

assignedNumber
status

isAvailableInThisRole
assessPerformance
assessValueToBusiness

<<moment-interval>>
MomentInterval

dateOrDateTimeOrInterval
number
priority
status
total

addDetail
calcTotal
recalcTotal
complete
cancel
generateNext
assessWRTPrevious
assessWRTNext

<<moment-interval>>
MomentInterval

dateOrDateTimeOrInterval
number
priority
status
total

addDetail
calcTotal
recalcTotal
complete
cancel
generateNext
assessWRTPrevious
assessWRTNext

<<place>>
Place

serialNumber
name
address
customValue

isAvailable
getCustomElseDefaultValue
assessAcrossRoles

<<mi-detail>>
MomentIntervalDetail

qty

calcTotal

<<mi-detail>>
MomentIntervalDetail

qty

calcTotal

<<thing>>
Thing

serialNumber
name
address
customValue

Extending UML with profiles

Define a profile:

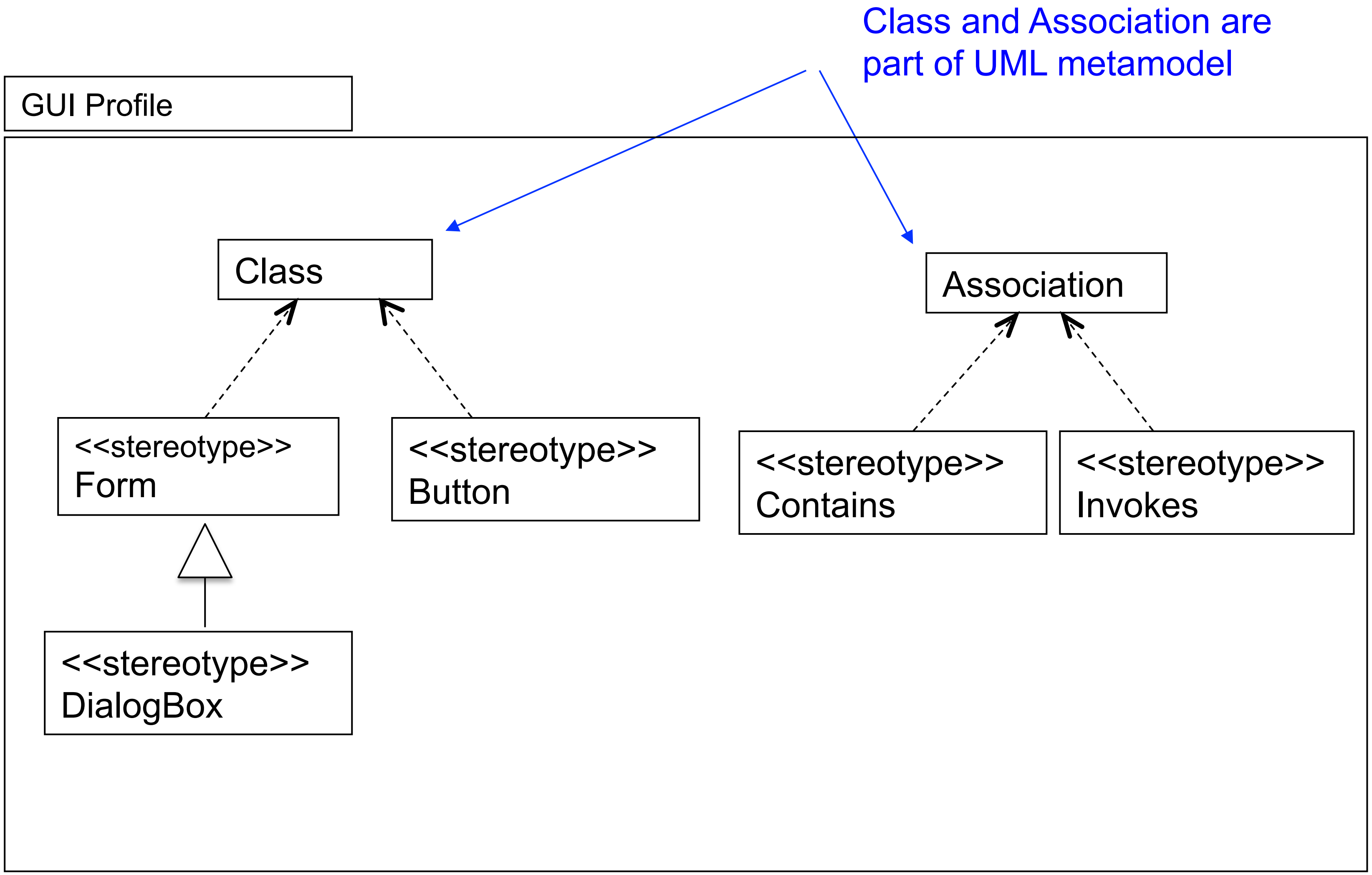
- Exploit existing constructs
  - Eg. component, connector, subsystem, etc.
- Define stereotypes for new constructs to model structural aspects of architectures
- Describe the behavioral semantics of the new constructs using OCL and dynamic diagrams

