KING'S College LONDON

University of London

Software Architecture & Design

6CCS3SAD

Dr Kevin Lano

KING'S College LONDON

University of London

Session 1: Simple Architectures; UML2

January 17th, 2020

Learning Outcomes

At the end of this session you should be able to:

- Define Software Architecture and its three key constituting concepts;
- Read and draw simple architectures using notation from the UML2; and
- Explain key properties of components, such as reusability and substitutability.

What is Software Architecture?

- **Perry and Wolf**
  - SA = {Elements (What), Form (How), Rationale (Why)}
- **Kruchten**
  - SA deals with design and implementation of high level structure of software
  - SA is about abstraction, decomposition, composition, style, and aesthetics

What is Software Architecture?

- **Shaw and Garlan**
  - SA [is a level of design that] involves the description of
    - elements from which systems are built
    - interactions among those elements
    - patterns that guide their composition
    - and constraints on these patterns

Representing Architectures

- **Architectures should be written down**
  - Enforces clarity of thinking
  - Enables analysis
  - Aids stakeholder communication and understanding

Representing Architectures (2)

- **Achievable via**
  - Using an architecture description language (ADL)
    - Textual or graphical notations
    - We will use UML2, but there are others!
  - Additional views: control flow, data flow, process, resource utilization
  - Explicit configuration modelling

Key Architectural Concepts

- **Components**
  - Computation/data
- **Connectors**
  - Information interchange
- **Configurations**
  - Instantiate components and connectors in particular arrangements

What is a Component?

- **There are many definitions...**
  - Cohesive encapsulated unit of system construction
  - A processing or data element (Perry & Wolf)
  - A unit of computation or a data store
- ***Loci* of computation and state**
  - Clients / Servers
  - Databases
  - Filters
  - Layers
  - Packages, classes

Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, Frantisek Plasil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens Szyperski. *What characterizes a (software) component?* Software – Concepts & Tools, 19(1):49–56, June 1998.

What is a Component? (2)

- **May be simple or composite**
  - Composite components = sub-systems
- **Encapsulate functionality**
  - Component interfaces expose what clients need to know to use it
  - Black-box vs grey-box vs white-box components

What is a Component? (3)

- **Catalysis definition**
  - A software package that can be independently replaced. It both provides and requires services based on specified interfaces.
- **Szyperski's definition**
  - ... a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.

http://www.catalysis.org

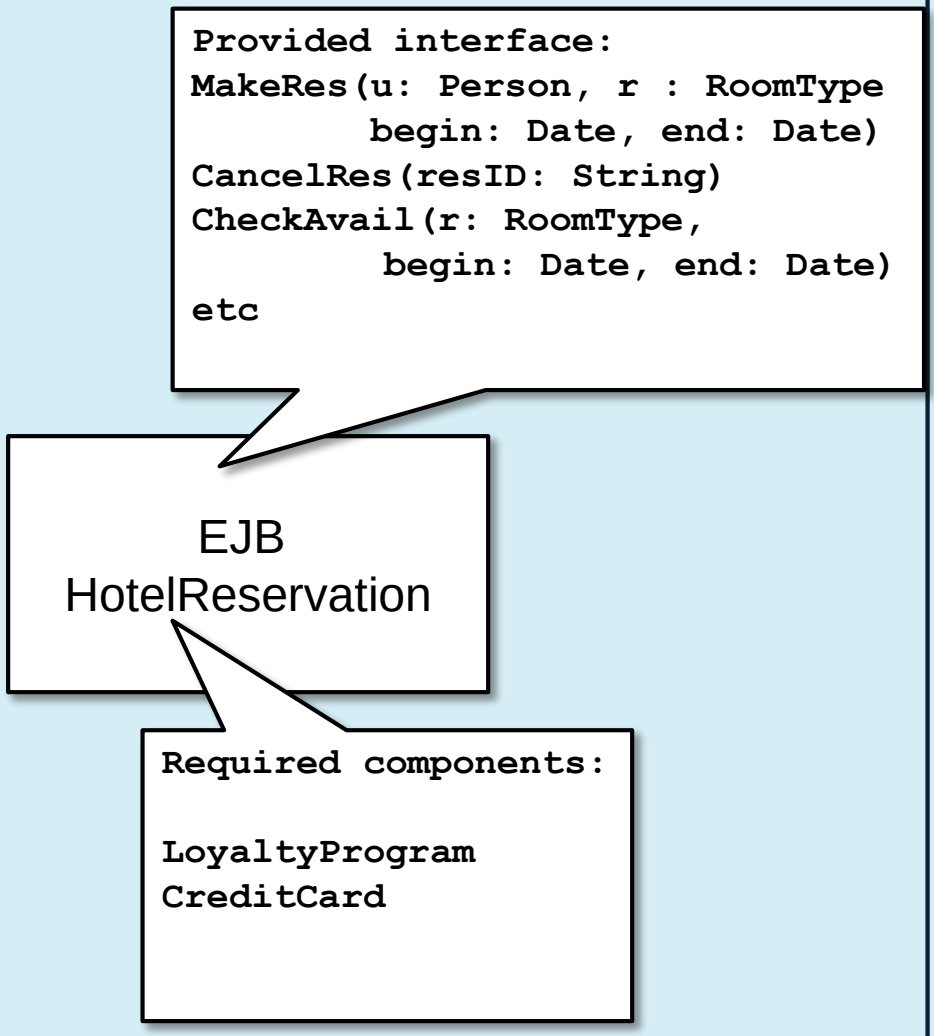
Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 2002

