

---

**PA103 - Object-oriented Methods for Design of Information Systems**

# **Interface as contract**

**© Radek Ošlejšek**  
**Fakulta informatiky MU**  
**oslejsek@fi.muni.cz**

---

# Lecture 2 / Part 1: **Interfaces in general**

# Interface

---

- The interface of the receiving system, to which the using system must relate and refer.
- Used in several different places in IT architecture:
  - Between human and computer, i.e. GUI.
  - API of libraries.
  - ➔ ▪ Interface of components.
  - ➔ ▪ Low-level interface of objects/classes.

# Interfaces of objects and components

---

- Interface has following **properties**:
  - Provides contact with outside world
  - Consists of operations and input/output parameters
    - So called *signature*
  - Must be well-documented
    - Including *contract*.
  - Must be on display and easily accessible
    - Visibility scope is often *public*
  - Can specify contracts
    - Usage constraints
    - e.g. **Comparable** in Java API

# Object and component interfaces (cont.)

---

- **Operations:**
  - Operations provide functionality.
  - Without operations, no functionality.
- **Input parameters:**
  - Data elements, which the operation needs to be able to perform the specified functionality
- **Output parameters:**
  - Data elements, which the operation returns after performing the specified functionality
  - Number of allowed output parameters depends on programming language
    - Return value is always single, e.g. single array of integers.
    - Some OO languages support out or in-out parameters.

# Information Hiding Rules

---

- Carefully define the public interface for classes as well as subsystems (components)
  - For subsystems use facade design pattern if possible
- Always apply the “Need to know” principle:
  - Only if somebody needs to access the information, make it publicly available
- The fewer details a class/component user has to know
  - the easier the class/component can be changed
  - the less likely will be affected by any changes in the class/component implementation

# Modeling Constraints with Contract

---

- Example of constraints in a sports arena:
  - An already registered player cannot be registered again
  - The number of players in tournament should not be more than *maxNumPlayers*
  - One can only remove players that have been registered
- These constraints cannot be modeled in UML, we model them with contracts (e. g. OCL)

# Contract

---

## Contract:

- A lawful agreement between two parties in which both parties accept obligations and on which both parties can found their right

## Object-oriented contract:

- Describes the services that are provided by object, subsystem or system if certain conditions are fulfilled
  - Services = „obligations“, conditions = „rights“
- For each service, it specifically describes two things:
  - The **conditions** under which the service will be provided
  - A **specification of the result** of the service
- Examples:
  - A **letter posted before 18:00** will be **delivered on the next working day** to any address in Czech Republic
  - For the **price of 4 Euros a letter with the maximum weight of 80 grams** will be **delivered anywhere in the USA within 4 hours of pickup**



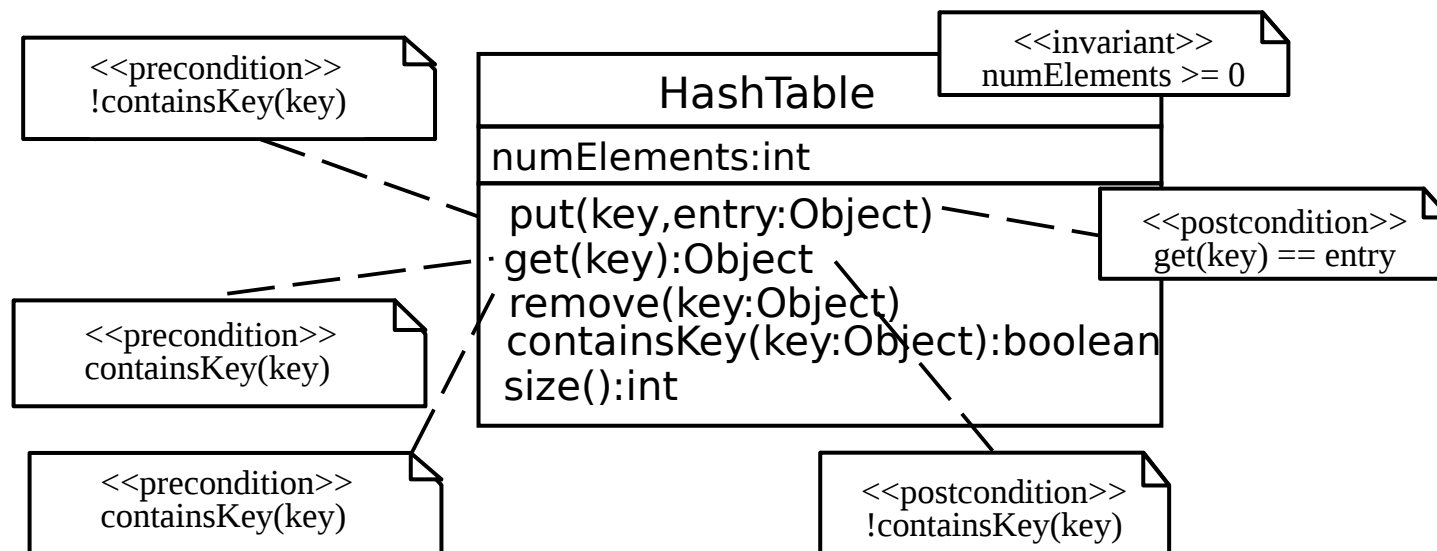
# OO Contract

---

- Contracts enable the caller and the provider to share the same assumptions about the class, component or subsystem
- A contract is an exact specification of the interface
- A contract include three types of constraints:
  - **Invariants**
    - A predicate that is always true for all instances of a class/component
    - Ex.: Pub is never out of beer
  - **Preconditions** („rights“)
    - Must be true before an operation/service is invoked
    - Ex.: for *visitPub* method: customer has money (otherwise the method behavior is unpredictable)
  - **Postconditions** („obligation“)
    - Must be true after an operation/service is invoked
    - Ex.: for *visitPub* method: customer has no money

# Modeling OO Contracts

- Natural language
- Mathematical Notation
- Models and contracts
  - OCL (Object Constraint Language) = language for the formulation of constraints with the formal strength of the mathematical notation and the easiness of natural language
  - => UML models + OCL constraints or text notes

