

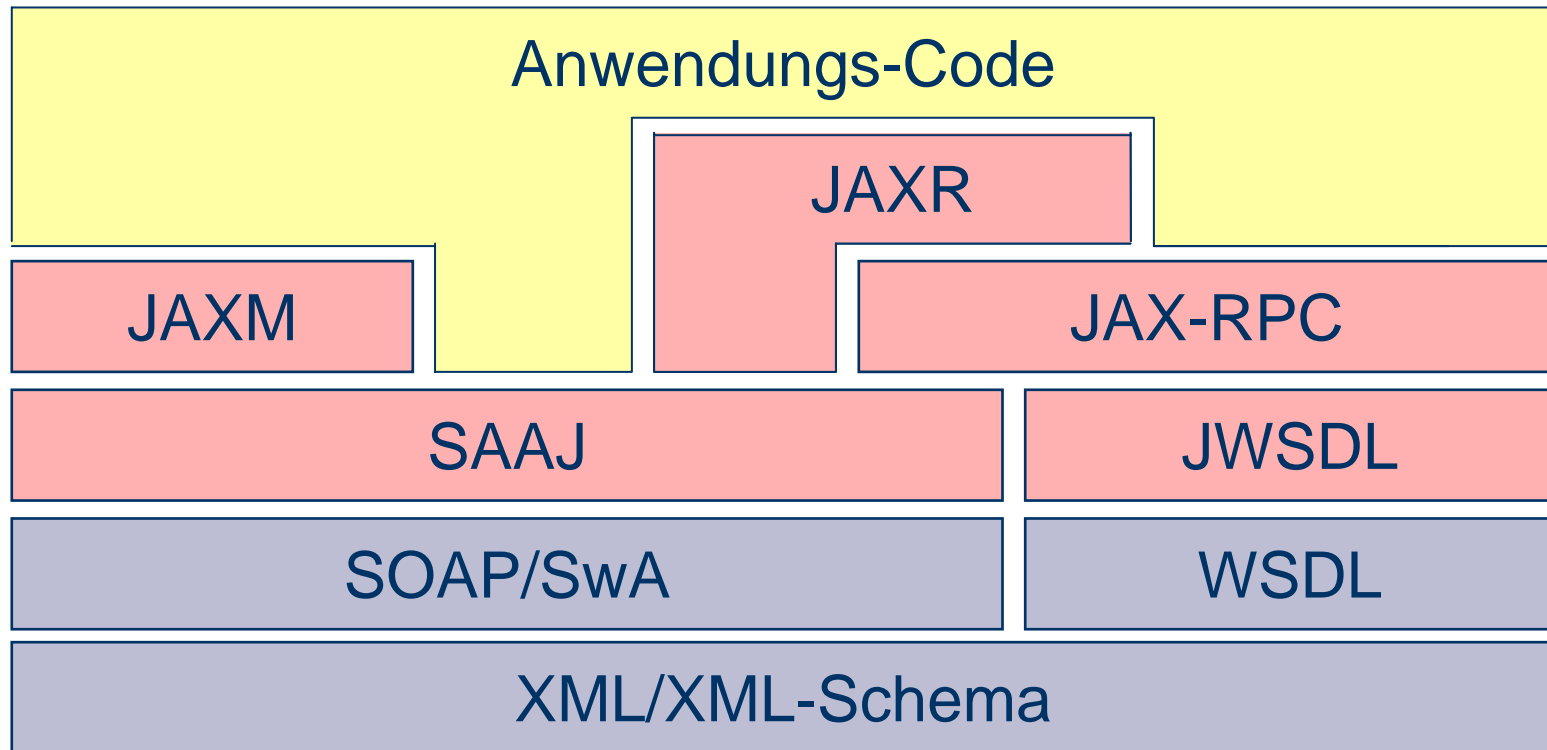
Web-Services

Implementierung mit Java

© J. Heinzlreiter

WS 2004/05

Java-APIs für Web-Services (1)



Java-APIs für Web-Services (2)

- **JWSDL: *Java APIs for WSDL***
 - Lesen, Generieren und Manipulieren von WSDL-Dokumenten.
- **SAAJ: *SOAP with Attachments API for Java***
 - Lesen, Generieren und Manipulieren von SOAP-Nachrichten,
 - Versenden/Empfangen von SOAP-Nachrichten (synchron).
- **JAXM: *Java API für XML Messaging***
 - Asynchroner Nachrichtenaustausch,
 - sichere Nachrichtenübertragung,
- **JAX-RPC: *Java-API for XML-based RPC***
 - API zur Implementierung von Web-Services und zur Erstellung von RPC-basierten Clients für Web-Services.
- **JAXR: *Java-API for XML-based Registries***
 - Interface für den Zugriff auf UDDI-Verzeichnisse.

