

# 11. Transparency Problems and the Decorator-Connector Pattern

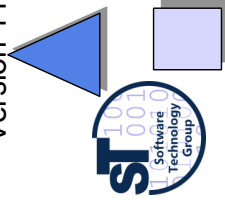
A Design Pattern that appears in all classical component systems

Prof. Dr. Uwe Alßmann

Technische Universität Dresden  
Institut für Software- und  
Multimediatechnik

<http://st.inf.tu-dresden.de>

Version 11-0.2, April 27, 2011



1. Transparency Problems
  1. Language Transparency
  2. Decorator Pattern
2. Decorator-Connector Pattern
3. Interface Definition Language
4. Location Transparency
5. Example YP Service
6. Name Transparency and Trading

CBSE, © Prof. Uwe Alßmann

1

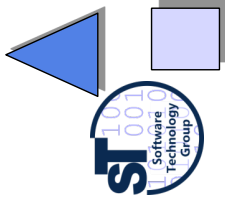
## Service-Oriented Architecture

- When the Object Management Group (OMG) was formed in 1989, **interoperability** was its founders primary, and almost their sole, objective:

*A vision of software components working smoothly together, without regard to details of any component's location, platform, operating system, programming language, or network hardware and software.*

Jon Siegel

# 11.1. Transparency Problems for COTS



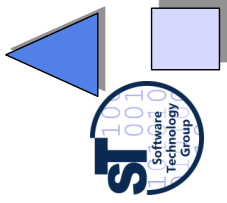
CBSE, © Prof. Uwe Alsmann

3

## Transparency Problems (*Middleware Concerns*)

- ▶ **Language transparency:** interoperability of components
  - on the same platform using different programming languages
- ▶ **Location transparency:** distribution of programs
  - Hiding, where a program runs
- ▶ **Naming transparency:** naming of services
  - Hiding, how a service is called
- ▶ **Lifetime transparency**
  - Hide whether server has to be started
- ▶ **Persistence transparency**
  - Hide whether server has persistent memory
- ▶ **Transactional transparency**
  - Hide whether server is embedded in parallel actions
- ▶ **Security scaling**
  - Plug-in authentication

# 11.1.2 Language Transparency



CBSE, © Prof. Uwe Alsmann

5

## Language Transparency Problems

- ▶ **Calling concept**
  - Procedure, Co-routine, Messages, ...
- ▶ **Calling conventions**
  - Call by name, call by value, call by reference, ...
- ▶ **Calling implementation**
  - Parameters on the stack, in registers, allocation and de-allocation
- ▶ **Data types**
  - Value and reference objects
  - Arrays, union, enumerations, classes, (variant) records, ...
  - Kind of inheritance (co-variance, contra-variance, ...)
- ▶ **Data representation**
  - Coding, size, little or big endian, ...
  - Layout of composite data
- ▶ **Runtime environment**
  - Memory management, garbage collection, lifetime ...



Prof. U. Alsmann, CBSE

6



## Options In General

- ▶ Direct language mapping:
  - 1:1 adaptation of pairs of languages:  $O(n^2)$
- ▶ Mapping to common language:
  - Adaptation to a general exchange format:  $O(n)$
- ▶ Compiling to common type system:
  - Standardize to a single format (like in .NET):  $O(1)$  but very restrictive, because the languages become very similar



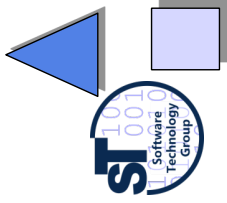
## Solutions in Classical Component Systems

- ▶ Calling concept:
  - standardized by the communication library (RPC)
- ▶ Calling conventions:
  - Standardized by the communication library (EJB - Java , DCOM - C)
  - Implementation for every single language (Corba)
- ▶ Calling implementation:
  - Standardized by the communication library (EJB - Java , DCOM - C)
  - Implementation for every single language (Corba)
- ▶ Data types:
  - Standard (EJB – Java types)
  - Adaptation to a general exchange format (IDL)
- ▶ Data representation:
  - Standard (EJB – Java representation, DCOM – binary standard)
  - Adaptation to a general format (IDL 2 Language mapping)
- ▶ Runtime environment
  - Standard by services of the component systems



# 11.1.3 The Decorator Design Pattern

(Repetition from DPF in winter)

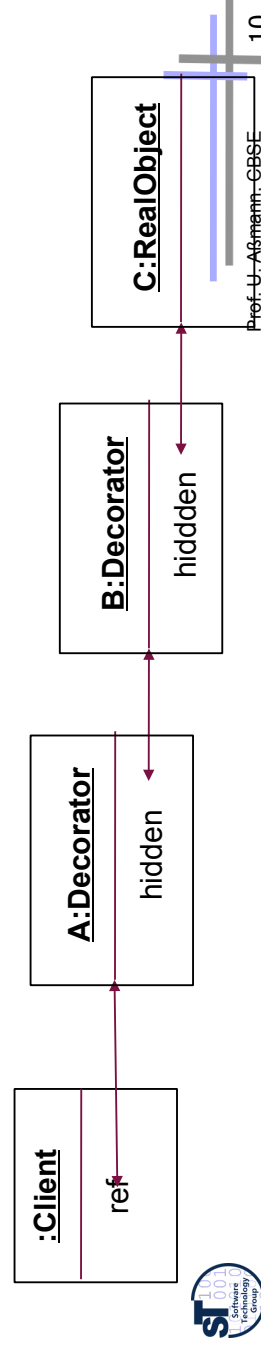


CBSE, © Prof. Uwe Alsmann

9

## Decorator Pattern

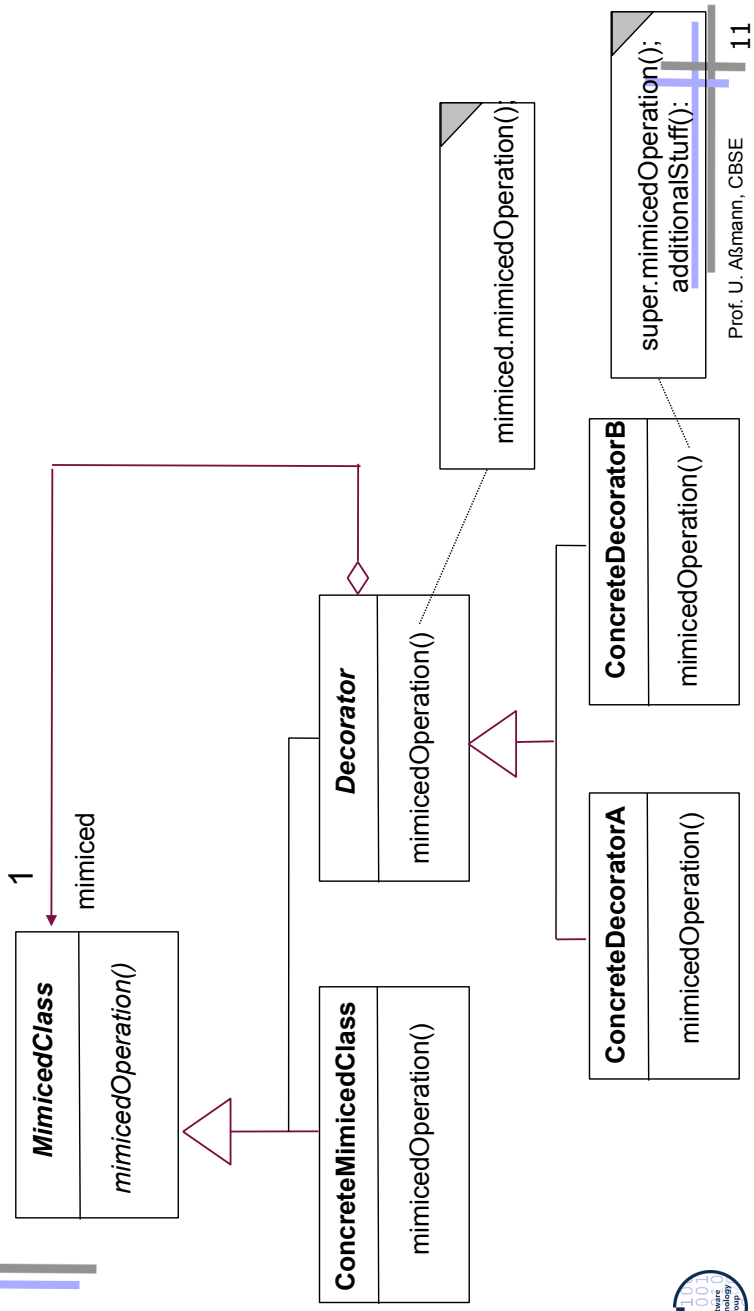
- ▶ A Decorator is a *skin* of another object
- ▶ It is a 1-ObjectRecursion (i.e., a restricted Composite):
  - A subclass of a class that contains an object of the class as child
  - However, only one composite (i.e., a delegatee)
- ▶ Combines inheritance with aggregation
  - Inheritance from an abstract Handler class
  - That defines a contract for the mimiced class and the mimicing class