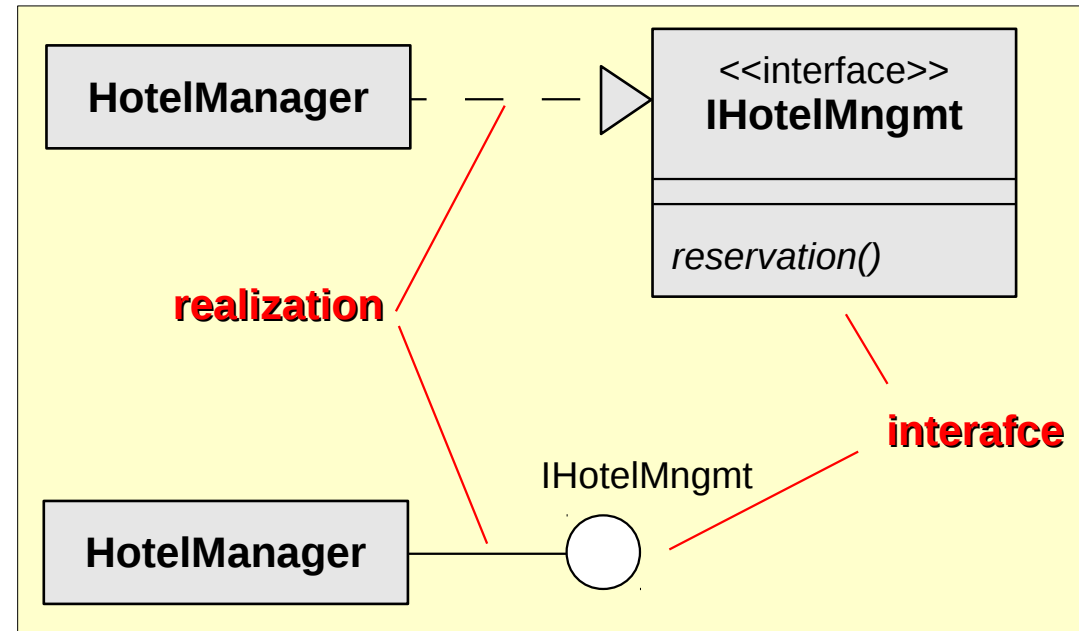


---

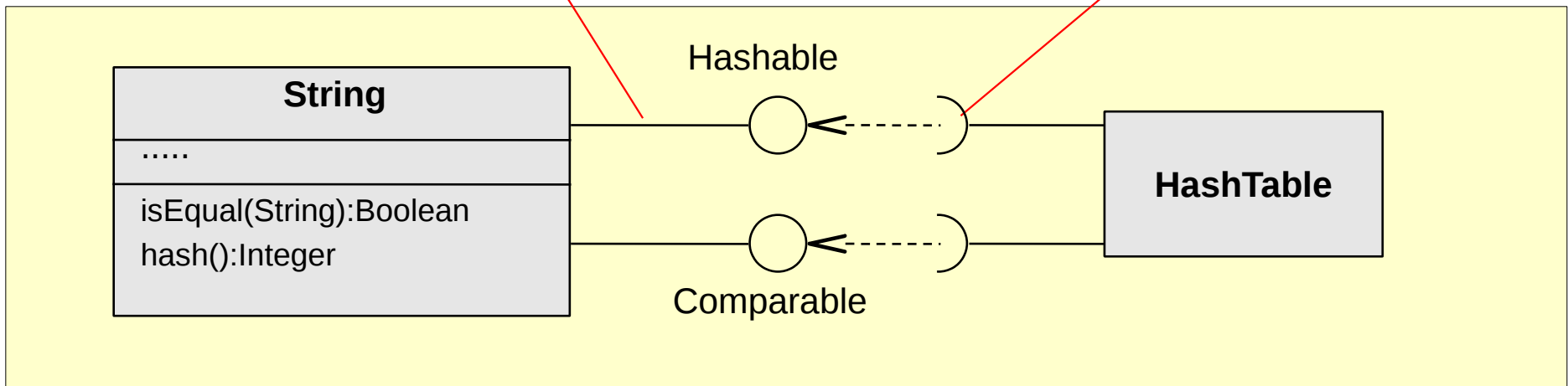
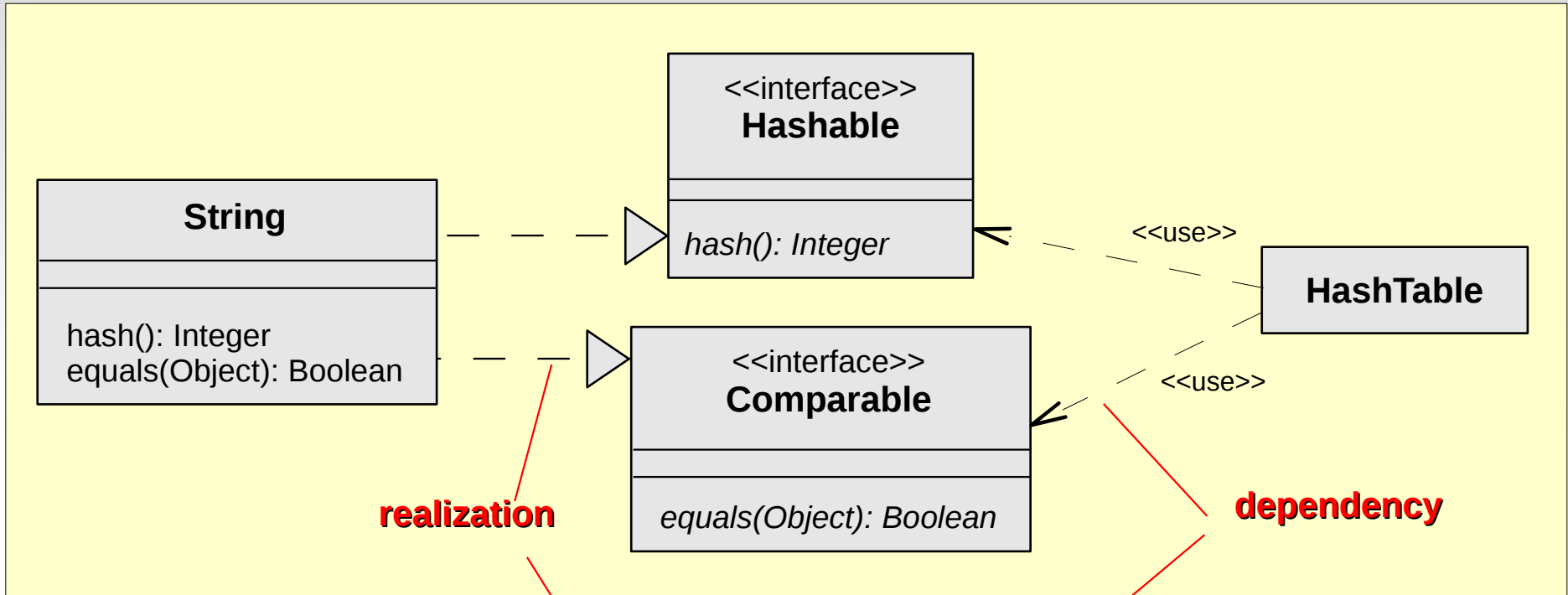
# Lecture 2 / Part 2: **Interfaces of classes/objects**

# <<interface>> stereotype

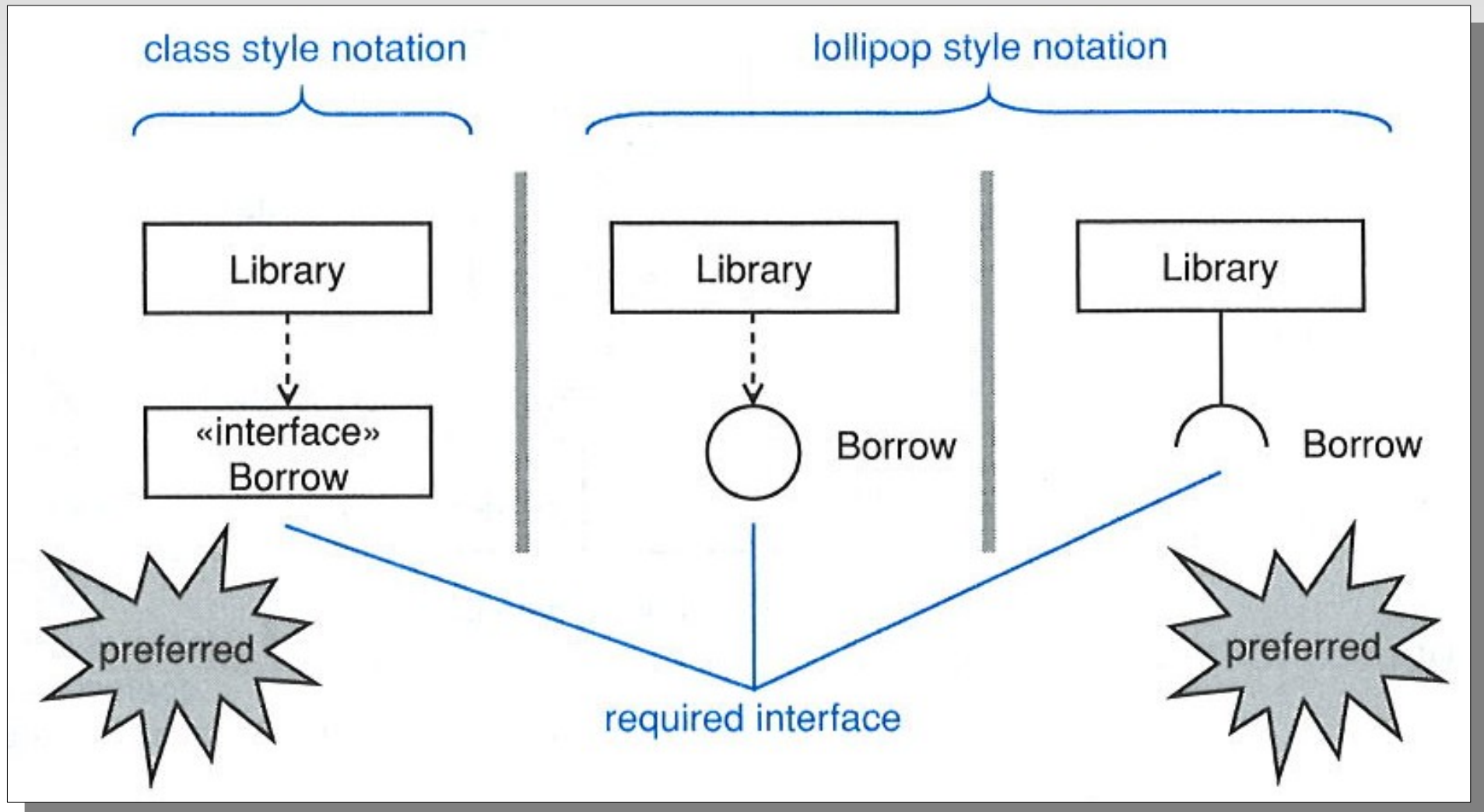
- Interface of “ordinary” class = list of provided methods
- Class with <<interface>> stereotype has special meaning in UML
  - Realization vs. inheritance
  - Two notations
  - Related to the interfaces of components
  - Often prescribe only partial behavior (and, on the contrary, class often implements multiple interfaces)



# Provided and Required Interfaces



# Preferred notations for required interfaces



# Visibility of Operations and Attributes

---

## **Public: +**

- Public elements (attributes, operations) can be accessed by any external code.

## **Private: -**

- Private elements can be accessed only by the class in which they are defined
- They cannot be accessed by subclasses or other classes

## **Protected: #**

- Protected elements can be accessed by the class in which they are defined and by any descendant of the class

## **Package: ~**

- Package-visible elements can be accessed by the class in which they are defined and by any class in the same package
- They cannot be accessed by classes in sub-packages
- They cannot be accessed by sub-classes in other packages.

---

# Lecture 2 / Part 3: **Interfaces of components**



# From Classes to Components

---

- Component model = design model.
- UML component diagram is generic, concrete component-based runtime environments (CORBA, EJB, ...) can have various limitations.
- Component is a logical, *replaceable* part of a system that conforms to and provides the realization of a set of interfaces.
- Unlike objects that arises and vanishes in the memory at runtime, components are carefully selected and connected at design time and then deployed in the form of binary replaceable artifacts (it is possible to replace a component with other that conforms to the same interface at runtime).
- Interfaces bridge logical and design models. For example, you may specify an interface for a class in a logical model, and that the same interface will carry over to some design component that realizes it.
- Interfaces allow you to build the implementation of a component using smaller components by wiring ports on the components together.

