7363 - Web-basierte Anwendungen
4750 – Web-Engineering

Eine Vertiefungsveranstaltung

RESTful Web Services

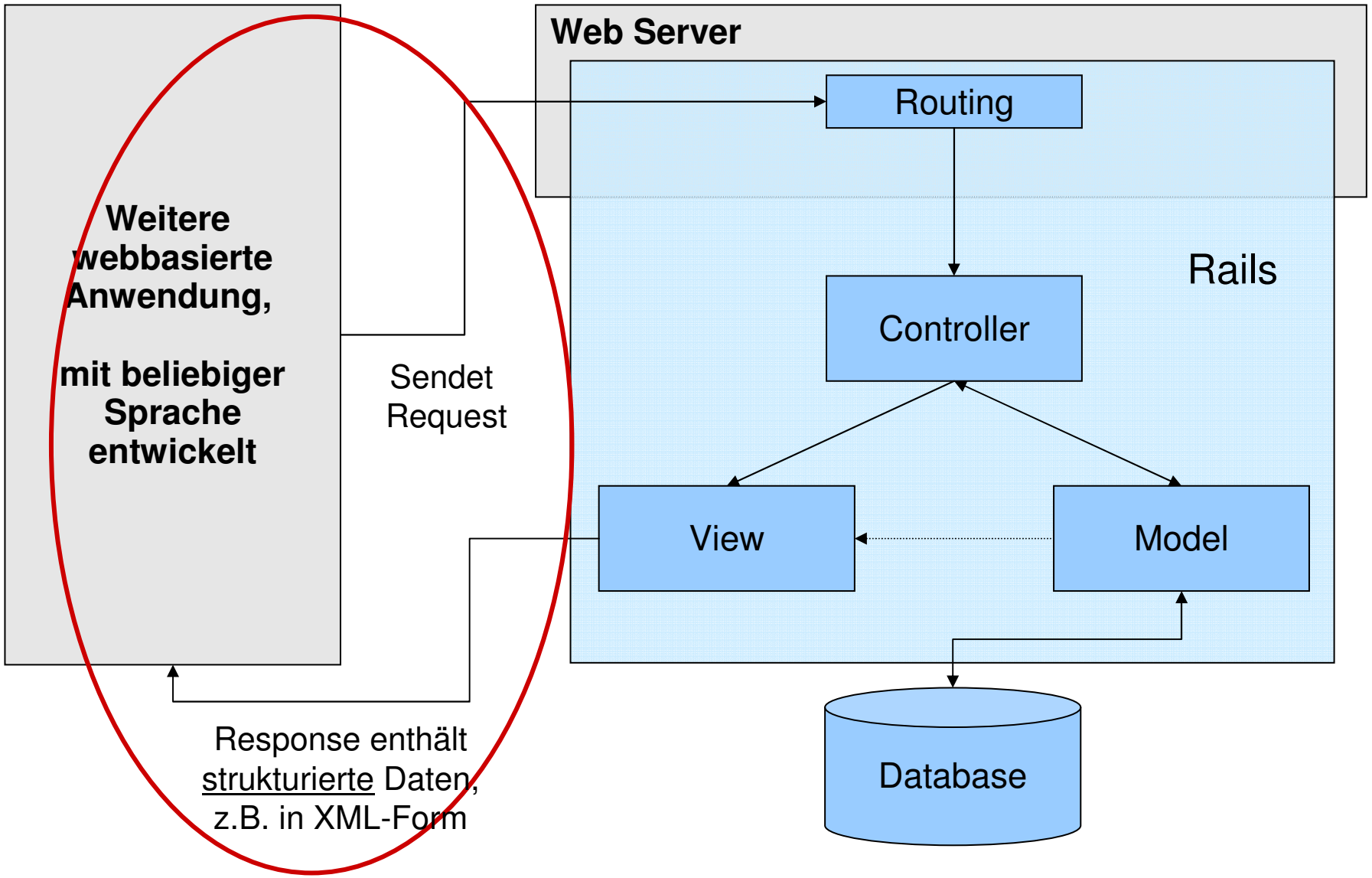
Oder:
Verteilte Anwendungen auf der Basis vom WWW