JAX-RPC

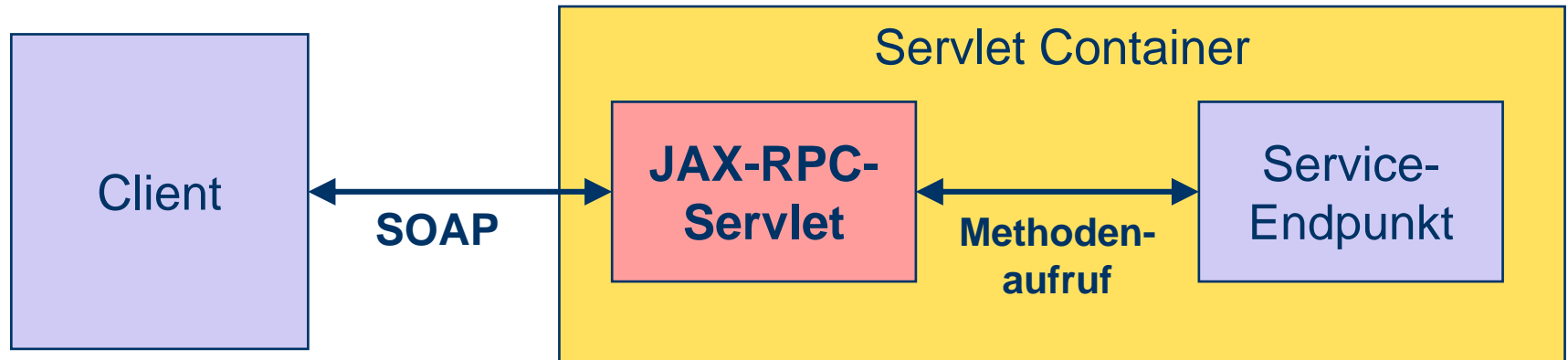
- API zur Implementierung von Web-Services und zur Erstellung von Clients für Web-Services.
 - RPC-basiertes Programmiermodell (ähnlich zu RMI und CORBA).
 - Protokollschicht wird vollständig gekapselt.



- Arten von Service-Endpunkten:
 - JAX-RPC Service-Endpoint (Servlet)
 - JAX-RPC EJB-Endpoint (Stateless Session Bean)

JAX-RPC Service-Endpunkt (JSE)