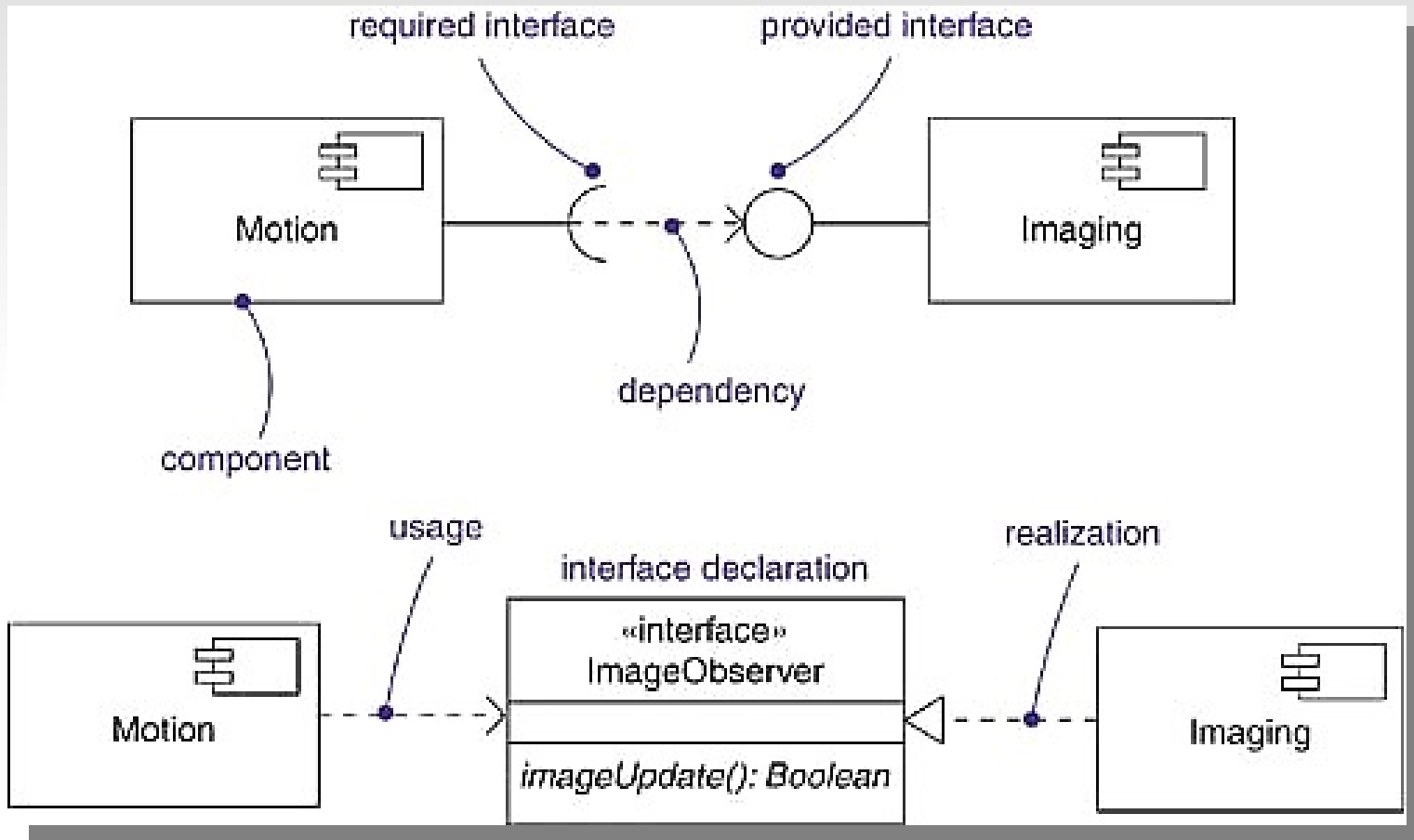
# Components – Terms and Concepts

---

- **Component** – replaceable part of a system that conforms to and provides the realization of a set of operations.
- **Interface** – collection of operations that specify a service that is provided by or required from a component.
- **Port** – specific window into an encapsulated component accepting messages to and from the component conforming to specified interfaces.
- **Internal structure** – implementation of a component by means of a set of parts that are connected together in specific way.
- **Part** – specification of a role that composes part of the implementation of a component.
- **Connector** – communication relationship between two parts or ports within the context of a component.

# Components and Interfaces

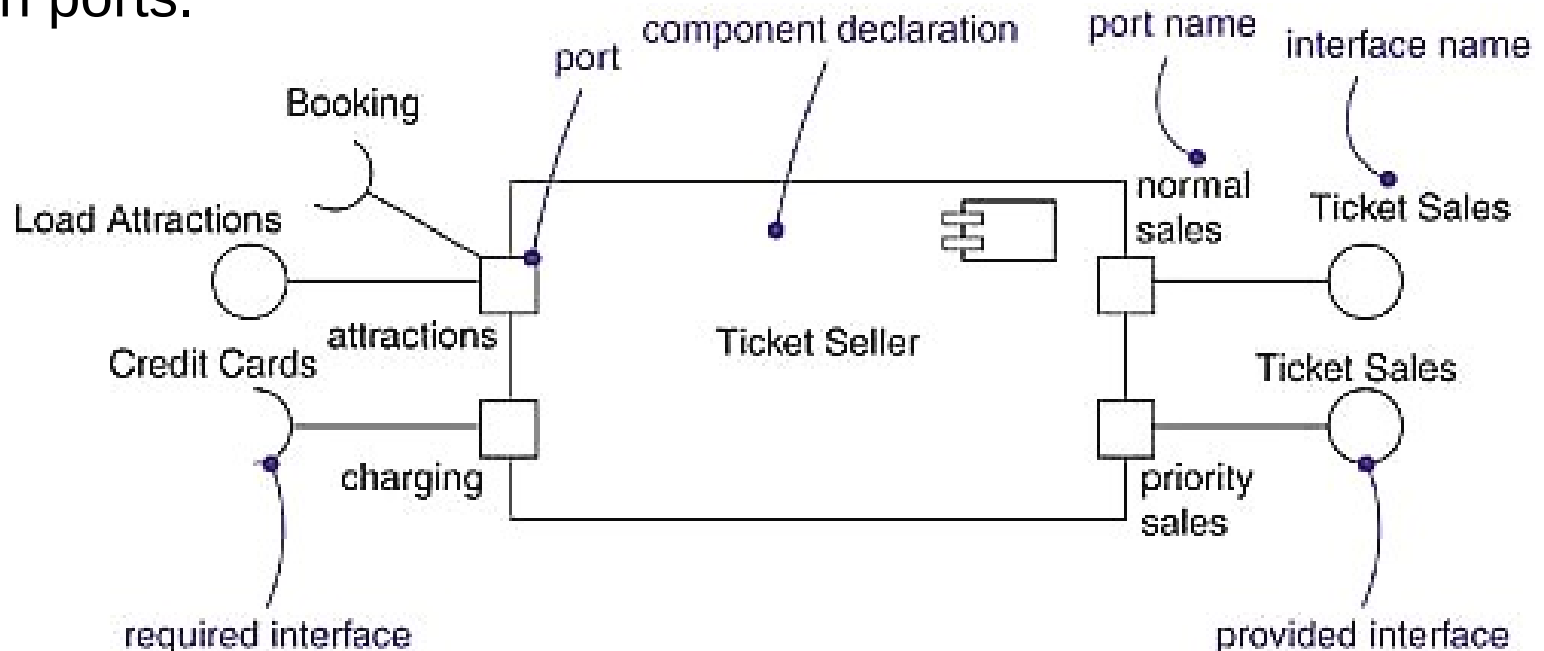
- Component interfaces are similar to interfaces in class diagrams...



**Note:** The dashed dependency lines show compatible provided and required interfaces, but when the interfaces have the same names the dependency lines are redundant and can be omitted.

# Ports

- ... except for ports.
- Interfaces are useful in declaring the overall behavior of a component, but they have no individual identity.
- Ports provide greater control over the implementation of interfaces.
- Port has identity. Another component can communicate with the component through a specific port.
- External interactions into and out of the component should pass through ports.