Example of a profile

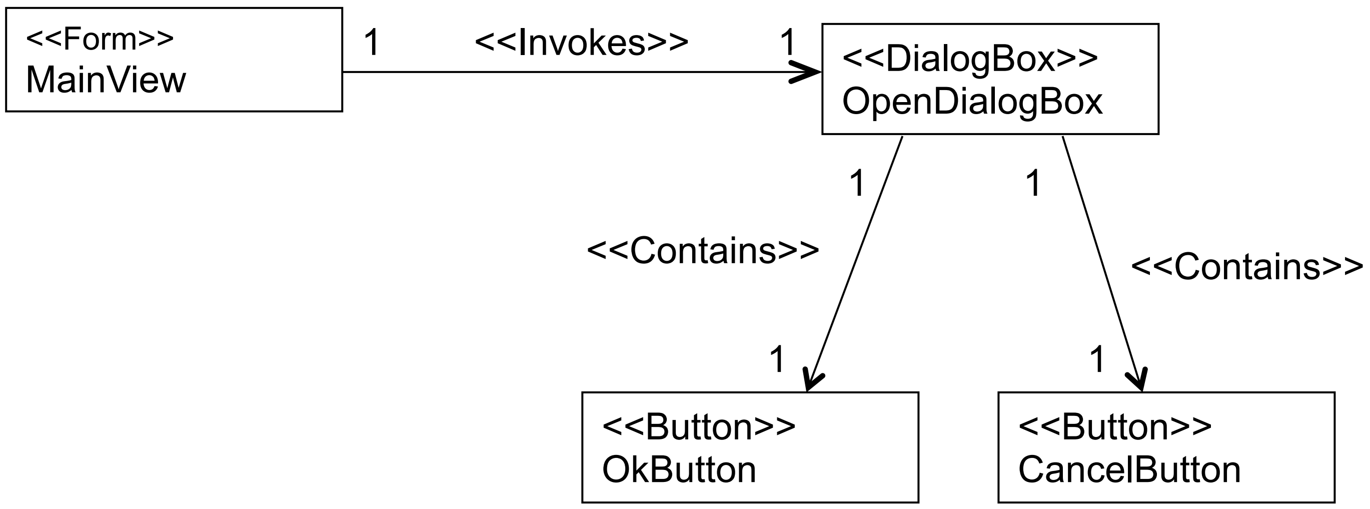
inspired by Cabot et al. (2003)

- Create a UML profile for representing basic GUI components
- The GUI contains the following components:
  - Forms (which can also be dialog boxes)
  - Buttons
- Constraints:
  - A form can invoke a dialog box
  - A form as well as a dialog box can contain buttons