What is a Component? (4)

- **We will largely follow Szyperski**
- **BUT distinguish**
  - Abstract Component
    - Conceptual representation of a component
    - Used in ADLs
    - May or may not be implemented as a component
  - Concrete Component
    - Actual component file
    - Ready to be deployed
    - Depends on component infrastructure (e.g., .NET)
    - Really, what Szyperski had in mind

Example Component

- **Hotel reservation component**
  - Concrete Component (Enterprise Java Bean, EJB)
  - Used through a provided interface
  - Is itself a client using two other components



Key Architectural Concepts

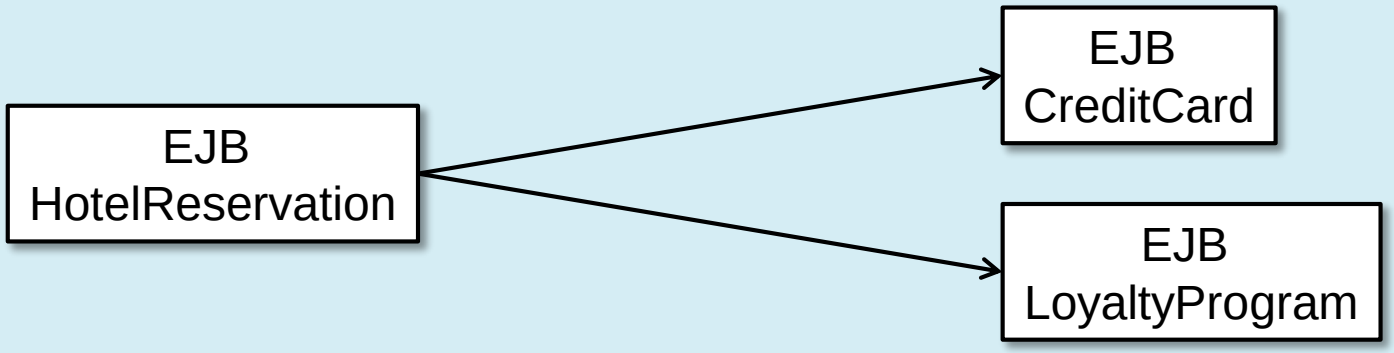
- **Components**
  - Computation
- **Connectors**
  - Information interchange
- **Configurations**
  - Instantiate components and connectors in particular arrangements

Connectors

- **Connectors model**
  - Communication among components
  - Rules that govern those communications
- **Types of interactions**
  - Simple
    - Procedure calls
    - Shared variable access
  - Complex and semantically rich
    - Client–server protocols
    - Database access protocols
    - Asynchronous event multicast
    - Piped data streams

Example Connector

- **HotelReservation should**
  - Check card validity and take a booking charge
    - synchronous communication (& transactional)
  - Notify LoyaltyProgram component if regular customer books
    - loosely coupled publish/subscribe communication



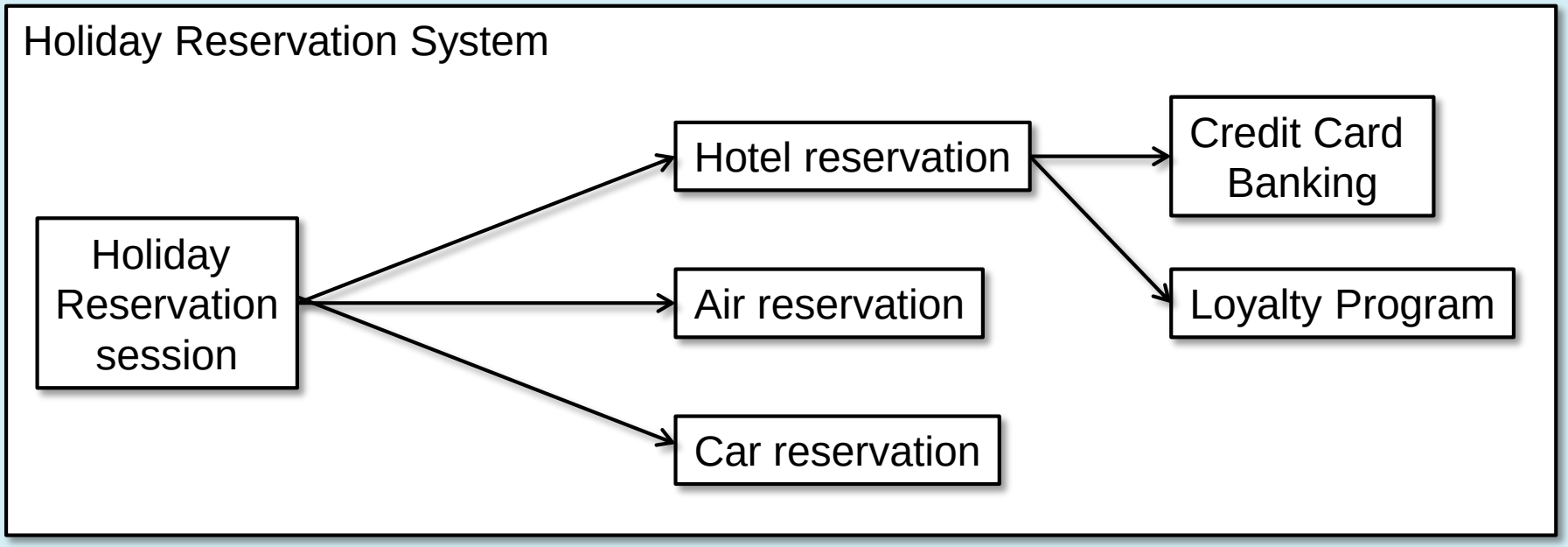
Key Architectural Concepts

- **Components**
  - Computation
- **Connectors**
  - Information interchange
- **Configurations**
  - Instantiate components and connectors in particular arrangements

