Hochschule **Rosenheim**
University of Applied Sciences

Verteilte Verarbeitung

Kapitel 8
Bestandteile einer Middleware
Grundlegende Entwurfsmuster

Lernziel

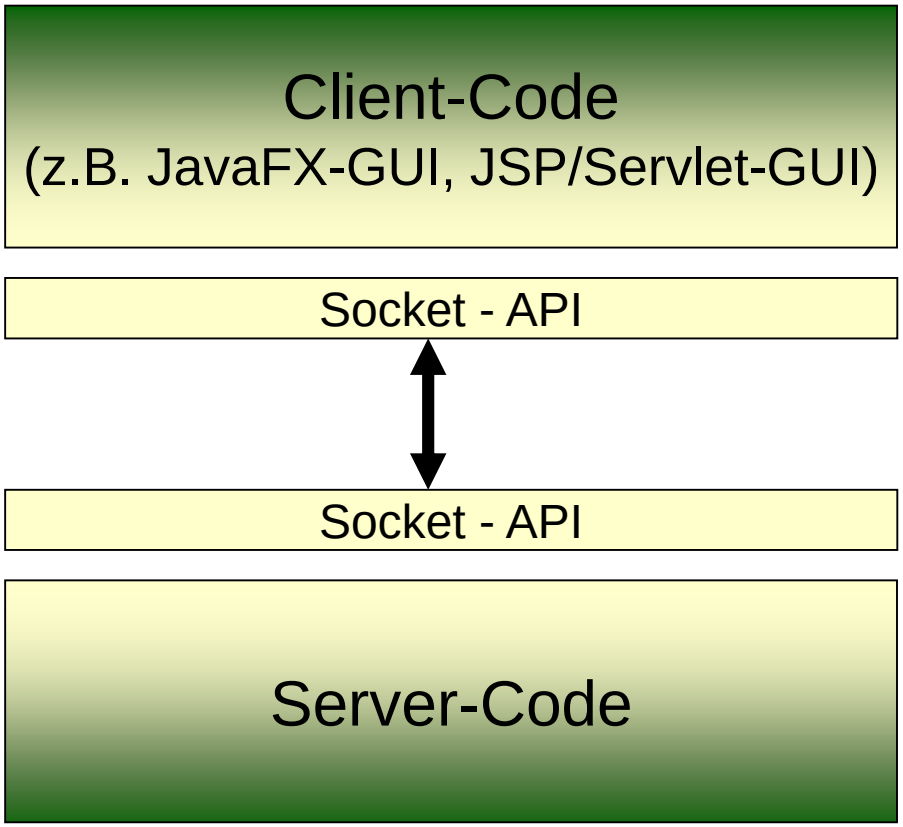
- Besseres Verständnis von
  - Kommunikationsprotokollen
  - Middleware
- Patterns zeigen Grundfunktionen / Architekturelemente, wie sie in jeder Middleware vorkommen [vgl. „POSA 1“ und „Remoting Patterns“]
- Beispiel im Foliensatz und Übungsaufgabe: Selbst gebautes RMI

- ***Wir bauen hier eine eigene Middleware***

Sockets und „Drauflos-Programmierung“

Socket-API (und Aufrufe)

Fachlicher Code



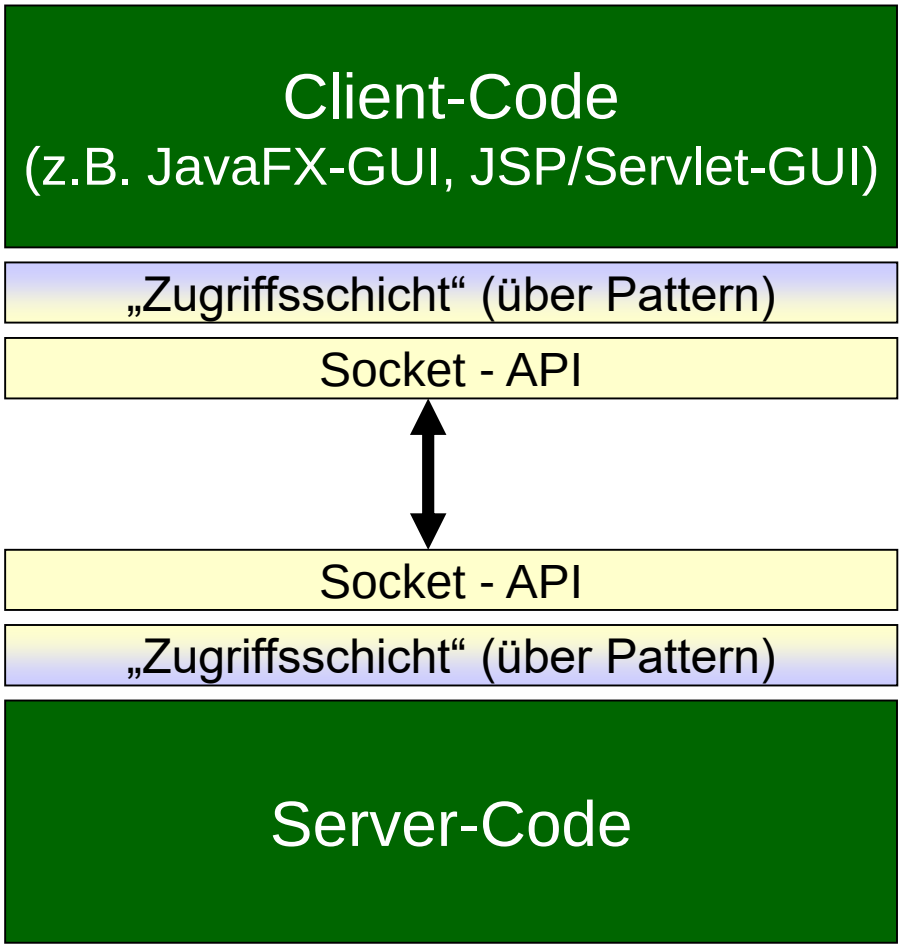
- Socket-API wird sehr oft im Code aufgerufen (ist ***verstreut***)
- Client / Server Protokoll ist verstreut
- Fehlerbehandlung verstreut

- Folgen:
  - jeder Programmierer muss alle Details des Socket-API kennen
  - Änderungen werden aufwändig
  - Code ist uneinheitlich („Software-Slum“)
  - Copy & Paste Fehler

Sockets und Pattern

Socket-API (und Aufrufe)

Fachlicher Code



- Socket-API wird gekapselt
- Für jede Aufgabe eigenes Architekturelement (Pattern)

- Folgen:
  - nur Spezialisten müssen Socket-API kennen
  - Änderungen einfacher
  - Netzwerk Code ist an einer Stelle konzentriert

Einkaufszettel: Entwurfsmuster

- **`Message`**
  - kapsele Nachrichten in eigene Klasse, z.B. Request, Response, Error
- **`Forwarder / Receiver`**
  - kapsele Versenden und Empfangen von Nachrichten
  - kapsele damit die Details des Socket-API
- **`Marshaller`**
  - Kapsele das (De-)Serialisieren von Nachrichten / Parametern
- **`Dispatcher`** (**`Broker`** „light“)
  - Kapsele das Auffinden des Servers
- **`Remote Proxy`** und Server-Skeleton
  - Lokales Objekt hat dasselbe Interface, wie entferntes Objekt
  - Remote Proxy und Server-Skeleton implementieren Kommunikation
- … einige weitere unterstützende Muster

Entwurfsproblem:

Nachricht = feste Bytefolge, jedes Byte andere Bedeutung

Kapitel: **Schaden** Stand: **17.05.2013**

Bezeichnung: **Allgemeine Schadendaten** Satzart: **4200**

Version: **4.0**

Teildatensatz 1

Nr.	Bezeichnung	Darst. AN/N	Anzahl Bytes	Byte-Adr.	M M/K	Inhalt / Erläuterung
1	Satzart	N	4	1	M	konstant 4200
2	Versionsnummer	N	3	5	M	konstant 004
3	VU-Nummer	AN	5	8		Das VU-Nr.-Verzeichnis kann bei der Bundesanstalt für Finanzdienstleistungsaufsicht in Bonn angefordert werden (Graurheindorfer Str. 108, 53117 Bonn) und steht online unter http://www.bafin.de/dok/2675598 zur Verfügung.
4	Versicherungsschein-Nummer	AN	17	13		Versicherungsschein-Nummer siehe allgemeine Informationen, Abschnitt 6 ********** Diese Feld ist nur noch aus Kompatibilitätsgründen vorhanden. Da die VSNr oftmals länger ist, sollte ausschließlich die 35-stellige VSNr im Vorsatz verwendet werden! **********
5	Schaden-Nummer des VU	AN	20	30		Schadennummer des Versicherers siehe allgemeine Informationen, Abschnitt 6.
6	Aktenzeichen des (IT-)Dienstleisterpartners	AN	20	50		z. B. Aktenzeichen/Rechnungsnummer einer (IT-)Dienstleisterpartnernummer (z.B. Werkstatt)
7	Satznummer	N	1	70	M	konstant 1
8	Schaden-Nummer des Vermittlers	AN	20	71		Schaden-Nummer der Vermittlers
	Schadenort					
9	- Länderkennzeichen	AN	3	91		KFZ-Länderkennzeichen, z.B. D = Deutschland B = Belgien DK = Dänemark F = Frankreich
10	- Postleitzahl	AN	6	94		Postleitzahl
11	- Ort	AN	25	100		Ort
12	- Straße	AN	30	125		Straße
13	- Ergänzende Beschreibung zum Schadenort	AN	80	155		Ergänzende Ortsbeschreibung (z.B. "Fahrzeug steht in der Garage")
14	Schadendatum	AN	8	235		Sollten Tag und/oder Monat nicht bekannt sein, muss "00" geschlüsselt werden

Projekt Schadennetze GDV

http://www.gdv-online.de
/snetz/release2013/ds4200.htm

Entwurfsprobleme – Bau einer Middleware

- Kapselung der Technik
  - Änderbarkeit/Optimierung an genau einer Stelle
  - Austauschbarkeit
  - Nur wenige Teammitglieder müssen Technik kennen
- Gute „Abstraktionen“ für Design/Programmierung
  - „Nachricht“ statt Byte-Wurst
  - „Sender/Empfänger“ statt Sockets
  - Proxy statt Sockets
- Denn: Sprache bestimmt das Denken!
  - vgl. Ludwig Wittgenstein und andere
  - Mentales-Modell des Lesers sollte nahe an dem sein, was sie mit ihrem Code meinen (= ihr nentales Modell)

Message-Pattern

- Idee: Eine Basisklasse für alle Nachrichten
  - Entspricht u.a. SOAP-Envelope
  - Entspricht Messages in Nachrichten orientierter Middleware
- Für jeden Typ eigene Unterklasse Message-Klasse (z.B. Request, Response, Fault, Document, Ack, …)
- Abstraktion: ***Netzwerk-Kommunikation == Nachrichtenaustausch***
  - Kapselung von (Meta-)Informationen über Nachrichten

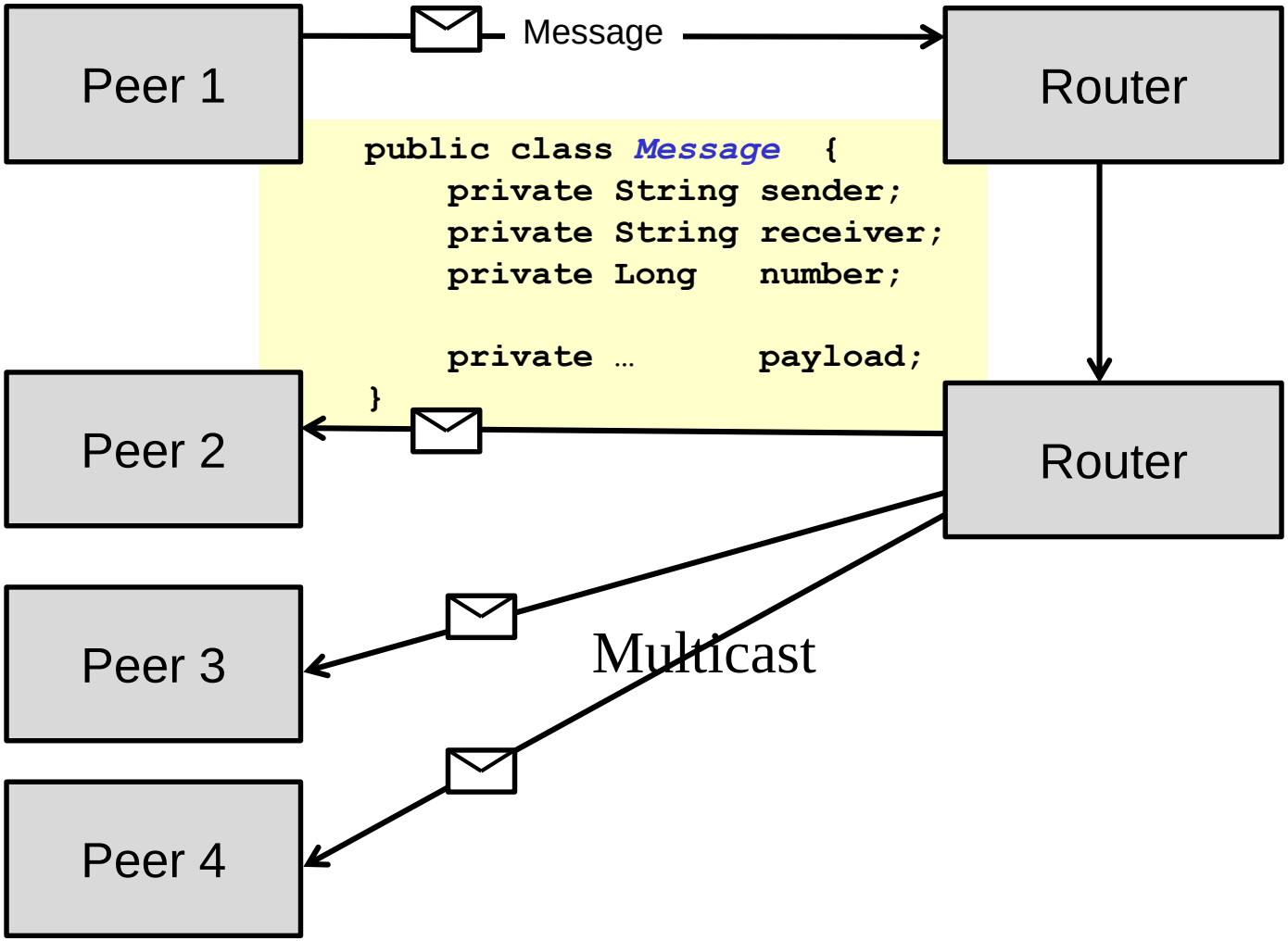
```
public class Message implements Serializable {
// …
}
```