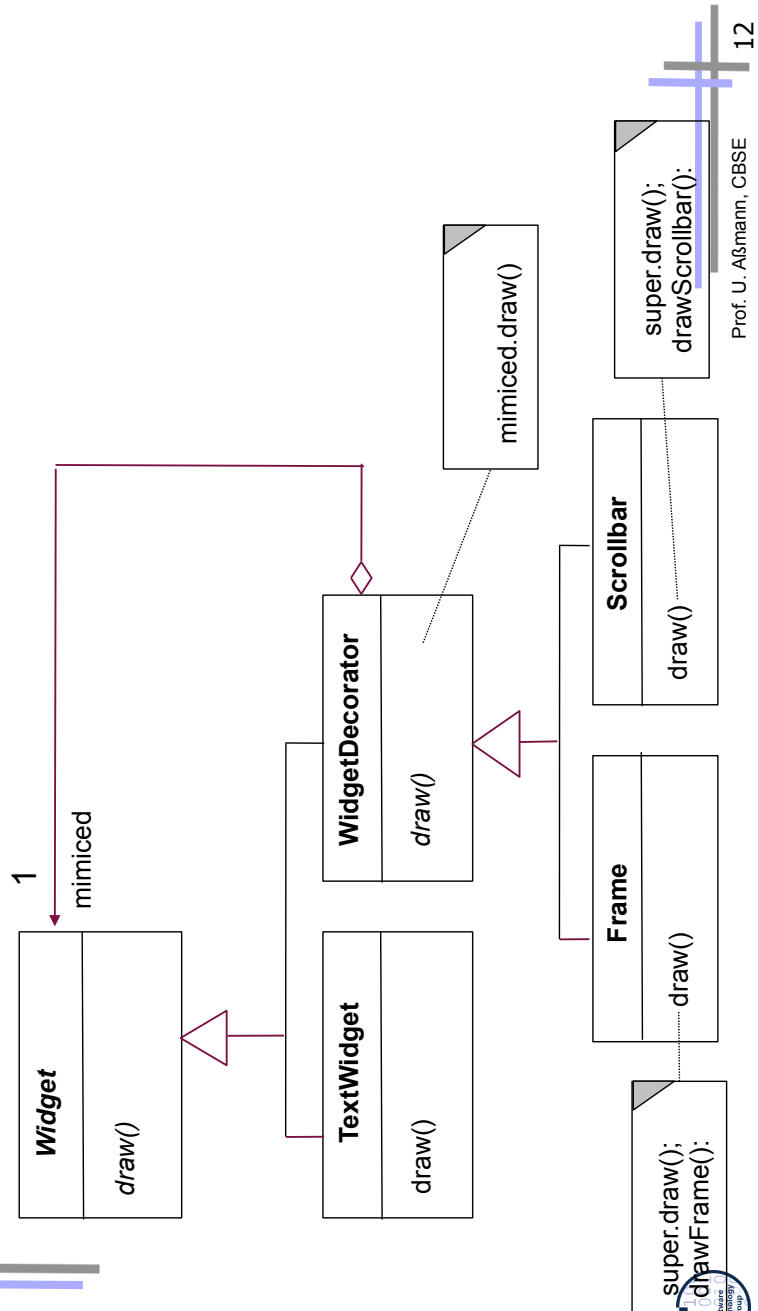
Prof. U.-Alsmann, CBSE

10

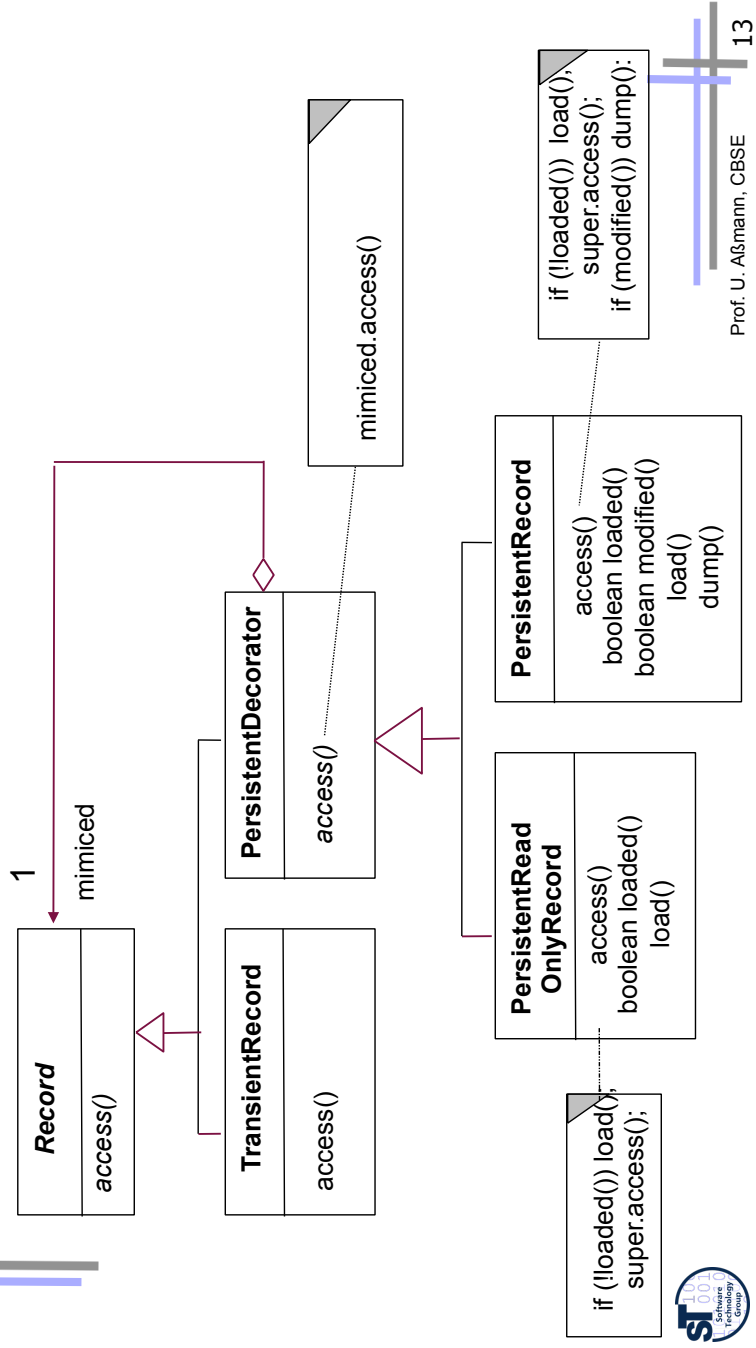
# Decorator – Structure Diagram



# Example: Decorator for Widgets



# Decorator for Persistent Objects

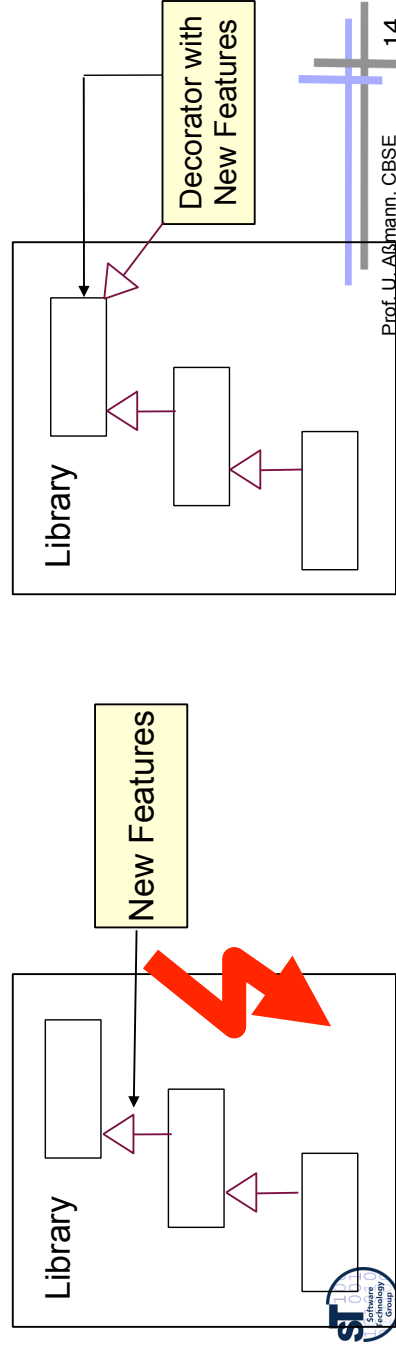


Prof. U. Alsmann, CBSE

13

# Purpose Decorator

- ▶ For extensible objects (i.e., decorating objects)
  - Extension of new features at runtime
  - Removal possible
- ▶ Instead of putting the extension into the inheritance hierarchy
  - If that would become too complex
  - If that is not possible since it is hidden in a library

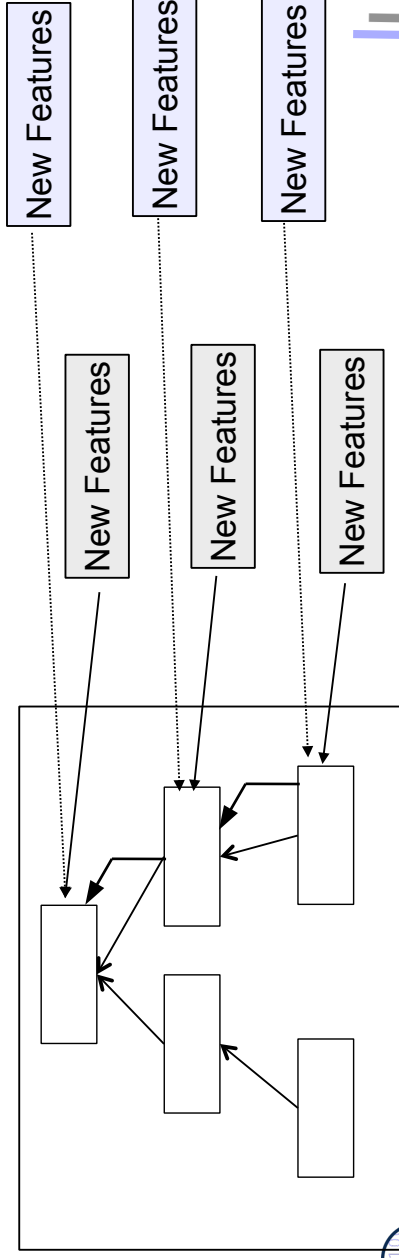


Prof. U. Alsmann, CBSE

14

## Variants of Decorators

- ▶ If only one extension is planned, the abstract super class Decorator can be omitted; a concrete decorator is sufficient
- ▶ Decorator family: If several decorators decorate a hierarchy, they can follow a common style and can be exchanged together
- ▶ Decorators can be chained to each other
- ▶ Dynamically, arbitrarily many new features can be added

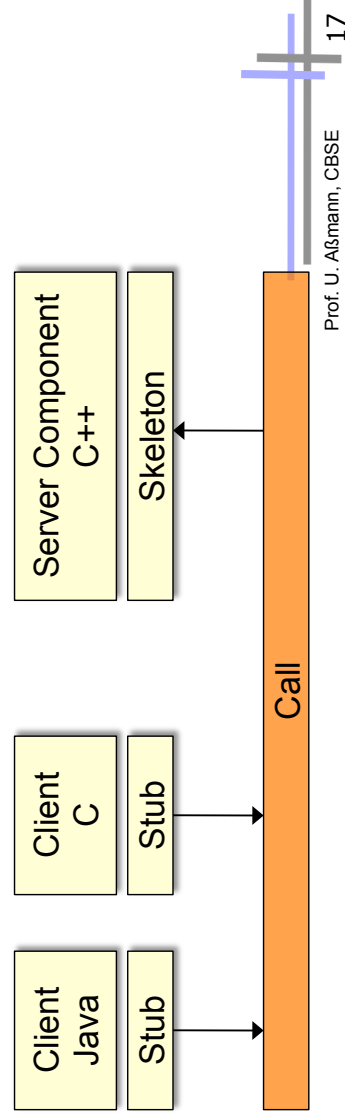


## 11.2 The Decorator-Connector Pattern



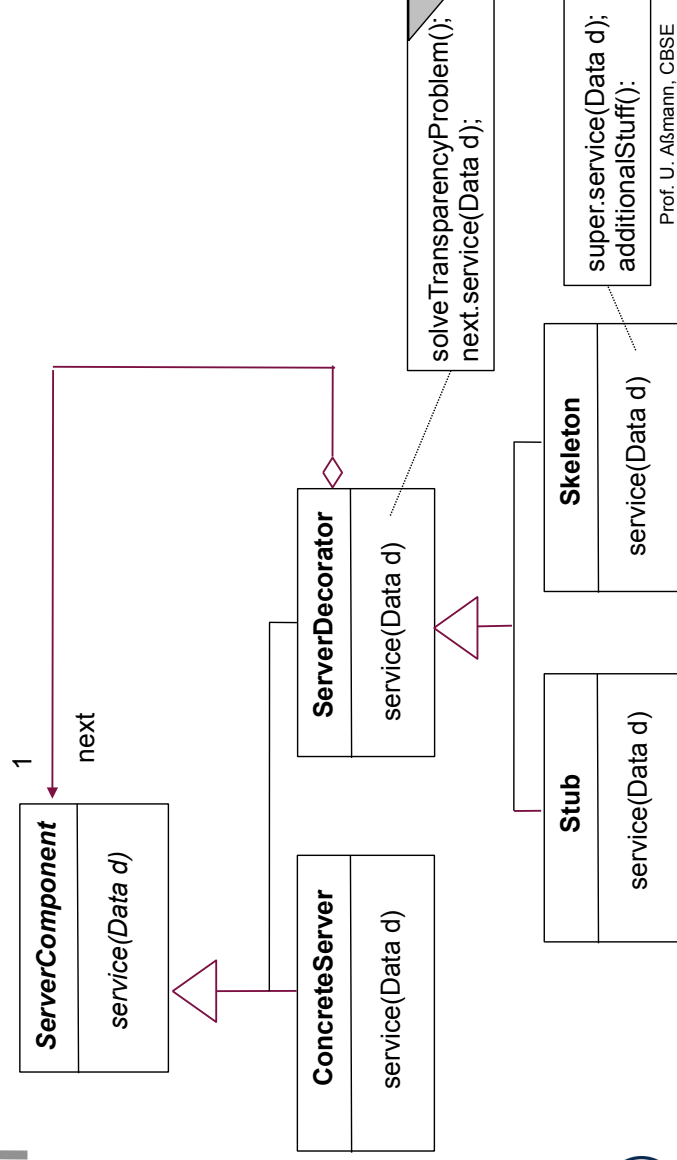
# Language Transparency With the Connector Pattern

- ▶ Connector Pattern (aka Stubs and Skeletons, Double-Decorator Pattern,  $n$ -Decorator Pattern):
  - Stub: Proxy of the client (decorator of the skeleton)
    - Takes calls of clients in language  $\mathcal{A}$  and sends them to the skeleton
  - Skeleton: Proxy (decorator) of the server
    - Takes those calls and sends the component implementation in language  $\mathcal{B}$
- ▶ Language adaptation in Stub or Skeleton (or both)
  - Adaptation deals with calling concepts, etc. (see above)
  - Based on a mapping of language constructs from both languages, defined by an Interface Definition Language (IDL)



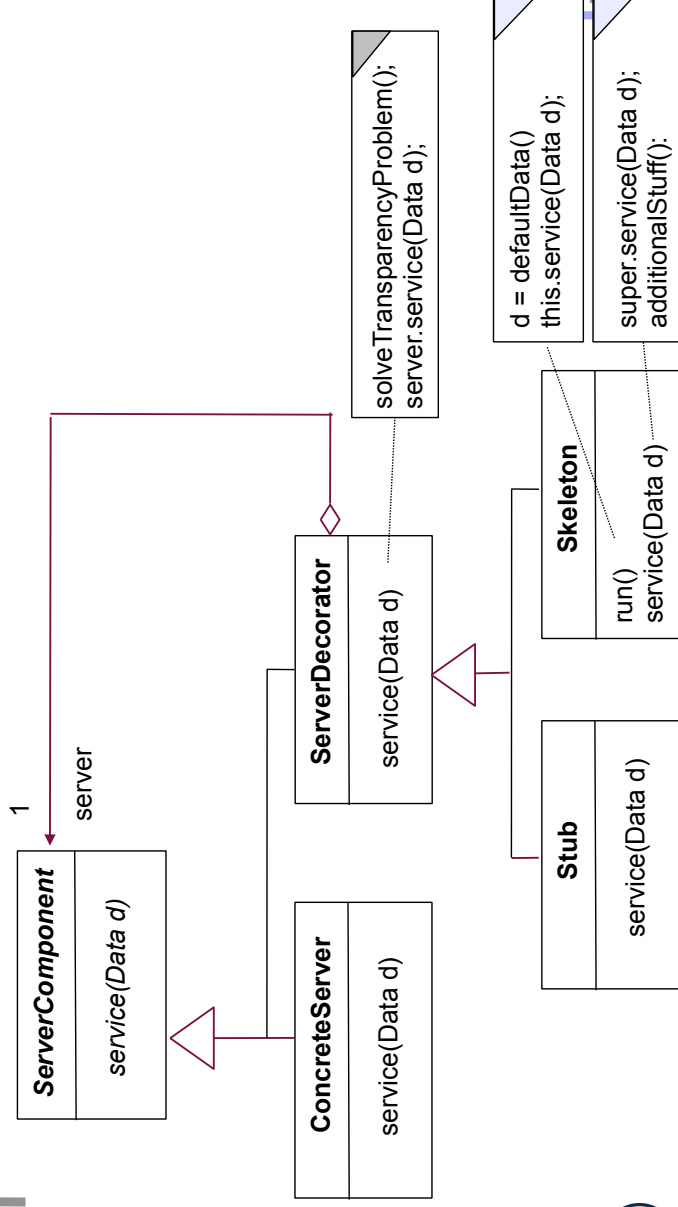
# Stubs and (Static) Skeletons

- ▶ A typical instance of the proxy or decorator pattern: two proxies on client and server
- ▶ Stub decorates skeleton, skeleton decorates server



