Verteilte Systeme
Entfernte Aufrufe und Webservices

Was ist ein verteiltes System?

Ein verteiltes System besteht aus …

- Einem oder mehreren Computer und
- Mehreren parallel laufenden Anwendungsprozessen

Die Koordination erfolgt …

- Durch den Austausch einfacher Nachrichten
- Jedoch nicht durch direkte Methodenaufrufe etc.

Unter einer Nachricht versteht man …

- Eine Datenstruktur mit festem Aufbau,
- Welche die zu übermittelnden Informationen beinhaltet

Beispiel: Nachrichten

HTTP-Anfrage

Anfragezeile
Kopfdaten (optional)
Leerzeile
Anfragedaten (optional)

```
POST /service.php?client=55 HTTP/1.1
Host: dhbw-karlsruhe.de
Content-Type: text/xml
Content-Encoding: UTF-8
Content-Length: 542

<s:envelope>
  <s:Body>
    <Vorlesung>
      Webentwicklung
    </Vorlesung>
  </s.Body>
</s:envelope>
```

In der Gruppenarbeit

Kommando
Beliebig viele Parameter
Trennzeichen: /

```
WELCOME/Mark
```

```
CURRENT STATUS/16/23/41/Y
```

Grundsätzlich ist jede Datenstruktur, egal ob binär oder textbasiert, für den Austausch von Daten zwischen zwei Systemen geeignet. Bei Verwendung einer Middleware ist das Nachrichtenformat in der Regel vorgegeben. Sonst muss es selbst definiert werden.

Nachrichtenaustausch

Einfache Datenströme

- Nur wenig Unterschiede zwischen Client und Server
- Beliebiger Datenaustausch und beliebige Protokolle möglich

Entfernte Aufrufe und Webservices

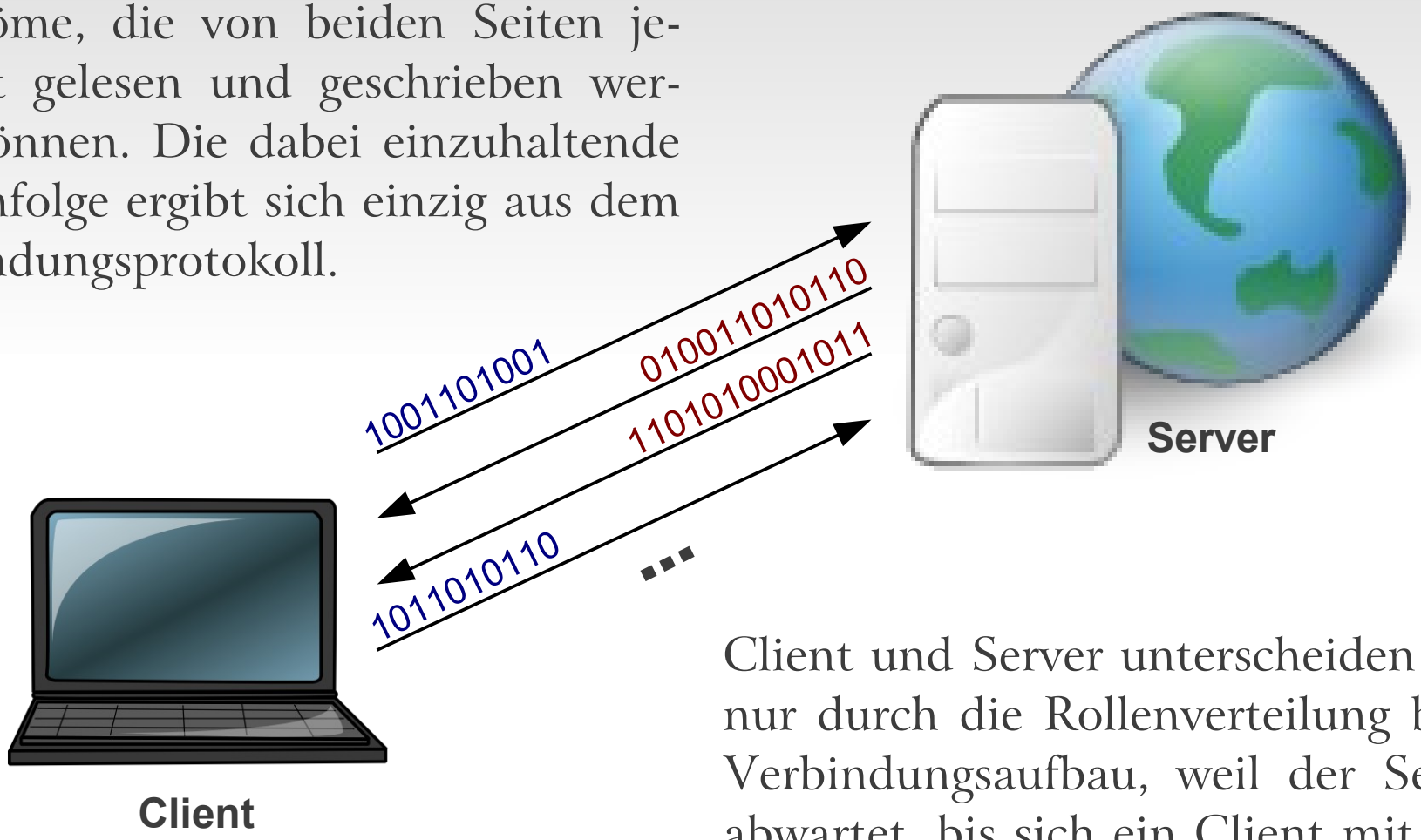
- Konzeptionelle Einschränkung auf Anfrage und Antwort
- Beliebige Clients senden eine Anfragestruktur an den Server
- Der Server sendet eine Antwortstruktur an den Client zurück

Nachrichtenbasierte Middleware

- Lauter gleichberechtigte Systeme, keine Clients und Server
- Entkopplung der Systeme durch einen Message Broker

Beispiel: Streams und Sockets

Der Datenaustausch erfolgt durch Datenströme, die von beiden Seiten jederzeit gelesen und geschrieben werden können. Die dabei einzuhaltende Reihenfolge ergibt sich einzig aus dem Anwendungsprotokoll.



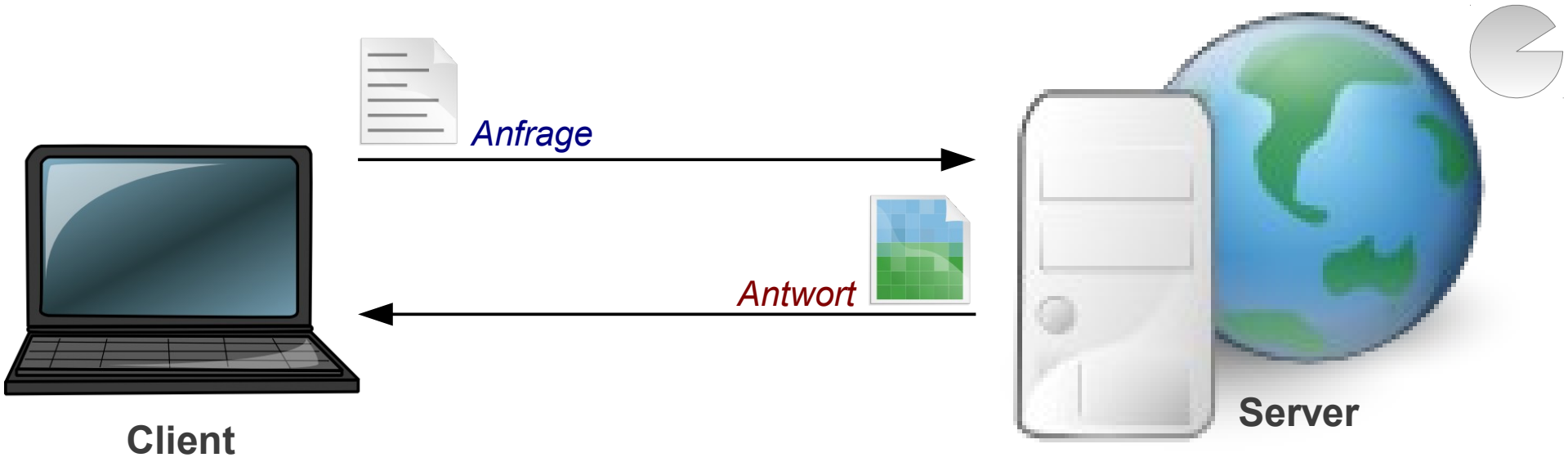
Client und Server unterscheiden sich nur durch die Rollenverteilung beim Verbindungsaufbau, weil der Server abwartet, bis sich ein Client mit ihm verbindet.

Beispiel: Entfernter Aufruf

1. Der Client sendet eine Nachricht an den Server. Diese Nachricht wird **Anfrage** genannt, weil sie den Server auffordert, etwas zu tun.

2. Der Server verarbeitet die Anfrage und führt die gewünschte Aktion aus. Der Client wartet derweil.

3. Anschließend wird das Ergebnis der Verarbeitung als **Antwort** an den Client geschickt und die Verbindung getrennt.



Entfernte Aufrufarten

Prozedural

Remote Procedure Call / Webservices

Die Serveranwendung beinhaltet eine sogenannte Bibliothek von entfernt aufrufbaren Funktionen, welche serverseitig ausgeführt werden können. Dabei besitzt jede Funktion einen Namen, Parameter sowie einen Rückgabewert.

int addCustomer(String)

void removeCustomer(int)

int placeOrder(int, float)

int cancelOrder(int)

Objektorientiert

Remote Method Invocation

Die Serveranwendung verwaltet eine Menge von Objekten, deren Methoden über das Netzwerk aufgerufen werden können. Jedes Objekt besitzt seine eigenen Daten, wobei es natürlich auch Referenzen auf andere Objekte beinhalten kann.

CustomerService

Customer newCustomer()
Customer findCustomer(String)

Customer

Address getAddressData()
Order placeOrder(int, float)
List<Order> getAllOrders()

Address myAddress
List<Order> myOrders
TimeStamp lastAccess

RPC/RMI-Implementierungen

CORBA
Herstellerunabhängiger Standard, der für viele Betriebssysteme und Programmiersprachen zur Verfügung steht.

Beinhaltet eine eigene *Interface Definition Language*, um Schnittstellen zu beschreiben.

Sun/Oracle RMI
Im Lieferumfang des JDK enthalten. Schnittstellen werden durch einfache Java Interfaces definiert.

SAP Java Connector
Proprietäre Bibliothek, um remotefähige Funktionsbausteine eines SAP-Systems aufzurufen.

XML-RPC Webservices
Leichtgewichtiger Standard für entfernte Prozeduraufrufe auf Basis von HTTP/XML.

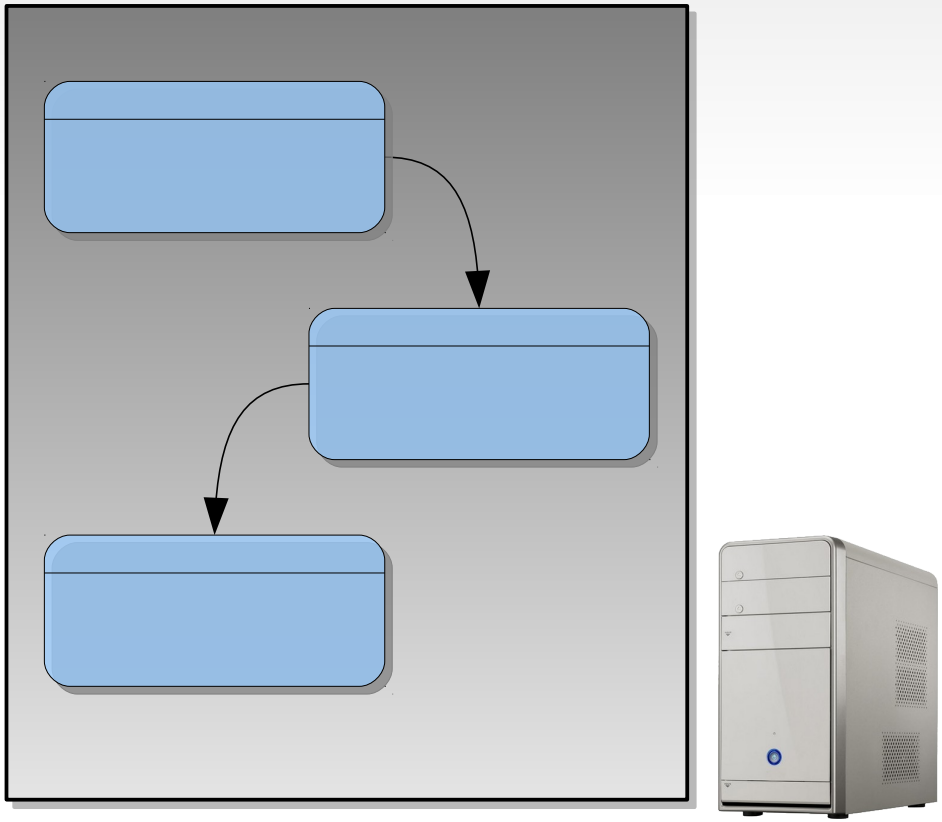
SOAP-Webservices
Nutzt die *Webservice Description Language*, um die ausgetauschten XML-Daten formal zu beschreiben.

Entfernte Methodenaufrufe

Lokale und entfernte Objekte

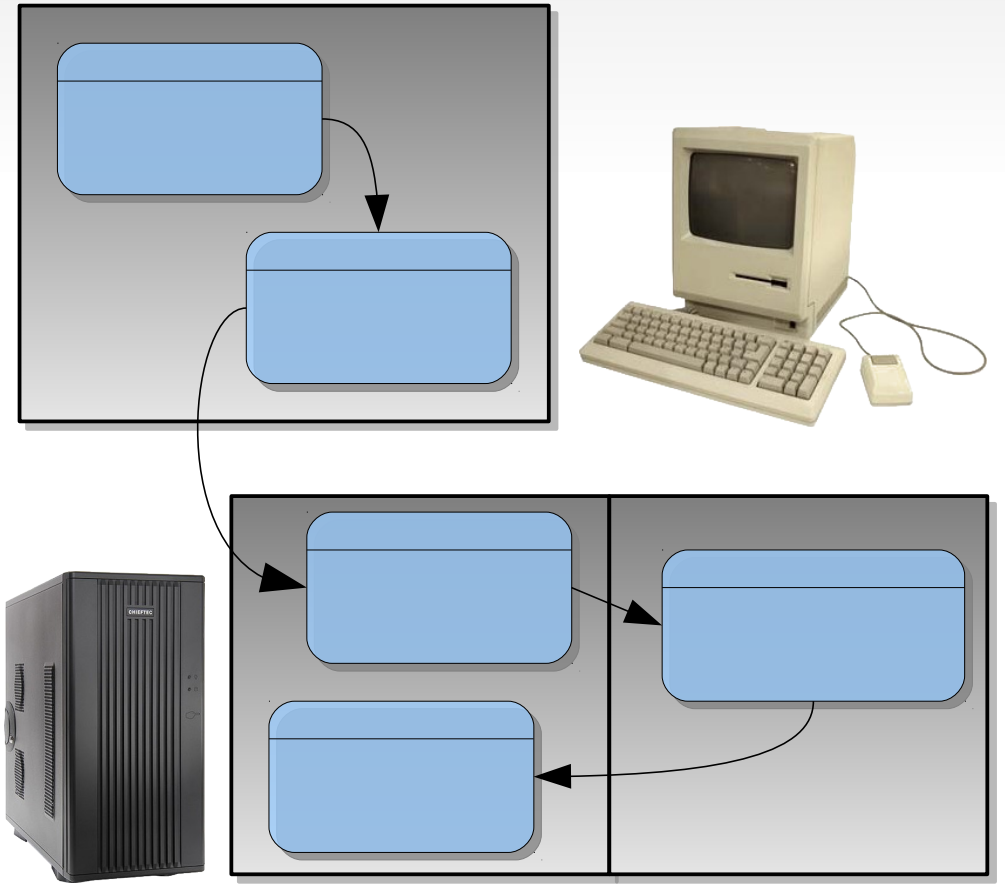
Lokale Objekte

Alle Objekte befinden sich im Speicher desselben Prozesses auf einem Computer.