Configurations

- **Connected graph of components and connectors that describes architectural structure**
  - proper connectivity
  - concurrent and distributed properties
  - adherence to design heuristics and style rules
- **Composite components are configurations**

Example configuration



ADLs

- **Architecture Description Languages**
  - Most popular in research community: Wright, ACME, Darwin, UniCon
  - Most popular in industry: EDOC, UML, reference model, development team's own personal hacked language

UML2 superstructure

- **Unified Modelling Language (UML)**
  - Version 2 (latest version) has its own built-in ADL
  - ADL is part of the UML2 “superstructure”
    - This is the part that most of us know as UML ☺
  - http://www.omg.org/spec/UML/

BREAK

Orientation

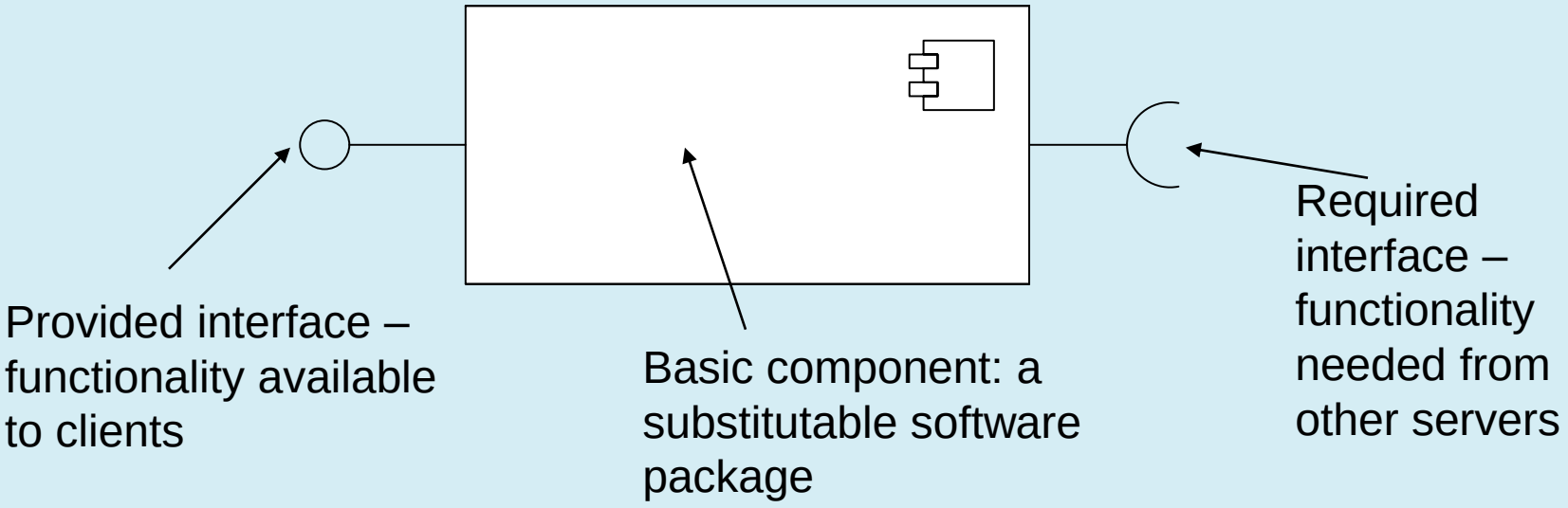
- **So far we have used box and line diagrams**
- **Now we will**
  - discuss components and connectors as abstract architectural entities
  - Give lightweight examples of what these entities can correspond to in an implementation
  - Introduce the UML 2 elements for modelling these things
- **These ideas are important later for**
  - Understanding configurations
  - Building actual architectures
  - Defining architectures for particular domains and implementation platforms

What is a Component?

- **Unit of composition**
  - Modular, encapsulating a set of functions, reusable
- **Can be independently deployed and composed**
  - Substitutable
- **Both provide and require services**
  - Based on specified interfaces

Components in UML 2

- **UML 2 has several representations of components**
- **We will use**



A warning before we proceed

- **Components are *design* elements in UML2**
  - Abstraction of something implemented
  - Some aspects correspond directly to implementation
  - Some aspects abstract things that are implicit in an implementation

A warning before we proceed

- **Component concepts will be illustrated using Java**
  - This is valid
    - Components are often abstractions of O-O classes
    - Share some of the same features
  - Design components are not O-O classes!
    - Two different levels of abstraction
  - For example:
    - Interfaces used in Java and specified in component design
    - Inheritance generally not important in component design

Components as Units of Composition

- **Properties**
  - *Modularity:*
    - Groups together sets of functions or data
  - *Cohesion:*
    - Functions and data are semantically related (have a common purpose)
  - *Encapsulation:*
    - How functions are implemented is hidden from clients
    - Clients do not need to look at the code to use the component
  - *Reusability:*
    - Can be used in different systems without change