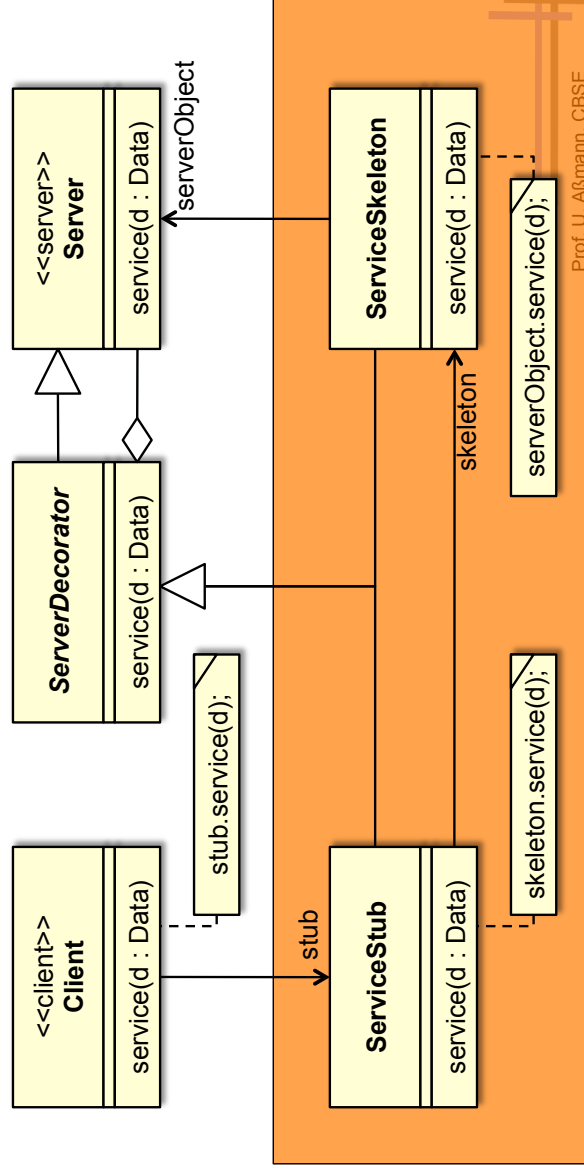
## Stubs and Dynamic Skeletons

- ▶ Skeleton contains a generic method `run()`
- Skeleton is an *Adapter*, mapping `run` to server method `service(Data)`



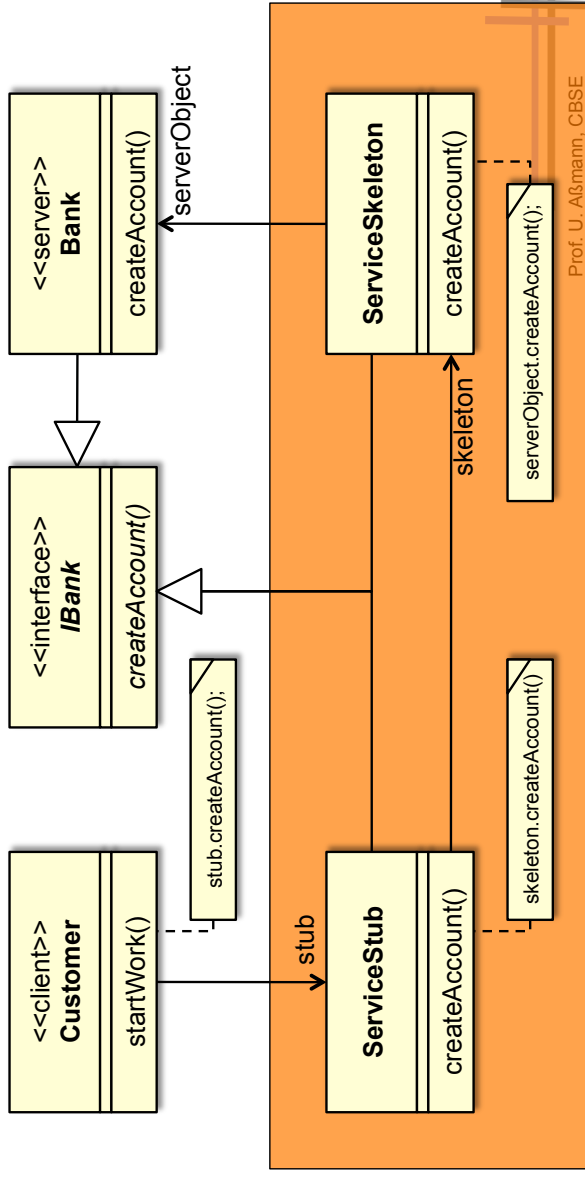
## The Connector Pattern (Alt. 1, with Decorator)

- ▶ Client and server are connected via a layer of stubs and skeletons (the *connector*)
- ▶ The connector consists of two decorators of the server
- ▶ Decorator chain is inherited



## A Connector with Server Interface (Alt. 2, with Abstract Superclass)

- ▶ Client and server are connected via a layer of stubs and skeletons (the *connector*)
- ▶ Server, Stubs and Skeletons inherit from same interface

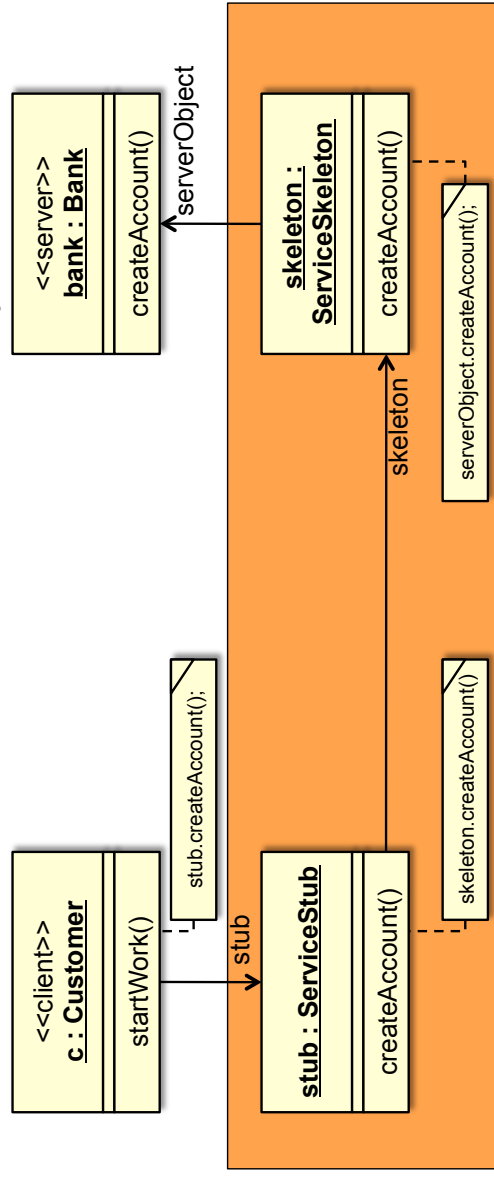


Prof. U. Alsmann, CBSE

21

## Object Diagram of Decorator-Connector Pattern

- ▶ Connector consists of a Decorator chain, in a layer

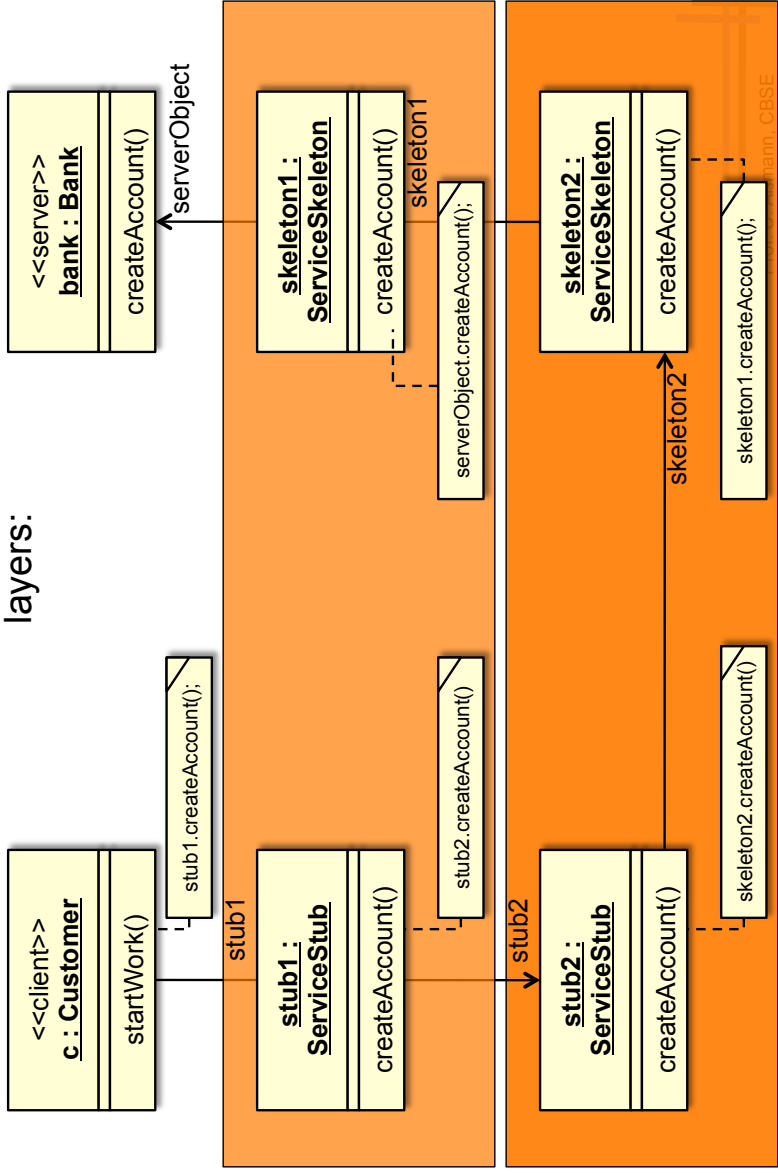


Prof. U. Alsmann, CBSE

22

# Layered Decorators

- ▶ More decorators can be stuffed into the connector in additional layers:



23

# Decorator vs Proxy vs Adapters vs Chain

- ▶ Why is it a Decorator?
  - Decorators allow for stacking of connectors (layering)
  - Proxy pattern: just one representative, no stacking possible
    - However, from the client and server's perspective, stub and skeletons are Proxies
  - Adapter: Adapted interface must be different from Adaptee
  - Chain: In a Chain, the processing may stop (not here..)
- ▶ However, Connectors can use all other basic "representant" patterns
  - Adapter-Connector: adapts required interface to server additionally
  - Chain-Connector: may stop processing
  - Proxy-Connector: just one layer possible



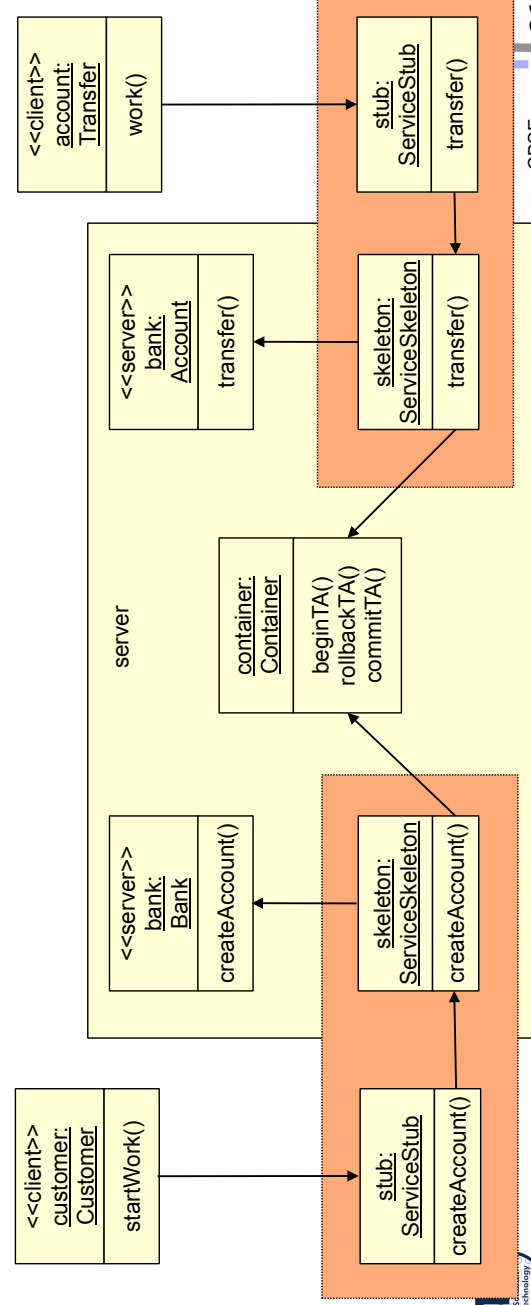
## Tasks of the Layers

- ▶ In a component model, every layer of decorator-pairs is devoted to a specific task for *transparency* (*middleware concern*)
  - Language mappings (language interoperability)
  - Distribution handling (serialization, deserialization)
  - Names (name mapping, name search)
  - Persistence
  - Transactions
  - etc.
- ▶ Layers can be composed (stacked) freely



## Containers – Infrastructure for all Connectors

- ▶ A **container** of a server component is an infrastructure for *all* connectors at run-time (all decorators/proxies).
  - Creation (server component factories for service families)
  - Transactions (begin, rollback, commit)
  - Persistence (activate, passivate)





## Who Realizes Stubs and Skeletons?

### ▶ Programmer

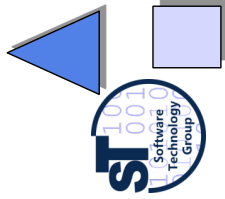
- Much handcrafting, using Decorator pattern. Boring and error prone

### ▶ Generator:

- Stub
    - Export interface is component dependent, independent of source language
    - Implementation is source language dependent
  - Skeleton
    - Import interface is component dependent, independent of source language
    - Implementation is target language dependent
- ▶ Idea: Generate export and import interfaces of Stub and Skeleton out of a component interface definition
- Take generic language adapter for the implementation