Typen von Nachrichten
Nachricht mit verschiedenen Bedeutungen …

- Kommandos
  - Beispiel: ***Request*** (anlBestellung(…)), ***Response*** (= OK)
- Dokumente / Daten
  - Beispiel: Bestellung 4711, Schadenereignis 0815
- Ereignisse
  - Beispiel: Bestellung 4711 ist eingegangen

- Kombinationen der Typen möglich,
  z.B. Ereignis + Dokument

Beispiel
Routebare Nachricht (wie UDP – Datagramme)



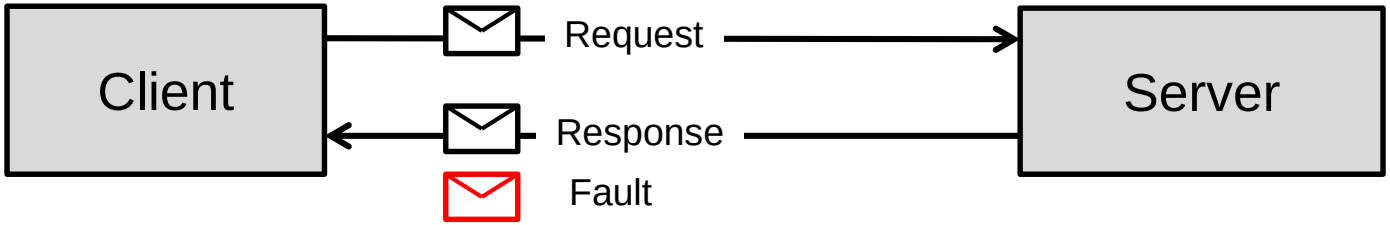
Nachrichten mit Queue als Zwischenspeicher


```
public class Message  {
    private String sender;
    private String receiver;
    private Long   number;
}
```

- Queue = Zwischenspeicher für Nachrichten
- Peer 1 sendet Nachricht
- Queue empfängt Nachricht
- Peer 2 holt Nachricht ab / Queue sendet Nachricht

Beispiel
Methodenaufrufe mit Nachrichten

Entfernter Prozedur-/Methodenaufruf: Request / Response



Request enthält:

- Entfernte Referenz auf das Objekt, das aufgerufen werden soll
- Name der Methode
- Parameter also z.B.: Object[] args

Response enthält:

- Rückgabewert der Methode

`Message` für Methodenaufrufe

Idee:

- ***Request-Nachricht*** enthält die ID des aufzurufenden Objekts sowie Namen der Methode und die Parameter
- ***Response-Nachricht*** enthält die Rückgabewerte
- ***Fault***-Nachricht zeigt Kommunikationsproblem oder Serverfehler

```
public class Request extends
Message
{
    private String objectId;
    private String methodName;
    private Object[] parameters;

    public Request(
        String objectId,
        String methodName,
        Object[] parameters)
    {
        // …
    }
    // …
}
```

```
public class Response
            extends Message {

    private Object result;

    public Response(Object result){
        // …
    }
    // …
}
```

```
public class Fault
            extends Message {

    private String reason;
    // ...
```

Beispiel: (SOAP-) WebService - Messages

Request-Nachricht

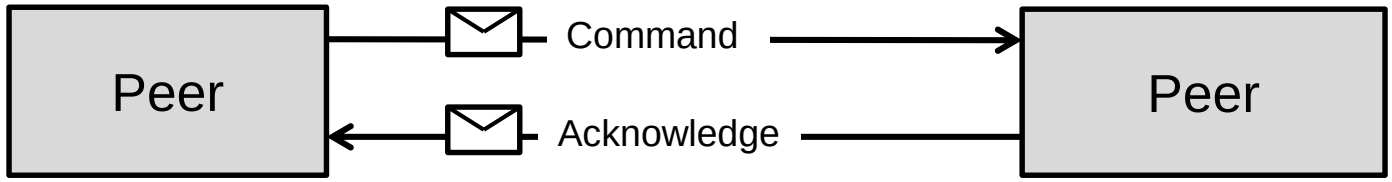
```
<s:Envelope ...>
<s:Header>
  <a:Action ...>
    http://.../IHelloService/SayHello
  </a:Action>
  ...
</s:Header>
<s:Body>
  <SayHello ...>
    <name>Willi Winzig</name>
  </SayHello>
</s:Body>
</s:Envelope>
```

Response-Nachricht

```
<s:Envelope ...>
<s:Header>
  <a:Action ...>
    http://.../IHelloService/SayHelloResponse
  </a:Action>
  ...
</s:Header>
<s:Body>
  <SayHelloResponse ...>
    <SayHelloResult>Hello Willi Winzig!
    </SayHelloResult>
  </SayHelloResponse>
</s:Body>
</s:Envelope>
```

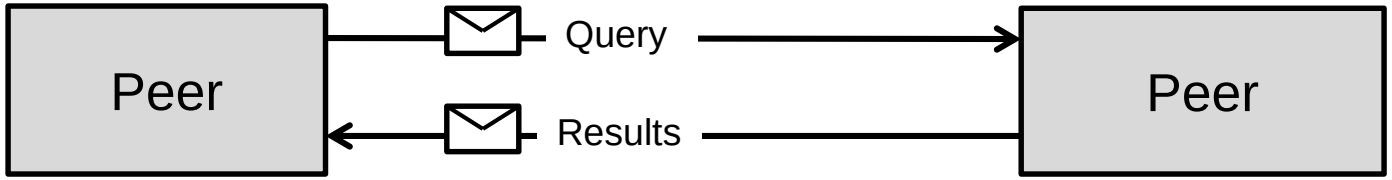
Beispiel Kommando

Entferntes Kommando



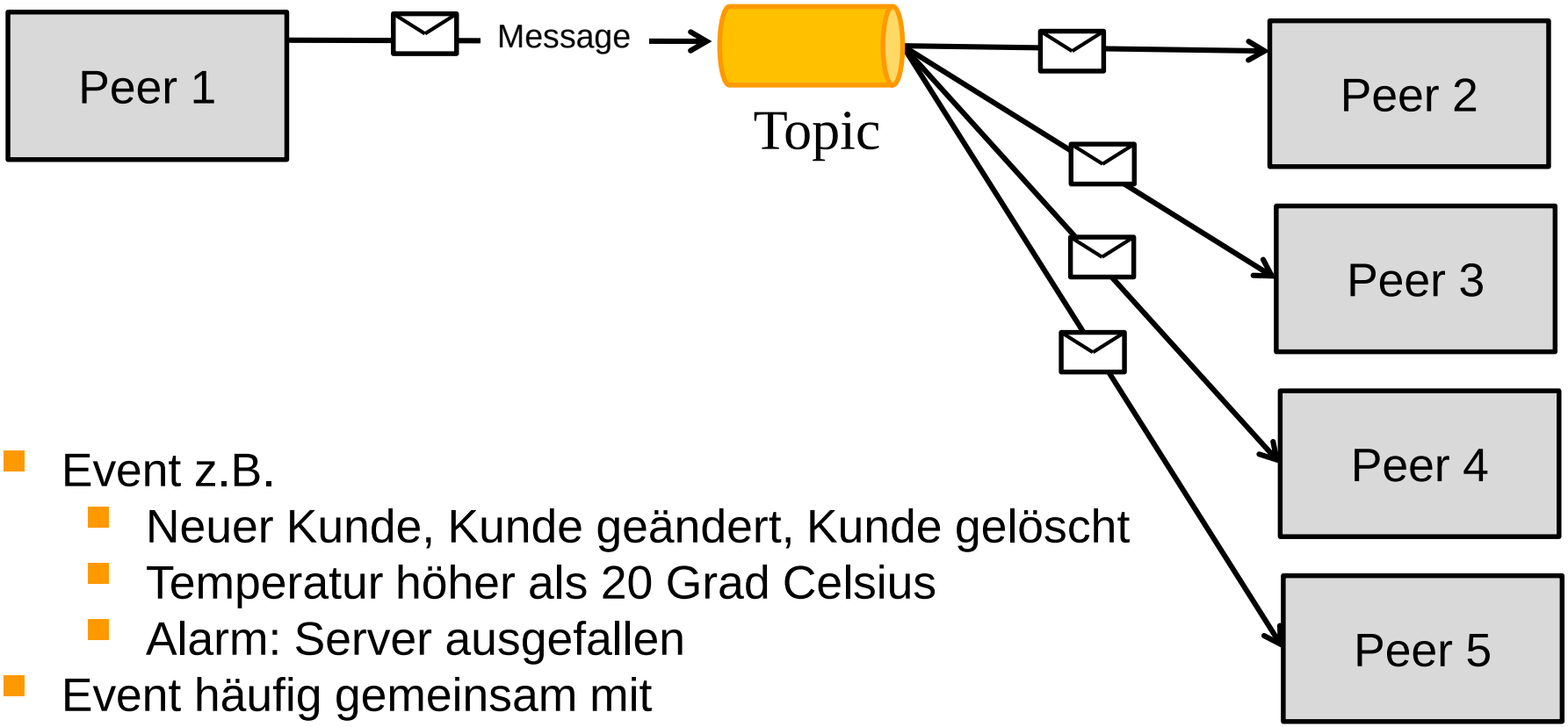
- Command enthält: Z.B. Name des Kommandos: ShutDown, Reset, …
- Acknowlede: Ggf. Nummer des Commands, damit es zugeordnet werden kann