REST

__Re__presentational __S__tate __T__ransfer

Ein Architekturstil zur Vereinfachung von Web Services

REST



REST vs. Web Services

- Derzeitiger „*mainstream*“:
  - Anwendungskopplung mit Web Services auf Basis von SOAP und WSDL
- SOAP
  - Ein Protokollrahmen (ehemals ein reines Protokoll)
    - RPC-Modus: „Verpacken“ von Funktionsaufrufen und allen Argumenten
    - Dokumentenmodus: „Eintüten“ eines XML-Dokuments in eine Verwaltungsstruktur (Envelope, Header)
    - Verschiedene „Bindings“: Meist HTTP, aber auch SMTP, FTP etc. möglich
- WSDL (*Web Service Description Language*)
  - Wie der Name schon sagt: Eine Sprache zur exakten Definition von Web Services (Methoden, Parameter, Bindings, Web-Adressen, Datentypen…)
  - Wird meist als Grundlage für Code-Generatoren genutzt

Entwicklungstrend

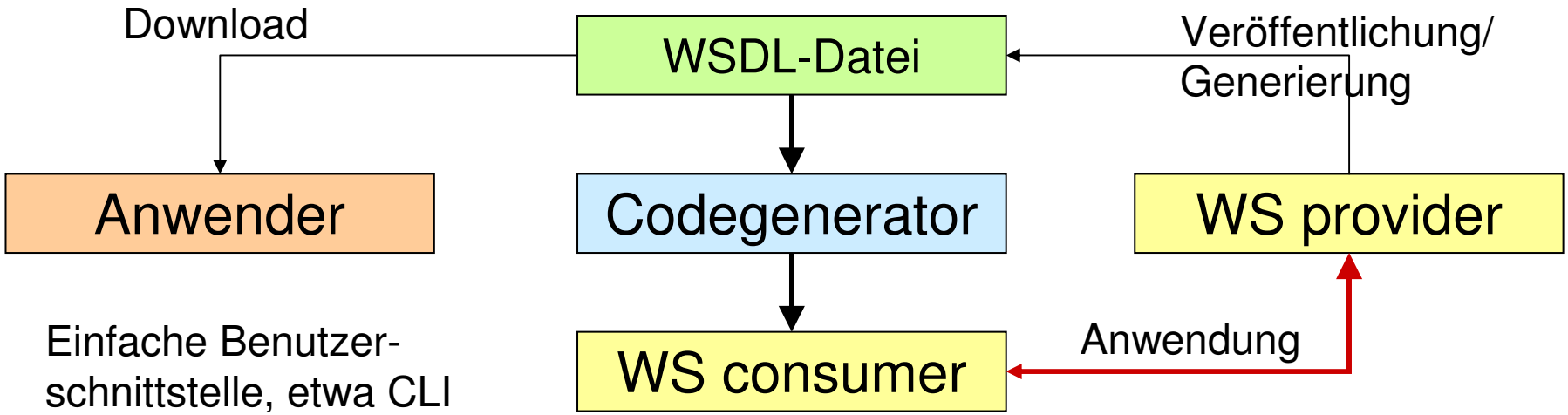
1. Enge Kopplung zwischen Anwendungen
   - Tiefe Integration; Ideal: Die Möglichkeiten integrierter Anwendungen
   - Ausdehnung des Konzepts "Prozedur-Aufruf" auf Anwendungs- und Rechnernetze
   - Techniken: CORBA, COM/DCOM, Suns RPC

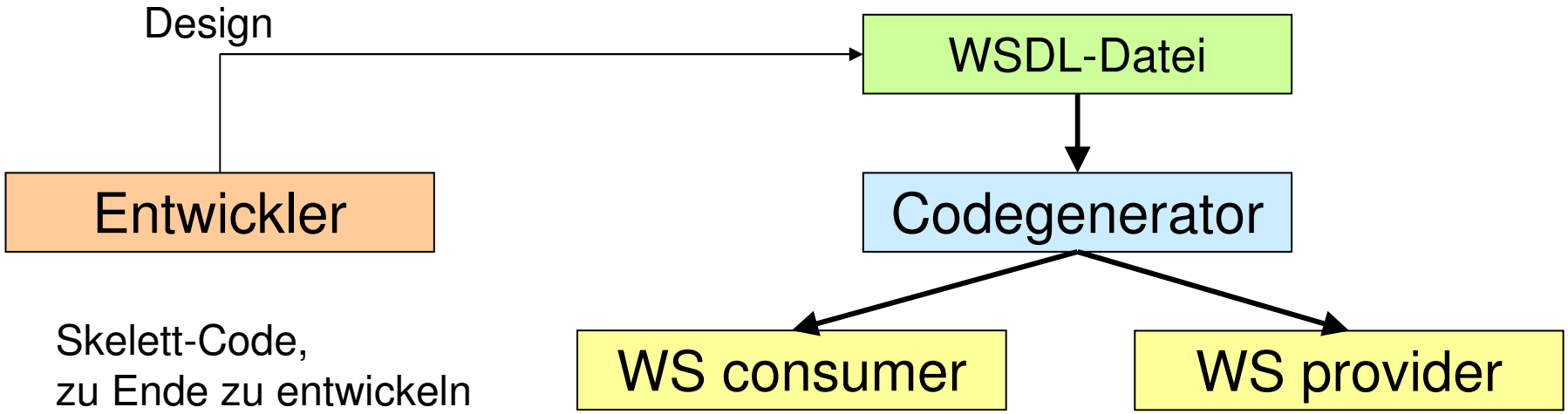
2. Übergang
   - Beibehaltung des Konzepts "Prozedur-Aufruf" bei lockerer Kopplung
   - Ausgliederung des *Messaging* an separate Schicht (etwa HTTP)
   - Verpackung ("*marshalling*") der Aufrufs- und Rückgabeparameter mit Standardmethoden (XML)
   - Techniken: XML-RPC