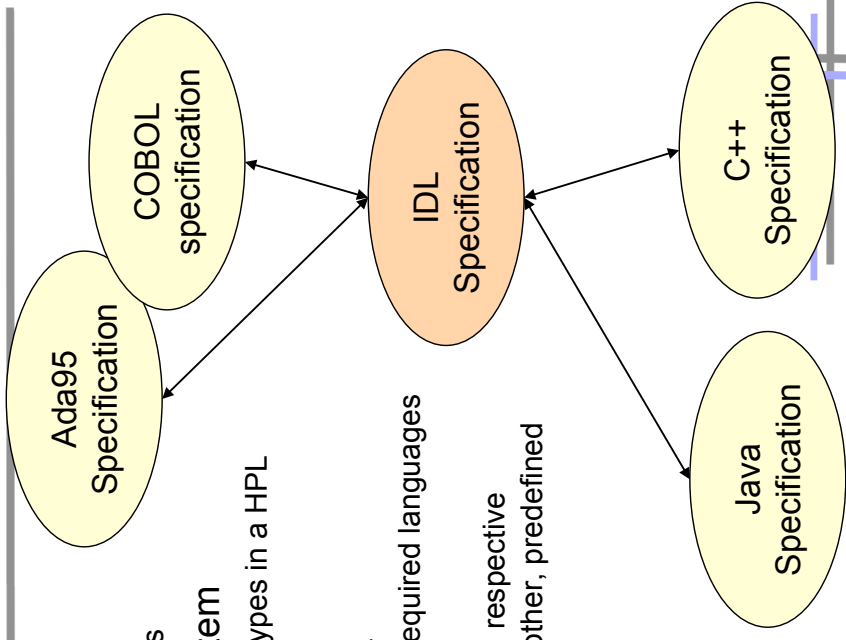


## 11.3 Interface Definition Languages for Mapping Different Languages



## Type Mapping with the Interface Definition Language (IDL)

- ▶ Language to define the
  - Interfaces of components
  - Data types of parameters and results
- ▶ Language independent type system
  - General enough to capture all data types in a HPL
- ▶ Procedure of construction
  - Define component interface with IDL
  - Generate stubs and skeletons with required languages using an IDL compiler
  - Implement the frame (component) in respective language (if possible reusing some other, predefined components)



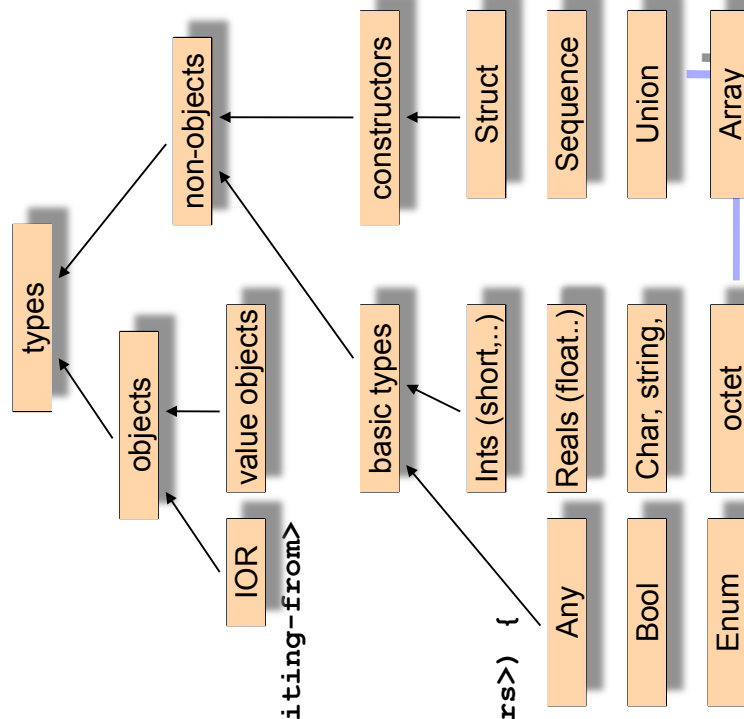
## Types in the Interface Definition Language

```

// IDL specification
modules <identifier> {
  <type declarations>
  <constant declarations>
  <exception declarations>
}

// classes
interface <identifier> : <inheriting-from>
{
  <type declarations>
  <constant declarations>
  <exception declarations>
  // methods
  optype <identifier>(<parameters>) {
    ...
  }
  ...
}

```





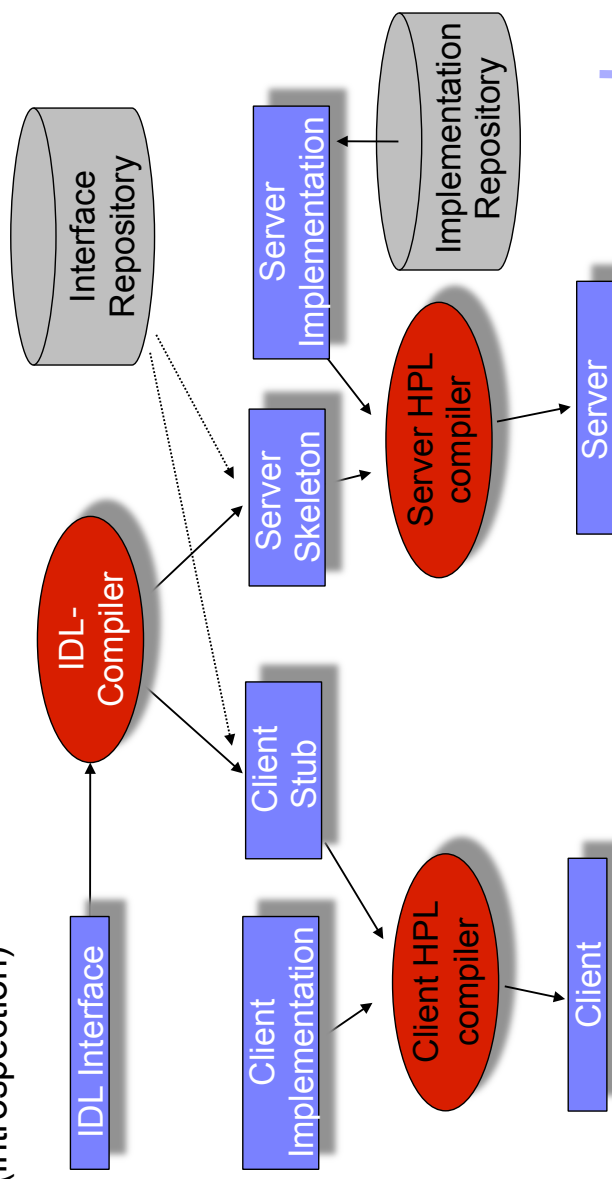
# Hello World IDL

```
module HelloWorld {  
    interface SimpleHelloWorld {  
        string sayHello();  
    };  
};
```



# Generation of Stubs and Skeletons from IDL

- ▶ Generation is done for every involved HPL
- ▶ Interface Repository is queried for component interfaces (introspection)





## Required Formal Properties of the IDL-To-Language Mapping

- Let  $\tau_{PL}: IDL \rightarrow TS_{PL}$  be the mapping from an interface definition language *IDL* to the type system *TS* of a programming language *PL*

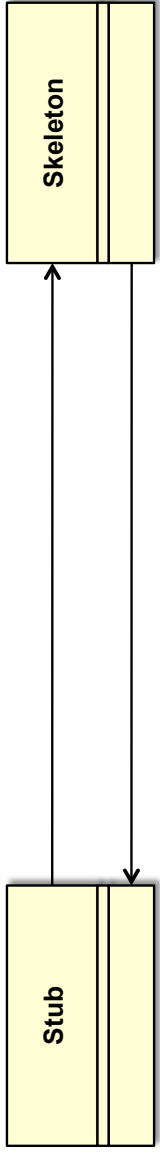
1. Well-definedness  
 $\forall PL : \tau_{PL}: IDL \rightarrow TS_{PL}$  is well defined
2. Completeness  
 $\forall PL : \tau_{PL}^{-1}: TS_{PL} \rightarrow IDL$  is well defined
3. Soundness

$$\forall PL : \tau_{PL}^{-1} \circ \tau_{PL}: IDL \rightarrow IDL \text{ is } \iota_{IDL}$$
$$\forall PL : \tau_{PL} \circ \tau_{PL}^{-1}: TS_{PL} \rightarrow TS_{PL} \text{ is } \iota_{PL}$$

## IDL Can Also Be Generated from Host Language

Specification of IDL and host language	Specification of host language
▶ Determined language binding, standardized IDL-to-Language mapping	▶ Retrieve the IDL out of the interface definitions (e.g., Java classes)
▶ Generation of stubs and skeleton is IDL compiler independent	▶ Have only one source of IDL compilers guaranteeing round-trip property of retrieval and generation for all languages
▶ Language specific IDL compilers	▶ Quasi standard
▶ OMG Corba	▶ Java, DCOM, .Net

# Stubs and Skeletons for Language Adaptation



Language 1

Map data to an exchange format

Call Skeleton

Language 2

Receive call from stub

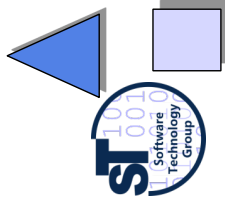
Retrieve data from the exchange format



Prof. U. Alsmann, CBSE

35

## 11.4 Location Transparency



CBSE, © Prof. Uwe Alsmann

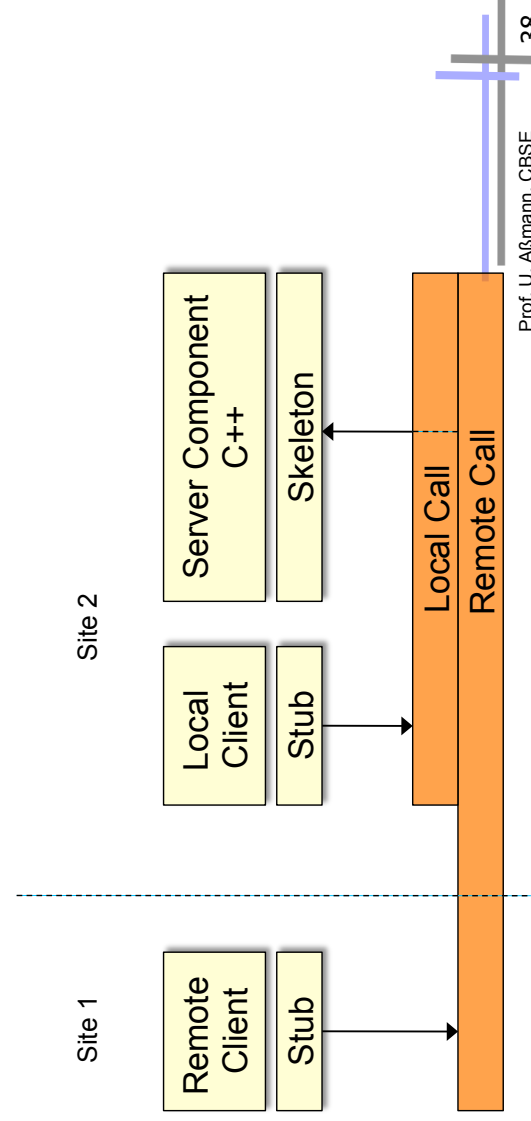
36

## Problem 2: Distribution

- ▶ *Location transparency*: interoperability of programs independently of their execution location
- ▶ Problems to solve
  - Transparent basic communication
    - Transparently initiate a local/remote call
    - Transparently transport data locally or remotely via a network
    - Transparent references
  - Distributed systems are heterogeneous
    - Platform transparent, concurrent execution?
    - So far we handled platform transparent design of components
  - Usual aspects in distributed systems
    - Transactions
    - Synchronization
    - ...