Entfernte Query



- Query enthält: Z.B. Name der Query oder direkt Anfrage
- Results: „Treffer“ der Query

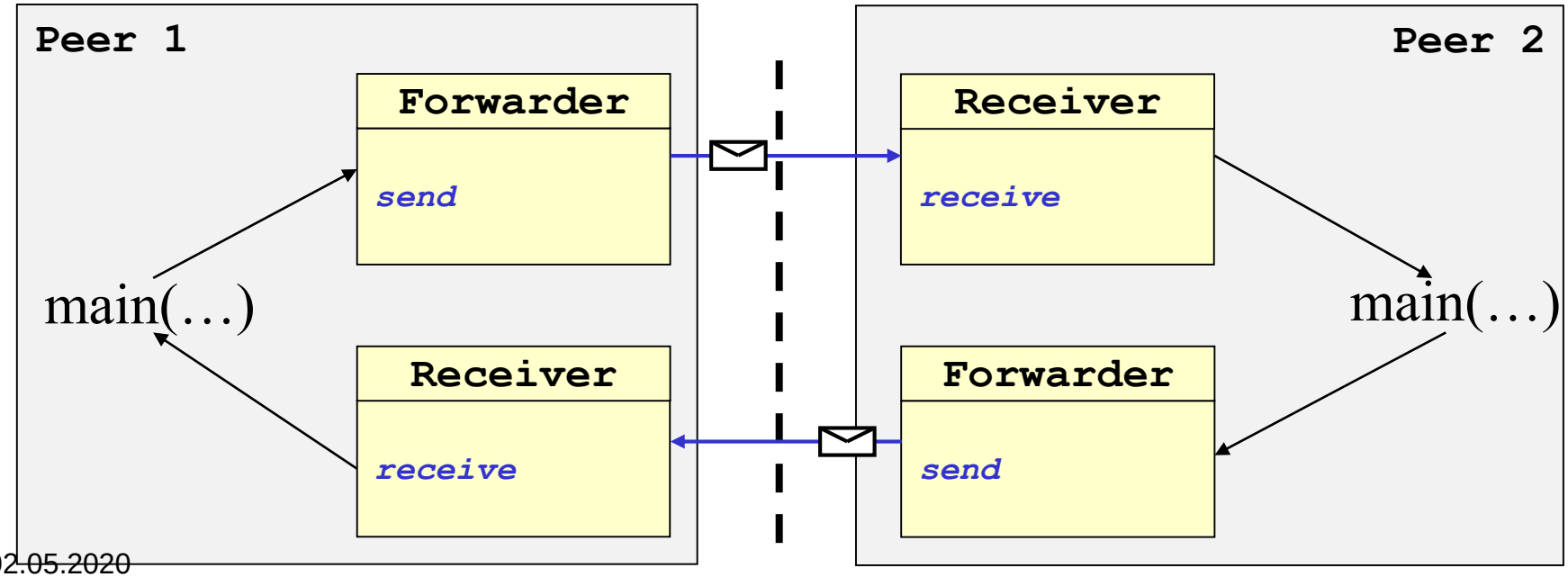
Beispiel Event



- Event z.B.
  - Neuer Kunde, Kunde geändert, Kunde gelöscht
  - Temperatur höher als 20 Grad Celsius
  - Alarm: Server ausgefallen
- Event häufig gemeinsam mit
  a) Observer-Pattern (synchron)
  b) Publisher Subscriber-Pattern (asynchron)

Forwarder / Receiver Pattern

- Kommunikation zwischen Client und Server abhängig von der Kommunikationstechnologie z.B. TCP / UDP Sockets
- Übermittlungsprotokoll ggf. verstreut
- Fehlerbehandlung/Retry-Code ggf. verstreut
- **Lösungsidee**: Forwarder/Receiver Pattern (POSA 1)
  - Insbesondere für Peer-To-Peer Architekturen



Interfaces für **`Forwarder`** und **`Receiver`**

- Versenden von Nachrichten (OneWay)

```
public interface Forwarder {
    public void send(Message message);
}
```

- ***Synchrones*** Empfangen von Nachrichten (OneWay)

```
public interface Receiver {
    public Message receive();
}
```

Achtung: Fehlerbehandlung noch offen / implizit

`Marshaller` Kapselung der Serialisierung

- Idee: Serialisieren der Nachrichten auslagern in eigene Klasse (= Strategy-Pattern)
- Vorteile:
  - Austauschbarkeit des Algorithmus zur Serialisierung (z.B. Java-Serialisierung, XML, IIOP)
  - Wiederverwendbarkeit der Mashaller-Implementierungen

```
public interface Marshaller {
   public byte[] marshall(Message m);
   public Message unmarshall(byte[]);
}
```

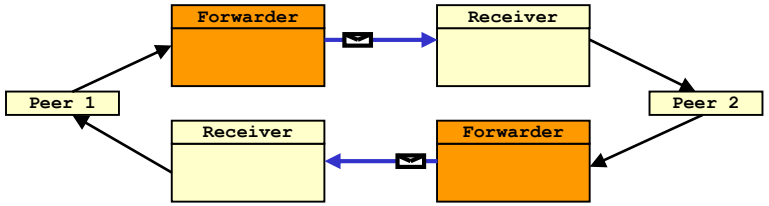
- Alternative: Neue InputFilterStream und OutputFilterStream-Klassen analog zu ObjectInputStream und ObjectOutputStream

Diskussion `Marshaller`

- **`Marshaller`** definiert Format, das über das Netz übertragen wird
- Idee: ***Format standardisieren*** und damit Plattformunabhängigkeit
- Abhängigkeiten von der Plattform, z.B.
  - Heterogene Hardware-Architekturen
    - Unterschiedliche Reihenfolge der Speicherung von Bytes (Little Endian / Big Endian)
    - Unterschiedliche Zeichencodierung: (ASCII auf PCs / EBCDIC auf IBM Mainframes / UNICODE)
  - Heterogene Programmiersprachen
    - Unterschiedliche Darstellung einfacher und komplexer Datentypen im Hauptspeicher.
- Beispiele für derartige Formate: XML, CDR (aus Corba), JSON…