The GUI profile package



An instance of the GUI Profile



<<Form>>
MainView

1 <<Invokes>> 1

<<DialogBox>>
OpenDialogBox

1 1

<<Contains>>

<<Contains>>

1 1

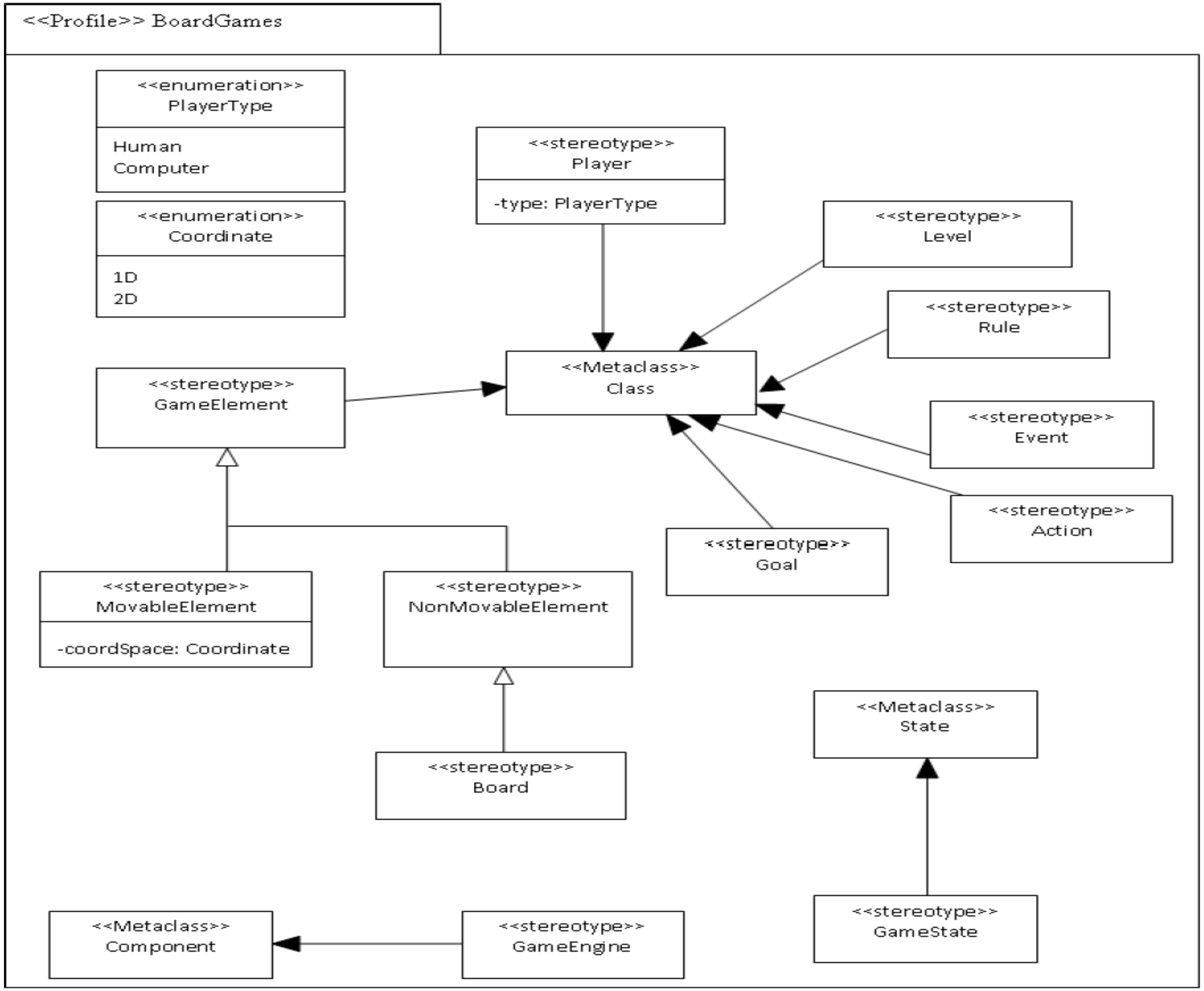
<<Button>>
OkButton

<<Button>>
CancelButton

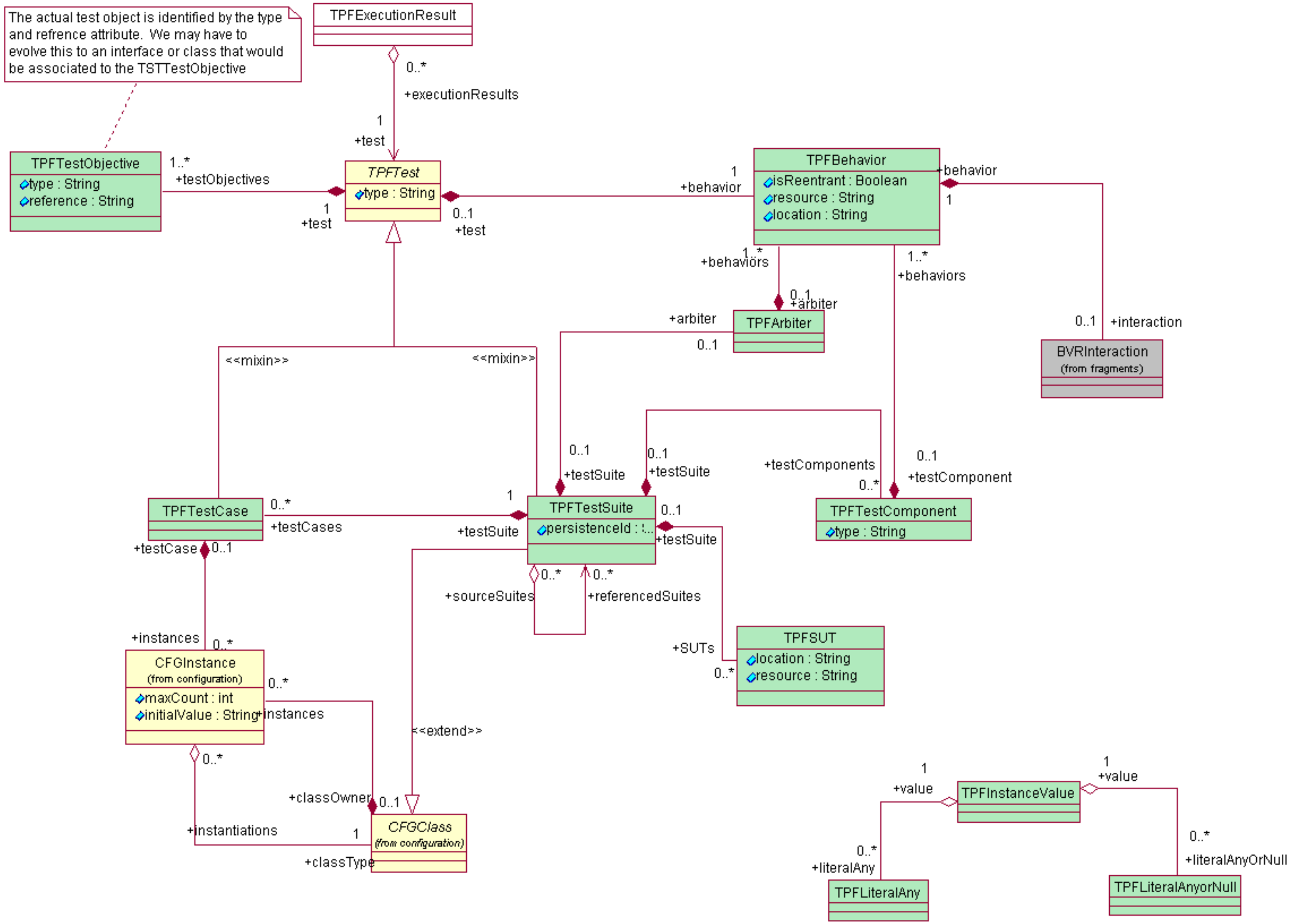
Profile diagrams

- A profile is a generic extension mechanism for customizing UML models for particular domains and platforms
- Profiles are extensions defined using stereotypes, tagged values, and constraints
- A **profile diagram** is a collection of such extensions specialized for a particular domain (eg. aerospace, healthcare, financial) or platform (eg. J2EE, .NET)