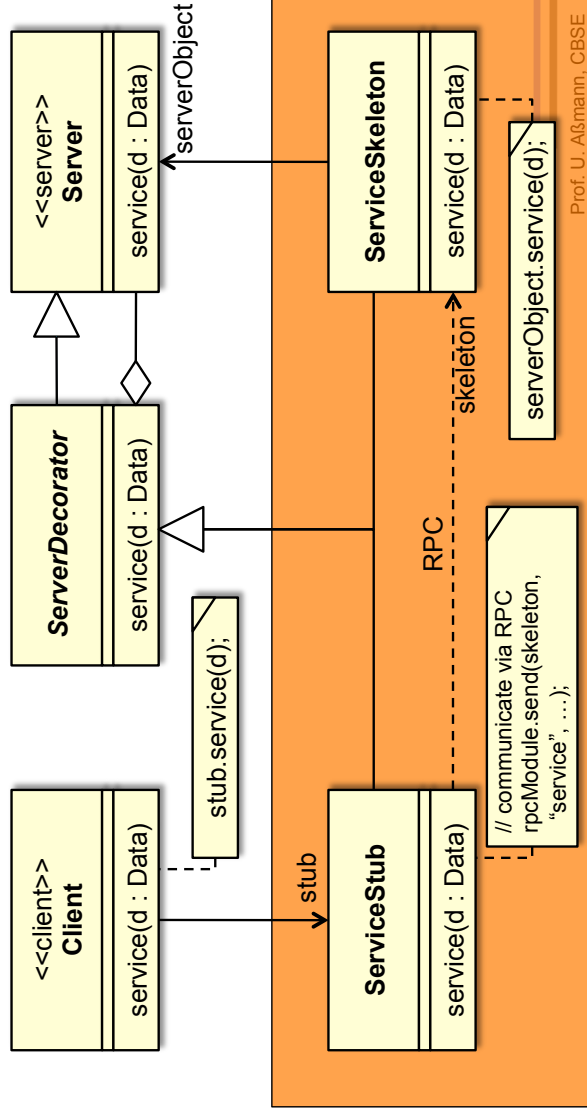
## Transparent Local/Remote Calls

- ▶ Communication over proxies/decorators
  - Proxies redirect call locally or remotely on demand
  - Proxies always local to the caller
- ▶ RPC for remote calls to a handler
  - Handler always local to the callee
- ▶ **Déjà vu! We reuse Stubs and Skeletons**

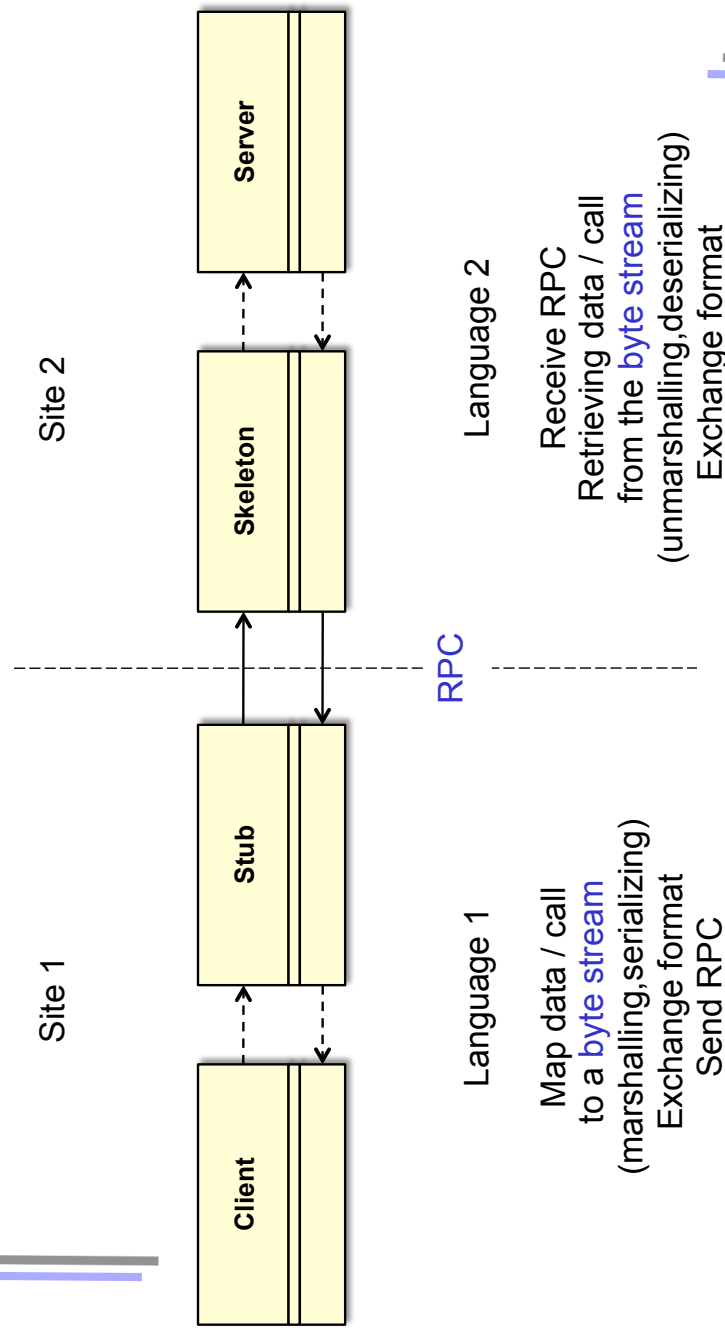


# Stubs and Skeletons for Distribution

- ▶ A variant of the Connector pattern, using remote procedure call (RPC) between the decorators

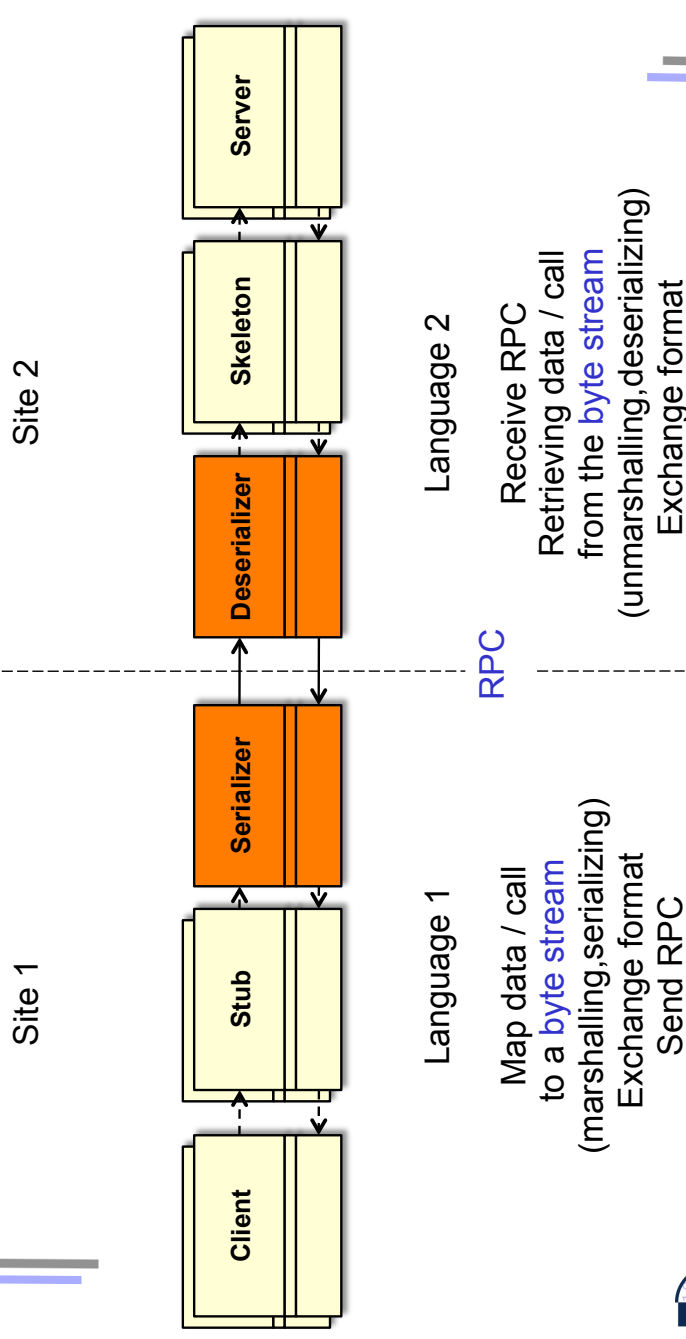


# Stubs and Skeletons for Distribution



## Stubs, Skeletons, and Serializers

- ▶ or with separate serializers/deserializers



## Problem 3: The Reference Problem (Name Transparency)

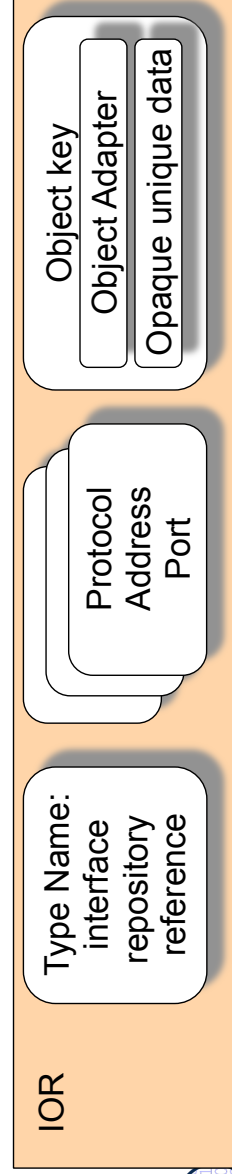
- ▶ How to reference something?
  - Target of calls (services)
  - Call by reference parameters and results
  - Reference data in composite parameters and results
- ▶ Scope of references
  - Thread/process
  - Computer
  - Agreed between communication partners
  - Net wide
- ▶ How to handle references transparently?

## Approach: Global Addresses

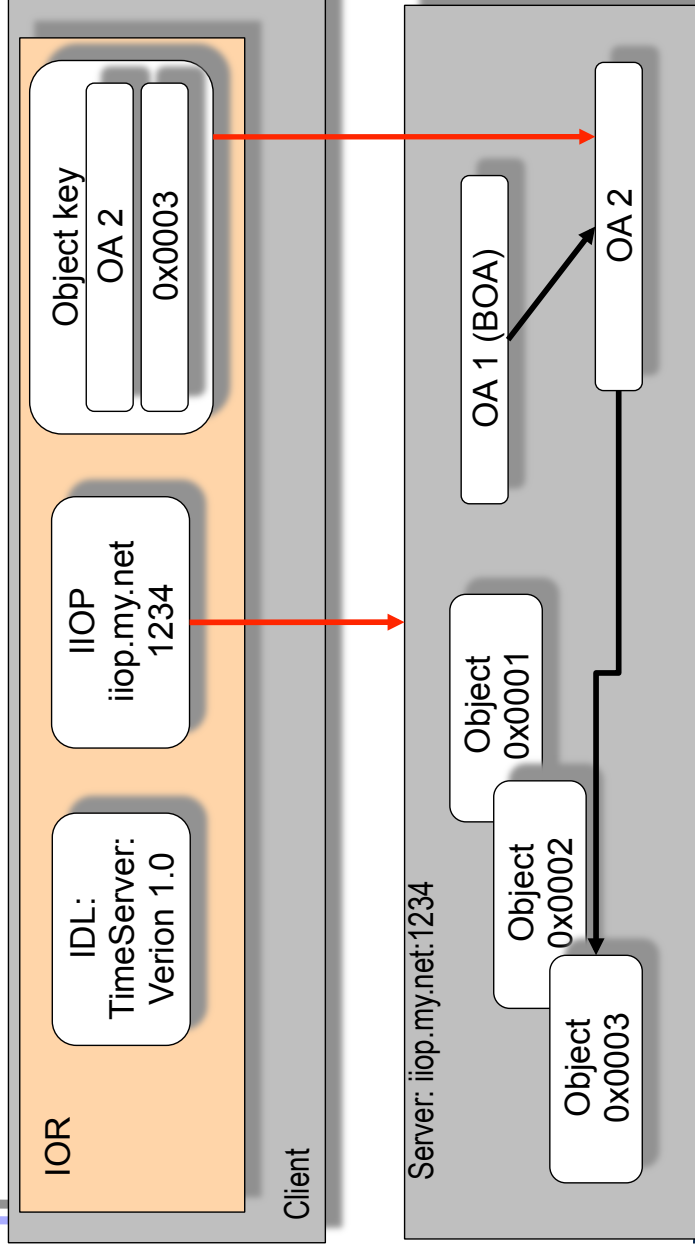
- ▶ World wide unique addresses
  - e.g., computer address + local address
  - URL, URI (Uniform Resource Identifiers)
  - CORBA IORs (Interoperable Object References)
  - AFS (Andrew File system) directory names
- ▶ Mapping tables for local references
  - Logical to physical
  - Consistent change of local references possible
- ▶ One server decorator per computer manages references
  - 1:n relation decorator to skeletons
  - 1:m relation skeletons to component objects
  - Lifecycle and garbage collection management
  - Identification (Who is this guy ...)
  - Authorization (Is he allowed to do this ...)

## Example: CORBA Interoperable Object Reference – IOR

- ▶ A unique key for an object
  - Uniquely mapped per language (for all ORBs)
  - Hides object references of programming languages
- ▶ Consists of:
  - Type name (code), i.e., index into Interface Repository
  - Protocol and address information (e.g., TCP/IP, port #, host name), could support more than one protocol
  - Object key:
    - Opaque data only readable by generating ORB (pointer)
    - Object decorator (adapter) name (for BOA)



## IOR Example



Prof. U. Alsmann, CBSE

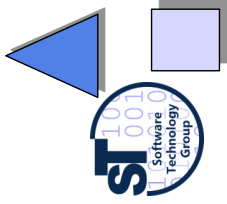
45

## Change of Local References

- ▶ **Why are you interested in a reference?**
  - Need a reference to computation service (function)
    - Sufficient to have a reference to the component
    - Decorator creates or hands out an arbitrary object instance on demand
  - Need a reference to store/retrieve data service
    - Use a data base
    - Decorator creates or hands out an arbitrary object instance wrapping the accesses to the data base
  - Need a reference for transaction to leave and resume
    - Decorator must keep correct mapping logical 2 physical address
    - Problems with use of self reference inside and outside service