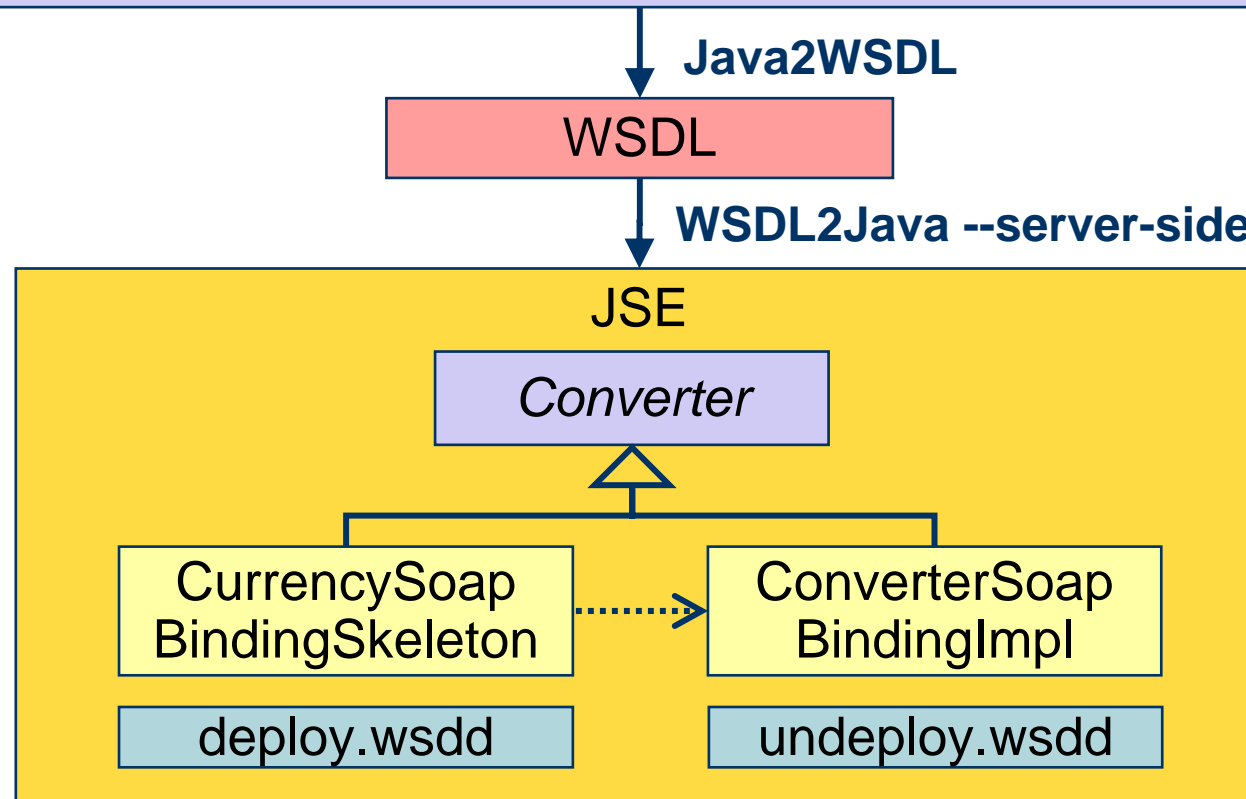
- Architektur



- JSEs werden in einem Servlet Container installiert.
- JSEs haben Zugriff auf dieselben Ressourcen und Kontextinformationen wie Servlets.
- Implementierung des JAX-RPC-Servlets ist herstellerabhängig:
 - ein Servlet für jeden JSE,
 - ein Servlet, das Requests an JSEs weiterleitet.

Implementierung von JSEs (1)

```
public interface Converter extends java.rmi.Remote {  
    double rateOfExchange(String fromCurr, String toCurr)  
        throws java.rmi.RemoteException;  
}
```



Implementierung von JSEs (2)

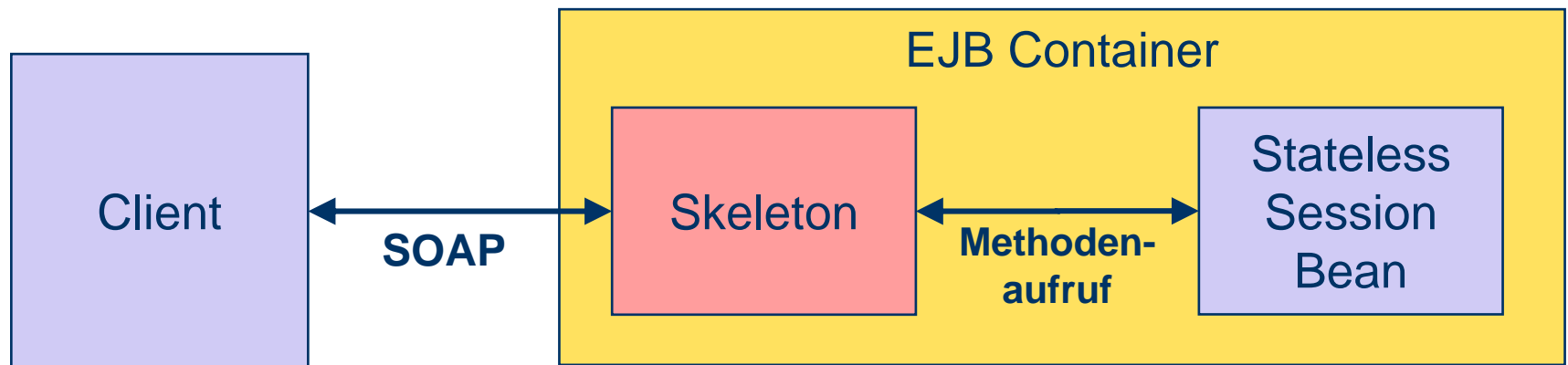
```
public class ConverterSoapBindingImpl implements Converter {
    public double rateOfExchange(String fromCurr, String toCurr)
        throws java.rmi.RemoteException, SOAPFaultException {
        return euroRate(toCurr) / euroRate(fromCurr);
    }
}
```

Deployment-Deskriptor (für Axis):

```
<deployment
  <service name="Converter" provider="java:RPC" style="rpc"
    use="encoded">
    <parameter name="wsdlTargetNamespace"
      value="http://webservices.swt5"/>
    <parameter name="className"
      value="swt5.webservices.ConverterSoapBindingSkeleton"/>
    <parameter name="allowedMethods" value="*" />
    ...
  </service>
</deployment>
```

JAX-RPC EJB-Endpunkt

- Stateless Session Beans können als Web-Service-Endpunkte exportiert werden.