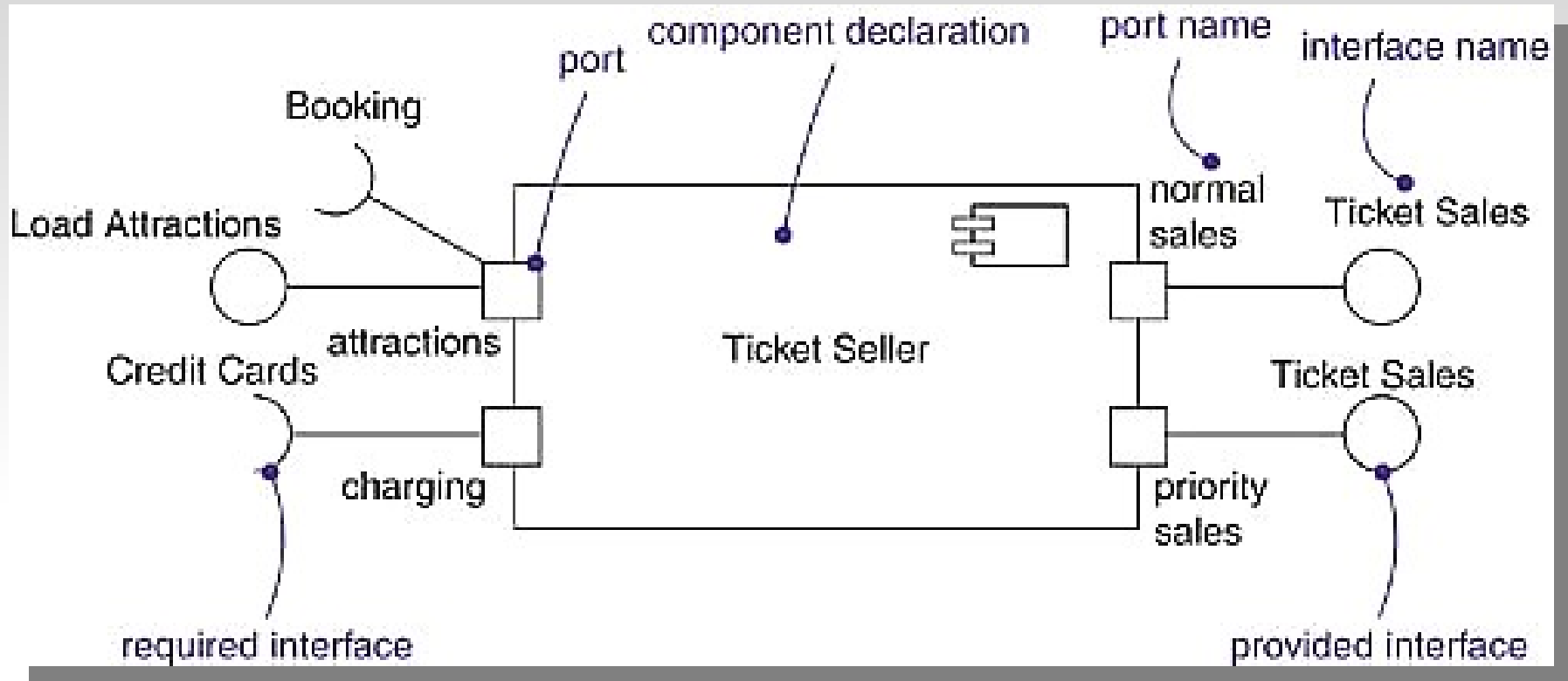


# Ports (cont.)

---

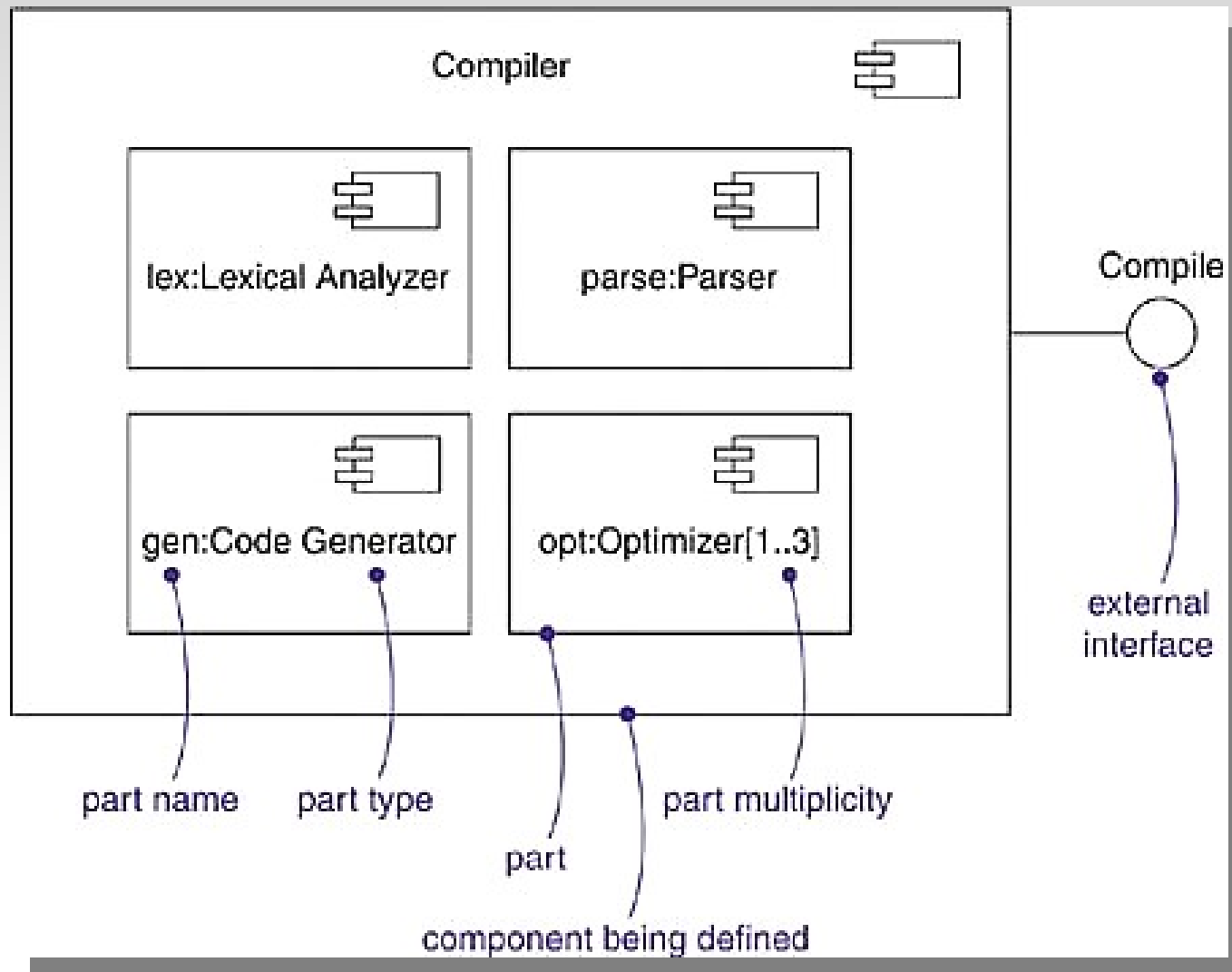
- Both provided and required interfaces may be attached to the port.
- Each port has a name so that it can be uniquely identified given the component and the port name.
- Ports are part of a component. Instances of ports are created and destroyed along with the instance of the component to which they belong.
- Ports may also have multiplicity; this indicates the possible number of instances of a particular port within an instance of the component.
  - Each port on a component instance has an array of port instances.
  - Although the port instances in an array all satisfy the same interface and accept the same kinds of requests, they may have different states and data values.
  - For example, each instance in an array might have a different priority level, with the higher-priority port instances being served first.

# Ports - example



There are two ports for ticket sales, one for normal customers and one for priority customers. They both have the same provided interface of type Ticket Sales. The credit card processing port has a required interface; any component that provides the specified services can satisfy it. The attractions port has both provided and required interfaces. Using the Load Attractions interface, a theater can enter shows and other attractions into the ticket database for sale. Using the Booking interface, the ticket seller component can query the theaters for the availability of tickets and actually buy the tickets.

# Internal Structure of Components



**Example:** A compiler component built from four kinds of parts. There is a lexical analyzer, a parser, a code generator, and one to three optimizers. The appropriate optimizer can be selected at run time.