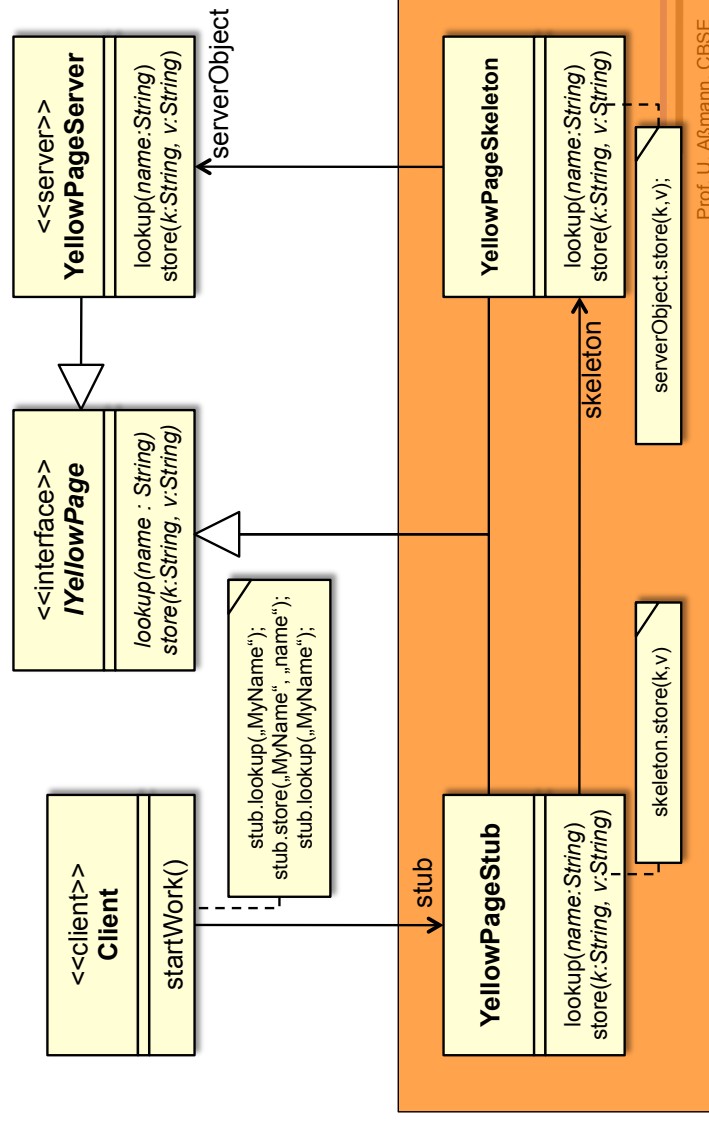
# 11.5 Example: A Remote Yellow Page Service

with remote access, serialization



## Remote Yellow Page Service

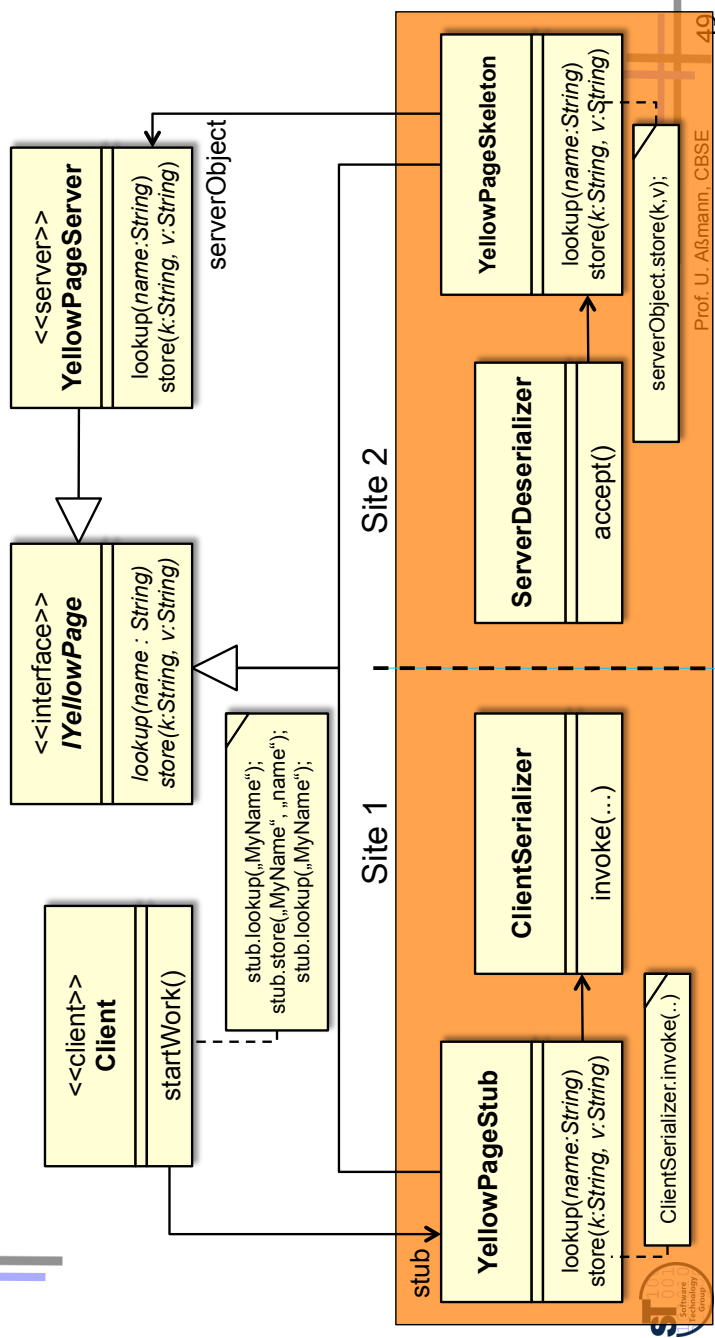
- Basic design without Serialization/Deserialization





# Remote Yellow Page Service

## With Serialization/Deserialization



# Service Interface

```
interface IYellowPageService {  
    String SERVICE_NAME = "Yellow Pages";  
    String lookup(String name);  
    void store(String name, String value);  
}
```

# Service Implementation

```
class YellowPageService implements IYellowPageService {
    private Hashtable<String,String> cache =
        new Hashtable<String,String>();

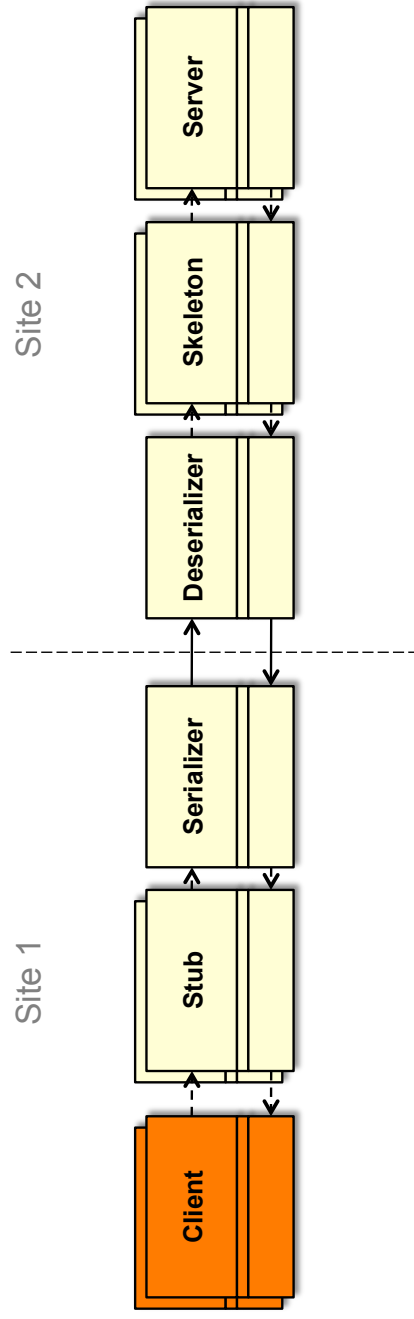
    private DataBasis db = ...;
    public String lookup(String name) {
        String res = cache.get(name);
        if (res == null)
            res = db.lookup(name);
            if (res != null) {
                cache.put(name,res);
            }
        return res;
    }

    public void store(String name, String value) {
        cache.put(name, value);
        db.store(name, value);
    }
}
```



# Client

- ▶ Wants to transparently use the Yellow Page service



## Example Client

- ▶ Client calls stub with service interface:

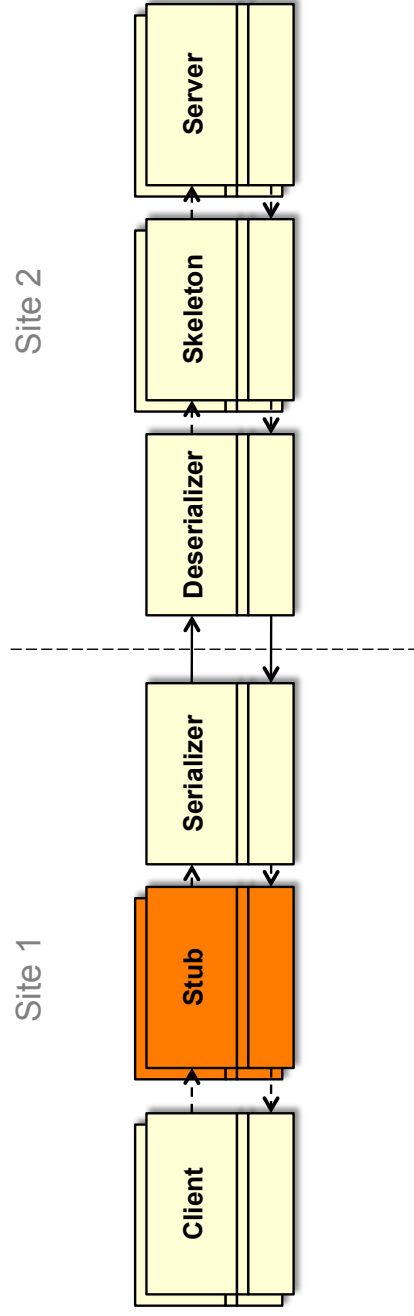
```
class Client {
...
// returns client stub
IYellowPageService yps =
    YellowPageFactory.create();
...
String res = yps.lookup("MyName");
...
}

class YellowPageFactory {
public IYellowPageService create() {
    return new YellowPageStub ();
}
}
```



## Stub (client side)

- ▶ Realizes 1:1 mapping of client to service component
- ▶ Uses 1:1 mapping of clients to stubs



## Example Client Stub - Implementation

```
class YellowPageStub implements IYellowPageService {
    private Integer logicalAddress = new Integer(-1);

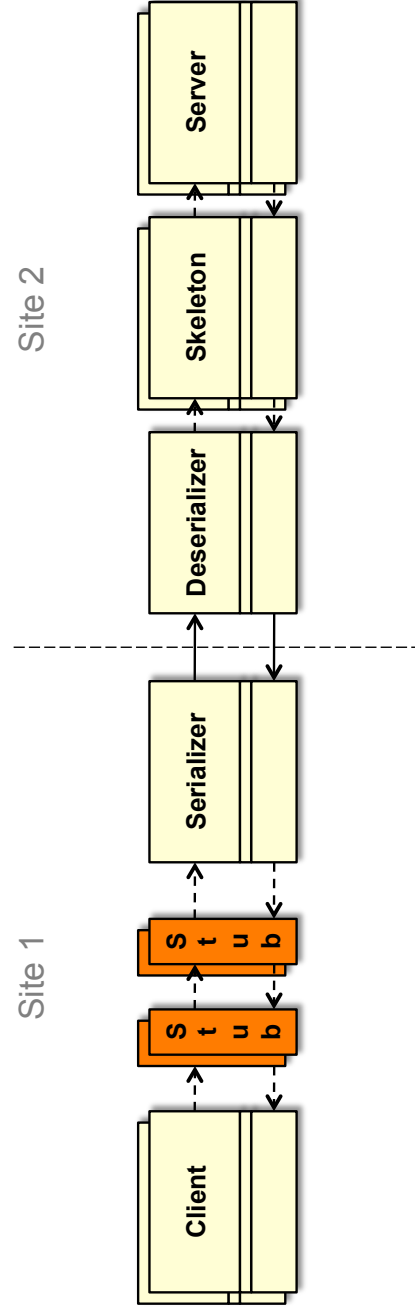
    public YellowPageStub () {
        logicalAddress = (Object) ClientSerializer.invoke(
            IYellowPageService.SERVICE_NAME, logicalAddress, "new", null);
    }

    public String lookup (String name) {
        Object res = ClientSerializer.invoke (IYellowPageService.SERVICE_NAME,
            logicalAddress, "lookup", new Object[] {name});
        return (String)res;
    }

    public void store (String name, String value) {
        ClientSerializer.invoke (IYellowPageService.SERVICE_NAME,
            logicalAddress, "store", new Object[] { name, value });
    }
}
```

## Scenario with Second Stub (client site)

- ▶ By using the Decorator pattern, stubs can be stacked onto each other
- ▶ Every stub solves another transparency problem (middleware concern)



# Client Stub 1 – This Time with Decorator Chain Implementation

```
// new stub: encryption decorator
class YellowPageStubEncryption implements IYellowPageService {
    private IYellowPageService clientDec;

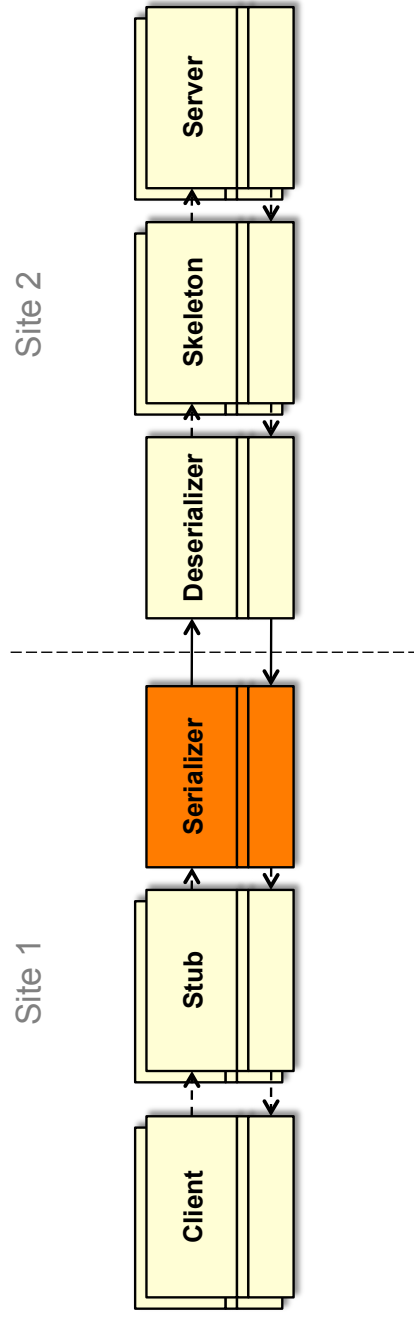
    // Security: encryption, decryption
    private String encrypt(String name);
    private String decrypt(String name);

    // client-side constructor
    public YellowPageStubEncryption() {
        clientDec = new YellowPageStub();
    }
    // lookup function, with encryption, decryption
    public String lookup(String name) {
        String res = clientDec.lookup(encrypt(name));
        return decrypt(res);
    }

    // store
    // ...
}
```