A profile for board games

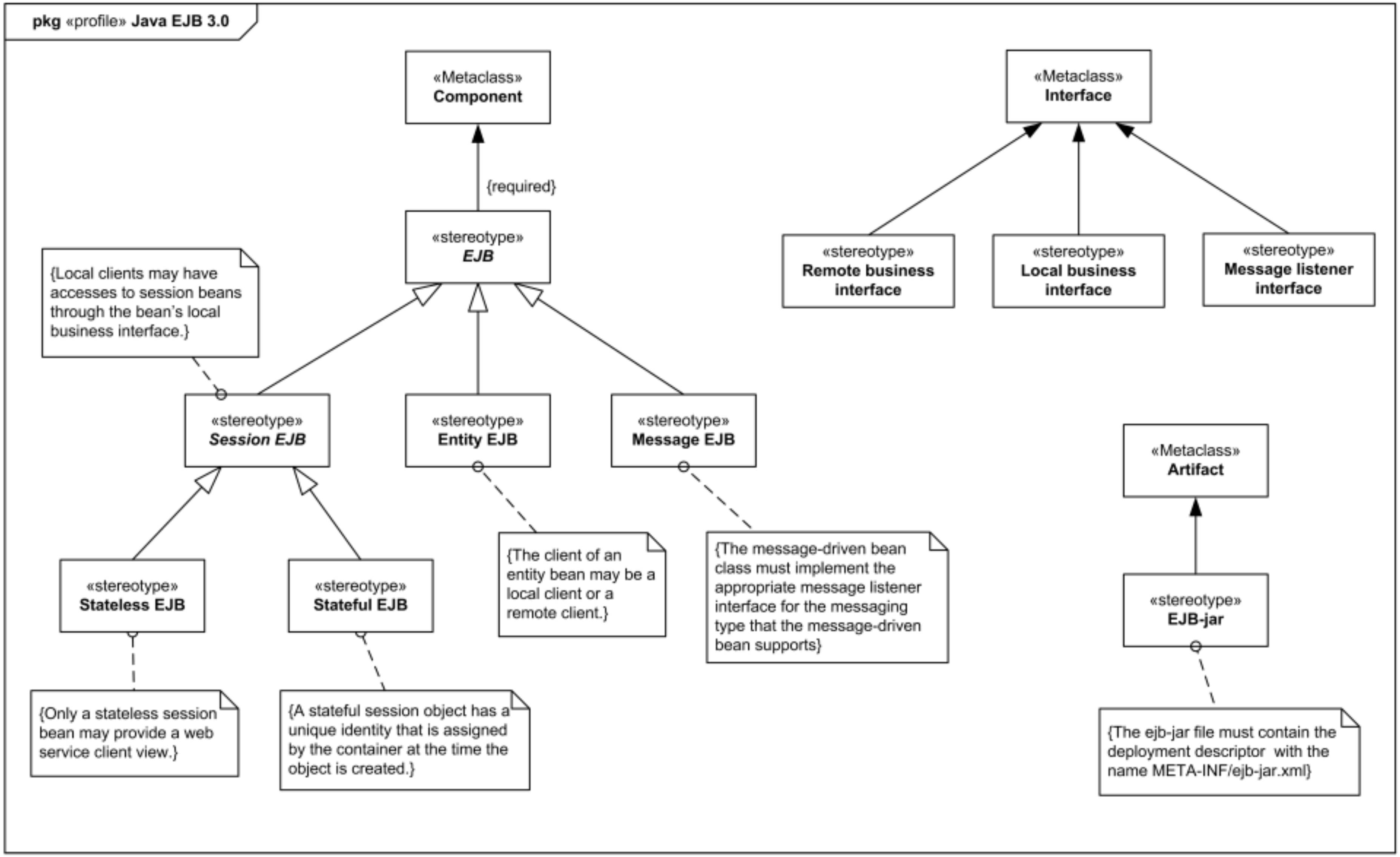


A testing profile



The actual test object is identified by the type and refrence attribute. We may have to evolve this to an interface or class that would be associated to the TSTTestObjective

EJB profile



pkg «profile» Java EJB 3.0

«Metaclass» **Component**

{required}

«stereotype» ***EJB***

{Local clients may have accesses to session beans through the bean's local business interface.}

«stereotype» ***Session EJB***

«stereotype» **Entity EJB**

«stereotype» **Message EJB**

«stereotype» **Stateless EJB**

«stereotype» **Stateful EJB**

{The client of an entity bean may be a local client or a remote client.}

{The message-driven bean class must implement the appropriate message listener interface for the messaging type that the message-driven bean supports}

{Only a stateless session bean may provide a web service client view.}

{A stateful session object has a unique identity that is assigned by the container at the time the object is created.}

«Metaclass» **Interface**

«stereotype» **Remote business interface**

«stereotype» **Local business interface**

«stereotype» **Message listener interface**

«Metaclass» **Artifact**

«stereotype» **EJB-jar**

{The ejb-jar file must contain the deployment descriptor with the name META-INF/ejb-jar.xml}

Some important profiles

- CORBA www.omg.org/technology/documents/formal/profile_corba.htm
- EDOC (profiles for Java and EJB) www.omg.org/technology/documents/formal/edoc.htm
- MARTE (Modeling and Analysis of Real-Time and Embedded Systems) www.omgmarte.org