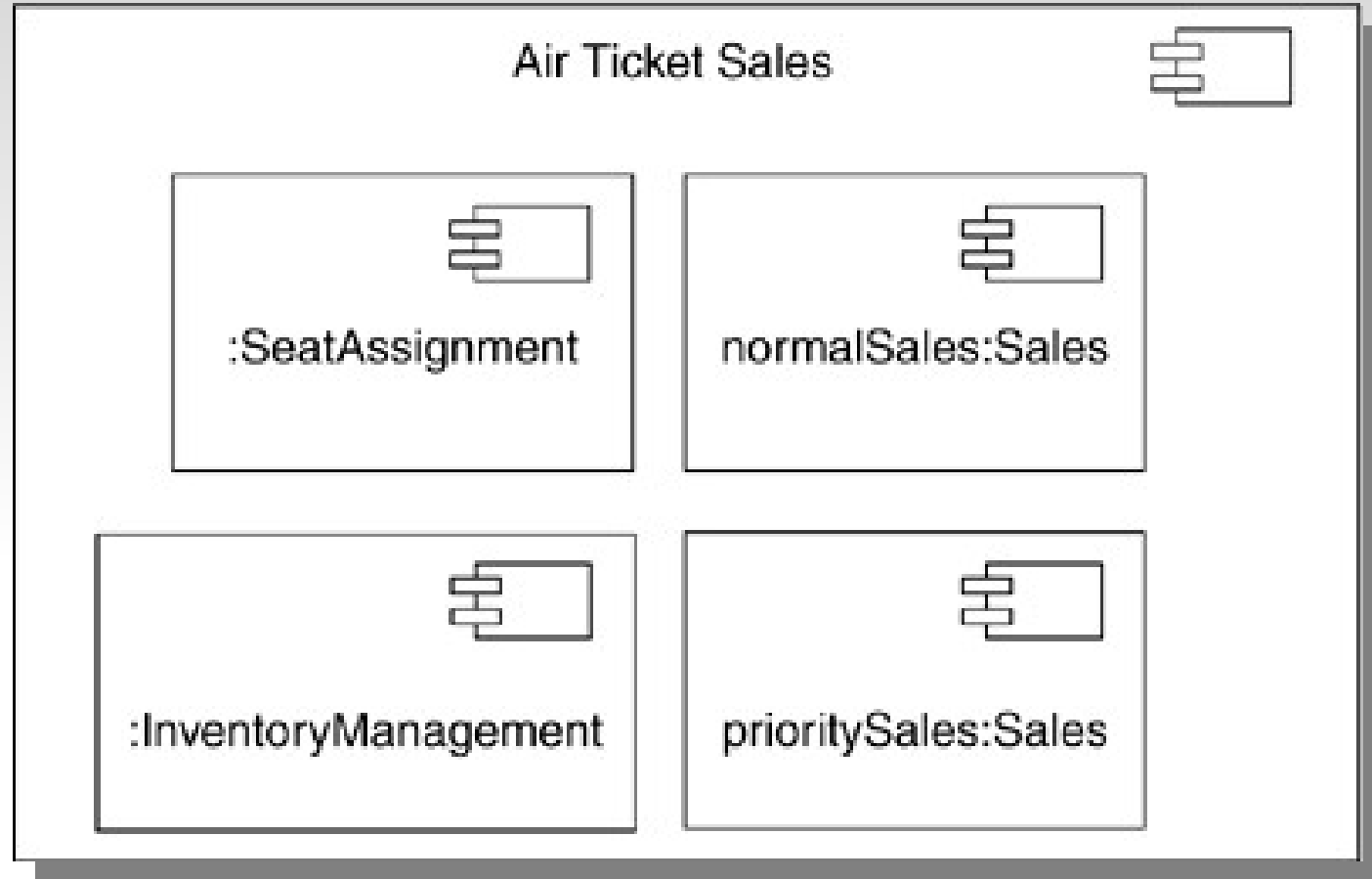
# Internal Structure of Components (cont.)

---

- A **part** is a unit of the implementation of a component.
- A part has a name and a type.
- In an instance of the component, there is one or more instance corresponding to each part having the type specified by the part.
- A part has a **multiplicity** within its component.
  - If the multiplicity of the part is greater than one, there may be more than one part instance in a given component instance.
  - If the multiplicity is something other than a single integer, the number of part instances may vary from one instance of the component to another.
  - A component instance is created with the minimum number of parts; additional parts can be added later.



# Parts of the Same Type



*Air Ticket Sales* component might have separate *Sales* parts for frequent fliers and for regular customers; they both work the same, but the frequent-flier part is available only to special customers and involves less chance of waiting in line. Because these components have the same type, they must have names to distinguish them. The other two components of types *SeatAssignment* and *InventoryManagement* do not require names because there is only one of each type within the *Air Ticket Sales* component.

# Connectors

---

**Connectors in UML** = connection of two structured parts within a structured classifier or a collaboration.

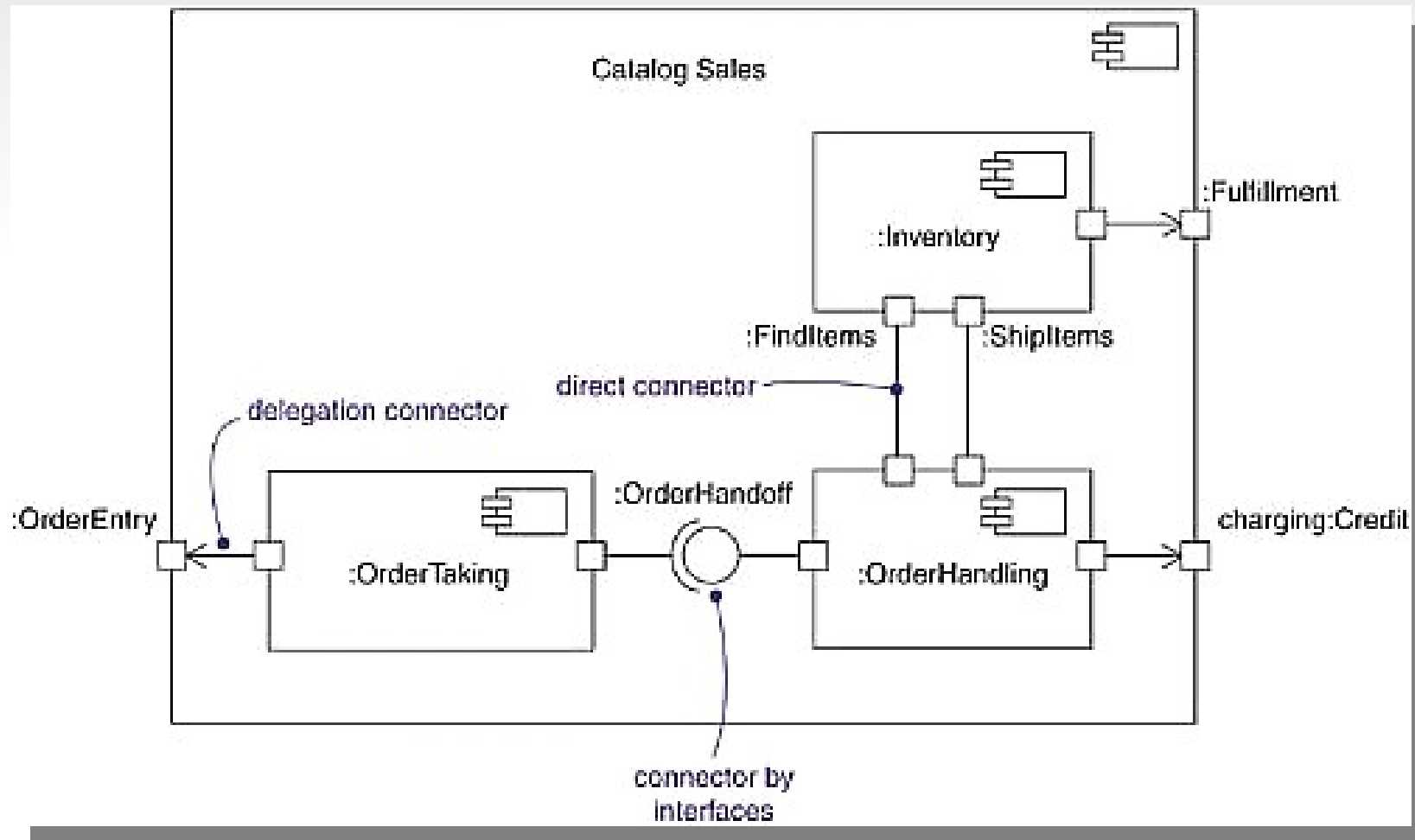
**Connectors in components** = A wire between two ports

- Realizes calls between compatible interfaces/ports; can be generated by tool

# Connectors - Example

## Connector by interfaces

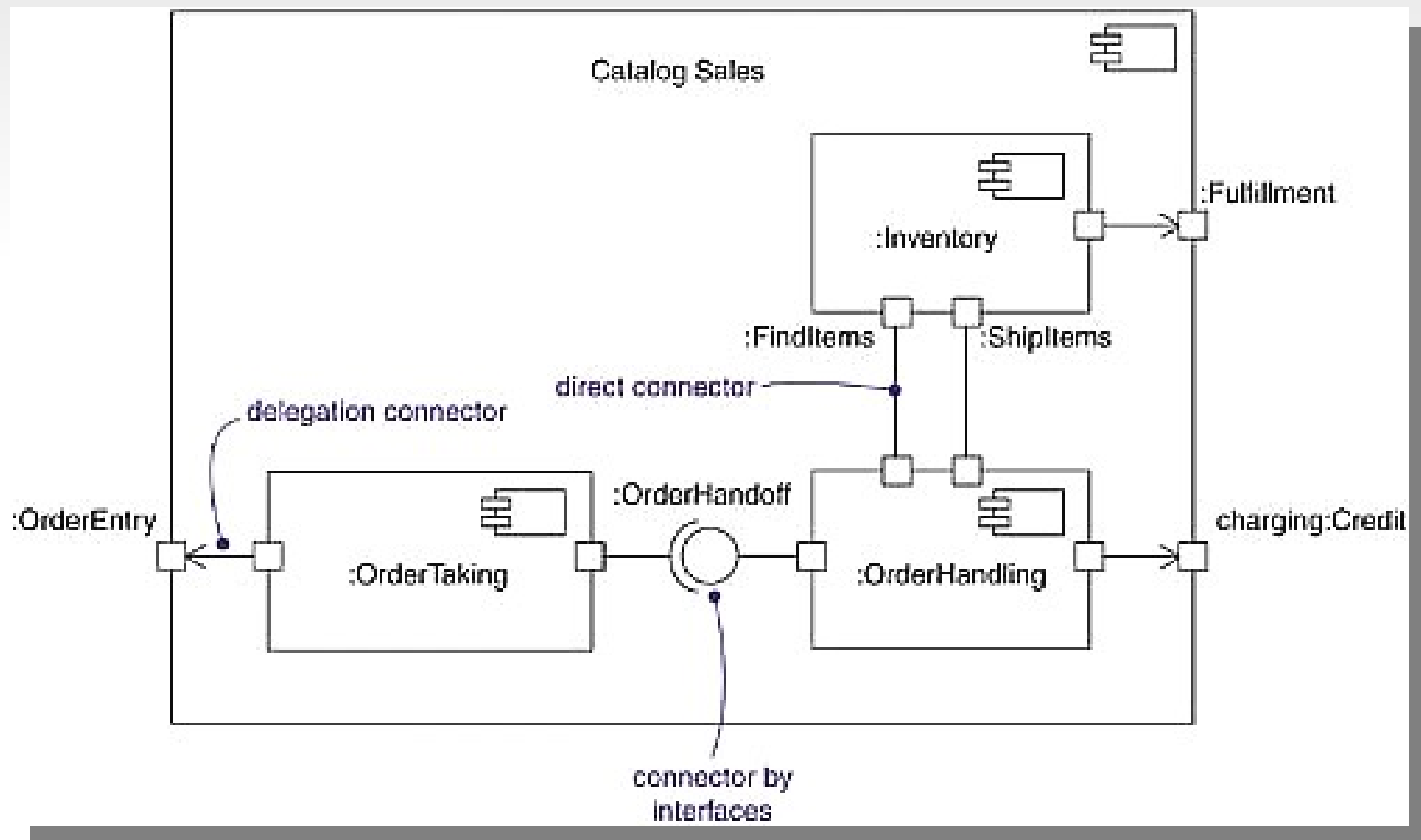
- Components are connected via compatible provided/required interfaces