Modularity and Cohesion

```
public class Transport implements Transportable {
  private String  smtpHost; private int smtpPort;
  public Transport() {}

  public void send (Message msg) throws TransportException {
    PrintWriter    out = null;
    BufferedReader in  = null;
    try {
      Socket s = new Socket (this.smtpHost, this.smtpPort);
      out = new PrintWriter (s.getOutputStream(), true);
      in = new BufferedReader (new InputStreamReader (s.getInputStream()));
      if (smtpErrorExists (in, "220")) {
        throw new TransportException ("Can't connect");
      }
    }
    catch (IOException ioe) {
      throw new TransportException ("Can't connect");
    }
    sendMessage (msg, in, out);
  }

  private void sendMessage (Message msg, BufferedReader in, PrintWriter out)
          throws TransportException { ... }

  private boolean smtpErrorExists (BufferedReader in, String errCode) { ... }
```

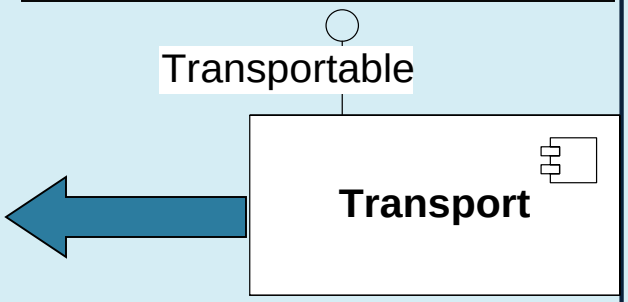
Modularity and cohesion

```
public class Transport implements Transportable {
    private String  smtpHost;
    private String  smtpResponse;
    private int  smtpPort;
    public Transport() {}

    public void send(Message msg) throws
TransportException {
        PrintWriter  out = null;
        BufferedReader in  = null;
        try { Socket s = new Socket( this.smtpHost,
this.smtpPort );
         out = new PrintWriter( s.getOutputStream(),
true );
         in = new BufferedReader( new
InputStreamReader( s.getInputStream() ) );
           if( smtpErrorExists( in,"220" ) ) {
              throw new TransportException( "Can't
connect to: " + this.smtpHost + ". Port: " +
this.smtpPort + "\t" + this.smtpResponse ); }
        }
 catch( IOException ioe ) {
           throw new TransportException( "Can't
connect to: " + this.smtpHost + ". Port: " +
this.smtpPort + "\t " );
        }
        sendMessage( msg, in, out );
    }

private void sendMessage(Message msg,
BufferedReader in, PrintWriter out) throws
TransportException {
        out.println( "HELLO " + this.smtpHost );
        if( smtpErrorExists( in,"250" ) ) {
           throw new TransportException( "SMTP error:
HELO failed.\t" + smtpResponse );
        }
        Address[] from = msg.getFrom();
        if( from == null ) {
           throw new TransportException( "SMTP error:
No FROM address specified, can't send email.\t " );
        }
        else {
                    CODE CONTINUES ...
}
```

```
public interface Transportable {
 public void send(Message msg )
   throws TransportException;
 public String getSmtpHost( );
 public int getSmtpPort();
 public void
   setSmtpHost(String smtpHost);
 public void
    setSmtpPort(int smtpPort);
}
```

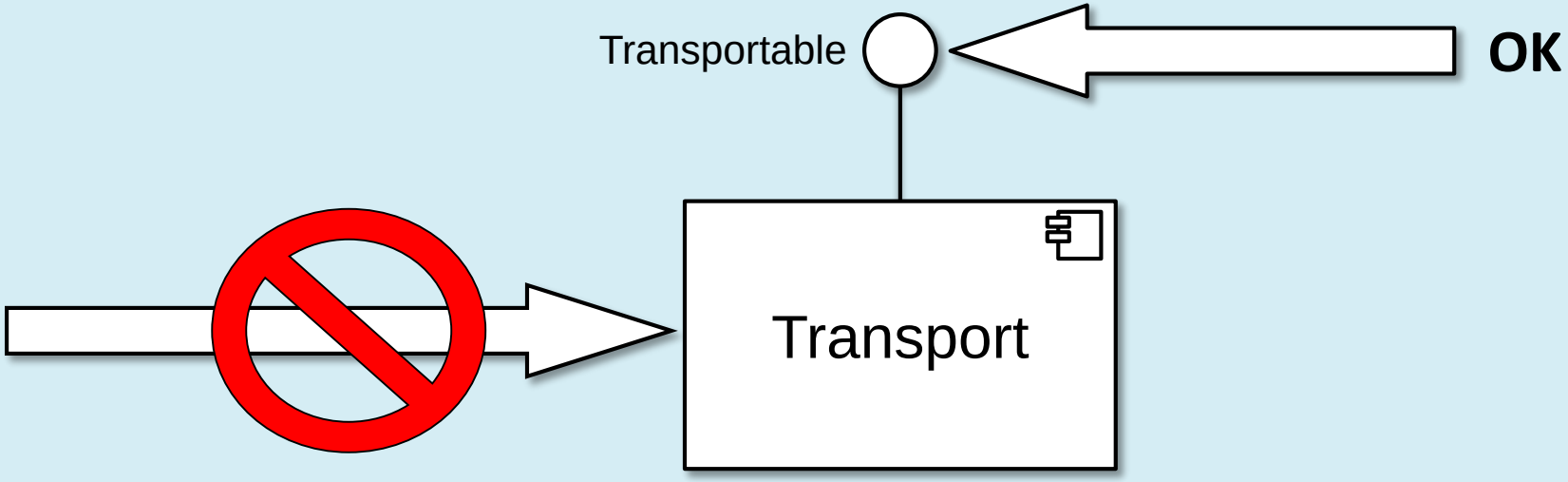


- **Here is a class that can be considered an implementation of a component because the class is modular and cohesive**
- **The class Transport is used to transport email messages via smtp**
- **This class implements the interface Transportable**
- **The class is *modular* because it groups together a *set* of methods and properties**
- **It is *cohesive*, because the set of methods and properties are all related by a common objective (they are all to do with sending email)**