Entfernte Objekte

Die Objekte befinden sich im Speicher verschiedener Prozesse, die teilweise auf unterschiedlichen Computern laufen.



Erinnerung: Objekte bisher

Objektsystem
Bezeichnet eine Sammlung von Objekten bestehend aus Attributen und Methoden

Objektreferenz
Eindeutige Identifikation eines Objekts innerhalb einer Anwendung

Schnittstelle
Definiert, welche Methoden eines Objekts aufgerufen werden können und welche Parameter diese besitzen

Ereignisse / Aktionen
Initiiert durch den Aufruf der Methoden eines Objekts. Häufig wird dabei der Zustand des Objekts verändert

Exceptions
Signalisieren ein fehlerhaftes Laufzeitverhalten und machen dieses behandelbar

Garbage Collection
Freigabe nicht mehr benutzten Speichers durch Zerstörung ungenutzter Objekte

Erweiterung um verteilte Objekte

Verteiltes Objektsystem
Das Objektsystem ist nicht mehr auf einen Prozess beschränkt. Die Objekte können auf mehreren Rechnern sein

Entfernte Objektreferenz
Eindeutige Identifikation eines Objekts über Prozess- und Rechnergrenzen hinweg

Entfernte Schnittstelle
Definiert die entfernt aufrufbaren Methoden. Dabei sind die verschiedenen Übergabesemantiken zu beachten!

Ereignisse / Aktionen
Werden sowohl durch lokale als auch durch entfernte Methodenaufrufe ausgelöst

Exceptions
Neue Fehlerursachen führen zu mehr Exceptions. Entfernte Aufrufe führen dazu, dass Ausnahmen in fremden Prozessen entstehen können

Garbage Collection
Muss auch entfernte Referenzen berücksichtigen können

Entfernte Objektreferenzen

Werden von der Middleware automatisch erzeugt

Sind über Raum und Zeit global eindeutig

Ermöglichen den Aufruf entfernter Objektmethoden

Realisiert durch lokale Stellvertreter (Vgl. Proxymuster)

Inhalt einer entfernten Referenz:

- Zieladresse des entfernten Rechners
- Portnummer und Zeitstempel (Lokaler Prozess)
- Lokale Objektnummer innerhalb des Prozesses
- Entfernte Schnittstelle des Objekts

Entfernte Schnittstellen

Definieren die aufrufbaren Methoden entfernter Objekte

Werden in einer middlewarespezifischen Sprache formuliert

Können von der lokalen Schnittstelle der Objekte abweichen

Inhalt einer entfernten Schnittstelle:

- Name der Schnittstelle
- Benötigte Datentyp- und Strukturdefinitionen
- Signaturen aller aufrufbaren Methoden (Name und Parameter)
- Rückgabewerte und Exceptions aller Methoden

Distributed Garbage Collection

Problem des Garbage Collectors:

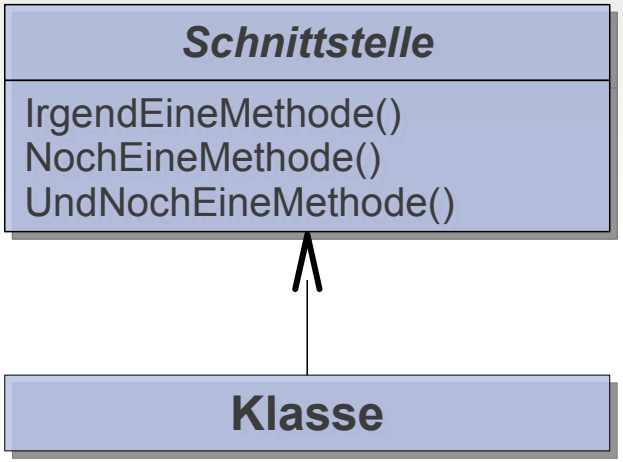
- Es können nur lokale Referenzen berücksichtigt werden
- Jedes Objekt ohne lokale Referenzen wird aufgeräumt
- Entfernte Referenzen werden dadurch ungültig gemacht
- Keine Möglichkeit für entfernte Systeme, darauf zu reagieren!

Lösungsansatz für verteilte Systeme:

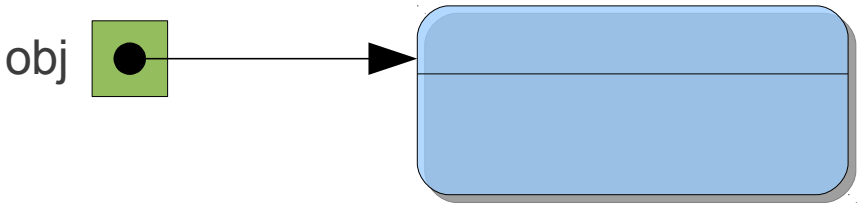
- Verwaltung eines Referenzzählers für jedes Objekt
- Eine neue, entfernte Referenz erhöht den Zähler automatisch
- Erniedrigung des Zählers, wenn eine Referenz aufgelöst wird
- Objekt darf nur zerstört werden, wenn der Zähler null ist

Entfernte Objekte im Detail

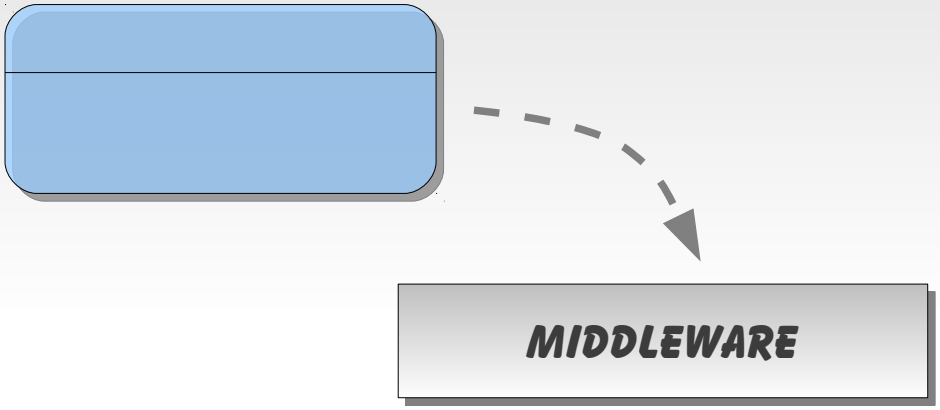
1. Der Serverprozess möchte, dass ein Objekt auch entfernt verfügbar ist. Damit dies funktioniert, muss das Objekt eine entfernte Schnittstelle besitzen.



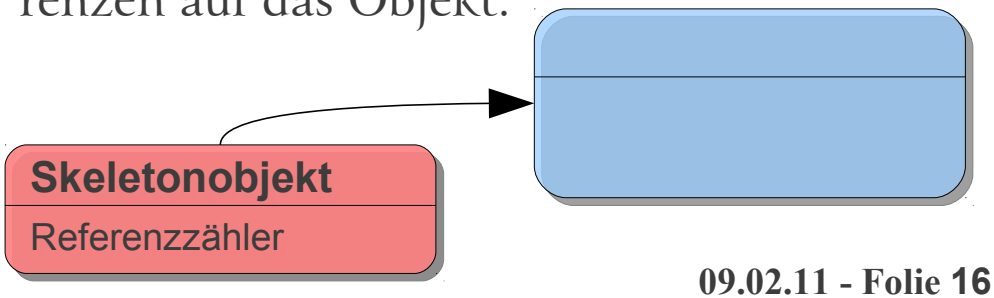
2. Der Server erzeugt das Objekt. Für den Server handelt es sich dabei um ein normales, lokales Objekt.



3. Um das Objekt für andere Prozesse zur Verfügung zu stellen, muss es der Middleware bekannt gemacht werden.

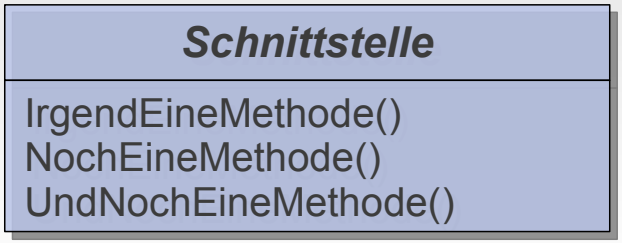


4. Die Middleware erzeugt ein sogenanntes Skeletonobjekt. Dieses Objekt ist für die serverseitige Netzwerkkommunikation zuständig. Außerdem verwaltet es den Referenzzähler der bekannten, entfernten Referenzen auf das Objekt.

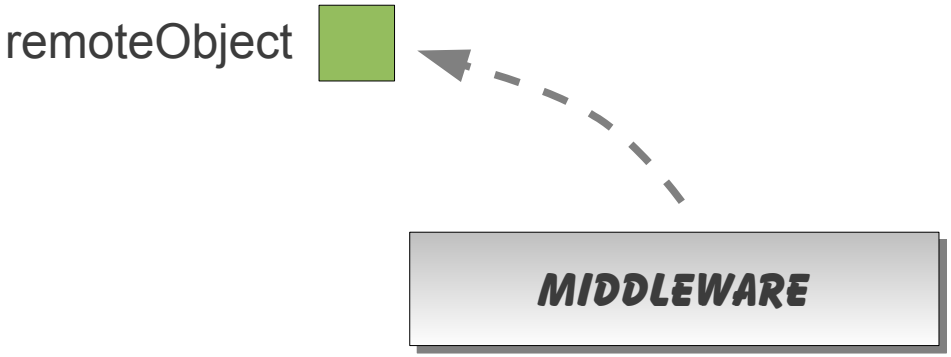


Entfernte Objekte im Detail

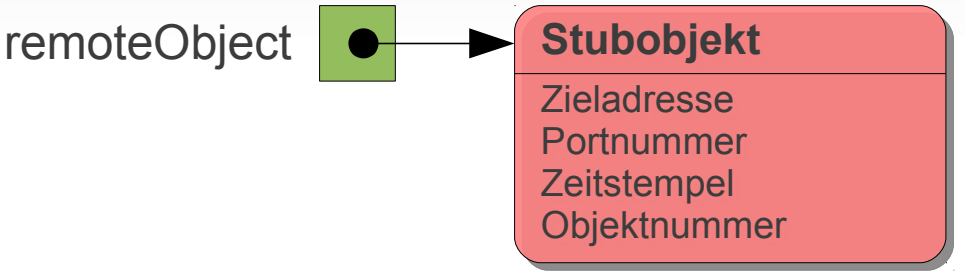
5. Ein Clientprozess möchte auf das entfernte Objekt zugreifen. Hierfür muss die Schnittstelle des Objekts bekannt sein.



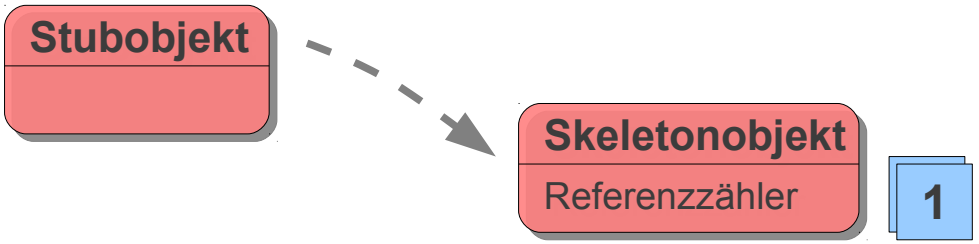
6. Der Client fordert von der Middleware eine Referenz auf das entfernte Objekt an. Sie soll verwendet werden, um die Methoden des Objekts aufzurufen.



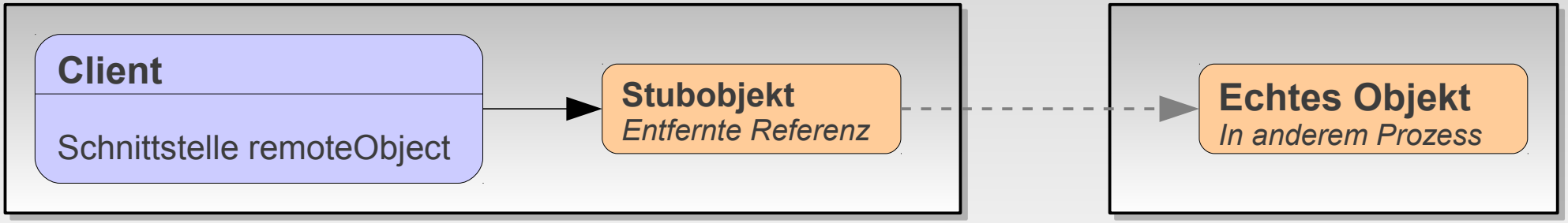
7. Die Middleware erzeugt ein sogenanntes Stubobjekt, das für die clientseitige Netzwerkkommunikation zuständig ist. Dieses Objekt lebt im Speicher des Clientprozesses und fungiert als lokaler Stellvertreter des entfernten Objekts, weshalb es dieselben Methoden besitzt (Vgl. Proxymuster!)



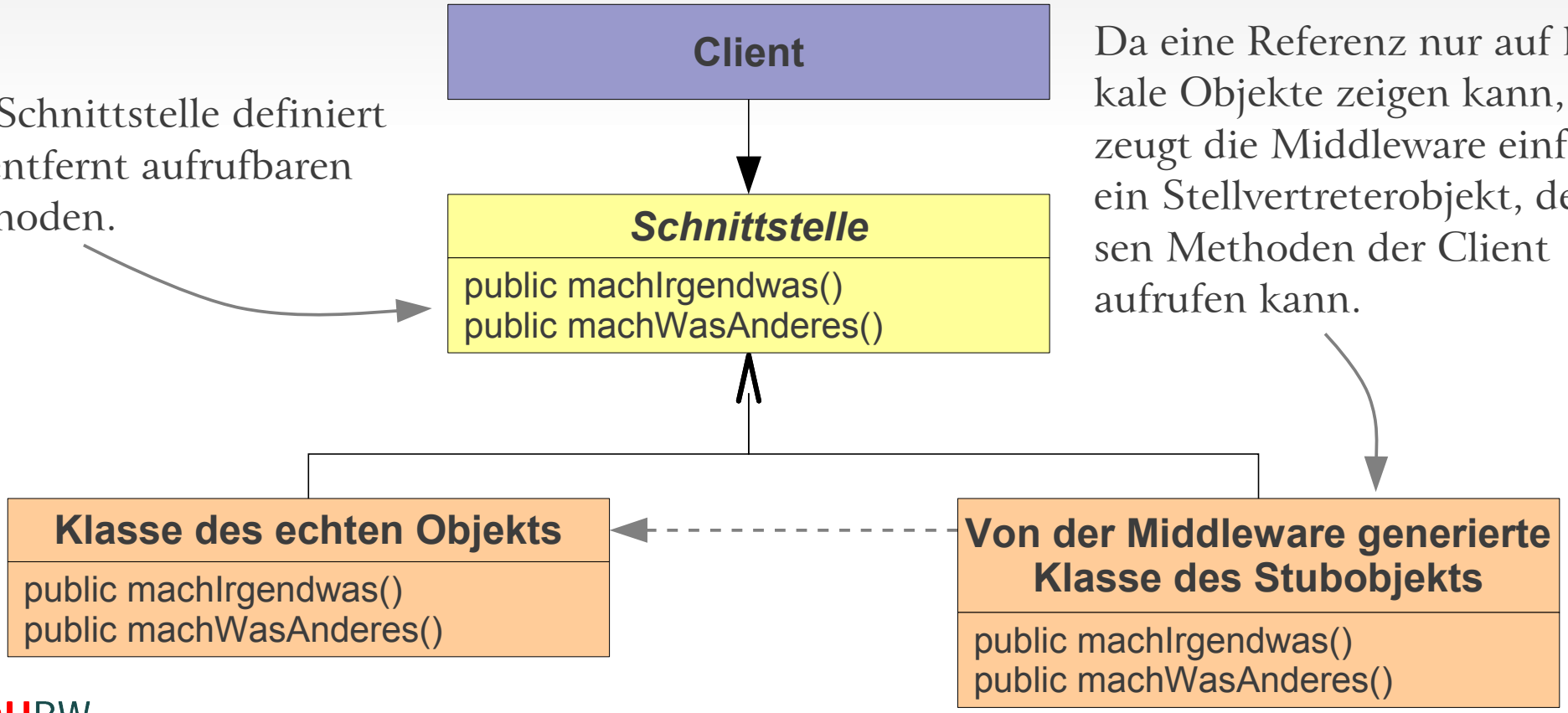
8. Das Stubobjekt sendet eine kurze Botschaft an den Server, damit dieser seinen Referenzzähler aktualisieren kann.