# Connectors - Example

## Direct connector

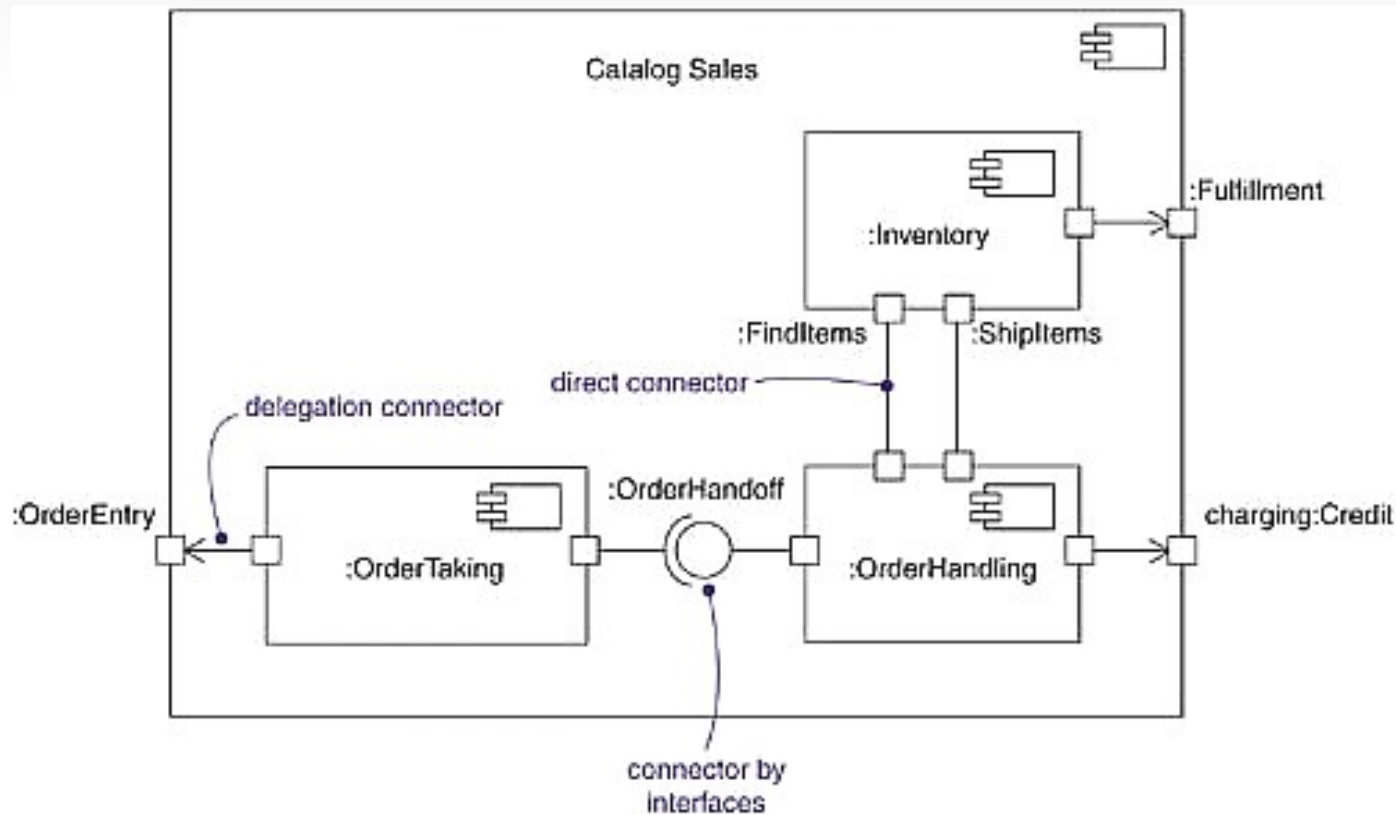
- Components are explicitly wired together, either directly or through ports



# Connectors - Example

## Delegation connector

- Connection of internal ports to external ports. Contains arrow.
- Interpretation 1: the internal port is the same as the external port; it has been moved to the boundary and allowed to peek through.
- Interpretation 2: any message to the external port is transmitted immediately to the internal port, and vice versa.



# Components – Summary

---

- Components allow you to:
  - Encapsulate the parts of your system.
  - Reduce dependencies.
  - Make dependencies explicit.
  - Enhance replaceability and flexibility when the system must be changed in the future.
- A good component:
  - Encapsulates a service that has a well-defined interface and boundary.
  - Has enough internal structure to be worth describing.
  - Does not combine unrelated functionality into a single unit.
  - Organizes its external behavior using a few interfaces and ports.
  - Interacts only through declared ports.

---

# Lecture 2 / Part 4: **Interface Description Languages**

# IDL – Interface Description Language

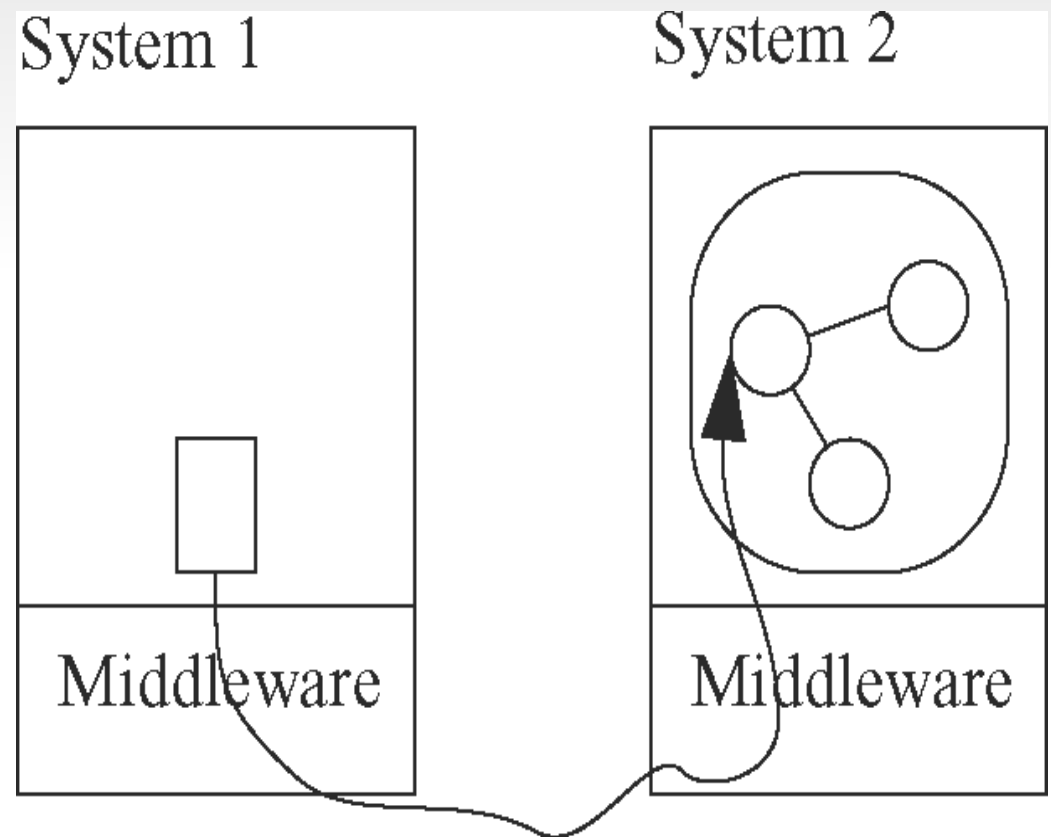
---

- Alternatively **Interface Definition Language**.
- Specification language used to describe a software component's interface in a language-independent way.
- IDL is not programming language – has no constructs.
- IDL is **not** a part of UML.
- Software systems based on IDLs include:
  - The Open Group's Distributed Computing Environment,
  - IBM's System Object Model,
  - the OMG CORBA,
  - Mozilla's XPCOM,
  - Facebook's Thrift,
  - WSDL for Web services,
  - ...