# Client-side Serializer

- ▶ Manages the basic communication on client side
- ▶ Is called from the client stubs
- ▶ Can be hidden in a Decorator (1:1), but can be also shared by all stubs





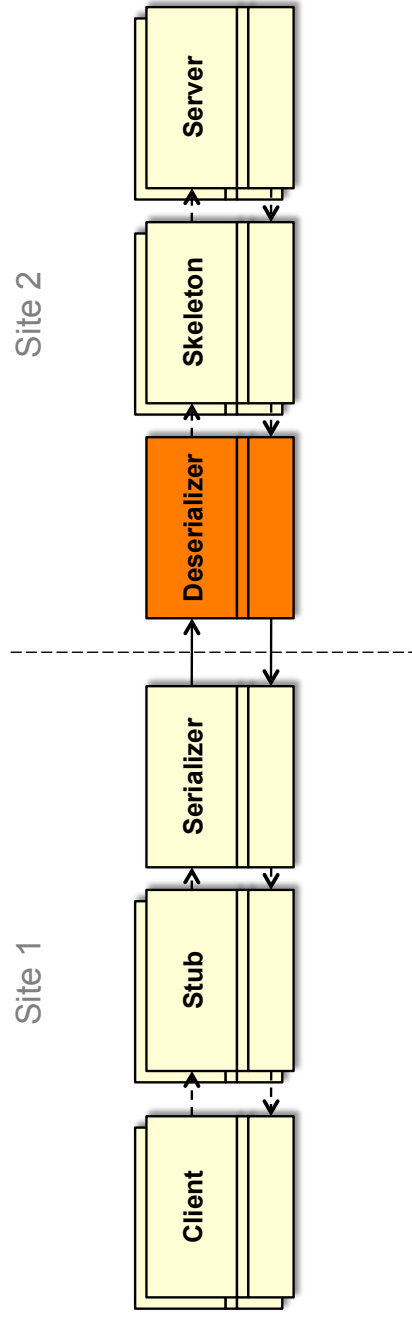
## Example Client Serializer

```
class ClientSerializer {  
    public static Object invoke(String service, Integer address,  
                               String method, Object[] args) {  
        Socket s = new Socket("yp-st.inf.tu-dresden.de", 1234);  
        ObjectOutputStream os = new ObjectOutputStream(s.getOutputStream());  
        ObjectInputStream is = new ObjectInputStream(s.getInputStream());  
        os.writeObject(service);  
        os.writeObject(address);  
        os.writeObject(method);  
        if (args != null) {  
            os.writeObject(args);  
        }  
        os.flush();  
        Object result = is.readObject();  
        s.close();  
        return result;  
    }  
}
```



## Server-side Deserializer

- ▶ Manages the basic communication on server side
- ▶ Calls the service skeletons (1:n mapping)





## Example Server Deserializer (1)

- ▶ Deserializer listens on the network is shared between different services
  - interprets incoming service names
  - can create/invoke several service skeletons (yellow page, phone book, ...)
  - lives always, but hides lifetime of the server

```
class ServiceDeserializer {
    public void run() {
        ServerSocket server = new ServerSocket(1234);
        while (true) {
            Socket client = server.accept();
            ObjectInputStream is = new ObjectInputStream(client.getInputStream());
            ObjectOutputStream os = new ObjectOutputStream(client.getOutputStream());
            String service = (String) is.readObject();
            if (service.equals(IYellowPageService.SERVICE_NAME)) {
                handleYellowPage(os, is);
            } else if (service.equals(IPhoneBook.SERVICE_NAME)) {
                handlePhoneBook(os, is);
            } else {
                System.err.println("Unknown service.");
            }
        }
    }
}
```



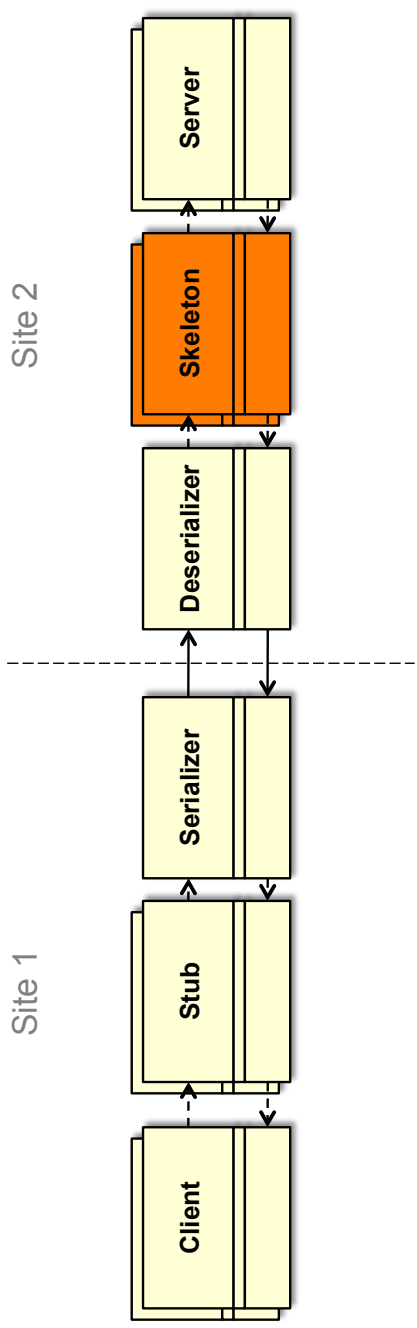
## Example Server Deserializer (2)

```
private void handleYellowPage(ObjectOutputStream os, ObjectInputStream is) {
    Integer address = (Integer) is.readObject();
    if (address == -1) { // creation of the service
        YellowPageSkeleton skeleton = new YellowPageSkeleton();
        os.writeObject(skeleton.getLogicalAddress());
    } else { // service query: interpretation of the symbolic service name
        IYellowPageService yp = new YellowPageSkeleton(address);
        String method = (String) is.readObject();
        Object[] args = (Object[]) is.readObject();
        if (method.equals("lookup")) {
            String res = yp.lookup((String)args[0]); // finally: call the service
            os.writeObject(res);
        } else if (method.equals("store")) {
            yp.store((String)args[0], (String)args[1]);
            os.writeObject(null);
        } else
            System.err.println("Unknown service method.");
    }
    os.flush();
}
```



## Skeleton (Server side)

- ▶ Manages service components of server on server side
- ▶ 1:1 mapping to service component

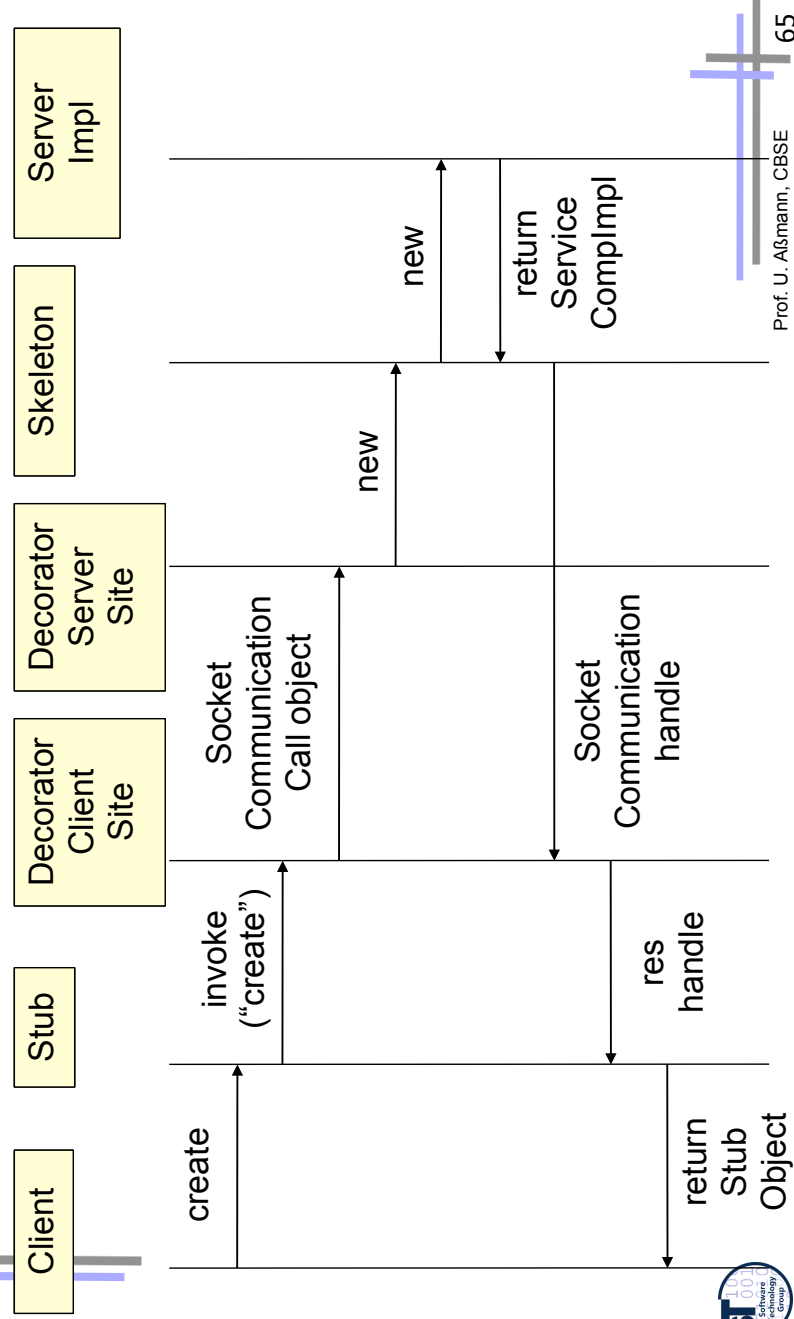


## Example Yellow Pages Server Skeleton (Service Lookup and Call, Adapter)

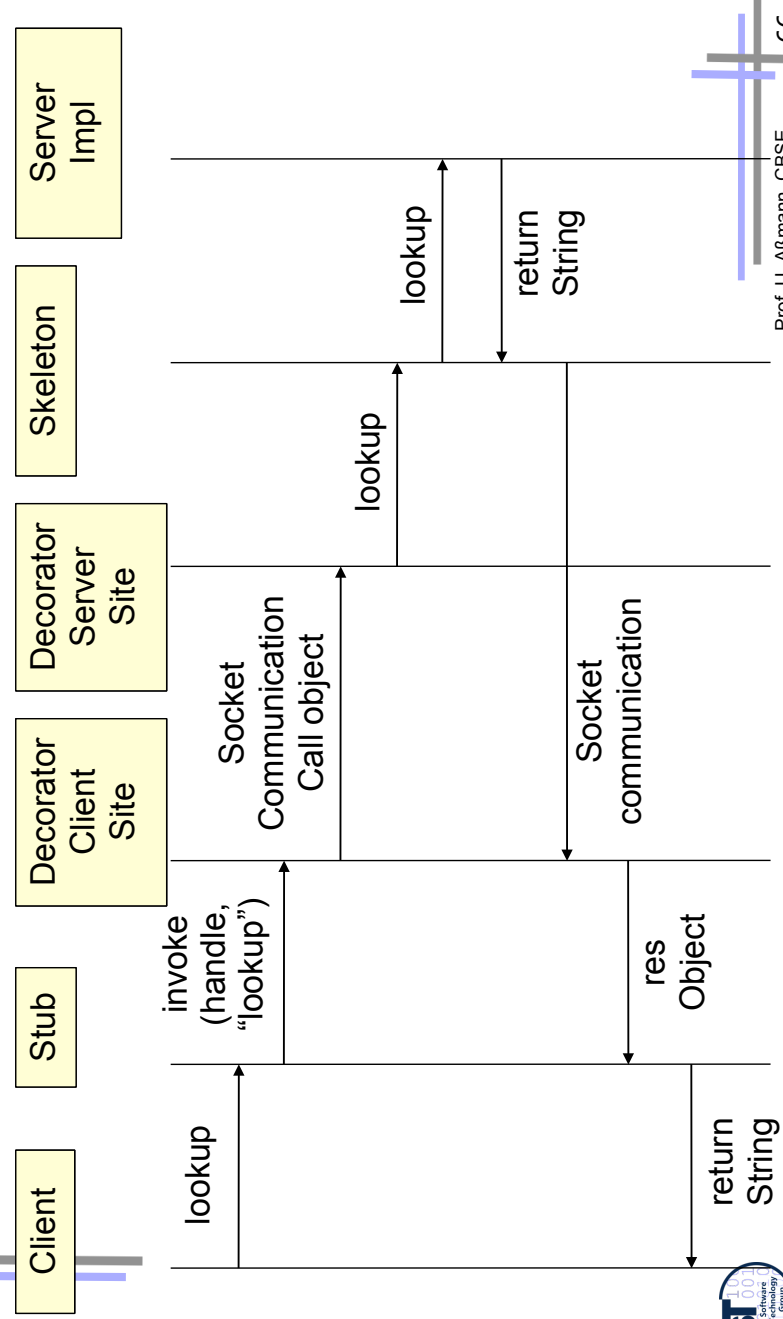
```
public class YellowPageSkeleton implements IYellowPageService {  
    private static Hashtable<Integer, IYellowPageService> yellowPageServices =  
        new Hashtable<Integer, IYellowPageService>();  
  
    private Integer logicalAddress;  
  
    public YellowPageSkeleton() {  
        this(new Integer(yellowPageServices.size()));  
        yellowPageServices.put(logicalAddress, new YellowPageService());  
    }  
  
    public YellowPageSkeleton(Integer address) {  
        logicalAddress = address;  
    }  
  
    public Integer getLogicalAddress() { return logicalAddress; }  
  
    public String lookup(String name) {  
        IYellowPageService service = yellowPageServices.get(logicalAddress);  
        return service.lookup(name);  
    }  
  
    public void store(String name, String value) {  
        IYellowPageService service = yellowPageServices.get(logicalAddress);  
        service.store(name, value);  
    }  
}
```