3. Lockere Kopplung zwischen Anwendungen
   - Robuste, fehlertolerante Anwendungsnetze
   - Das Konzept "Dokumentenaustausch" herrscht vor
   - "Aufruf" und Verarbeitung bzw. "Antwort" erfolgen meist asynchron
   - Techniken: SOAP, REST

WSDL

- WSDL: Anwendungsszenarien





REST vs. Web Services

- Kritik an SOAP & Co.
  - Sehr (zu?) komplex
  - Gewaltiger Overhead
  - Funktioniert nach völlig anderen Kriterien als das WWW
  - Skalierbarkeit?

- Die Position von REST
  - Endgültig Abschied vom RPC-Bild, Beschränkung auf HTTP
  - Anwendungen im WWW verwalten (abstrakte) Ressourcen
  - Ressourcen werden mittels URIs identifiziert
  - Ressourcen besitzen verschiedene Darstellungen, z.B. als HTML-Seite, XML-Repräsentation, RDF-Dokument, PDF, …
  - Anwendungen werden gekoppelt über
    - den Austausch solcher Darstellungen und
    - Kommandos, was mit (per URI identifizierten) Ressourcen geschehen soll

REST

- Ursprung von REST

  [1] Dissertation: Roy T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, UC Irvine, 2000.
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

- Einführende Online-Literatur

  [2] Thomas Bayer, REST Web Services – eine Einführung, OIO, 11-2002
  - http://www.oio.de/public/xml/rest-webservices.htm

    (Vorsicht – der Artikel enthält einige Fehler!)

  [3] Ralf Wirdemann, Thomas Baustert, RESTful RAILS, b-simple.de, Hamburg, 01-2007
  - http://www.b-simple.de/download/restful_rails_de.pdf

  [4] Rails-Beispiele: http://api.rubyonrails.org/
  - **Files: vendor/rails/activeresource/README**
  - **Classes: ActiveResource:Base**

REST

- Die Sicht von REST: Lauter Ressourcen!
  - Ressourcen sind Objekte, auf die sich per URL verweisen lässt, insbesondere:
    - XML-Dokumente
    - HTML-Dateien
    - RDF-Dateien, …
- REST – oder: Die Wiederentdeckung von HTTP
  - Von allen HTTP-Methoden verwenden Browser als auch Web Service-Komponenten praktisch nur GET und POST
  - REST greift das ursprünglich von HTTP verfolgte Konzept der verteilten Autorenschaft wieder auf.
  - REST verwendet dazu auch die HTTP-Methoden PUT, DELETE sowie HEAD und OPTIONS
- HTTP-Kommandos aus Sicht von REST:
  - Kommando = Satz, Methode = Verb, URI = Nomen

REST

- HTTP und CRUD
  - Der Lebenszyklus eines Datenbankeintrags / einer Ressource, (meist aus objekt-relationaler Sicht) wird durch folgende Methoden realisiert:
    - **C** reate
    - **R** ead
    - **U** pdate
    - **D** elete
  - Direkte Entsprechung bei HTTP
    - C reate - POST <url>
    - R ead - GET <url>
    - U pdate - PUT <url>
    - D elete - DELETE <url>

- Weitere HTTP-Methoden für REST?
  - HEAD
    - Fordert Metadaten zur angegebenen Ressource an
  - OPTIONS
    - Ermittlung der für eine Ressource verfügbaren Methoden

(leider noch keine Beispiele gefunden)

REST

- Konsequenzen für webbasierte Anwendungen
  - Entfernung der Controller-Methoden aus den URLs
  - Enge Kopplung zwischen Controller-Methoden und HTTP-Methoden
  - Controller und Ressource verschmelzen zu einer Einheit
  - URLs werden zu reinen, einheitlich strukturierten *resource identifiers*

- URL-Analyse, gewöhnliche URLs
  - Szenario: Anlegen einer Bestellung

URLs enthalten ID, Controller und Methode!