# Distributed Computing

- The object reference contains:
  - The network address.
  - The port number (transport protocol).
  - Object (agent) name or ID.



Object Reference

# CORBA

---

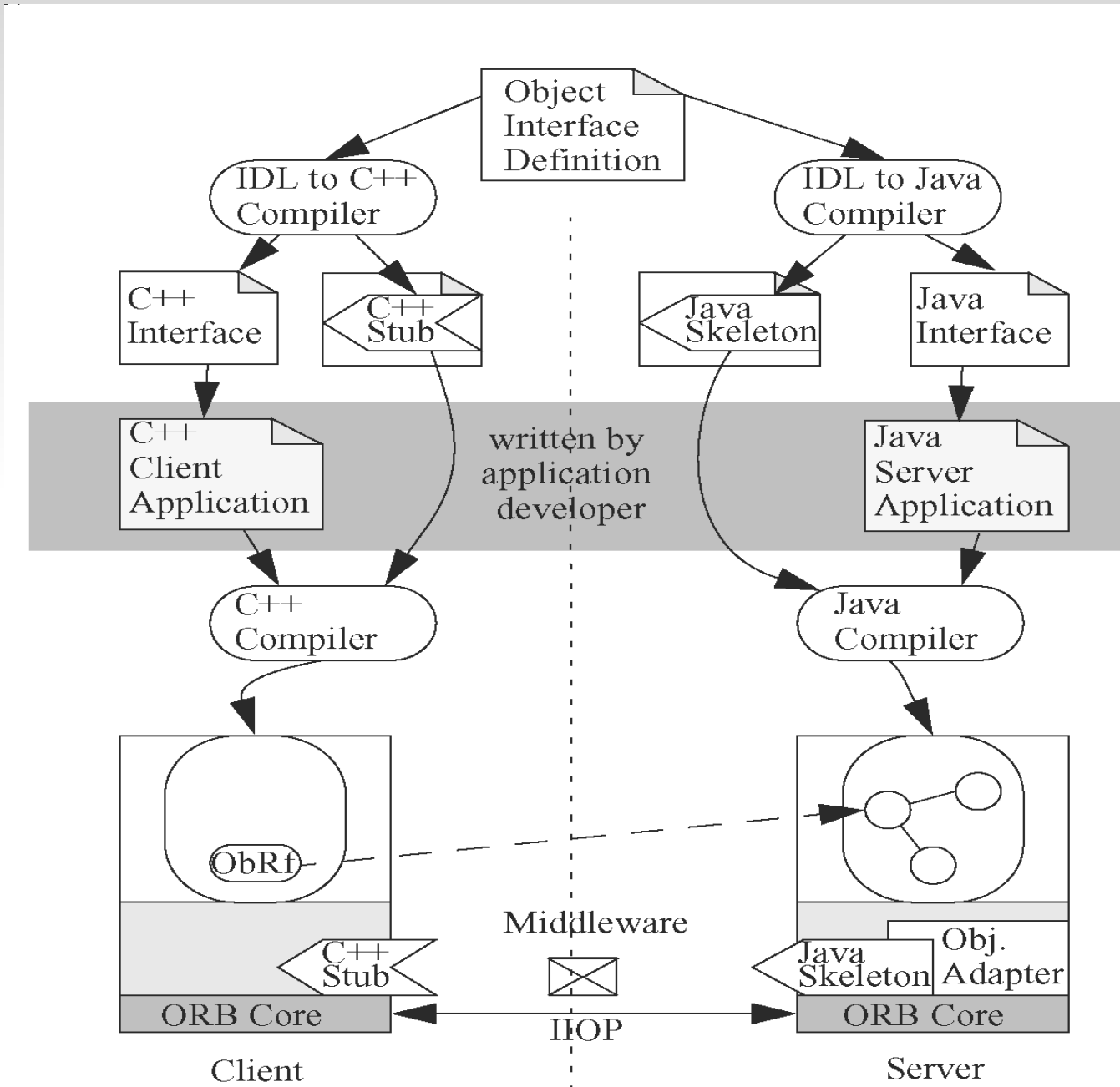
- CORBA = **C**ommon **O**bject **R**equest **B**roker **A**rchitecture, specified by OMG (Object Management Group)
  - Open and vendor independent architecture and infrastructure that computer applications use to work together over the network
- ORB (**O**bject **R**equest **B**roker) mediates the communication between applications:
  - Locating the remote object
  - Activating the remote object
  - Communicating the client request to the object
  - Communicating the reply after carrying out the reply

# CORBA IDL

- Declares data members, methods and parameters
- Has its own type system
- Maps to many programming languages like C, C++, Java and COBOL via OMG standards (interface compilers)

```
module BankSimple {
    typedef float CashAmount;
    interface Bank {
        ...
    };
    interface Account {
        // account owner and balance
        readonly attribute string name;
        readonly attribute CashAmount balance;
        // operations available on the account
        void deposit (in CashAmount amount);
        void withdraw (in CashAmount amount);
    };
};
```

# CORBA Workflow



# Mapping IDL to Java/C++

---

IDL	Java	C++
module	package	namespace
interface	interface	abstract class
operation	method	member function
attribute	pair of methods	pair of functions
exception	exception	exception

# Declaring Data Members in CORBA IDL

---

- Data members are declared using the `attribute` keyword.
- The declaration must include a name and a type.
- Attributes are readable and writable by default.
- To make a read-only attribute, use the `readonly` keyword.
- IDL compiler generates public read and write methods for the data member as required.

```
attribute long assignable;
```

generates

```
int assignable();
```

```
void assignable (int i);
```

# Declaring Data Members (cont.)

---

- CORBA `const` values declared in an IDL map to `public static final` fields in the corresponding Java interface.
- Constants not declared inside the interface are mapped to `public interface` with the same name containing a field `value`.

```
const float sample = 2.3;
```

- It is also possible to define your own types using the `typedef` keyword:

```
typedef string name;
```

# Declaring Methods in CORBA IDL

---

- Methods are declared by specifying name, return type, and parameters.

```
float calculate(in float val1, in char operator);
```

- Methods can optionally throw exceptions. User-defined exceptions must be declared in the IDL.
- Methods are *synchronous* by default. The client program will wait for the remote method to execute and return.
- *Asynchronous* methods are defined using the **oneway** keyword. **oneway** methods have no return value, can have input parameters only and cannot throw exceptions. The client makes the call to the **oneway** method and continues processing while the remote object executes it – the client is not blocked.



# Declaring Parameters in CORBA IDL

---

- Parameters can be of following types:
  - Basic (`char`, `long`, `short`, `float`, `bool`, etc.),
  - Constructed (`struct`, `union`, `array`, `sequence`),
  - Typed objects,
  - `any`.
- Parameters can be declared as `in`, `out` or `inout`.
  - `in` parameters are copied from client to server
  - `out` parameters are copied from server to client
  - `inout` parameters are used both for incoming and outgoing information and are copied both ways.
- CORBA 2.0 supports only pass-by-value for non-object data types. Objects are passed by reference
- CORBA 3.0 supports pass-by-value for objects by using the `valuetype` keyword.

# Web Services

---

- **W3C definition:**
- “A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other web-related standards”.