Vergleich mit Proxymuster



Die Schnittstelle definiert die entfernt aufrufbaren Methoden.

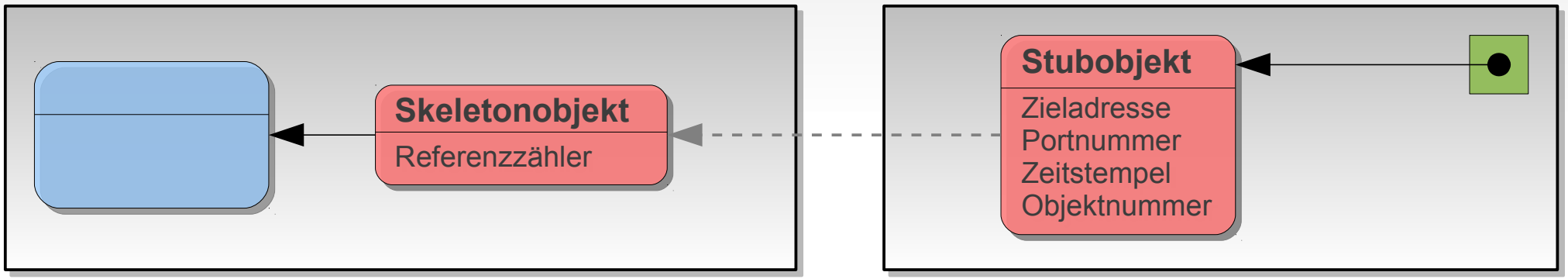


Da eine Referenz nur auf lokale Objekte zeigen kann, erzeugt die Middleware einfach ein Stellvertreterobjekt, dessen Methoden der Client aufrufen kann.

Entfernte Aufrufe im Detail

1. Der Server erzeugt ein entferntes Objekt und meldet es bei der Middleware an. Es entsteht ein lokales Skeletonobjekt, um das sich der Server allerdings nicht kümmern muss.

2. Der Client fordert eine Referenz auf das entfernte Objekt an und erhält daraufhin ein lokales Stubobjekt, dessen Methoden er aufrufen kann.



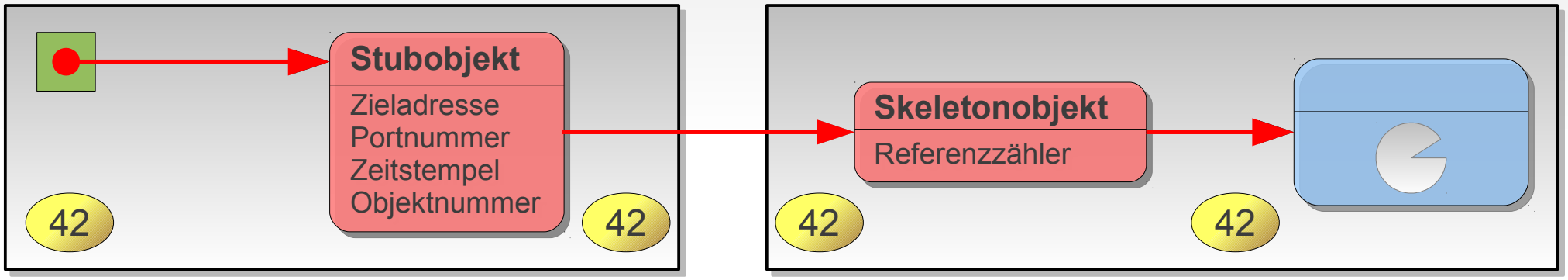
Entfernte Aufrufe im Detail

3. Der Client ruft eine Methode des erhaltenen Stubobjekts auf.

4. Der Stub versendet die Anfrage über das Netzwerk an das Skeleton.

5. Das Skeletonobjekt empfängt die Anfrage und ruft die echte Methode auf.

6. Der Rückgabewert der Methode wird zurückgesendet und an den Client übergeben.



Übergabesemantiken

Frage: Wie werden Aufrufparameter und Rückgabewerte zwischen den kommunizierenden Prozessen übertragen?

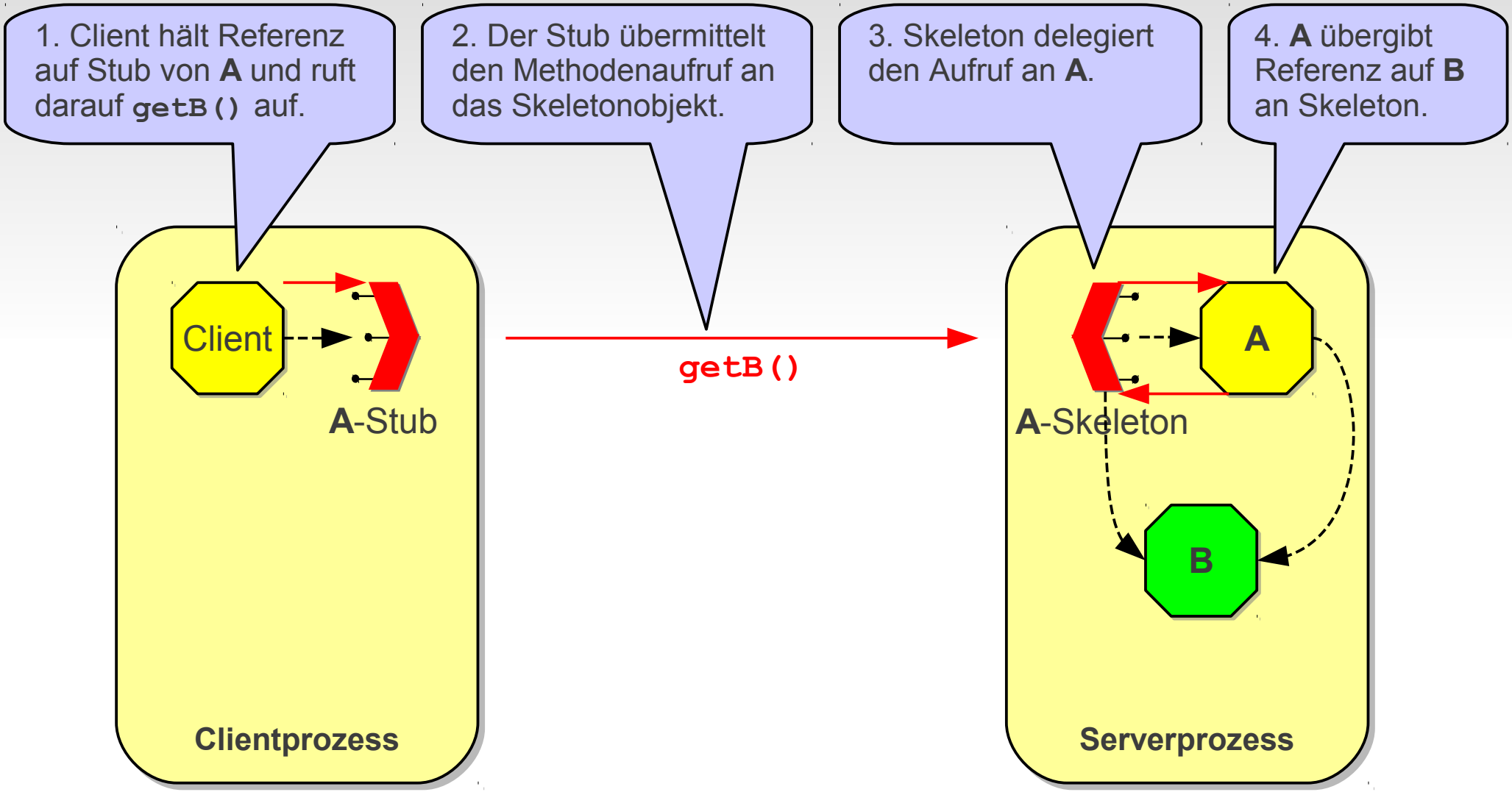
Übertragung einer Kopie (Call By Value):

- Normale Objekte und elementare Werte werden einfach kopiert
- Übertragung einer serialisierten Version der Objekte
- Achtung: Alle referenzierten Objekte werden ebenfalls kopiert
- Lokale Kopie der Objekte bei Client und Server vorhanden

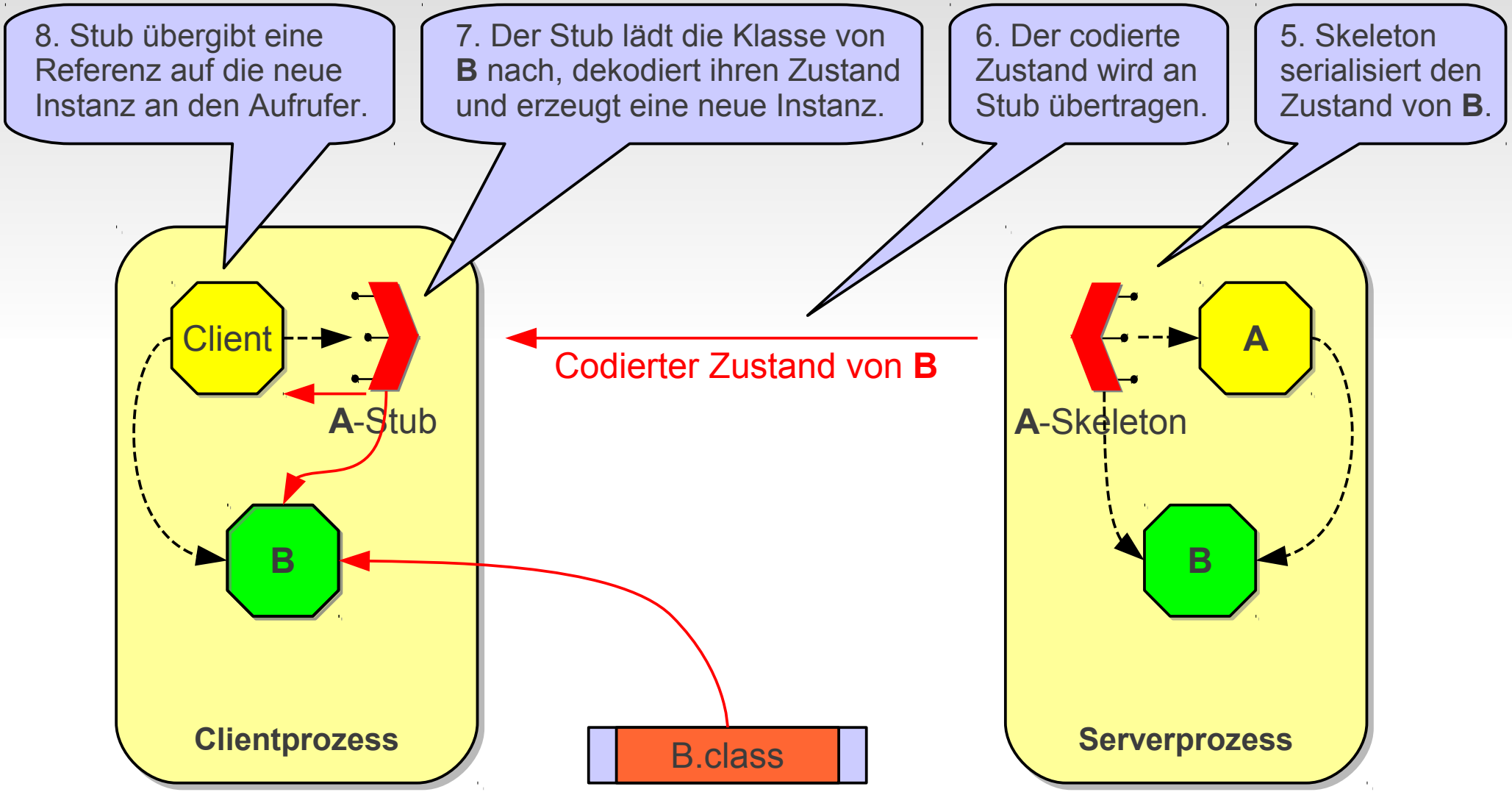
Übertragung einer Referenz (Call By Reference):

- Anstelle eines remotefähigen Objekts wird der Stub übertragen
- Der Client erhält einfach eine entfernte Referenz auf das Objekt
- Entfernte Objekte werden also nicht serialisiert oder kopiert