Encapsulation

- **`Transport` encapsulates emailing functionality**
  - A client can use the class by
    - Importing it via **`import email.Transport;`**
    - Instantiating it via
      `Transportable tr = new Transport();`
  - Client programmer does not need (and cannot see) method code of Transport
  - Uses class with no problems via its interface

Encapsulation (2)



No direct access to component, only via its interface

Reuse

- **Applications that could reuse `Transport`**
  - A servlet email client program
  - A holiday reservation system that sends itineraries and bills via email
  - A spamming program

Reuse (2)

- **Why is reuse important?**
  - Reduced development time and cost
  - Improved reliability and quality
  - Potential for a reusable component marketplace
    - Specialist component developers selling their components to component assemblers

Problems with Reusability

- **Technical difficulties**
  - Component functionality may not be reusable
    - e.g., components often encapsulate business logic specific to an organisation
  - Component granularity too coarse or too fine
  - Components do not provide exactly what's needed
    - Need wrappers to adapt
  - Component integration is unpredictably complex

Problems with Reusability (2)

- **Economic problems**
  - Designing for reuse requires a higher up-front investment
  - Requires a long-term vision and buy-in from the management
  - ££ benefits of reuse must outweigh the ££ risks

How Reusable Should a Component Be?

- **Ideally, components should be defined independently**
  - To support reuse in a desirable range of contexts
  - To support interconnections that might occur in the future
  - Difficult in practice – but try as best as you can

Rule of good architecture design:
Define components for reusability, whenever appropriate

Provided and Required Interfaces

- **A component also**
  - Specifies
    - The services it provides to its clients
    - The services it requires from other components
  - By exposing provided and required interfaces

What is an Interface?

- **Interfaces in Java**
  - A list of methods with input/output types
  - Classes implement interfaces using **`implements`**
  - Used to define what a class is to do
    - Without saying how it is done
    - E.g., implementing Cloneable means objects can be cloned, but not how this happens
- **Interfaces separate implementation from functionality**
  - Allows implementations to be changed without breaking the system

Interface use in Java

Interface explains *what* an implementing class is to do:

Send email messages

```
public interface Transportable {
  public void send(Message msg)
      throws TransportException;
  public String getSmtpHost();
  public int getSmtpPort();
  public void setSmtpHost
              (String smtpHost);
  public void setSmtpPort
              (int smtpPort);
}
```

Two different implementations of **`Transportable`** functionality

```
public class Transport
 implements Transportable { ... }
```

```
public class EmailClient
 implements Transportable { ... }
```

```
sendMailWithClient(Transportable a,
                   String msg)
```

Can use either **`Transport`** or **`EmailClient`** instance without changing expected outcome!

```
sendMailWithClient(new Transport(), "hi there")

                   =

sendMailWithClient(new EmailClient(), "hi there")
```

Component Interfaces

- **Essentially, the same idea**
- **A component interface is**
  - *A signature:* a list of functions with input and output types
  - *A semantics:* Something telling us what these functions are meant to do and how they should be used

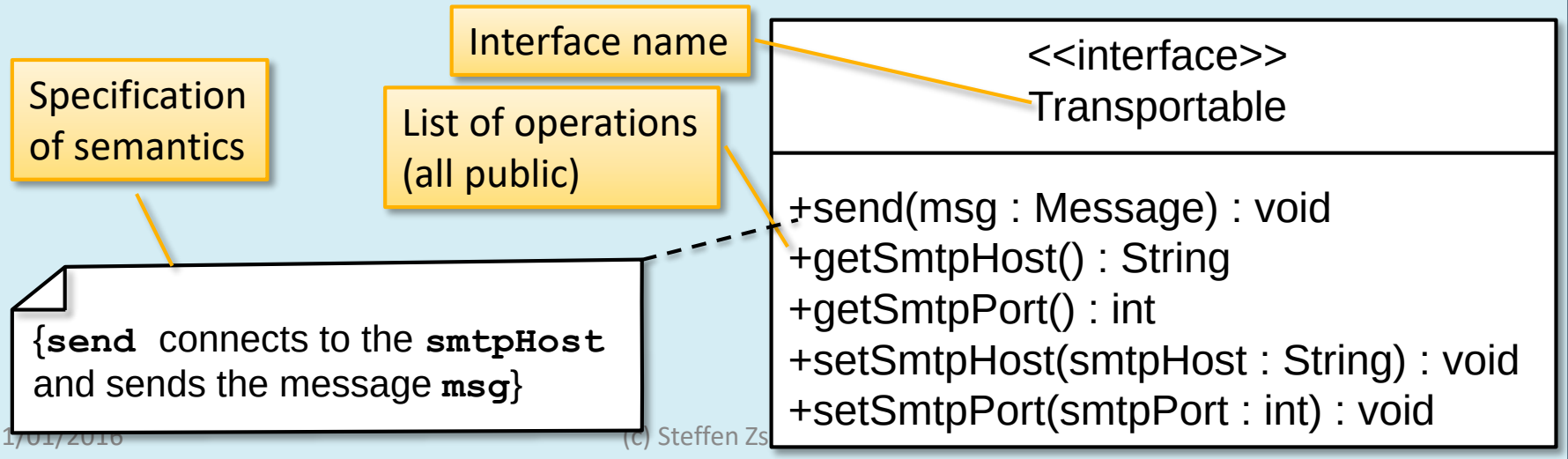
One component may have several provided interfaces; one interface may have several implementations