GET *my_base_url*/cgi-bin/bestellen.pl?id=54321&action="create"

POST *my_base_url*/cgi-bin/bestellen.pl?id=54321&action="create"

HTTP-Methode

Zuständiger Controller

ID der Bestellung

Aktion / Methode

  - GET oder POST? Beides könnte funktionieren!
    GET wäre aber schlechter Stil, da „Lesen“ keine Seiteneffekte haben sollte.

REST

- Beispiele
  - Abruf einer HTML-Seite
    - Gewöhnlich: GET *my_base_url*/index.html
    - REST-Stil: GET *my_base_url*/some_pages/
  - Anlegen einer neuen Bestellung
    - Schlecht: GET *my_base_url*/cgi-bin/bestellen.pl?id=54321&action="create"
    - Besser: POST *my_base_url*/cgi-bin/bestellen.pl?action="create"
    - REST-Stil: POST *my_base_url*/bestellungen
  - Löschen einer vorhandenen Bestellung
    - Gewöhnlich: POST *my_base_url*/cgi-bin/bestellen.pl?id=54321&action="delete"
    - REST-Stil: DELETE *my_base_url*/bestellungen/54321

- HTTP-Analyse, REST-Stil

URLs identifizieren nur Ressourcen

POST *my_base_url*/bestellungen

HTTP-Methode → Controllermethode

Controller implizit in *resource* URL

ID der Bestellung ist der gesamte URL laut IP-Verbindung und http-header „Location“!

REST

- Beispiel
  - Quelle: http://rest.blueoxen.net/cgi-bin/wiki.pl?ResourcesAsObjects

```
C->S: GET http://example.com/lightbulb/on HTTP/1.1
C<-S: HTTP/1.1 200 OK
C<-S: Content-Type: text/plain
C<-S:
C<-S: false

C->S: PUT http://example.com/lightbulb/on HTTP/1.1
C->S: Content-Type: text/plain
C->S:
C->S: true
C<-S: HTTP/1.1 200 OK
```

REST

- Beispiel
  - (Im Sinn von KNX modifiziert) Man erkennt: Selbst Textmodus genügt

```
C->S: GET http://example.com/lights/ga_1_0_0 HTTP/1.1
C<-S: HTTP/1.1 200 OK
C<-S: Content-Type: text/plain
C<-S:
C<-S: off
```

Nein!

Statusabfrage: Ist das Licht mit Gruppenadresse 1/0/0 an?

```
C->S: PUT http://example.com/lightbulb/ga_1_0_0 HTTP/1.1
C->S: Content-Type: text/plain
C->S:
C->S: on  # oder: toggle
C<-S: HTTP/1.1 200 OK
```

Gewünschter Zustand

Statusänderung: Schalte das Licht mit Gruppenadresse 1/0/0 !

REST: Zusammengesetzte Objekte

```
GET /warenkorb/5873 HTTP/1.0
```

Warenkorb anfordern

Quelle: [2]

```
HTTP/1.1 200 OK
Content-Type: text/xml
Host: …
```

Ergebnis (Header ergänzt)

```
<?xml version="1.0"?>
<warenkorb xmlns:xlink="http://www.w3.org/1999/xlink">
  <kunde xlink:href="http://shop.oio.de/kunde/5873">5873</kunde>
  <position nr="1" menge="5">
    <artikel xlink:href="http://shop.oio.de/artikel/4501" nr="4501">
      <beschreibung>Dauerlutscher</beschreibung>
    </artikel>
  </position>
  <position nr="2" menge="2">
    <artikel xlink:href="http://shop.oio.de/artikel/5860" nr="5860">
      <beschreibung>Earl Grey Tea</beschreibung>
    </artikel>
  </position>
</warenkorb>
```

REST

- Zusammengesetzte Objekte
  - Kunden, Warenkörbe und Artikel werden als separate Ressourcen behandelt und geführt – d.h. mit eigenen URLs
  - Referenzierung von Objekten geschieht mittels deren URLs und dem Standard XLink (vgl. LV „XML-Technologie“)
  - Es besteht Konsistenz mit WWW-Gewohnheiten
    - Das XML-Dokument ließe sich mit CSS kombiniert direkt im Browser anzeigen
    - Die XLink-Elemente würde ein Browser beim Anklicken verfolgen (→ *drill down*)!