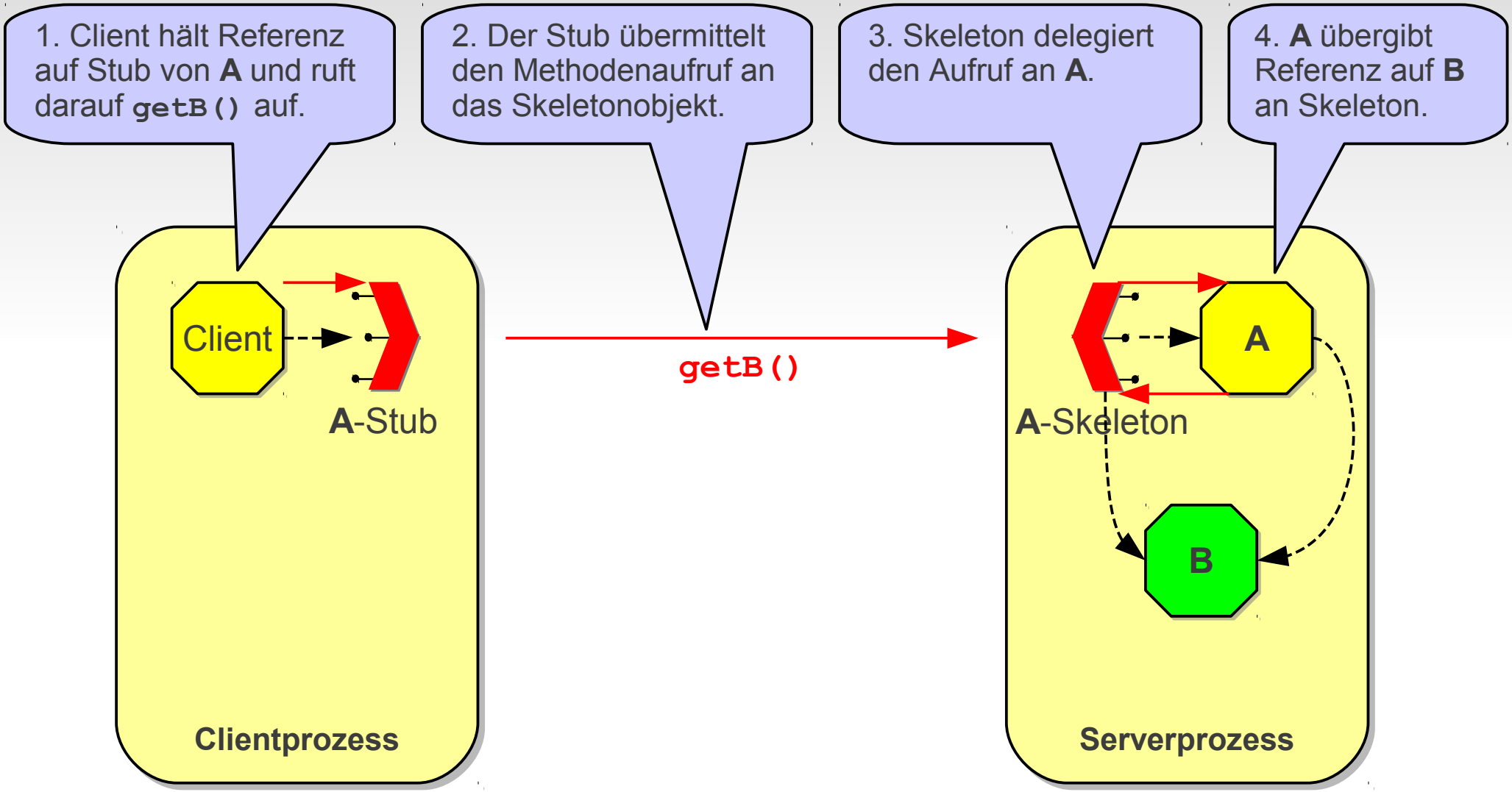
Beispiel: Kopiersemantik



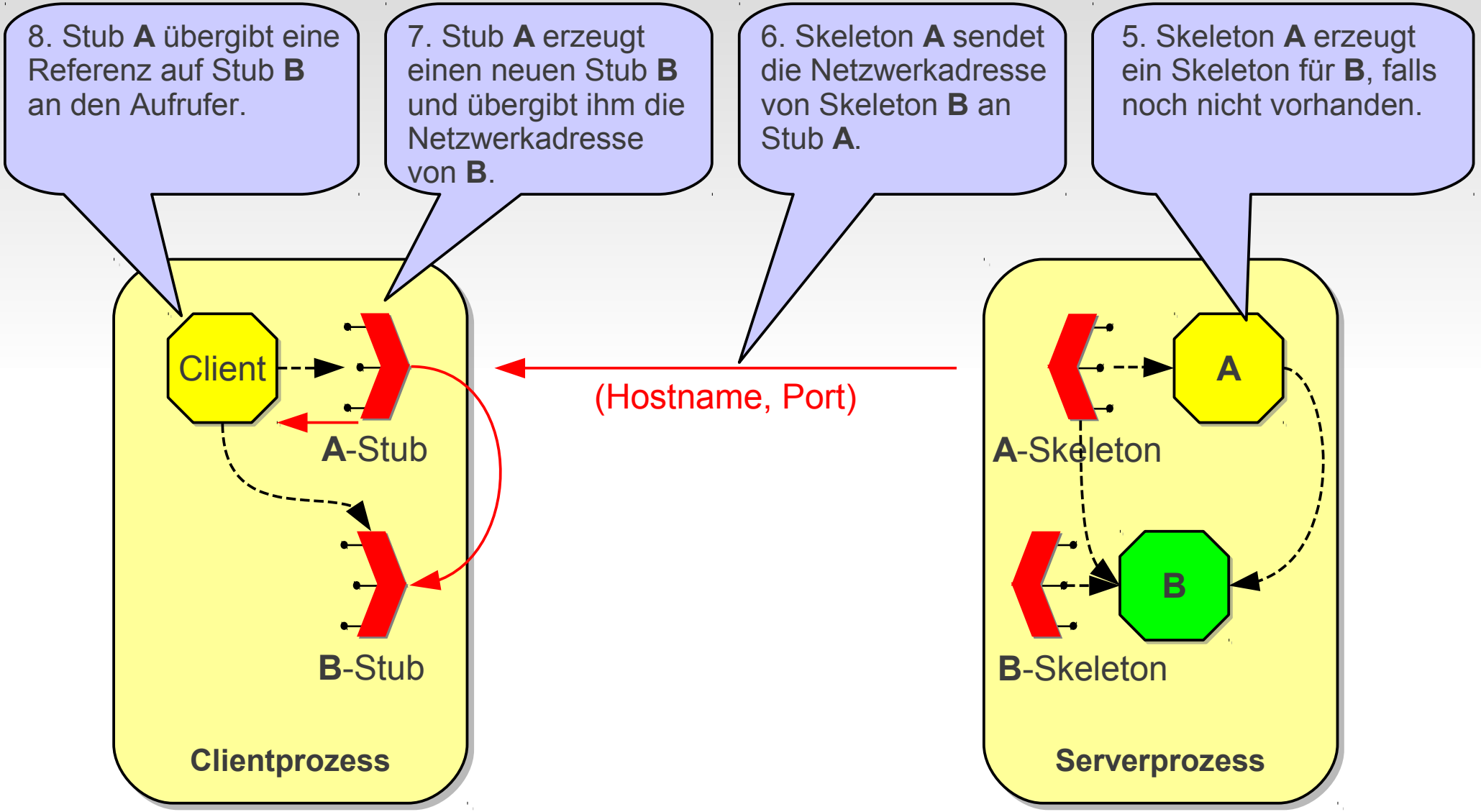
Beispiel: Kopiersemantik



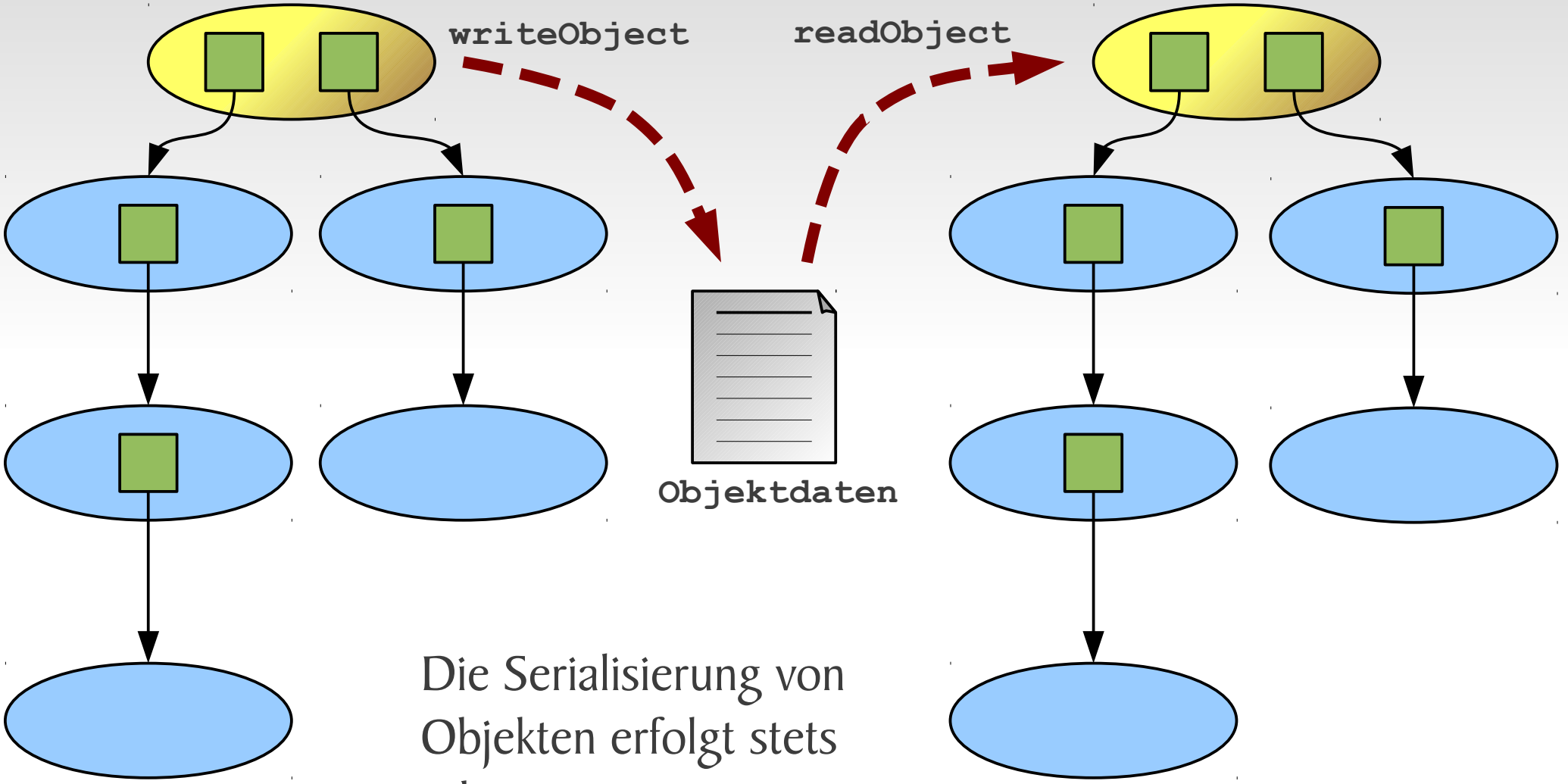
Beispiel: Referenzsemantik



Beispiel: Referenzsemantik

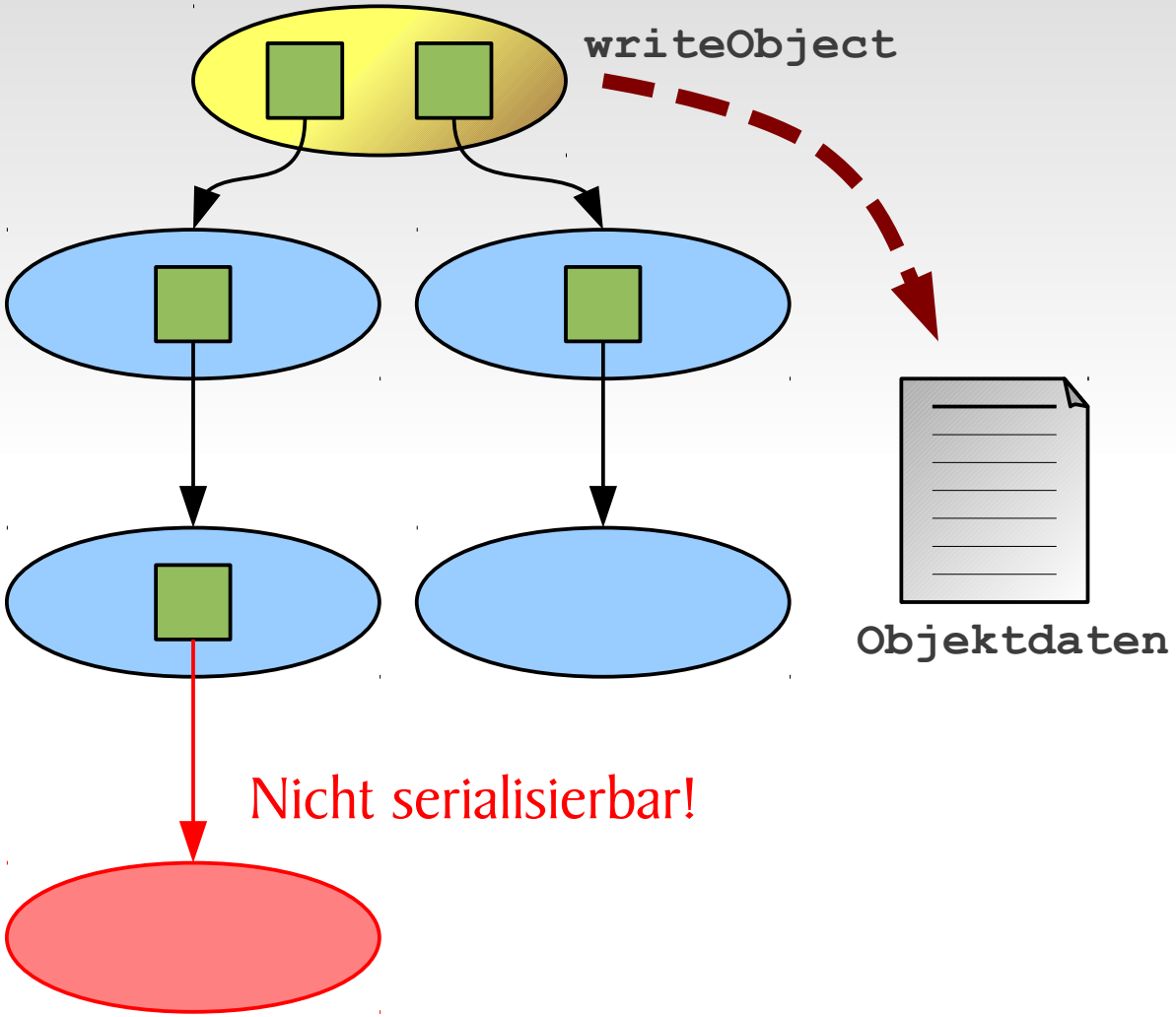


Probleme der Kopiersemantik



Die Serialisierung von Objekten erfolgt stets rekursiv.

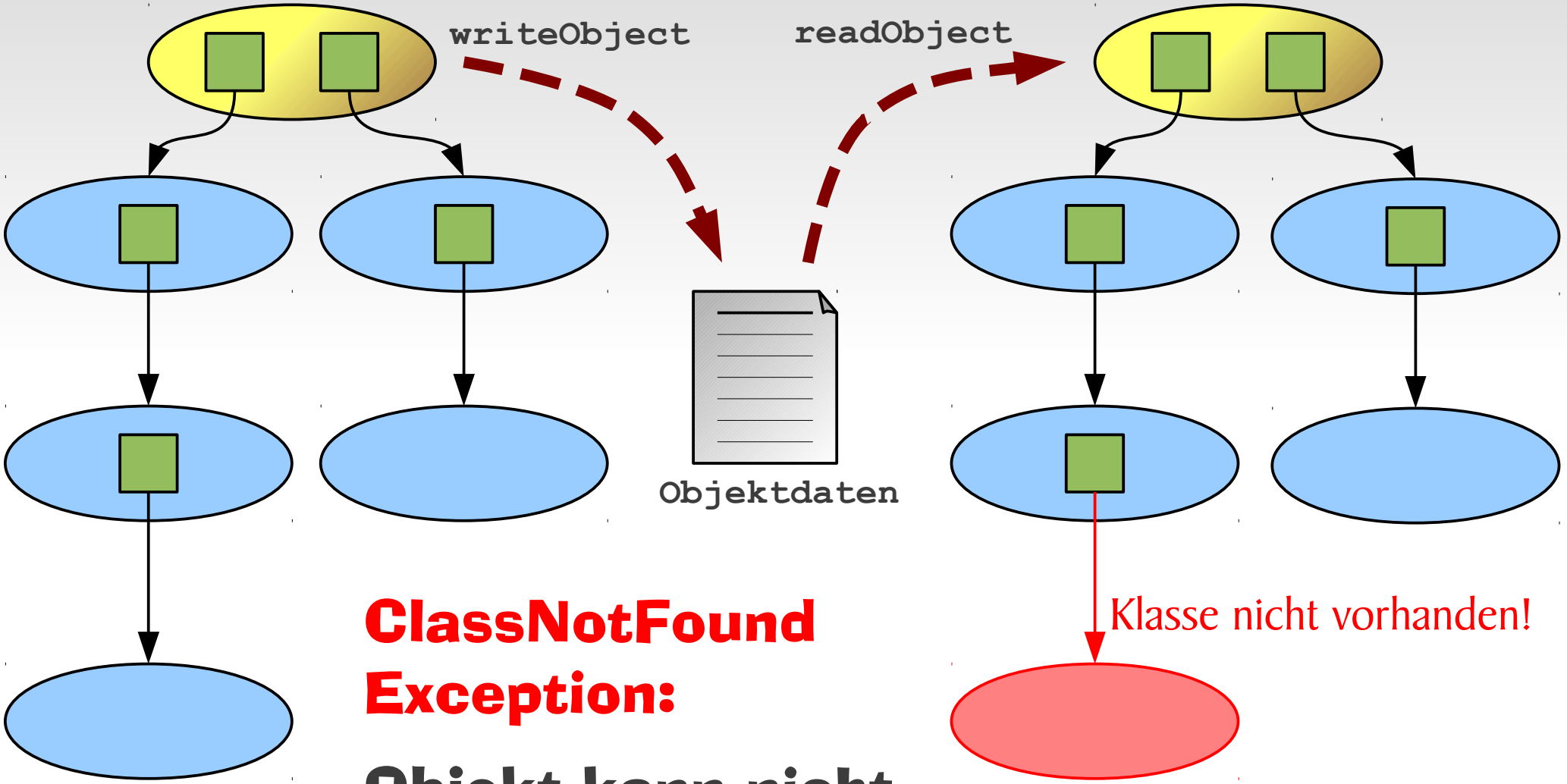
Probleme der Kopiersemantik



IOException:

Objekt kann nicht serialisiert werden

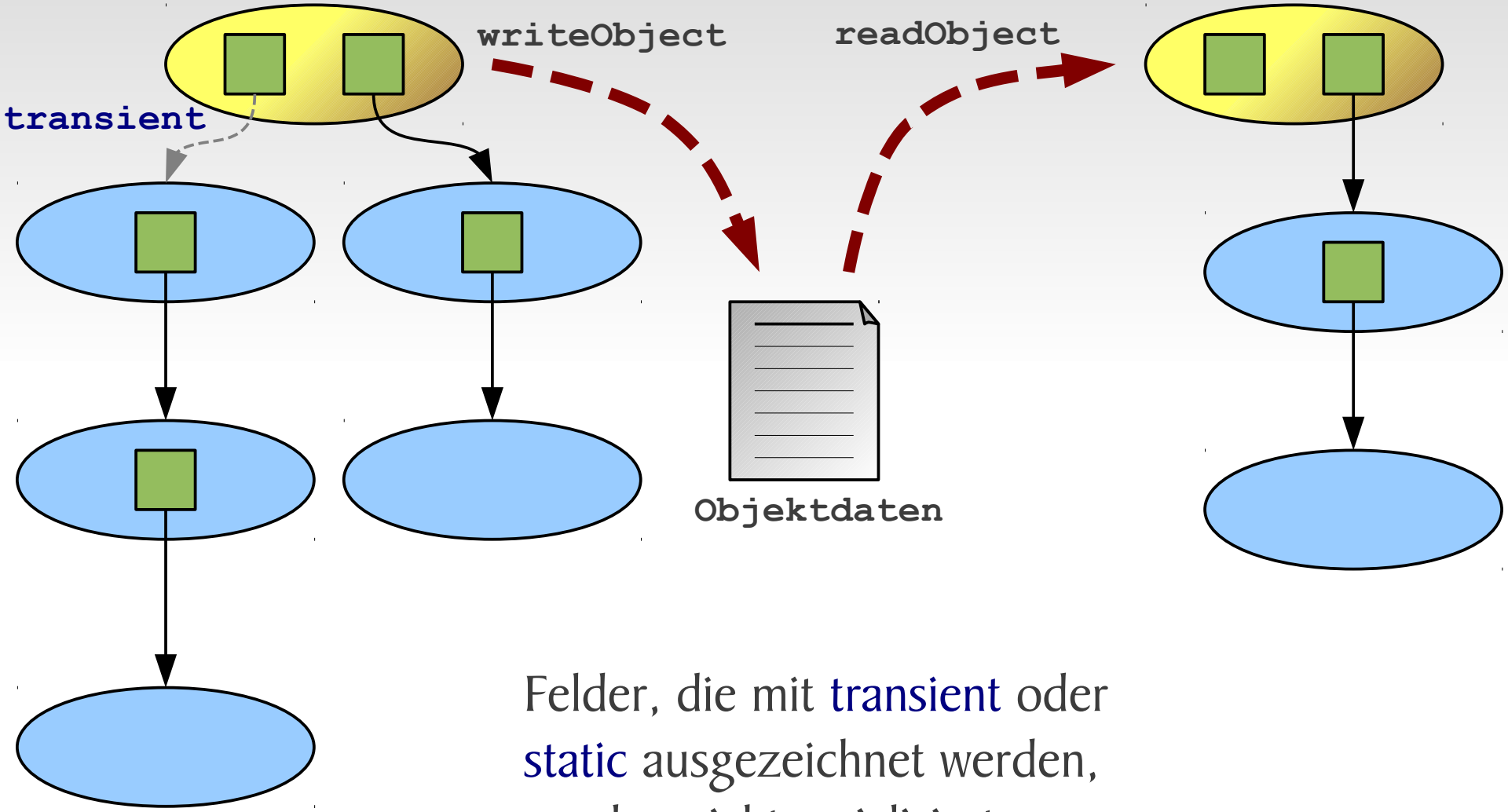
Probleme der Kopiersemantik



ClassNotFound Exception:

Objekt kann nicht deserialisiert werden

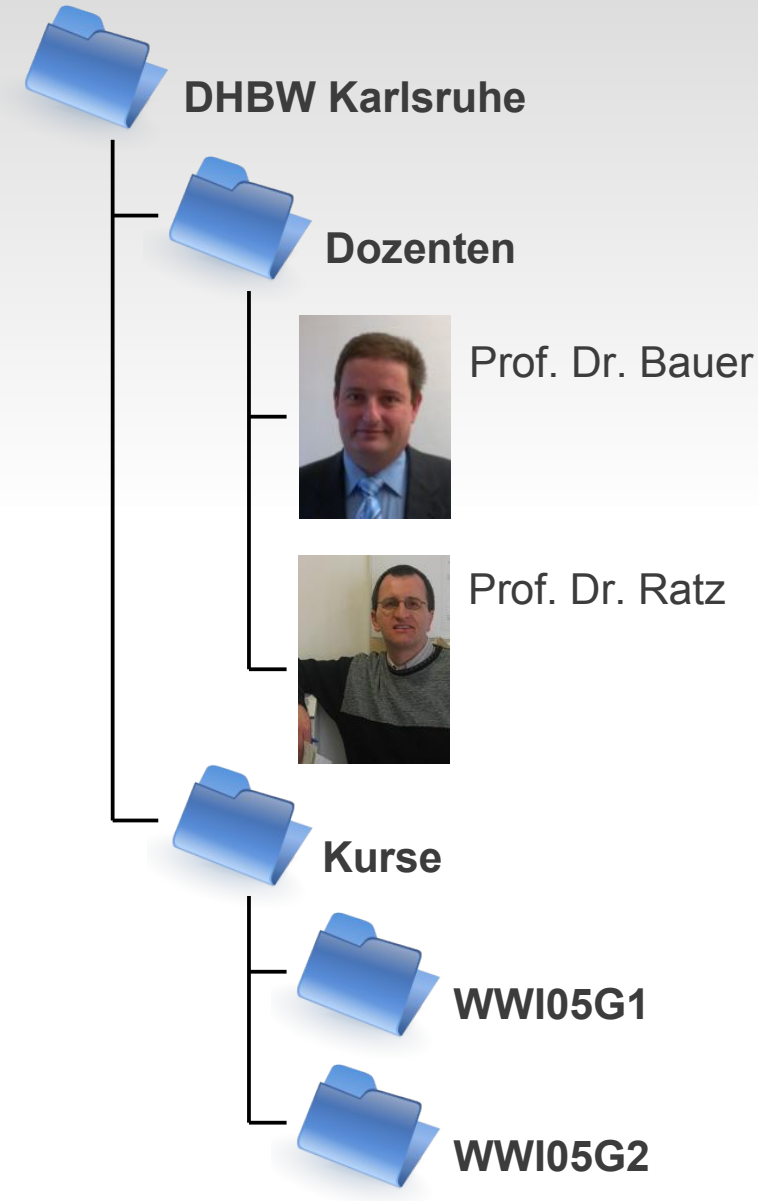
Probleme der Kopiersemantik



Felder, die mit transient oder static ausgezeichnet werden, werden nicht serialisiert.

Entfernte Objekte suchen

Die Middleware verwaltet alle entfernten Objekte mithilfe eines Namens- oder Verzeichnisdienstes



Namensdienste

- Verwalten eine Menge von Objekten
- Jedes Objekt erhält einen Namen
- Verwaltung als flache Liste oder hierarchische Baumstruktur je nach Dienst

Verzeichnisdienste

- Erweiterung der Namensdienste
- Ordnen den Objekte zusätzliche Metadaten zu (z.B. Zugriffsrechte)

Orts- und Mobilitätstransparenz

Clients kennen die Position der entfernten Objekte nicht. Denn Jedes Objekt wird nur über seinen Namen gesucht.

Ändert sich die Position des Objekts, muss nur sein Eintrag im Namensdienst angepasst werden

Objekte können darum beliebig verschoben werden, ohne dass negative Auswirkungen entstehen.

Clients müssen allerdings ihre Referenzen auffrischen, wenn ein Objekt verschoben wurde.

Sun/Oracle RMI

Was ist Sun/Oracle RMI?

- Im Lieferumfang von Java enthaltene Middleware
- Ermöglicht die Programmierung verteilter Javaprogramme
- Bildet die technische Grundlage für Enterprise Java Beans
- Nur wenig Unterstützung für andere Programmiersprachen

Bestandteile von Sun/Oracle RMI:

~~`rmic`~~	~~Codegenerator für Stubs und Skeletons~~
`rmiregistry`	Namensdienst für entfernte Objekte
`rmid`	On-Demand Erzeugung entfernter Objekte
`java.rmi`	Paket der Programmierschinttstellen

rmiregistry und rmid

Möglichst frühe Objekterzeugung:

- `rmiregistry` kann nur bereits erzeugte Objekte verwalten
- Anmeldung der erzeugten Objekte durch das Serverprogramm
- Objekte befinden sich im Speicher solange das Programm läuft
- Ungeeignet für Systeme mit sehr vielen Objekten

Bedarfsorientierte Objekterzeugung:

- `rmid` kann neue Java Runtimes starten und Objekte erzeugen
- Geringfügig umständlichere Programmierung auf Serverseite
- Zugriff auf die Objekte weiterhin über `rmiregistry`
- Darum kein Unterschied in der Programmierung für Clients

Vorgehen bei der Entwicklung

Auf Serverseite

Remote Interface definieren

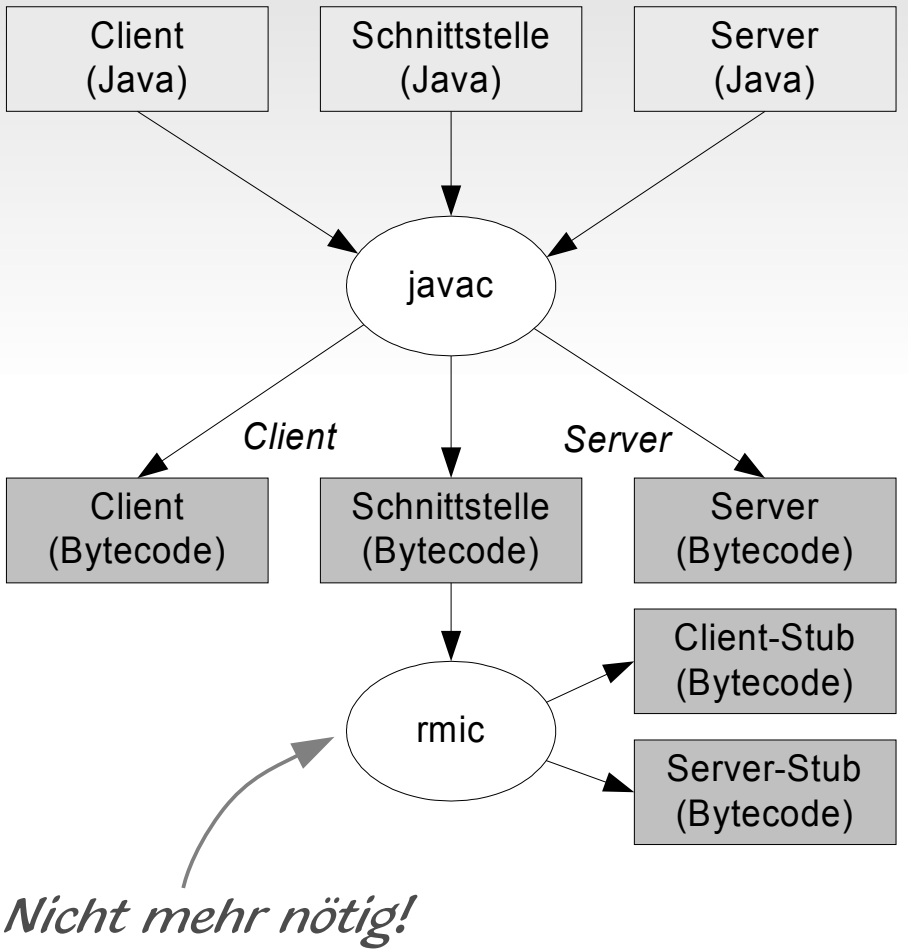
Ausprogrammieren mindestens einer Serviceklasse, die das Remote Interface implementiert

Entwicklung einer Anwendung, die ein Objekt der Serviceklasse erzeugt und beim Namensdienst registriert

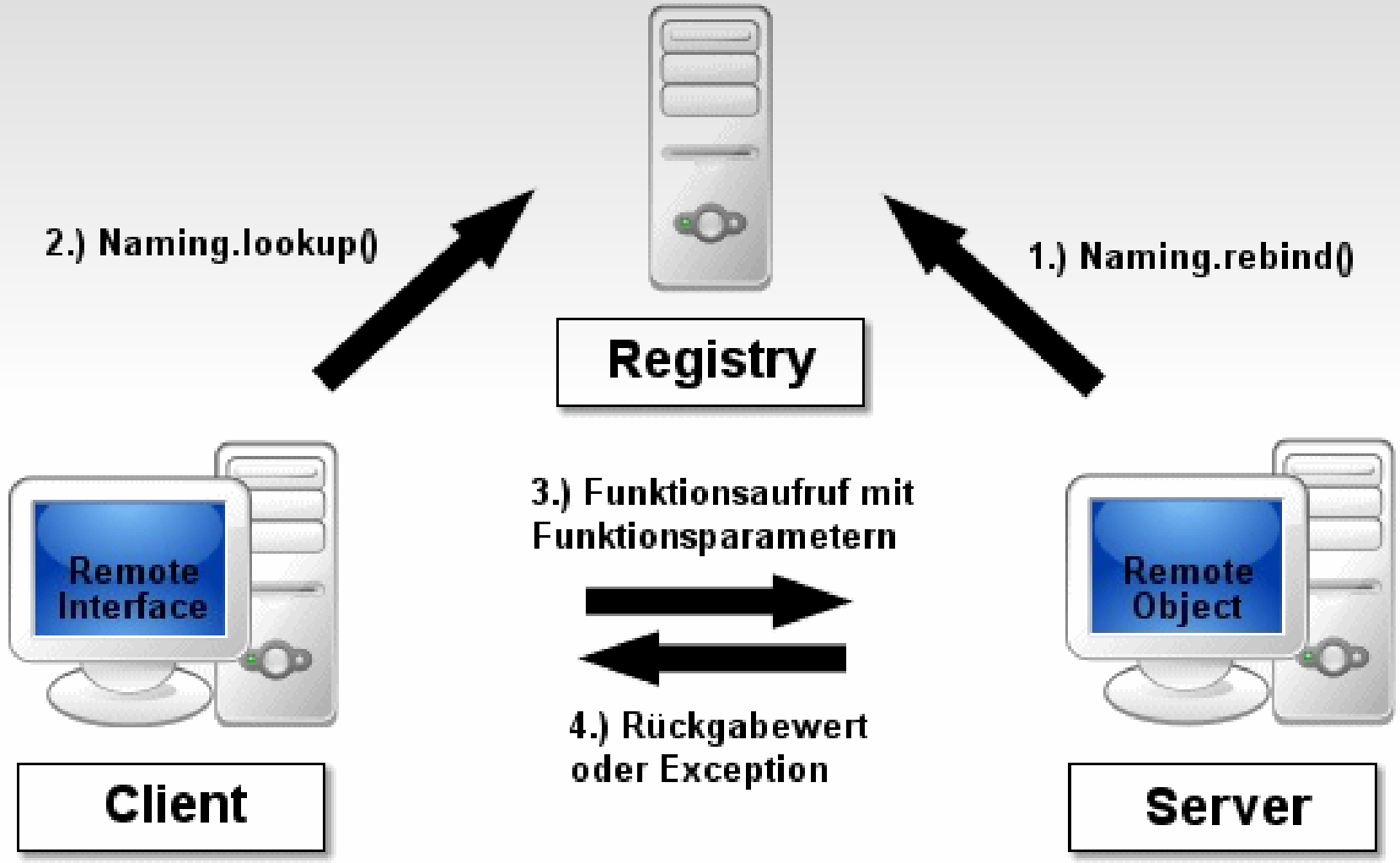
Auf Clientseite

Das kompilierte Remote Interface in die Clientanwendung kopieren

Ausprogrammieren des Clients



Vorgehen bei der Entwicklung



Remote Interface definieren

Definition der Schnittstelle durch ein Java Interface:

- Das Interface muss von **`java.rmi.Remote`** abgeleitet werden
- Jede Methode muss eine **`RemoteException`** deklarieren
- Davon abgesehen sind keine Einschränkungen vorhanden

Beispiel:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface TimeService extends Remote {
    public String getDate() throws RemoteException;
    public String getTime() throws RemoteException;

    public void setTimezone(Timezone tz)
        throws InvalidTimeZoneException, RemoteException;
}
```

Serviceklasse entwickeln

Ausprogrammieren des eigentlichen Diensts:

- Die Klasse muss das Remote Interface implementieren
- Alle Methoden des Interfaces müssen ausprogrammiert werden
- Zusätzlich muss die Klasse von **`UnicastRemoteObject`** erben

Beispiel:

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class ExampleTimeService
extends UnicastRemoteObject implements TimeService {
    public String getDate() throws RemoteException { ... }
    public String getTime() throws RemoteException { ... }
    public void setTimezone(Timezone tz) throws ...
    ...
}
```

Serveranwendung schreiben

Anmelden des Objekts beim Namensdienst:

- Erst muss das Objekt vom Serverprogramm erzeugt werden
- Dann bekommt es einen Namen in der **rmiregistry**

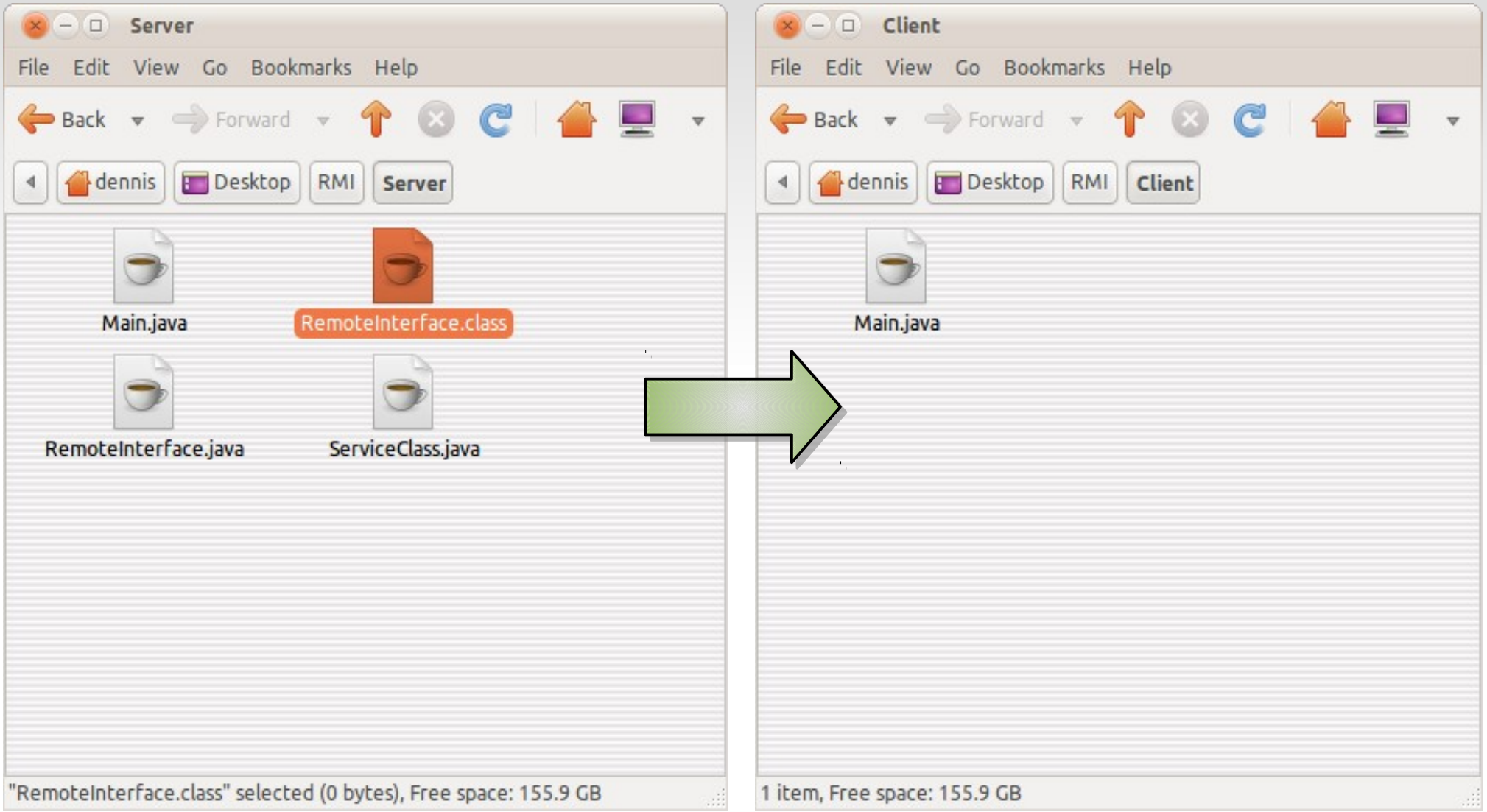
Beispiel:

```
import java.rmi.AlreadyBoundException;
import java.rmi.Naming;

public class Main {
    public static void main(String[] args) {
        TimeService timeService = new TimeService();

        try {
            Naming.bind("TimeService", timeService);
        } catch (AlreadyBoundException ex) { ... }
    }
}
```

Remote Interface kopieren



Clientanwendung schreiben

Entfernte Referenz des Objekts anfordern:

- Stub der entfernten Referenz im Namensdienst nachschlagen
- Aufruf der entfernten Methoden mit dem Stubobjekt durchführen
- Dabei auftretende **`RemoteExceptions`** behandeln

Beispiel:

```
import java.rmi.Naming;
import java.rmi.RemoteException;

public class Main {
    public static void main(String[] args) {
        try {
            TimeService ts = (TimeService)
                Naming.lookup("rmi://localhost:1099/TimeService");
            System.out.println(ts.getDate());
        } catch (RemoteExcetpion ex) { ... }
    }
}
```

Methoden der Klasse Naming

`public static void bind(String name, Remote obj)`
Registriert das durch **`obj`** referenzierte Objekt unter dem Namen **`name`**.
Gab es bereits ein Objekt dieses Namens, wird eine Ausnahme ausgelöst.

`public static void rebind(String name, Remote obj)`
Registriert das durch **`obj`** referenzierte Objekt unter dem Namen **`name`**.
Ein eventuell vorhandenes Objekt gleichen Namens wird ersetzt.

`public static void unbind(String name, Remote obj)`
Entfernt ein zuvor registriertes Objekt aus der RMI-Registry.

`public static Remote lookup(String name)`
Liefert die Referenz zu dem unter **`name`** registrierten Objekt.

`public static String[] list()`
Liefert eine Liste der Namen aller registrierten Objekte.

RMI und Java Security

Das Java Sicherheitsmodell:

- Überwachung aller Aufrufe durch den Security Manager
- Alle unsicheren Aufrufe sind standardmäßig verboten
- Entfernte Aufrufe daher nur für denselben Rechner erlaubt
- Zusätzliche Konfiguration für verteilte Anwendungen nötig

Anpassungen an den Programmen:

- Installation eines speziellen RMI-Security Managers
- Konfiguration des Security Managers durch Policy Files
- Dateiname des Policy Files beim Programmstart angeben

Beispiel: Security Manager

```
// Installation des Security Managers (Client und Server)
if (System.getSecurityManager() == null) {
    System.setSecurityManager(new RMISecurityManager());
}
```

```
// Policy File für einen Client
grant {
    permission java.net.SocketPermission
        "127.0.0.1:*", "connect,resolve";
};

// Policy File für einen Server
grant {
    permission java.net.SocketPermission
        "127.0.0.1:*", "connect,resolve";
    permission java.net.SocketPermission
        "127.0.0.1:*", "accept";
};
```

Zeig mal!

Webservices

Was ist ein Webservice?

Pragmatische Definition:

- Entfernter Prozeduraufruf auf Basis von HTTP und XML
- Ausgetauschte Nachrichten werden mit XML formuliert
- Transport der Nachrichten als HTTP-Request/Response

Ehemalige Definition des W3C:

Ein Webservice ist eine Anwendung, die mit einem Uniform Resource Identifier eindeutig identifizierbar ist und deren Schnittstelle als XML-Artefakt definiert, beschrieben und gefunden werden kann.

Was bedeutet das?

Verschiedene XML-Formate dienen …

(*definiert*) … dem Datenaustausch mit Webservices

(*beschrieben*) … der formalen Definition von Webservices

(*gefunden*) … dem Suchen und Finden von Webservices

Einschränkungen dieser Definition:

- Manche Webservices nutzen ein anderes Austauschformat wie JSON anstelle komplizierter XML-Beschreibungen
- Nicht jeder Webservice besitzt eine formale Definition
- Webservice-Verzeichnisse werden praktisch kaum genutzt

Unterschied XML ...

```
<?xml version="1.1" encoding="utf-8"?>
<ArticleList client="amazonas" date="2011-01-02">
    <Article id="1" name="Bleistiftsammlung">
        <Description>
            Belistiftesammlung des berühmten Malers Pablo Picasso.
        </Description>
        <Price>
            <Value>299.0</Value>
            <Currency>EUR</Currency>
        </Price>
    </Article>
    <Article id="2" name="Zeichenblock, DIN A4">
        <Description>
            DIN A4 Zeichenblock mit historisch anmutender Maserung.
        </Description>
        <Price>
            <Value>74.99</Value>
            <Currency>EUR</Currency>
        </Price>
    </Article>
</ArticleList>
```

... und JSON

```
{
    "client": "amazonas",
    "date": "2011-01-02",
    "articles": [
        {
            "id": 1,
            "name": "Bleistiftsammlung",
            "description": "Belistiftesammlung des berühmten Malers ...",
            "price": {
                "value": 299.0,
                "currency": "EUR",
            }
        }, {
            "id": 2,
            "name": "Zeichenblock, DIN A4",
            "description": "DIN A4 Zeichenblock mit historisch ...",
            "price": {
                "value": 74.99,
                "currency": "EUR",
            }
        },
    ]
}
```

Bedeutung von Webservices

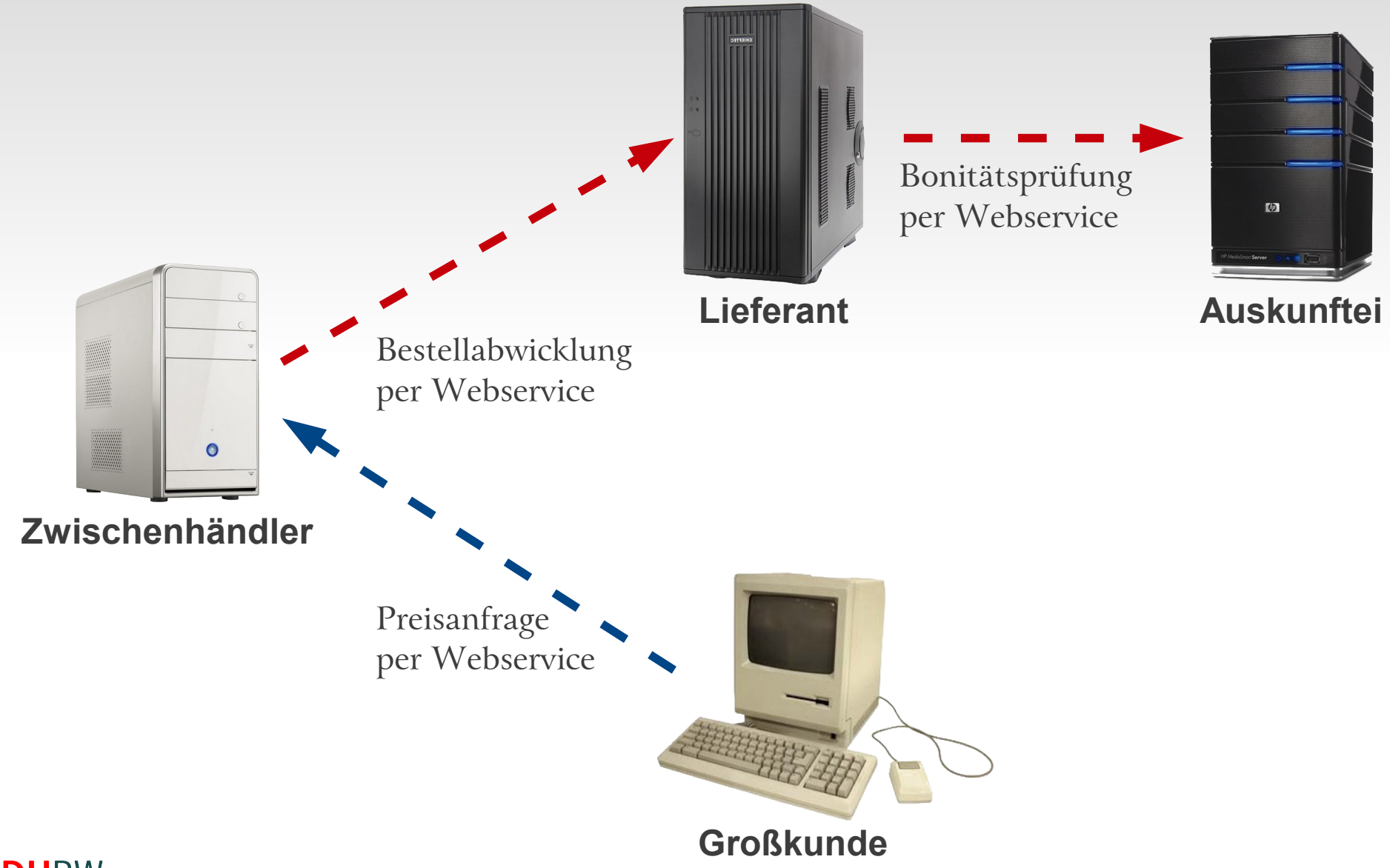
Vorteile von Webservices:

- Weite Verbreitung aufgrund gut dokumentierter Standards
- Unabhängig von Programmiersprache und Betriebssystem
- Ermöglichen die Zusammenarbeit heterogener Systeme
- Flexible Verarbeitung der XML-Nachrichten möglich
- Können mit SSL und HTTP-Proxies abgesichert werden