ForwarderTCPSocket

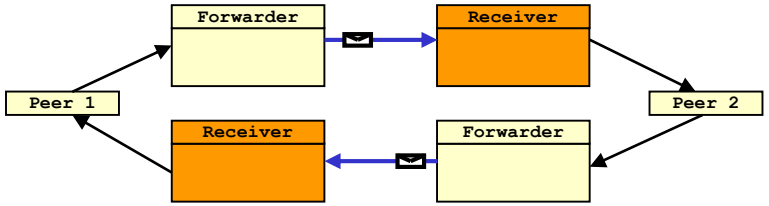
(Achtung: kein produktiver Code!)




```
public class ForwarderTCPSocket implements Forwarder {
    public void send(Message message) {
        deliver(message.getDestination(), marshal(message));
    }

    // Kapselt das versenden von „Bytewuersten“
    private void deliver(String destination, byte[] data) {
       Socket socket = null; OutputStream os = null;
       try { // Besser waere hier Pooling von Verbindungen
            socket = new Socket(
                InetAddress.getByName(destination), 10014);
            os = socket.getOutputStream();
            os.write(data);
            os.flush();
       }
       catch (IOException ex) { … }
       finally { … }
    }

    private byte[] marshal(Message message) { … }
}
```

ReceiverTCPSocket

(Achtung: kein produktiver Code!)




```
public class ReceiverTCPSocket implements Receiver {
    public ReceiverTCPSocket(String name) { this.name = name; }

    public Message receive(int timeout) {
        return unmarshal(deliver(timeout));
    }  // Asynchrone Version nur mit ObserverPattern

    private byte[] receiveBytes(int timeout) {
       ByteArrayOutputStream readBuffer = new ByteArrayOutputStream();
       InputStream isr = null;
       byte[] result = null; ServerSocket serverSocket; Socket clientSocket;

       try { // Auch hier besser Wiederverwendung der Sockets
          serverSocket = new ServerSocket(…); clientSocket = serverSocket.accept();

          isr = clientSocket.getInputStream();
          int fromStream = 0;
          while ((fromStream = isr.read()) != -1) {
             readBuffer.write(fromStream);
          }

          result = readBuffer.toByteArray();
       } catch(Exception x) {…} finally {…}

    public Message unmarshal (byte[] array) { … }
}
```

Diskussion Forwarder/Receiver

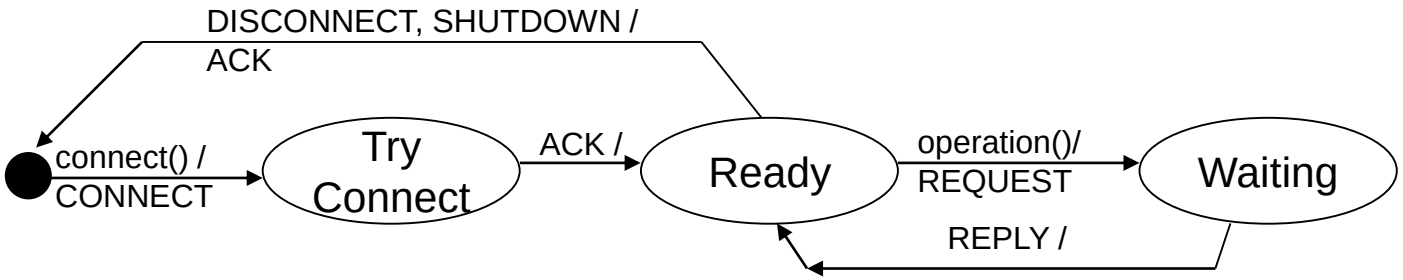
- Vorteile
  - Forwarder/Receiver kapselt Kommunikations API
    - Beispiel: Implementierung mit TCP Sockets und Java-Serialisierung
    - Ggf. auch Kapselung UART / SPI / I2C
  - Kommunikation an genau einer Stelle im Programm
    - Team muss sich nicht mit den Details der Socket-Programmierung auskennen
    - Optimierungen des Zugriffs sind zentral möglich
- Nachteile
  - Zusätzliche Komplexität
- Alternativer Name für Pattern: Client-Request-Handler/Server-Request-Handler

(Protokoll)-Automat

- Protokoll = Wann darf der Client dem Server welche Nachricht schicken? Wie Antwortet der Server? Was wird als Fehler angesehen?
- Idee: Kapsele ***Abwicklung des Protokolls***
  - Modelliere / Implementiere Protokoll über einen endlichen deterministischen (Mealy-)Automaten
    - Kommunikation hat einen Zustand (initialising, ready, waiting, shutdown)
    - Ankommende Nachrichten als Zeichen eines Eingabealphabetes sein (Eingabealphabet also z.B. Request / Response / Fault)
    - Ausgehende Nachrichten können Zeichen eines Ausgabealphabetes sein
    - Übergangsfunktionen implementieren das „eigentliche“ Protokoll
- Wichtig insbesondere für ***asynchrone*** Kommunikation

Implementierung der Übertragungsfunktion

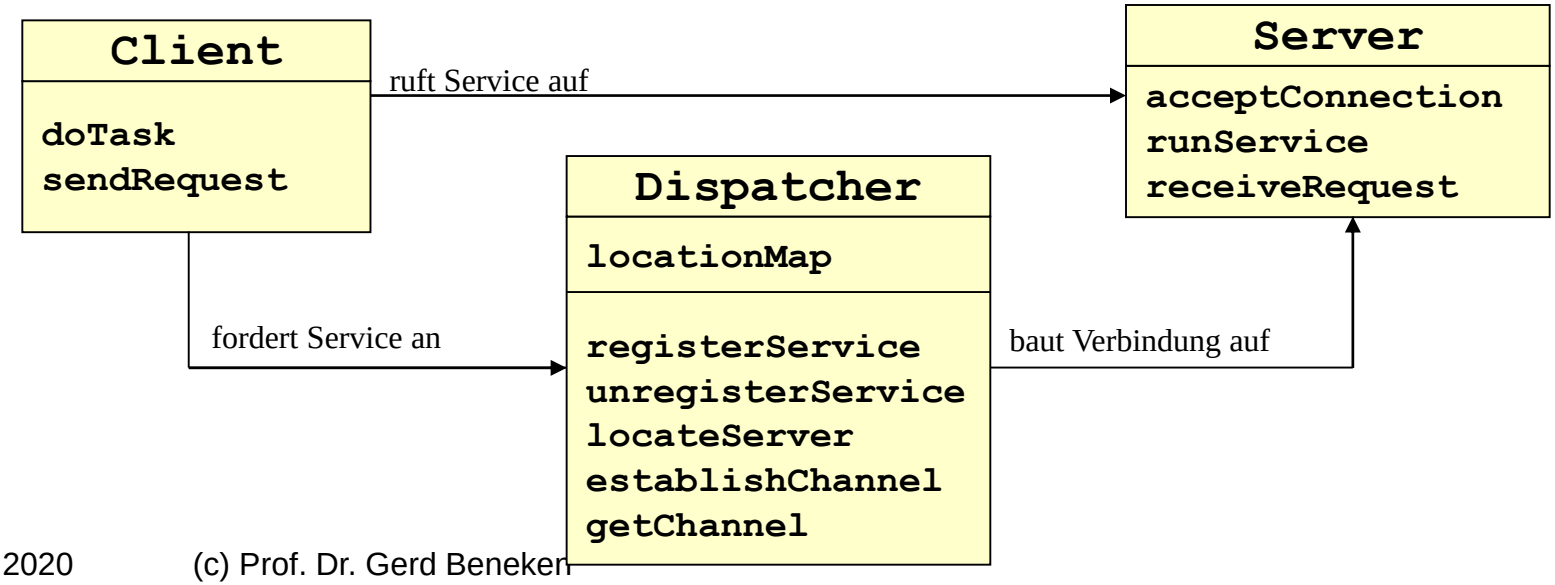
- Beispiel:



`Dispatcher` (= Namensdienst)

Jetzt populär wegen Microservice / Cloud Comp.