- Kommentare zum Beispiel
  - Warenkorb-ID und Kunden-ID sind identisch, ihre URLs nicht
  - Der Dokumententyp stellt einen Kompromiss zwischen Lesbarkeit für Menschen und maschineller Verarbeitbarkeit dar.
    - Einige Angaben sind redundant, z.B. Artikel- und Kundennummer
    - Das Element „Beschreibung“ ist für maschinelle Verarbeitung unnötig

  Anregung: Analoges Vorgehen bei Veranstaltung, Wettkampf, Teilnehmer?

- Aktuelle Diskussionen
  - Ruby on Rails (RoR) 2.x:
    - ActiveService-Modul (XML-RPC, SOAP) wird ausgegliedert
    - Nachfolge: REST-Modul **ActiveResource**
    - Grund: Einfacher, besser mit RoR und Web-Prinzipien verträglich
  - Kritik an SOAP
    - SOAP hat sich von einem Protokoll zu einem Protokoll-Framework entwickelt.
    - SOAP versucht inzwischen derart viele Dinge zu ermöglichen, dass neue Verwirrung entsteht. Beispiel: RPC- versus Dokumentenmodus
    - SOAP-Typisierung ist inzwischen von XML Schema ersetzt worden. Dies war aber ein zentrales Anliegen, SOAP zu entwickeln.
    - SOAP funktioniert nicht so wie das WWW im allgemeinen (kein Ressourcen-Modell), REST sehr wohl.

- Aktuelle Diskussionen
  - Kritik & Fragen an REST
    - Sicherheitsaspekte, die über HTTPS hinausgehen, werden nicht berücksichtigt
    - Was wird aus WSDL und aus der darauf basierenden Code-Generierung?
    - Wie beschreibt man die auszutauschenden Ressourcen *einheitlich*? Wie beschreibt man insb. Nicht-XML-Ressourcen?
    - SOA auf Basis von SOAP & WSDL ist inzwischen in vielen Unternehmen eine akzeptierte Technologie. Sie erfährt umfangreichen Support u.a. von Microsoft, IBM und Sun. REST ist bei weitem nicht so weit verbreitet und unterstützt. (Wie schnell) wird sich das ändern?

- Rails-Demo: Ein REST-Client für Book-Objekte
  - TCPmon starten (Einstellen: auf Port 3001 lauschen, an 3000 weiterleiten)

```
java -cp axis.jar org.apache.axis.utils.tcpmon
```

  - Im Lib-Projekt Rails-Konsole starten, folgenden Code eingeben:

```
class Book < ActiveResource::Base
  self.site = "http://localhost:3001/"
end
# Analogie beachten: class Book < ActiveRecord::Base

# Client-Anwendung
bk = Book.find 1     # XML-Datenaustausch sowie HTTP-Kommandos
bk = Book.new        # beachten
bk.title = "Test-Titel" # usw., alle Mussfelder belegen!
bk.save              # TCPmon-Ausgabe beachten. Fehlerfall?
Book.find :all       # Auch Arrays von Objekten erhältlich
```

REST

- Rails-Demo: Fazit
  - Zugriff auf Objekte einer entfernten Anwendung erfolgt weitgehend transparent und fast wie bei lokalen Objekten über ActiveRecord
  - Die Darstellung beruht auf einem hinterlegten Mapping zwischen einer automatisch generierten bzw. ausgewerteten XML-Struktur und dynamisch verfügbaren Attributen bzw. enthaltenen Objekten
  - Das lokal erzeugte Objekt muss zum auf dem entfernten System hinterlegten Modell / Datenbankschema passen, ansonsten werden Schritte wie „save“ scheitern
  - Auch Fehlermeldungen werden vom Protokoll berücksichtigt
  - Keine Formatbeschreibungen wie WSDL, keine strenge Typisierung – ganz in der Skriptsprachen-Tradition

- Insgesamt:
  - Eine derart einfache und bequeme Art der Anwendungskopplung, dass man SOAP nicht vermisst