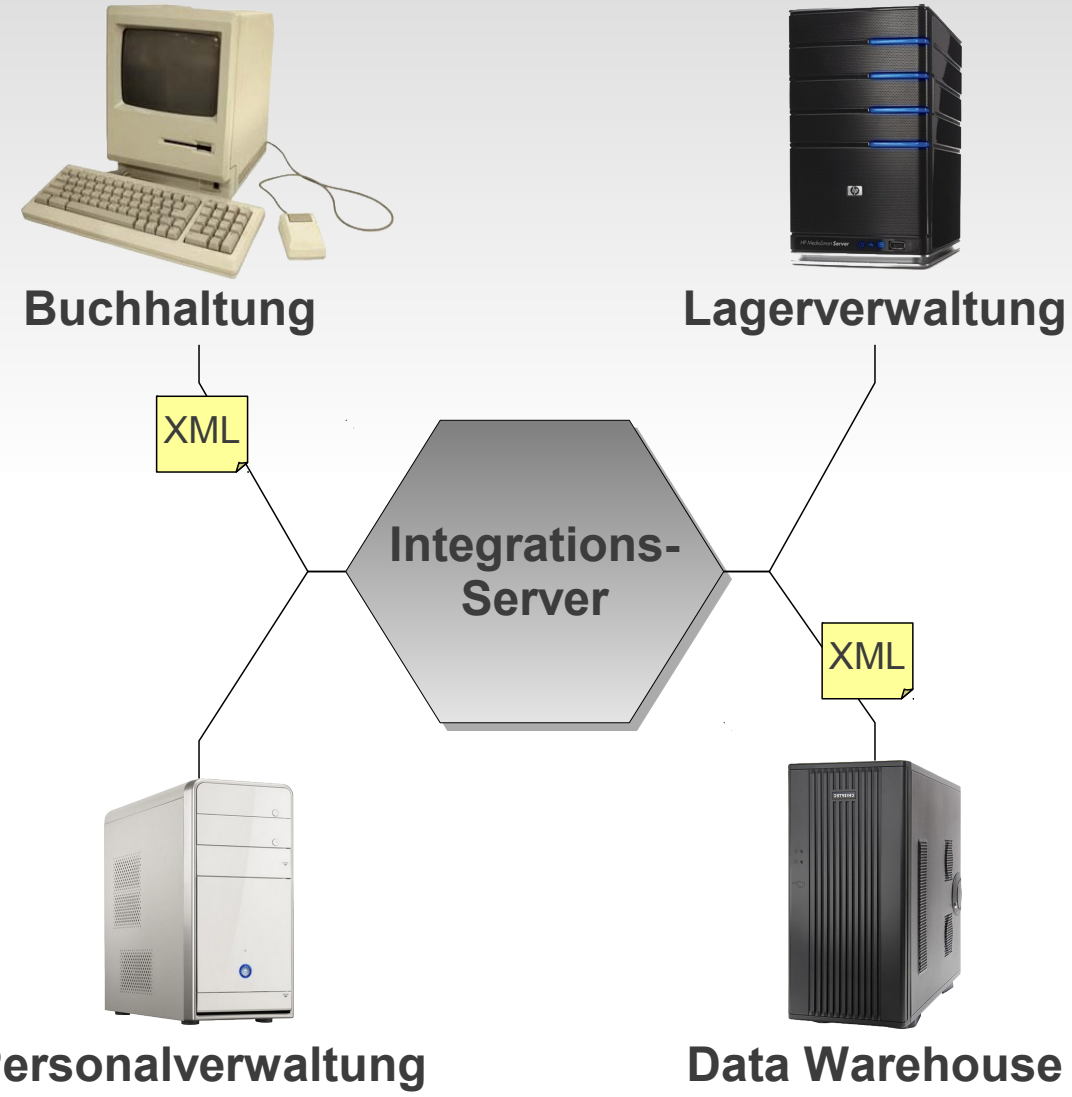
Nachteile von Webservices:

- Erhöhter Overhead aufgrund des XML-Datenaustauschs
- Daher geringere Performance als bei binären Protokollen
- Unübersehbare Anzahl an Frameworks vorhanden

Beispiel: Business-To-Business



Beispiel: Systemintegration



Zueinander inkompatible Systeme werden häufig über einen Integrationsserver miteinander verbunden. Dieser bietet daher verschiedene Schnittstellen zu den System, wobei sehr häufig Webservices benutzt werden.

Der Integrationsserver ist immer der Empfänger eines Aufrufs. Er nimmt verschiedene Aufbereitungen an den empfangenen Botschaften vor, bestimmt die betroffenen Zielsysteme und leitet die Nachrichten dann weiter.

Abgrenzung zu Web APIs

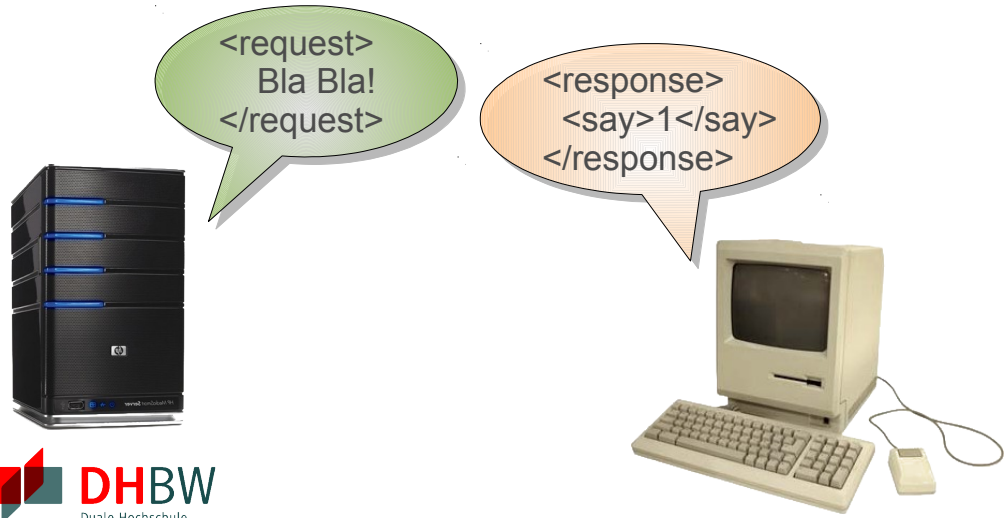
Webservices

Remote Procedure Call mit HTTP und XML

Schwerpunkt: Interoperabilität zwischen heterogenen Systemen, Service Orientierte Architekturen, Systemintegration

Einsatzgebiete: Business Anwendungen, Systemübergreifende Geschäftsprozesse, B2B-Anwendungen

Funktionsweise: Entfernte Aufrufe mit XML und HTTP als Transportmedium



Web APIs

HTTP-Basierte Onlinedienste

Schwerpunkt: Onlinedienste, Web 2.0

Einsatzgebiete: Webanwendungen, Online Communities, Mash Ups

Funktionsweise: Abruf dynamisch generierter Inhalte per HTTP-Request, Oftmals optimiert für Javascript

Wer macht was?

Anbieter:

Stellt einen Webservice zur Verfügung. Häufig handelt es sich hierbei um SOAP-Services, die durch eine WSDL-Beschreibung definiert sind.

Verzeichnis:

Listet die Webservices verschiedener Anbieter auf, damit diese einfacher gefunden werden können. Das Verzeichnis beinhaltet auch die WSDLs der Services.

Konsument:

Ruft die Webservices eines Anbieters auf. Meistens werden Services einfach direkt aufgerufen, aber auch eine Suche in einem Verzeichnis ist möglich.

Veröffentlichung von Webservices

Universal Description, Discovery and Integration:

- Branchenverzeichnis für Unternehmen und Webservices
- Listet alle Webservices auf, die ein Unternehmen anbietet
- Zugriff auf das Verzeichnis erfolgt ebenfalls per Webservice
- Ermöglicht es einem Client, Webservices automatisch zu finden
- Konnte sich allerdings nie durchsetzen, heute kaum noch relevant
- Universal Business Registry daher seit 2006 wieder eingestellt

Bestandteile des UDDI-Datenmodells:

- **White Pages:** Auflistung und Identifizierung von Unternehmen
- **Yellow Pages:** Klassifizierung der angebotenen Webservices
- **Green Pages:** Technische Beschreibungen der Webservices

Beschreibung von Webservices

Webservice Description Language:

- Schnittstellensprache zur Beschreibung von Webservices
- Definiert alle aufrufbaren Operationen eines Webservices
- Beschreibt die beim Aufruf ausgetauschten Nachrichten
- Beinhaltet Schemata aller Strukturen und Datentypen

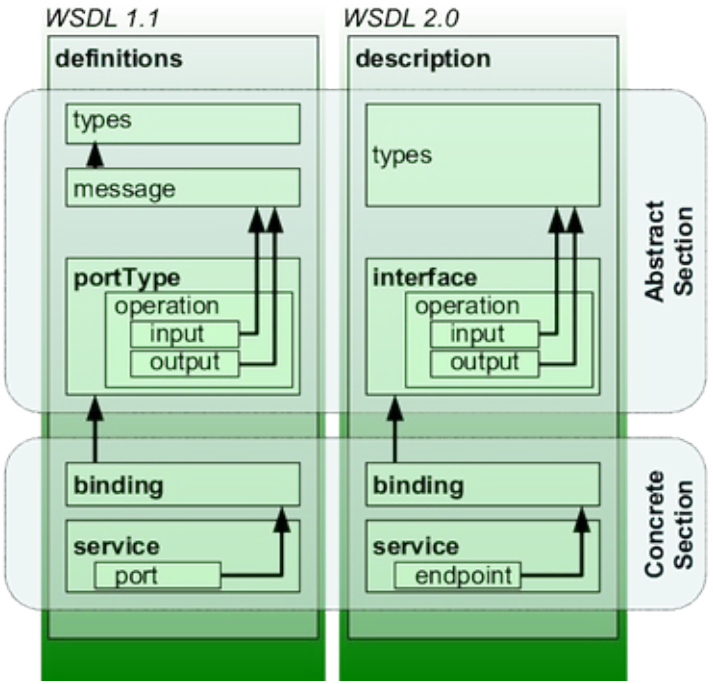
Inhalte der WSDL:

Typdefinitionen

Interfaces

Protokollbindungen

Endpunkte (URL-Mapping)



Inhalt der WSDL-Beschreibung

Typdefinitionen

Alle Datenstrukturen, die mit dem Webservice ausgetauscht werden, müssen mit XMLSchema beschrieben werden.

Interfaces / Operationen

Ein Interface fasst mehrere, aufrufbare Operationen zusammen. Dabei wird für jede Operation definiert, welche Nachrichten Client und Server austauschen und welchem Typ diese entsprechen.

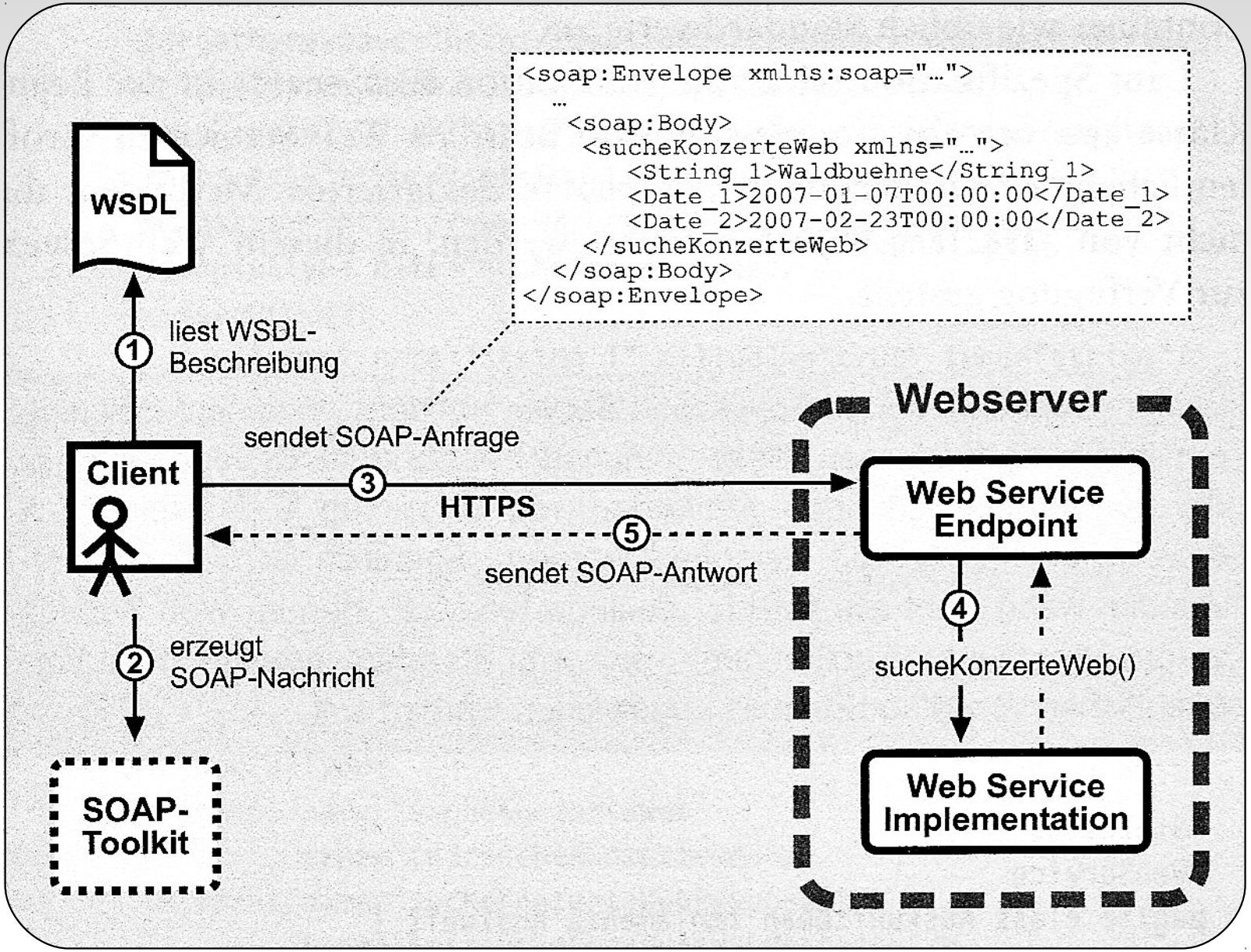
Protokollbindungen

Zu jedem Interface muss es mindestens ein Binding geben, das definiert, wie Nachrichten übertragen werden. (Zum Beispiel direkt per HTTP-Aufruf oder eingebettet in ein SOAP-Dokument)

Services / Endpunkte

Jeder Service umfasst mindestens einen Endpunkt. Über die Endpunkte wird den Bindings eine URL zugeordnet.