- Idee: Client soll nicht wissen, von wo der Service kommt
- Dispatcher = Namensdienst (Ortstransparenz)
  - Server registriert sich beim Dispatcher, Clients fragen nach Servern
  - Dispatcher baut bei Bedarf einen Kommunikationskanal zum Server auf
  - Client sendet dann über die Aufgebaute Kommunikation direkt an den Server



Dispatcher und Service

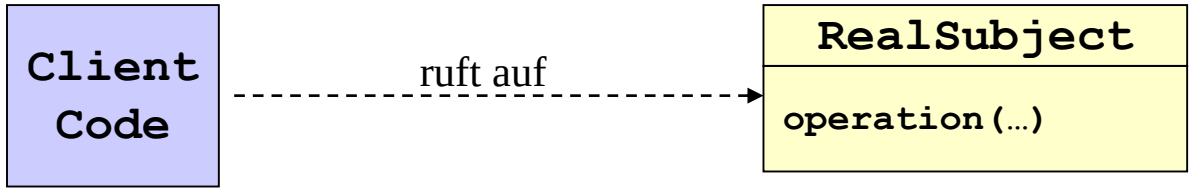
```
public class Dispatcher {
    private Map<String, Service> registry;
    public Dispatcher() { registry = new HashMap<String, Service>();}

    public void register(String serviceName, Service service) {
        registry.put(serviceName, service);
    }
    public Service locate(String serviceName) {
        return registry.get(serviceName);
    }

}

public abstract class Service {
    private String nameOfService; private String nameOfServer;
    public Service(String nameOfService, String nameOfServer)
    {
        this.nameOfServer = nameOfServer;
        this.nameOfService = nameOfService;
    }

    abstract public void service();  // Verbindungsaufbau etc.
}
```

Entwurfsmuster für Methoden / Prozeduren / Ressourcen im Netzwerk

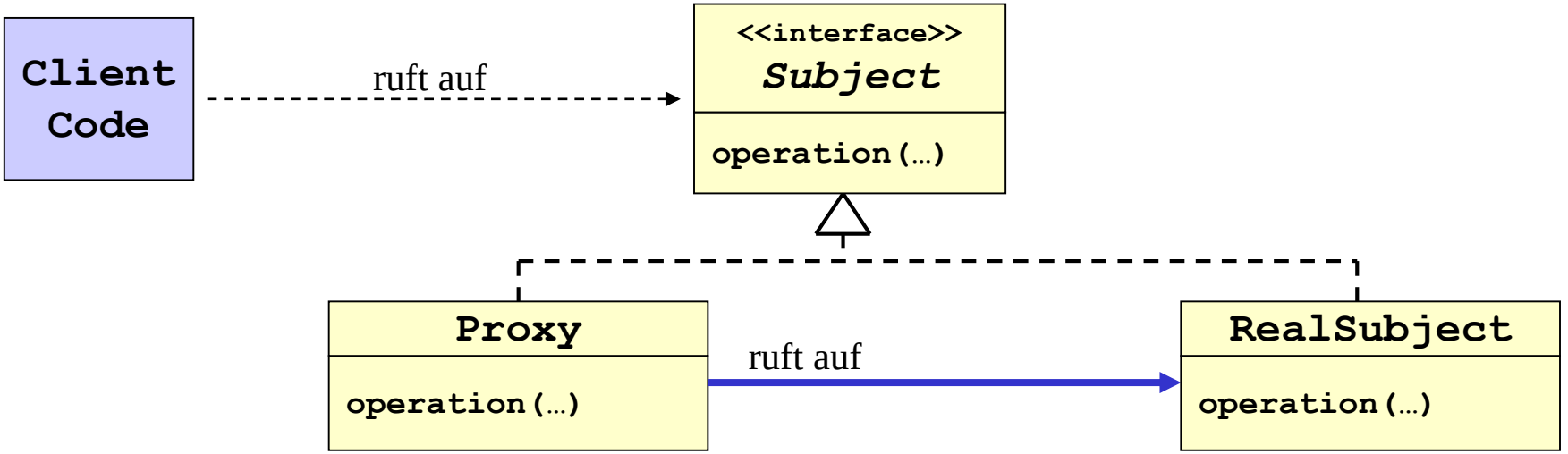
- **`Remote Proxy`** und Server-Skeleton
  - Lokales Objekt hat dasselbe Interface, wie entferntes Objekt
  - Remote Proxy und Server-Skeleton implementieren Kommunikation
- **`Interface Description`**
- **`Invoker`** und **`Requestor`**

Proxy-Pattern

(Remote)Proxy-Pattern

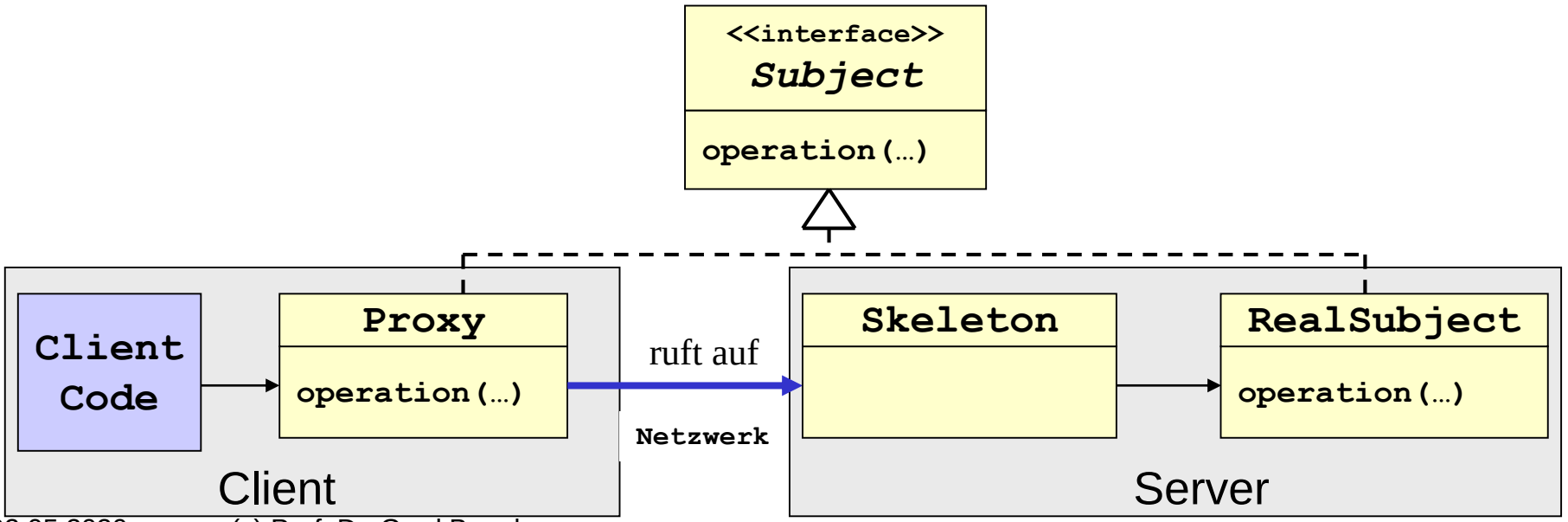


- Ziel: Ortstransparenz
- Client ruft auf lokalem Interface einen entfernten Dienst auf