# Creation of YP Service

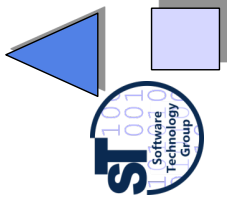


# Call (Lookup) YP Service



# 11.6 Name Transparency and Trading

Mapping names to locations by name servers



CBSE, © Prof. Uwe Alsmann

67

## More Flexible Service Management

### ▶ How to spare server skeletons?

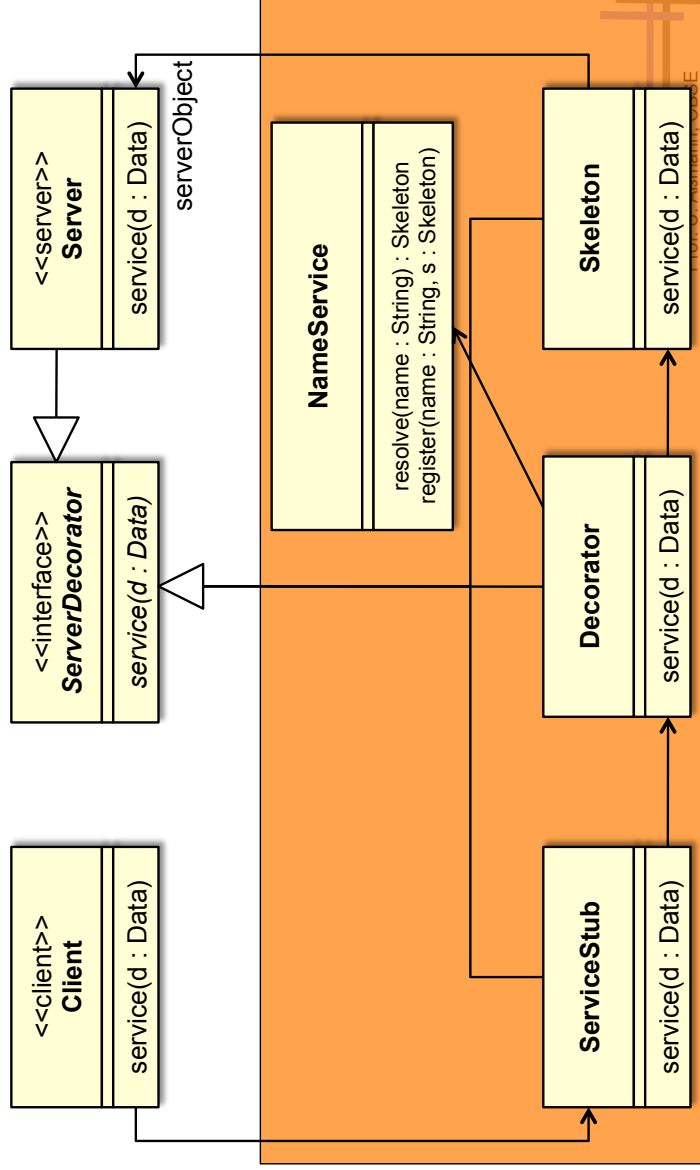
- IDL compiler must generate code for server decorator deserializer (example code contained the service dispatcher)
- Solution: only one server side Decorator per site – independent of client components provided
  - flexible service method with name lookup
  - the current solution prevents dynamic loading of services, because code has fixed names; requires regeneration of Decorator

### ▶ Solution: *name service*

- Decoupling of decorator and skeletons
- Provide a basic name service for identifying the components (skeletons) of a site, so that the number of services is dynamically extensible
- Server components register in a service directory (name service) with name and reference
- Generic adapter looks up, creates, and provides the appropriate service

# Name Service

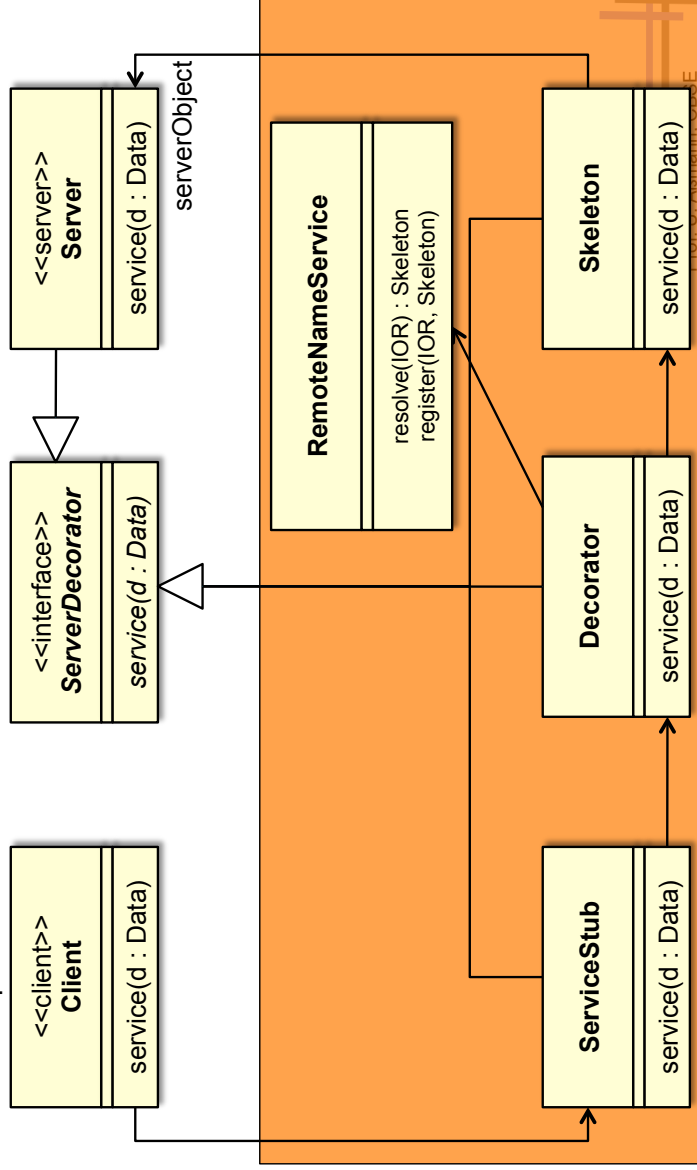
- ▶ Name to Location
- ▶ Located in the container as an associative array (map)



69

# Name Service Generalized (1)

- ▶ *Distributed name service (name to location):*
  - If name of server is known, search for the right site providing a desired component

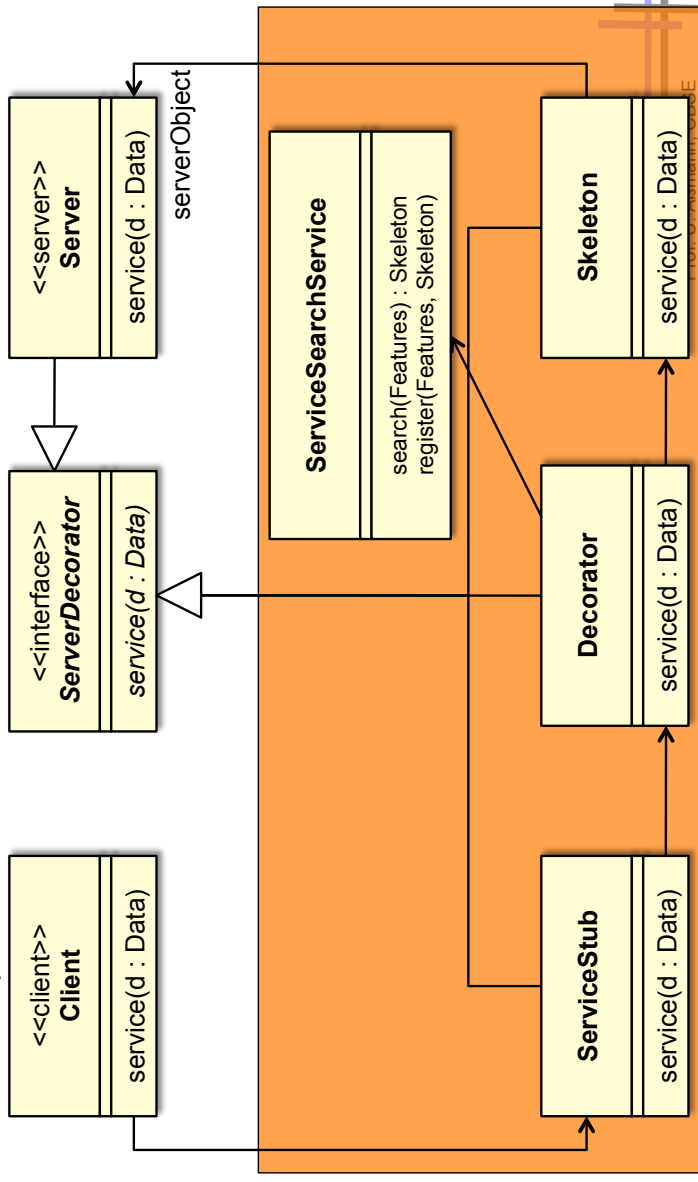


70

## Name Service Generalized (2)

### Extended name service, dynamic call:

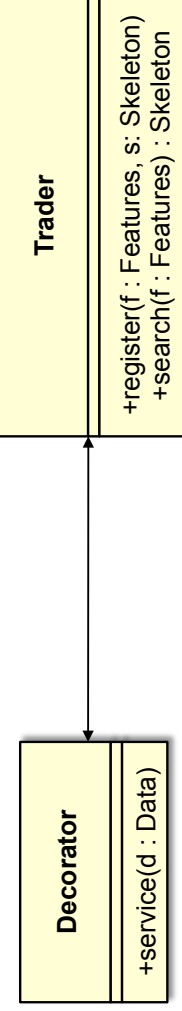
- If name of server is **not** known, search for the right service with faceted feature description

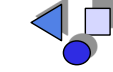


## Traders as Generalized Name Servers

### Trader service, traded call map properties to name or properties to location

- Search for a server component with known properties, but *unknown* name
- Server components register at a *trader* with name, reference, and lookup properties (metadata)
  - The trader has a component repository (*registry*)
  - Instead of names, lookup of service matches properties (metadata)
  - Return reference (site and service)
- Matching relies on standardized properties
  - Terminology, Ontology in facets (see “Finding components”)
  - Functional properties (domain specific functions ...)
  - Non-functional properties (quality of service ...)





## Rept.: Reflection & Reflective Invocation

- ▶ Reflection
  - to inspect the interface of an unknown component
  - for automatic/dynamic configuration of server sites
  - to call the inspected components
- ▶ Access to interfaces with IDL
  - Standardize an IDL run time representation and access
  - Define a IDL specification for IDL representation and access
  - Store IDL specifications in *interface repositories* which can be introspected



## Example Generic Skeleton (Reflective Skeleton)

- ▶ A **generic skeleton** is a special case of a name service: using reflection to look up the name for a method

```
class ReflectiveSkeleton {
    // serverObjects is the server implementation repository
    static ExtendedHashtable serverObjects = new ExtendedHashtable();
    ObjectOutputStream os;
    ObjectInputStream is;

    ...
    public Object handleGeneric() { ...
        Integer addr= (Integer) is.readObject();//handler
        String mn = (String) is.readObject();//method name
        Class[] pt = (Class[]) is.readObject();//parameter types
        Object[] args= (Object[]) is.readObject();//parameters

        // get server object reference by reflective call to implementation repository
        Object o      = serverObjects.getComponent(addr);
        Method m      = o.getClass().getMethod(mn,pt); //method object by
        reflection
        Object res     = m.invoke(o,args);          //method call by
        reflection
        os.writeObject(res);
        os.flush();
    } ...
}
```





## **Remark: Skeletons and Containers**

- ▶ Can be started and consulted by skeletons
- ▶ May offer many other aid functionality
  - Transactions: consistent management of multiple clients and service requests
  - Security
  - Persistence
  - Interception (hooks into which new functionality can be entered)
  - Support for aspects



## **What Classical Component Systems Provide**

- ▶ Technical support: remote, language and platform transparency
  - Stub, Skeleton
    - One per component (technique: IDL compiler)
    - Generic (technique: reflection and dynamic invocation)
  - Decorators on client and server site
    - Individual
    - Generic (technique: Name services)
- ▶ Economically support: reusable services
  - Basic: name, trader, persistency, transaction, synchronization
  - General: print, e-mail, ...
  - Domain specific: business objects, ...





## Summary

- ▶ Component systems provide many component secrets
  - Location, language and platform transparency
  - Transactional, persistence, security, name service
- ▶ Component secrets are realized with the Connector Pattern (Stub, Skeleton-Pattern)
  - One pair or tuple of Decorators per component in a layer, but several layers, stacking Decorators on top of each others
  - On the server side, adapters help to make services generic
  - Decorators, Proxies, Adapters, Chains on client and server site
- ▶ Generated by IDL compiler
  - Is the IDL compiler essential?
  - No! Generic stubs and skeletons are possible, too. Technique: Reflective invocation



## The End

- ▶ Many slides courtesy to Prof. Welf Löwe, Växjö University, Sweden.