# WS – Key Technologies

---

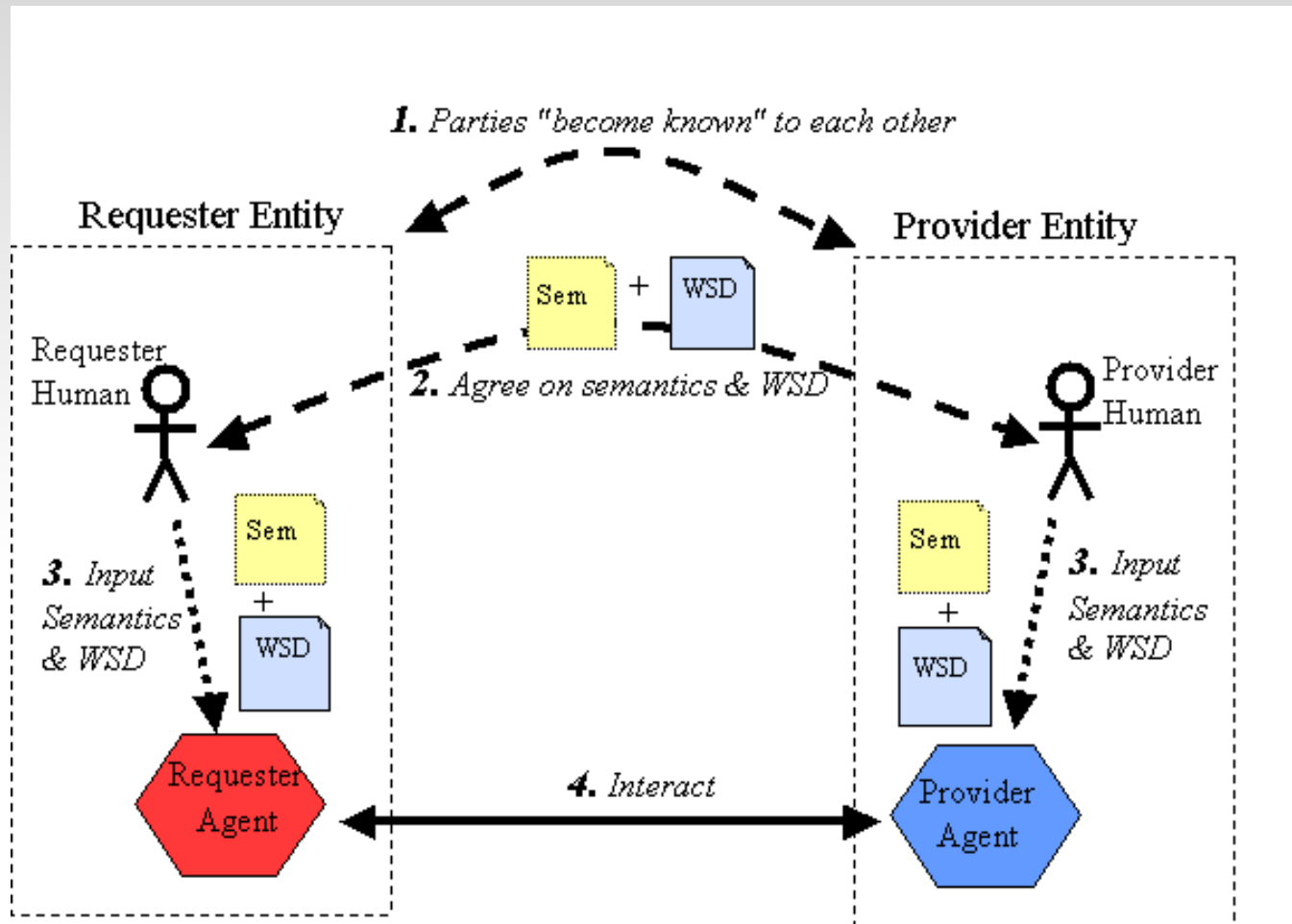
- **XML (eXtensible Markup Language):**
  - is a markup language for exchanging structured messages.
- **SOAP:**
  - is a XML based protocol for exchanging messages between nodes over different transport protocols (HTTP, SMTP, ...)
- **WSDL (Web Services Description Language):**
  - Based on XML Schema
  - The interface description (operations, operations parameters, data types)
  - The implementation description (service location, transport protocol)
  - Generated automatically, e.g. from annotated EJB
  - Convertible from IDL

# WS – Key Technologies (cont.)

---

- **UDDI (Universal Description, Discovery and Integration):**
  - Discovery agency that provides find and publish services for the requester and provider agents.
  - Provides two kinds of information: business related information and technical information.
  - The communication between the UDDI and the requester and provider agents uses SOAP messages.

# The General Process of Engaging a WS



(1) the requester and provider entities become known to each other (or at least one becomes know to the other); (2) the requester and provider entities somehow agree on the service description and semantics that will govern the interaction between the requester and provider agents; (3) the service description and semantics are realized by the requester and provider agents; and (4) the requester and provider agents exchange messages

# WSDL Example

```
<?xml version="1.0" ?>
- <definitions name="BNQuoteService"
  targetNamespace="http://www.xmethods.
  net/sd/BNQuoteService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XML
  Schema"
  xmlns:soap="http://schemas.xmlsoap.org/
  wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/
  "
  xmlns:tns="http://www.xmethods.
  BNQuoteService.wsdl">
- <message name="getPriceRequest">
  <part name="isbn" type="xsd:string" />
</message>
- <message name="getPriceResponse">
  <part name="price" type="xsd:float" />
</message>
- <portType name="BNQuotePortType">
- <operation name="getPrice">
  <input message="tns:getPriceRequest"
    name="getPrice" />
  <output message="tns:getPriceResponse"
    name="getPriceResponse" />
</operation>
</portType>
- <binding name="BNQuoteBinding" type="tns:BNQuotePortType">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http" />
- <operation name="getPrice">
  <soap:operation soapAction="" />
- <input name="getPrice">
  <soap:body use="encoded" namespace="urn:xmethods-
  BNPriceCheck"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</input>
- <output name="getPriceResponse">
  <soap:body use="encoded" namespace="urn:xmethods-
  BNPriceCheck"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
</output>
</operation>
</binding>
- <service name="BNQuoteService">
  <documentation>Return book price at BN.com given an ISBN
  number</documentation>
- <port name="BNQuotePort" binding="tns:BNQuoteBinding">
  <soap:address
    location="http://services.xmethods.net:80/soap/servlet/rpcrouter
    " />
</port>
</service>
</definitions>
```

**Message type**

**Port type**

**Data types**

**Binding**

**Port**

# CORBA - WS Comparison

Aspect	CORBA	Web Services
Computing model	Objects distribution	SOAP messages exchanging
Interface definition	IDL	WSDL
Location transparency	Object reference	URI
Registry	Interface repository	UDDI
Service discovery	Naming and trader services	UDDI
Implementation language	Any language with an IDL compiler	Any language
Message encoding	Binary format	Unicode
Transport protocol	GIOP/IIOP	HTTP, SMTP, HTTPS and other transport protocols
Parameter passing	By reference & by value	By value
State	Stateful	Stateless

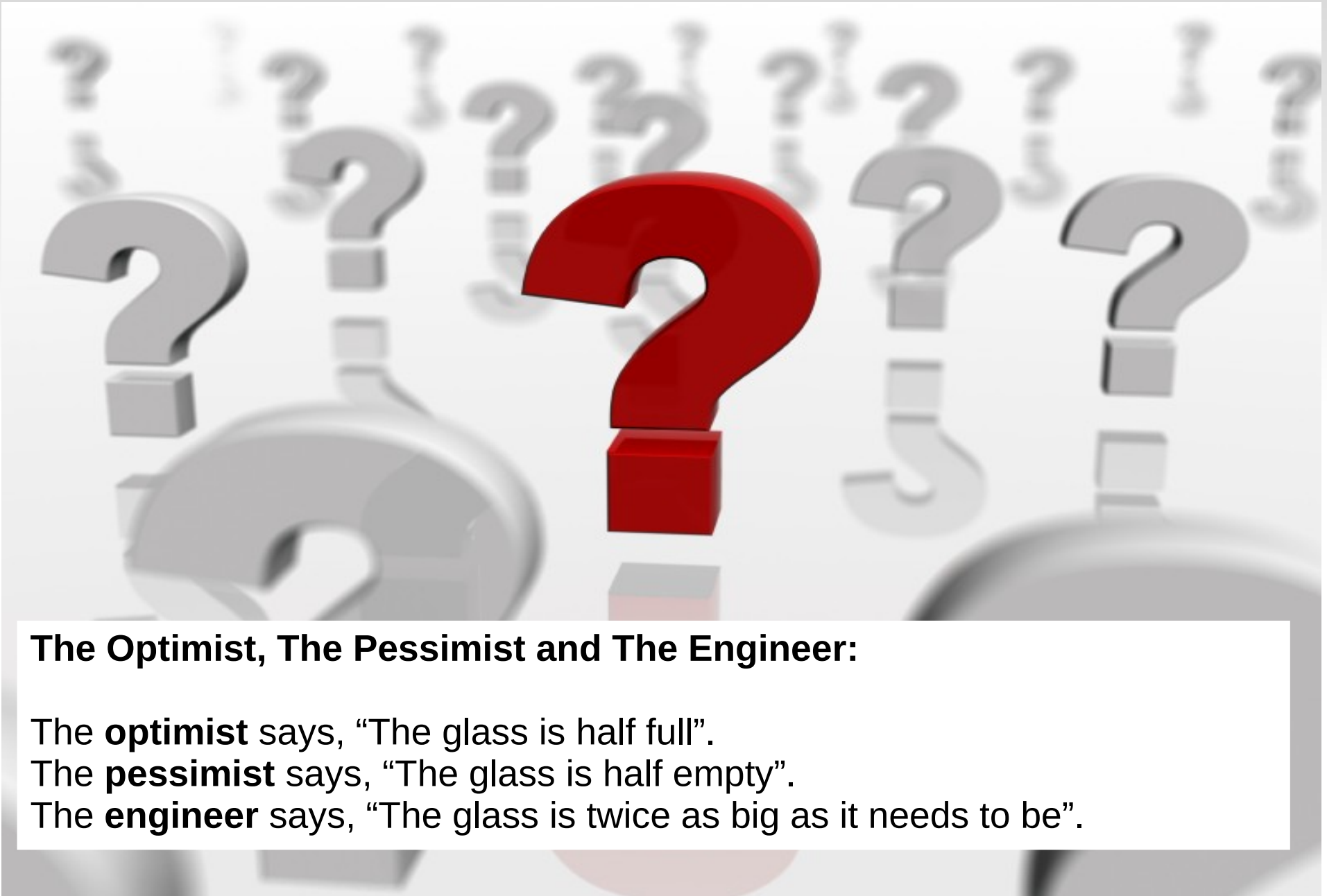
# CORBA - WS Comparison (cont.)

---

- The differences between both technologies are due to the fact that they were developed for different reasons using different technologies.
- Web Services require less effort and cost for deployment of the technology components.
- A lot of work has been done to implement SOAP messaging over CORBA IIOP.
- WSDL can be generated from CORBA IDL and vice versa.
- CORBA and Web Services can play a complementary role.



# Questions?



## **The Optimist, The Pessimist and The Engineer:**

The **optimist** says, “The glass is half full”.

The **pessimist** says, “The glass is half empty”.

The **engineer** says, “The glass is twice as big as it needs to be”.