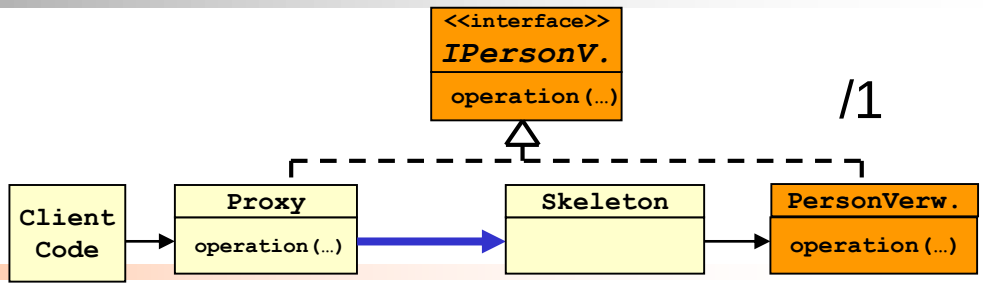
Proxy mit Server-Anteil

- Idee: Klasse am Server nicht modifizieren, sondern Proxy einen Server-Anteil spendieren
- Serveranteil (Server-Stub / Skeleton / Invoker)
  - kommuniziert mit Proxy am Client
  - Ruft ***lokal*** Server-Objekt auf



Selbstgebautes RMI „Der Dienst“

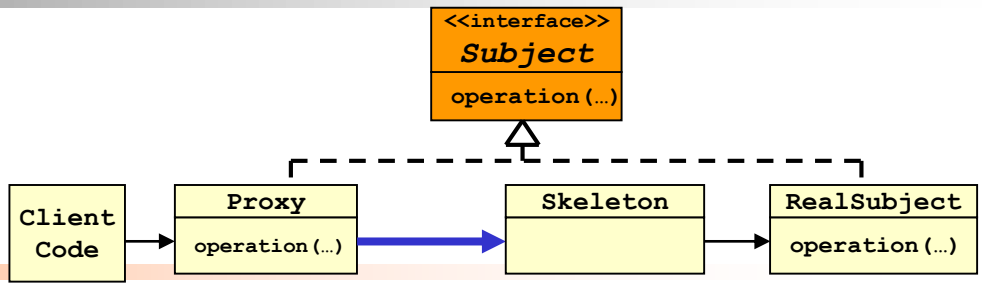
/1




```
public interface IPersonVerwaltung {
   public Person createPerson(String vorname, String nachname);
   public List<Person> findByNachname(String nachname);
   public Person findById(Long id);
}
```

```
public class PersonenVerwaltung implements IPersonVerwaltung {
     private Map<Long,Person> datenbank = new HashMap<>();
     private static long counter = 0;

     @Override
     public Person createPerson(String vorname, String nachname){
       counter++;
       Person p = new Person(counter, vorname,nachname);
       datenbank.put(p.getId(), p);
       return p;
     }
// …
```

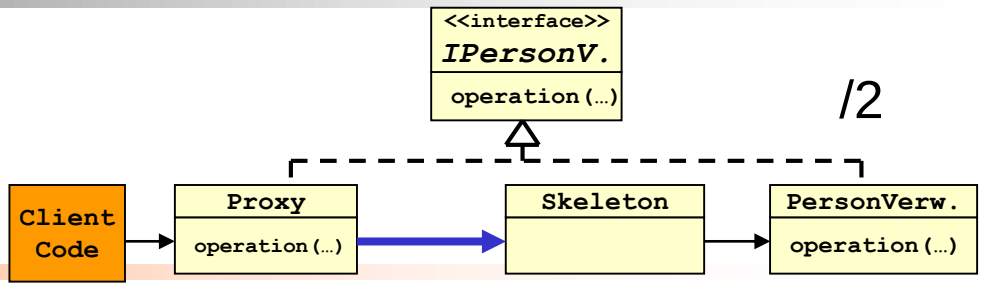
Interface Description Pattern



- Beschreibt die ***Schnittstelle*** eines ***entfernten Objektes / Dienstes / Ressource***
- ***= Vertrag zwischen (Proxy am) Client und Server-Objekt / Dienst / Ressource***
- kann zur automatischen Generierung des Proxies und der Server-Anteile (z.B. Invoker siehe unten) verwendet werden
  - Generierung von Code vor der Compile-Zeit oder
  - Generische Implementierung (z.B. über Reflection)
  - = Typisch für die meisten Remoting-Technologien (WebServices, RMI, WCF, COM, RPC …)
- Häufig eigene Sprachen zur Schnittstellenspezifikation
  - IDL = Interface Definition Language (CORBA)
  - COM-IDL (Microsoft COM / COM+)
  - WSDL = Web Service Definition Language + XSD
  - Auch Programmiersprachen Interfaces (C# oder Java Interfaces)
- Kann genutzt werden, um Server und Client in verschiedenen Sprachen zu implementieren.

Selbstgebautes RMI „Der Client“

/2



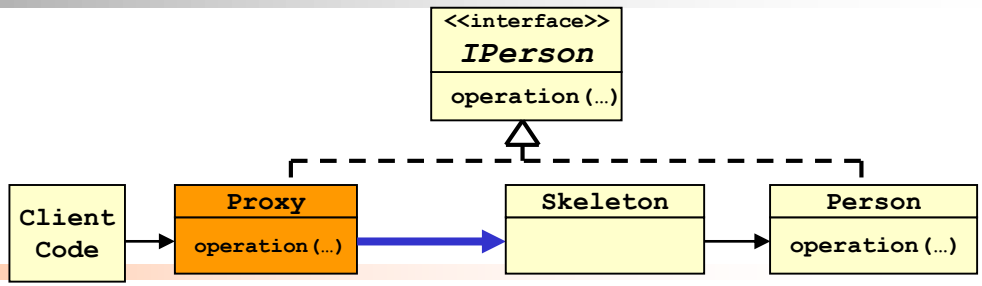
- Idee: Client ruft Service-Interface **`IPersonVerwaltung`** auf
- Programmierung am Client, als ob **`PersonVerwaltung`**-Objekt lokal verfügbar wäre (Ortstransparenz)
- Tatsächlich erfolgt aber ein Netzwerkaufruf
- (Achtung: IPerson ist eine sehr schlechte Remote-Schnittstelle)

Proxy!

```
public class Client {
  public static void main(String[] args) {
    IPersonVerwaltung verwaltung = new PersonenVerwaltungProxy();
    Person p1 = verwaltung.createPerson("Flori", "Flunk");

    List<Person> familieBarsch =
       verwaltung.findByNachname("Barsch");
    familieBarsch.forEach(System.out::println);
  }
}
```