Component Interfaces (2)

- **Programmers often consider interfaces as signatures only**
  - As in Java
  - **But semantics are vital!**
  - Can be given in English as informal documentation
  - In UML 2, may give semantics via informal English or OCL logical constraints
  - See later for formal semantics

Interfaces in UML2

- **Two ways**
  - Component usage: lollipop and socket notation
  - Detailed signature and specification
    - Like UML classes, but with **`<<interface>>`** keyword on top
    - **Always** provide detailed interface specifications before referring to interface name in component diagrams



UML Syntax for Operations Reminder

- **For detailed UML interface syntax**

```
visibility name(parameter-list) : return-type-expression
```

```
+ assignAgent(a : Agent) : boolean
```

- **Visibility:**
  - public (+), protected (#), private (-)
  - Always public for interfaces

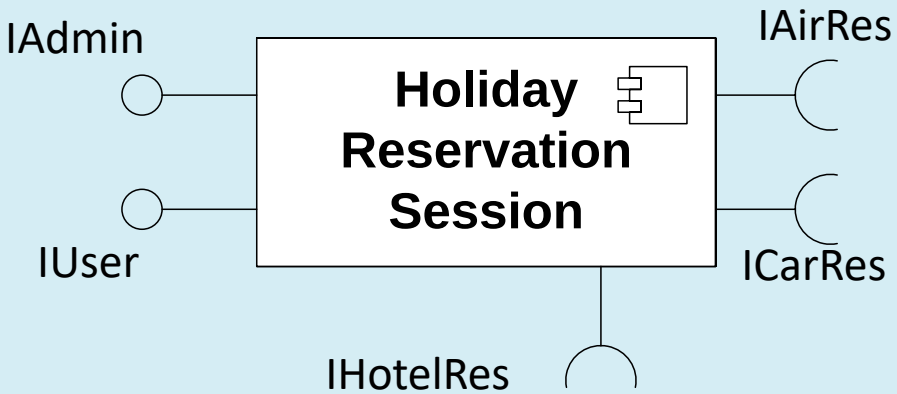
Provided & Required Interfaces

- **Components expose two kinds of interface**
  - *Provided interfaces* tell clients how to use component
  - *Required interfaces* specify required functionality of external components

- **In Java**
  - Provided interfaces are explicit
    - Given via **`implements`** keyword
  - Required interfaces are implicit
    - Defined by whatever interface types are used by the class

Example UML2 Component

- **Holiday reservation component**
  - Two different kinds of clients
    - Ordinary user, system administrator
    - Two provided interfaces, one for each client
  - Uses three different components
    - Hotel reservation, Car reservation, and Air reservation
    - Three required interfaces, one for each required component



Implementing Provided & Required Interfaces

Provided functionality given by implementing interfaces

```
public class HolidayResSession implements IUser, IAdmin {
  IHotelRes myHotelRes;
  ICarRes myCarRes;
  IAirRes myAirRes;

  public HolidayResSession (IHotelRes aHotelRes,
                            ICarRes aCarRes,
                            IAirRes aAirRes) {
    this.myHotelRes = aHotelRes;
    this.myCarRes = aCarRes;
    this.myAirRes = aAirRes;
  }

  ...
```

References to external components of required functionality typed by interfaces

Constructor assigns components of required interface types ("Inversion of Control")

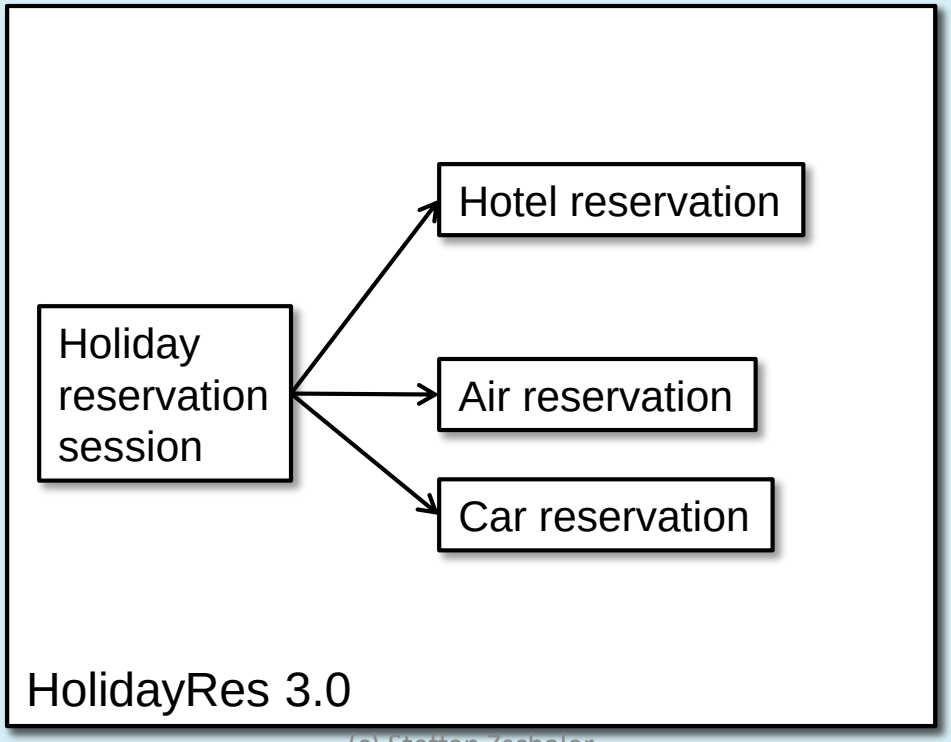
Inversion of Control: E.g., Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern, 2004
http://www.martinfowler.com/articles/injection.html
Safer/Cleaner implementation: ArchJava, see http://archjava.fluid.cs.cmu.edu/