Verwendung der WSDL



Beispiel: WSDL-Beschreibung

```
<description>
    <!-- Beschreibung der Datentypen und Strukturen -->
    <types>
        <xs:schema xmlns:xs="...">
            <element name="GetAllBooksRequest"> ... </element>
            <element name="GetAllResponse"> ... </element>
            <element name="Book"> ... </element>
        </xs:schema>
    </types>

    <!-- Definition der aufrufbaren Operationen -->
    <interface name="BookStoreInterface">
        <operation name="GetAllBooks"
                   pattern="http://www.w3.org/ns/wsdl/in-out">
            <input messageLabel="In"
                   element="tns:GetAllBooksRequest"/>
            <output messageLabel="Out"
                    element="tns:GetAllBooksResponse"/>
        </operation>
    </interface>
```

Beispiel: WSDL-Beschreibung

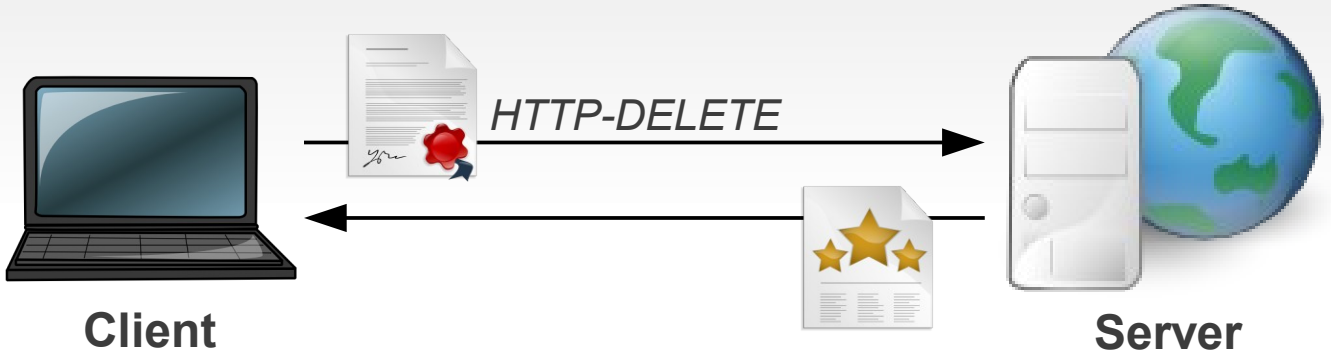
```
    <!-- Einzelne Operationen per SOAP verfügbar machen -->
    <binding name="BookStoreBinding"
             interface="tns:BookStoreInterface"
             type="http://www.w3.org/ns/wsdl/soap" ... >
        <operation ref="tns:GetAllBooks"/>
    </binding>

    <!-- Endpunkte / URLs definieren -->
    <service name="BookStoreService"
             interface="tns:BookStoreInterface">
        <endpoint name="BookStoreSoapEndpoint"
                  binding="tns:BookStoreBinding"
                  address="http://example.com/soapws/BookStore"/>
    </service>
</description>
```

Verschiedene Bindings

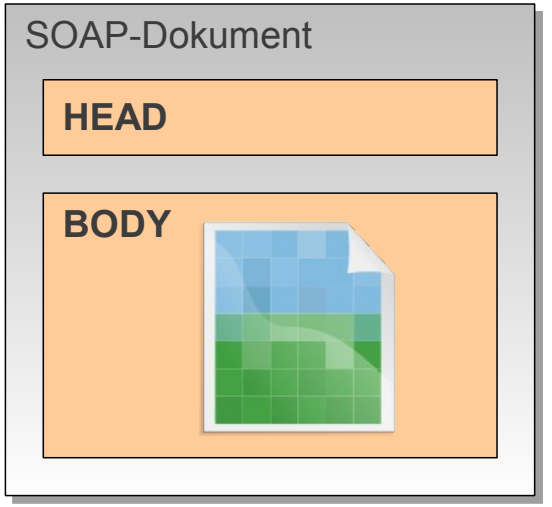
RESTful Binding:

Die Nachrichten werden unverändert so übertragen, wie sie in der Webservice-Beschreibung definiert wurden. Da eine URI immer einer Ressource entspricht (z.B. einem Datensatz in der Datenbank) werden jedem HTTP-Verb andere Nachrichten zugeordnet.



SOAP-Binding:

Die übertragenen Nachrichten werden in einem SOAP-Dokument verpackt, das in die drei festen Bereiche *Header*, *Body* und *Fault* gegliedert ist. Da es für den Webservice in der Regel nur eine URI gibt, sind nur HTTP-POST Anfragen zugelassen.



XML-RPC Webservices

Entfernte Aufrufe mit XML-RPC:

- Leichtgewichtiger Standard für webbasierte Prozeduraufrufe
- Transportiert Prozeduraufrufe ausschließlich über HTTP/XML
- Keine formale Beschreibung der Webservices vorgesehen
- Definiert nur, wie Prozeduraufrufe zu XML serialisiert werden

Datentypen des XML-RPC Standards:

Booleans
Datums-/Zeitangaben
Zeichenketten
Ganzzahlen
Base64-Binärdaten
Arrays
Kommazahlen
Komplexe Strukturen

Die Apache XML-RPC Bibliothek:

- Open Source-Implementierung des Standards für Java
- Ermöglicht die einfache Entwicklung von Clients und Servern
- Erweitert den Standard um alle eingebauten Java Datentypen

Beispiel: XML-RPC Request

```
<?xml version="1.1" encoding="utf-8"?>
<methodCall>
    <!-- Name der aufgerufenen Prozedur -->
    <methodName>CalculatorService.add</methodName>

    <!-- Parameter der Methode -->
    <params>
        <param>
            <value>
                <int>42</int>
            </value>
        </param>
        <param>
            <value>
                <int>38</int>
            </value>
        </param>
    </params>
</methodCall>
```

Beispiel: XML-RPC Response

```
<?xml version="1.1" encoding="utf-8"?>
<methodReponse> <params> <param>
    <value>
        <!-- Rückgabe einer komplexen Struktur -->
        <!-- anstatt eines einfachen Werts -->
        <struct>
            <member>
                <name>FirstValue</name>
                <value>42</value>
            </member>
            <member>
                <name>SecondValue</name>
                <value>38</value>
            </member>
            <member>
                <name>Result</name>
                <value>80</value>
            </member>
        </struct>
    </value>
</param> </params> </methodResponse>
```

Beispiel: XML-RPC Server

Datei WEB-INF/de/dhbw/xmlrpc/Calculator.java

Eine ganz normale Klasse ohne irgendwelche Besonderheiten!

```
public class Calculator {
    public int add(int i, int j) {
        return i + j;
    }

    public int sub(int i, int j) {
        return i - j;
    }

    ...
}
```

Datei WEB-INF/org/apache/xmlrpc/webserver/XmlRpcServlet.properties

```
CalculatorService=de.dhbw.xmlrpc.Calculator
```

Beispiel: XML-RPC Server

Deployment Descriptor WEB-INF/web.xml

Weist dem XML-RPC Servlet eine URL zu

```
<web-app version="3.0">
    <servlet>
        <servlet-name>XmlRpcServlet</servlet-name>
        <servlet-class>
             org.apache.xmlrpc.webserver.XmlRpcServlet
        </servlet-class>
        <init-param>
            <param-name>enabledForExtensions</param-name>
            <param-value>true</param-value>
            <description>Apache Erweiterungen zulassen</description>
        </init-param>
    </servlet>

    <servlet-mapping>
        <servlet-name>XmlRpcServlet</servlet-name>
        <url-pattern>/xmlrpc</url-pattern>
    </servlet-mapping>
</web-app>
```

Beispiel: XML-RPC Client

```
import org.apache.xmlrpc.client.*;
import org.apache.xmlrpc.*;
import java.net.*

public class TestCalculator {
    public static void main(String[] args)
    throws MalformedURLException, XmlRpcException {
        // XML-RPC Client erzeugen
        XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();
        config.setServerURL(new URL("http://localhost:8080/ws/xmlrpc"));

        XmlRpcClient client = new XmlRpcClient();
        client.setConfig(config);

        // Entfernten Webservice aufrufen
        Object[] parameters = new Object[] {
            new Integer(28), new Integer(42)
        };
        Integer result = (Integer)
            client.execute("Calculator.add", parameters);
    }
}
```

JAX-WS und SOAP

SOAP-Basierte Webservices:

- Konzeptionelle Weiterentwicklung des XML-RPC Standards
- Unabhängig von HTTP und XML zum Nachrichtenaustausch
- Mehr Aufrufarten, zum Beispiel Anfragen ohne Antwort
- Basiert auf dem Versand frei definierbarer Nachrichten, welche in ein SOAP-Dokument eingebunden werden
- Formale Beschreibung der Webservices durch WSDL-Artefakte

Java API for XML-Webservices:

- Einheitliche API zum Erstellen und von SOAP-Webservices
- Einfache Entwicklung mit Annotationen statt XML-Konfiguration
- Fester Bestandteil des Java Development Kit seit Java 5

Beispiel: SOAP-Request

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:soap="beispiel.de">
   <soapenv:Header/>
   <soapenv:Body>
      <soap:orderBooks>
         <book>
            <author>Dietmar Ratz</author>
            <isbn>978-3-446-41655-0</isbn>
            <price>44.9</price>
            <title>Grundkurs Programmieren in Java</title>
         </book>
         <book>
            <author>John Vlissides</author>
            <isbn>978-3-8273-1544-1</isbn>
            <price>29.99</price>
            <title>Entwurfsmuster anwenden</title>
         </book>
      </soap:orderBooks>
   </soapenv:Body>
</soapenv:Envelope>
```

Eigentliche Nachricht

Beispiel: SOAP-Response

```
<soapenv:Envelope xmlns:soapenv="..." xmlns:soap="beispiel.de">
    <soapenv:Header/>
    <soapenv:Body>
        <soap:orderBooksResponse>
            <orderNumber>4389289239-21</orderNumber>
            <total>74.89</total>
        </soap:orderBooksResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Antwortnachricht

SOAP-Dokumente unterscheiden nicht zwischen Anfrage und Antwort, weil es sich nur um eine Verpackung frei definierbarer Nachrichten handelt.

Entwicklung mit JAX-WS

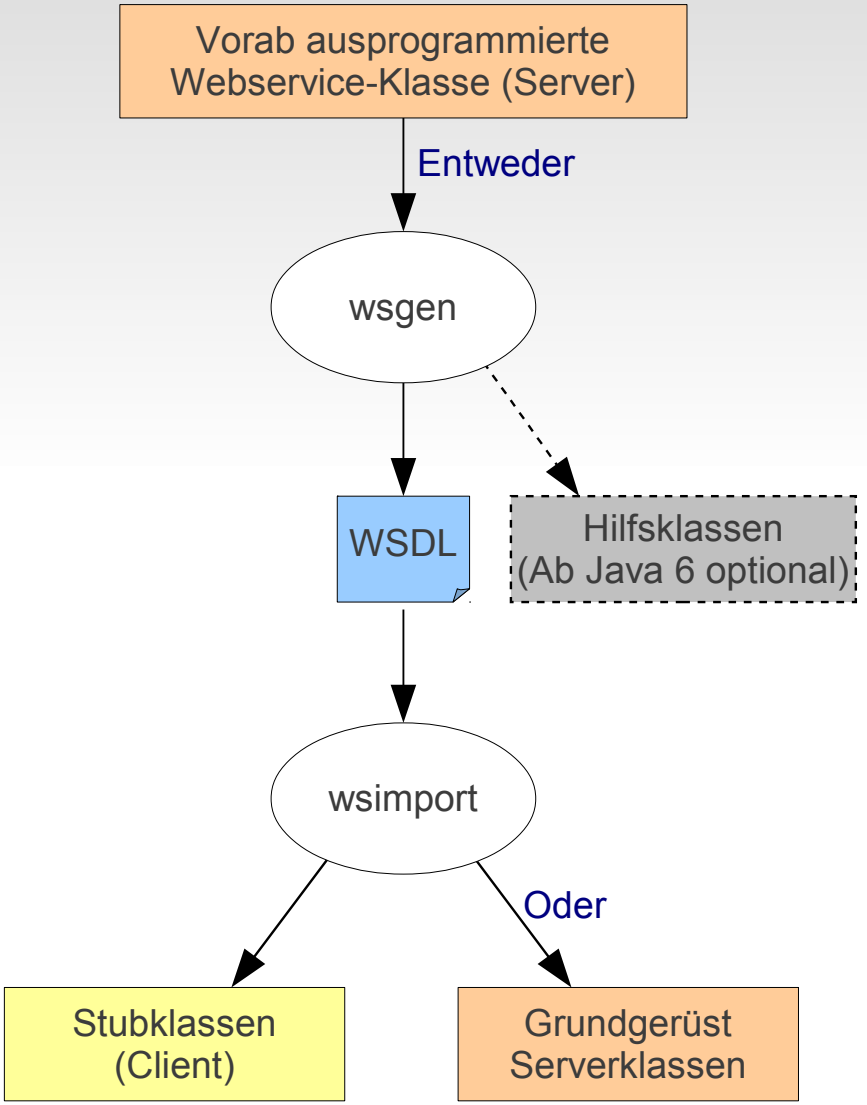
Auf Serverseite

Code First: Umwandlung einer bestehenden Klasse zu einem Webservice. Die WSDL wird automatisch generiert.

Contract First: Manuelle Erstellung der WSDL-Beschreibung. Aus dieser wird das Grundgerüst der auszuprogrammierenden Klassen generiert.

Auf Clientseite

Automatische Erzeugung der Stubklassen aus der WSDL-Definition.



JAX-WS Annotationen

@WebService

Kennzeichnet eine öffentliche, nicht-abstrakte Klasse mit öffentlichem und parameterlosten Konstruktor als Webserviceimplementierung.

@WebMethod

Markiert eine öffentliche, nichtstatische Methode, so dass diese eine aufrufbare Operation des Webservices darstellt.

@WebParam

Ermöglicht es, einzelne Parameter einer Methode abweichend von den Standardvorgaben des Frameworks zu konfigurieren, indem den Parametern zum Beispiel andere Namen gegeben werden.

@WebResult

Erlaubt es, den Namen des Rückgabewerts zu ändern.

Beispiel: SOAP-Webservice

```
import java.util.List;
import javax.jws.*;

@WebService(name="BookStore", targetNamespace="soap.dhbw.de")
public class BookStoreFacade {

    public BookStoreFacade() { }

    @WebMethod
    @WebResult(name="book")
    public List<Book> getAvailableBooks() { ... }

    @WebMethod
    @WebResult(name="total")
    public double orderBooks(@WebParam(name="book")
                             List<Book> books) {
        ...
    }
}
```

Beispiel: Testserver

```
import javax.xml.ws.Endpoint;

public class SoapTestServer {
    public static void main(String[] args) {
        String address = "http://localhost:8080/BookStore";
        BookStoreFacade service = new BookStoreFacade();
        Endpoint endpoint = Endpoint.publish(address, service);
    }
}
```

Dieses kleine Programm implementiert einen einfachen Testserver, der den Aufruf des Webservices ermöglicht. Er eignet sich hervorragend für Minianwendungen und lokale Tests des Webservices. In einer echten Anwendung würden Sie den Webservice stattdessen in einem Webcontainer oder Applikationsserver zum Laufen bringen.

Wenn Sie einen Webservice in einem Applikationsserver deployen, ist die Entwicklung sogar noch einfacher, weil Sie neben der Webserviceklasse nichts weiter programmieren müssen. (Siehe nächste Folie).

Beispiel: EJBs als Webservice

```
import java.util.List;
import javax.ejb.Stateless;
import javax.jws.*;

@Stateless
@WebService(name="BookStore", targetNamespace="soap.dhbw.de")
public class BookStoreFacade {
    public BookStoreFacade() { }

    @WebMethod
    @WebResult(name="book")
    public List<Book> getAvailableBooks() { ... }

    @WebMethod
    @WebResult(name="total")
    public double orderBooks(@WebParam(name="book")
                             List<Book> books) {
        ...
    }
}
```

Beispiel: SOAP-Client

```
import de.dhbw.soap.*;     // Generierte Clientklassen
import java.util.*;

public class SimpleSOAPClient {
    public static void main(String[] args) {
        // Lokalen Stellvertreter für den Webservice erzeugen
        BookStoreService service = new BookStoreService();
        BookStore webservice = service.getBookStorePort();

        // Webservice aufrufen
        List<Book> books = webservice.getAvailableBooks();

        for (Book book : books) {
            System.out.println("Titel:  " + book.getTitle());
            System.out.println("Author: " + book.getAuthor());
            System.out.println("ISBN:   " + book.getIsbn());
            System.out.println("Preis:  " + book.getPrice() + " EUR");
            System.out.println();
        }
    }
}
```

1. Sicherstellen, dass der Webservice aufgerufen werden kann
2. **`wsimport http://localhost:8080/BookStore?wsdl`**
3. Clientanwendung kompilieren