Selbstgebautes RMI

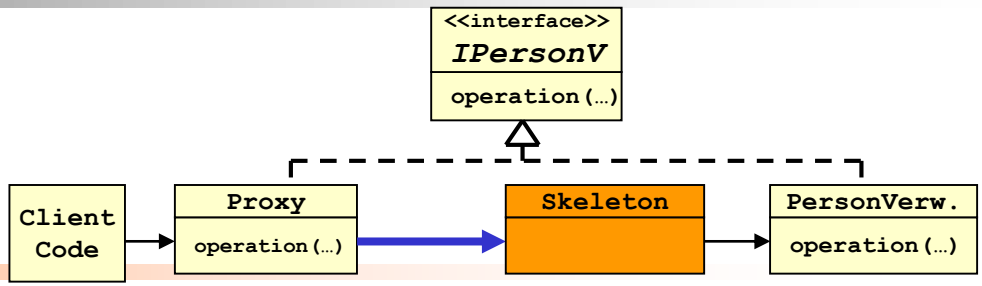
Proxy am Client




```
public class PersonVerwaltungProxy implements IPersonVerwaltung {
    private Forwarder forwarder; private Receiver receiver;

    public PersonVerwaltungProxy(Forwarder fwd, Receiver rec) {
         this.forwarder = fwd; this.receiver = rec;
    }

    public Person createPerson(String vorname, String nachname){
        Request request = new Request("createPerson",
                                  new Obect[]{vorname, nachname});
        forwarder.send(request);
        Response response = receiver.receive();
        Person result = (Person) response.getResult();
        return result;
    }
    // …
}
```

Selbstgebautes RMI

Proxyanteil am Server




```
public class PersonVerwaltungSkeleton {
   private Forwarder forwarder; private Receiver receiver;
   private PersonVerwaltung personverwaltung;

   public PersonVerwaltungSkeleton(PersonVerwaltung v,
          Forwarder f, Receiver r) {
      this.forwarder = f; this.receiver = r;
      this.personverwaltung = v;
   }

   public void processRequest() { // Fehlerbehandlung entfernt
         Request methodCall = (Request) receiver.receive();
         Message response = null;

         if ("createPerson".equals(methodCall.getNameOfProcedure())) {
               response = new Response(personvrewaltung.createPerson(…));
            } else {  // Fehlerbehandlung weitgehend entfernt
               response = new Fault(„Unbekannte Methode");
            }
            forwarder.sendMessage(response);
} } }
```

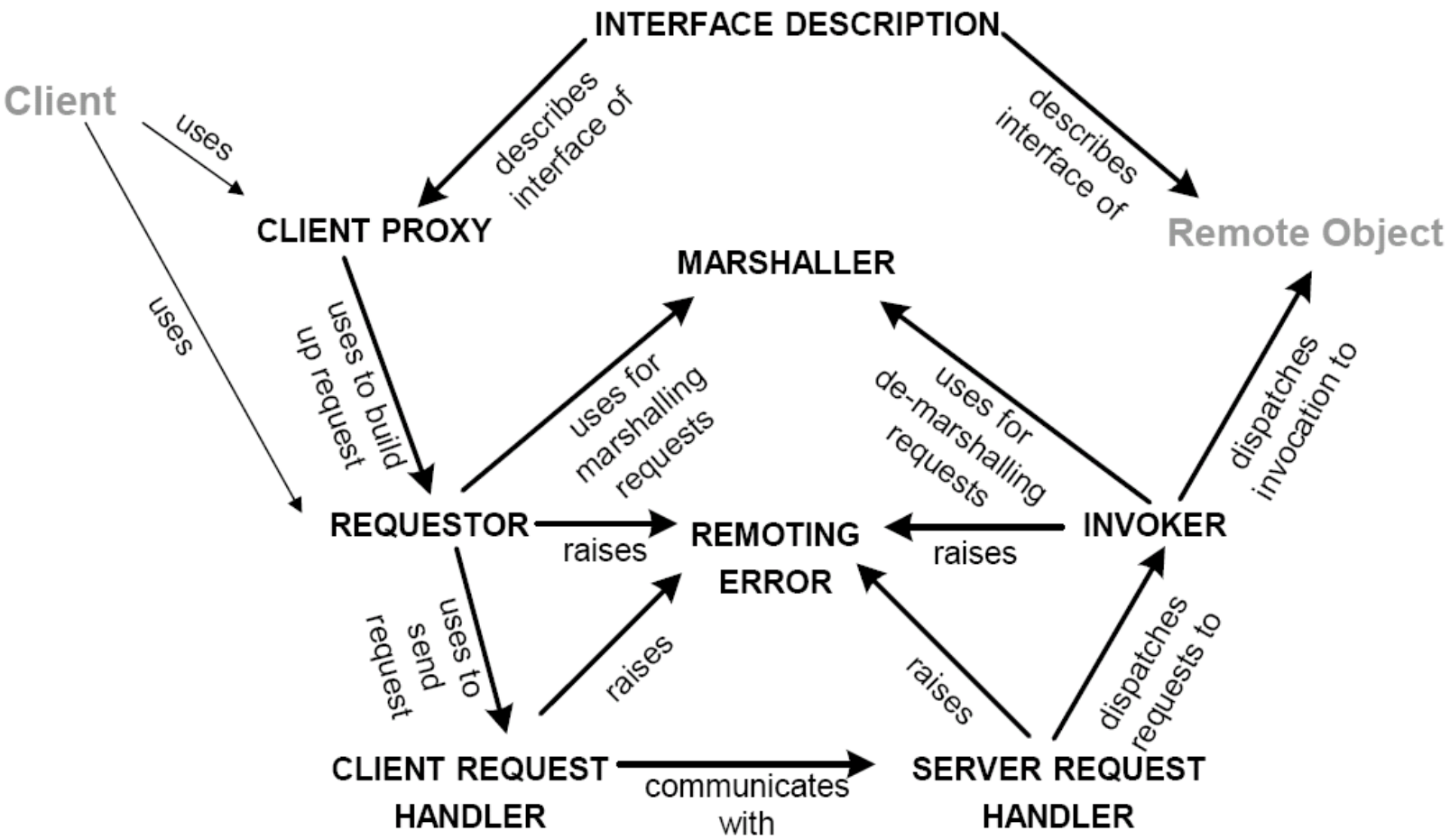
Diskussion des Beispiels

- Beispiel zeigte entfernten Methodenaufruf
- Proxy und Server-Anteil sind manuell implementiert
  - Eigenes Protokoll Request, Response, Fault
  - Eigene Auswertung der Rückgabewerte
- Technologien wie: RPC, RMI, Web-Services, CORBA, .NET-Remoting, COM+, … funktionieren ähnlich
  - Interface spezifiziert einen „Dienst“ am Server
  - Remote Proxy und Server-Anteil werden entsprechend generiert oder sind generisch implementiert

Offene Fragestellungen des Beispiels

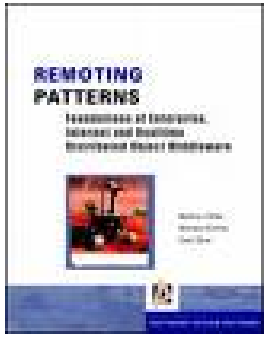
- Verteilte Objekte: Server kann …
  - mehr als einen Service anbieten, Identifikation des Services?
  - verschiedene Personen-Objekte verwalten
    (Thema: ***Identität verteilter Objekte***), hier nur einfache Map
- Fehlerbehandlung
  - Fehler können während der Übertragung auftreten
  - Fehler nicht im Interface sichtbar
  - Code des Beispiels behandelt kaum Fehler (nur Fault)
- Code nicht optimal
  - Adresse des Servers fest verdrahtet (Dispatcher)

Remoting Pattern-Language (Völter et al.: Remoting Patterns)



Literatur zum Nach-/Weiterlesen

- Völter, Kircher, Zdun: ***Remoting Patterns***: Foundations of Enterprise and Realtime Distributed Object Middleware, Wiley, 2003



- Buschmann, Meunier, Rohnert, Sommerlad, Stal: ***Pattern Oriented Software Architecture (POSA)***, Wiley, 1996 (die „Siemens“-Patterns)
- POSA Band 2 bis 5
  (Details und andere Pattern Languages)