Components as Substitutable Units

- **Substitutable unit**
  - Can be replaced at design or run time
    - By a component that offers equivalent functionality
    - Based on compatibility of its interfaces

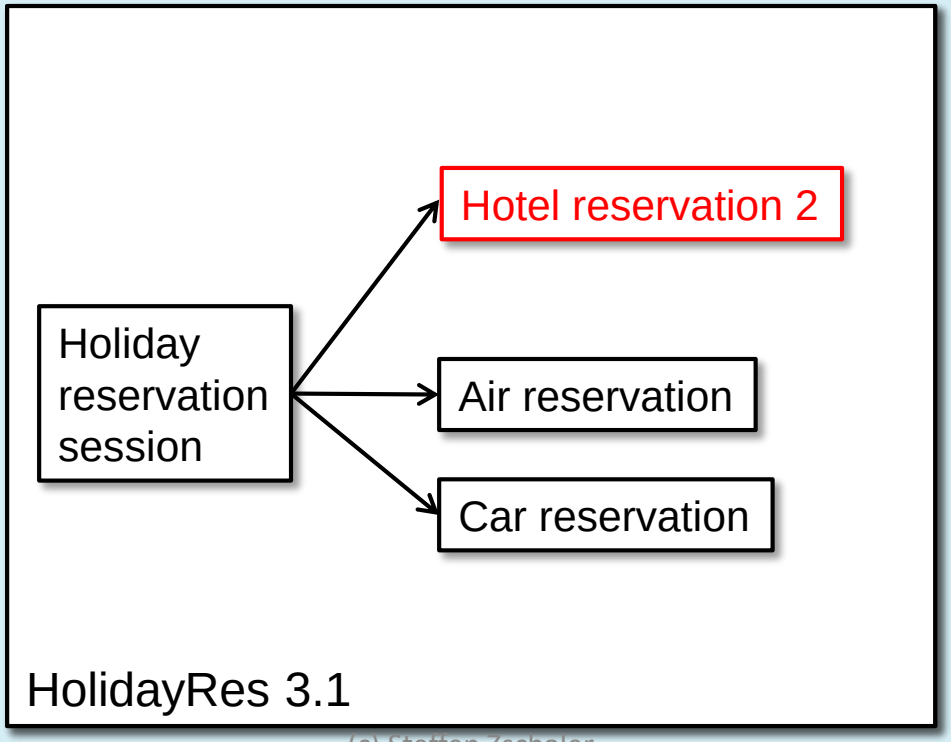
Components as Substitutable Units (2)

- **Can replace a component with an alternative or updated version**
  - Without breaking the systems in which the component is used



Components as Substitutable Units (2)

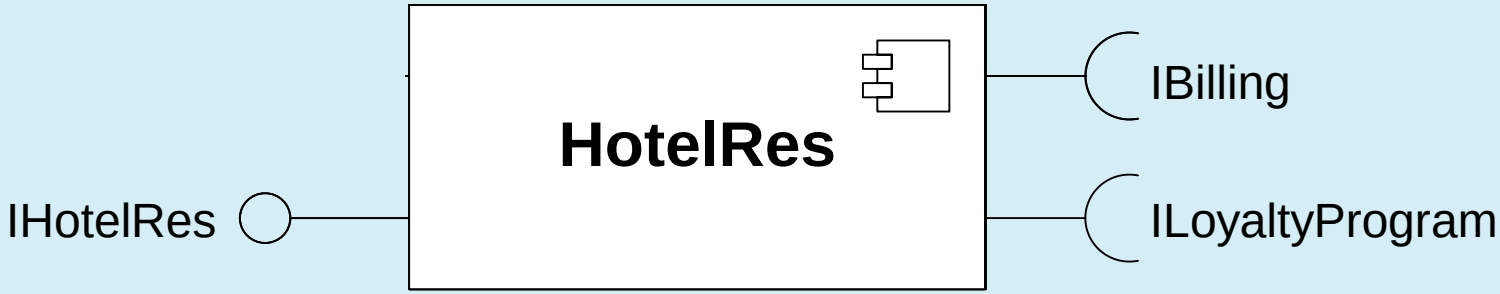
- **Can replace a component with an alternative or updated version**
  - Without breaking the systems in which the component is used



Laws of Substitutability

- **Component *B* can be substituted for component *A* if, and only if,**
  - *A*'s provided interfaces are preserved by *B*
    - Signature and expected behaviour of all operations of all provided interfaces are preserved
  - *B*'s required interfaces are the same as or fewer than *A*'s
    - *B* cannot require more functionalities than A did
- **Constraints must be satisfied to avoid system breakage**

Substitutability Example



- Add new interface for conference bookings
- Loyalty program no longer operational

Substitutability Example

- Added new interface for conference bookings
- Loyalty program no longer operational