- SysML (profile for system engineering with UML) www.sysml.org
- RUP profile for Business Modeling www.ibm.com/developerworks/rational/library/5167.html
- SOAML (UML profile for SOA) www.omg.org/spec/SoaML
- UPDM Unified Profile for DoDaf/MoDaf www.updm.com

Some profiles are “officially” endorsed by OMG

See the list of OMG profiles in www.omg.org/technology/documents/profile_catalog.htm

Questions

- What is a composite structure?
- What is a package?
- What is a “merge” dependency?
- What is a component?
- What is the difference between package and component?
- What is a stereotype?
- What is a profile and what it is good for?

References on advanced UML

- OMG, *UML Superstructure* 2.4.1, 2011
- `www.agilemodeling.com`
- `advanceduml.wordpress.com`
- `www.uml-diagrams.org`
- `www.ibm.com/developerworks/rational/rationaledge/`
- `www.ibm.com/developerworks/rational/library/05/321_uml/`

Readings

- Ambler, *The Elements of UML2 Style*, Cambridge UP, 2005
- Cheesman and Daniels, *UML Components. A Simple Process for Specifying Component-Based Software,* Addison Wesley, 2000
- Bell, UML Basic: The Component Diagram, 2004 `www.ibm.com/developerworks/rational/library/dec04/bell/`
- Ivers et al., Documenting Component and Connector Views with UML 2.0, TR8 SEI, 2004
- Bock, UML2 Composition Model, *JOT 3:10,* 2004, `www.jot.fm/issues/issue_2004_11/column5/`

Tools for advanced UML

- www.sei.cmu.edu/architecture/arche.html
- www-01.ibm.com/software/awdtools/swarchitect/websphere/
- www.sparxsystems.com.au/products/ea/index.html
- www.papyrusuml.org
- diaspec.bordeaux.inria.fr
- lci.cs.ubbcluj.ro/ocle/index.htm

Questions?