Connectors

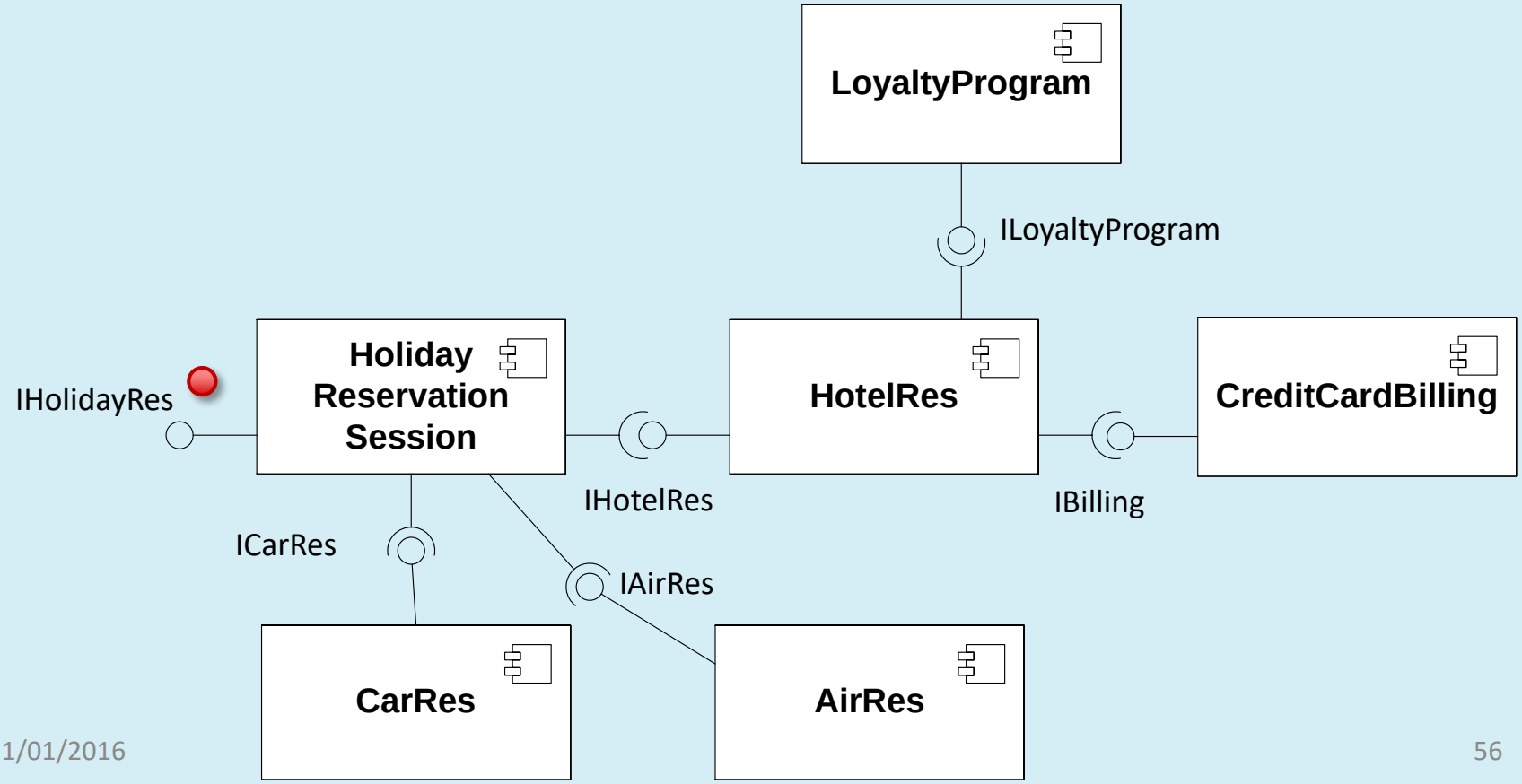
- **UML 2 provides two kinds of connectors**
  - Delegation connectors (later)
  - Assembly connectors (now)

Assembly Connectors

- **Connection between two components**
  - One provides services required by other
- **Defines communication between components**
  - Providing component = server to the requiring component
  - The requiring component calls methods of the provided interface

Connectors in UML2

- **Depicted by sticking the provided (lollypop) to the required (socket) interface**



Implementing Connectors *The Simple Way*

```
public class HolidayReservationSession implements IHolidayRes {
  IHotelRes myHotelRes;
  ICarRes myCarRes;
  IAirRes myAirRes;  /* These are the supplier interfaces */

  public HolidayReservationSession(IHotelRes aHotelRes,
                                   ICarRes aCarRes,
                                   IAirRes aAirRes) {
    this.myHotelRes = aHotelRes;
    this.myCarRes = aCarRes;
    this.myAirRes = aAirRes;  /* Supplier components instantiated */
  }
  ...
}
...
new HolidayReservationSession(new HotelRes(new CreditCardBilling(),
                                           new LoyaltyProgram()),
                              new CarRes(),
                              new AirRes());
```

Simple Architectures in UML2

- **Now you can design simple architectures in UML2**
- **Simple architectures consist of**
  - Detailed interface specifications
  - Components with provided and required interfaces
  - Connections between components
- **Complex architectures involve**
  - Compound components (components that have other components inside them)
  - Other kinds of connection
- **Simple architectural views sufficient for many medium-scale systems**
  - So you've learnt a lot today!

Coming Up

- **A simple process for architecture design**
- **Compound components and complex architectures**
- **Implementation of components and component architectures**
  - Beyond basic Java classes
  - Complex forms of communication
  - Domain-specific modelling issues