- Schnittstelle wird durch ein Interface, das *java.rmi.Remote* erweitert definiert.
- Implementierung des Web-Service erfolgt im Session-Bean.
- Remote- bzw. Local-Interface kann, muss aber nicht definiert werden.
- Deployment-Descriptor muss erweitert werden.

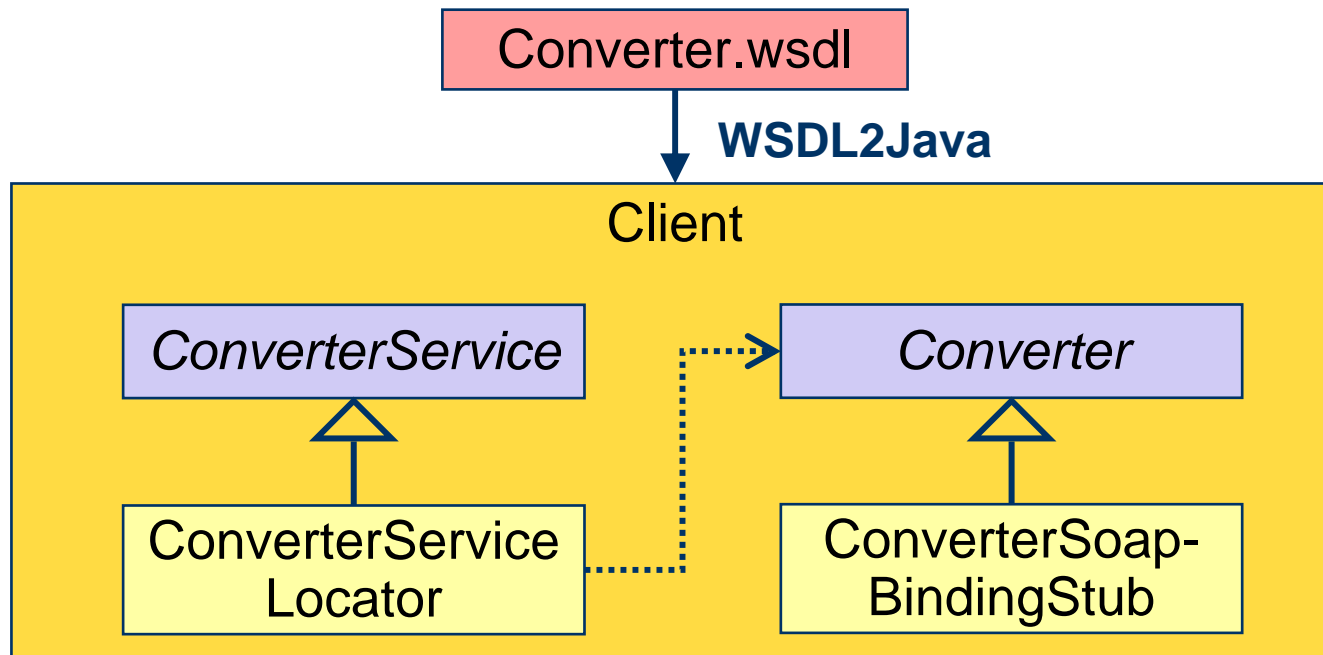
Implementierung eines EJB-Endpunkts

```
public interface Shop extends java.rmi.Remote {
    ArticleValue getArticle(String articleID)
        throws java.rmi.RemoteException;
}
```

```
public class ShopEJB implements SessionBean {
    public void setSessionContext(SessionContext ctx) { ... }
    public void ejbCreate() { ... }
    public void ejbRemove() { ... }
    public void ejbActivate() { ... }
    public void ejbPassivate() { ... }
    public ArticleValue getArticle(String articleID) { ... }
}
```

- Deployment-Deskriptoren
 - *ejb-jar.xml*: Definition eines Interfaces als *service-endpoint*.
 - *webservices.xml*: Definition WSDL, Verbindung Web-Service/Bean
 - Mapping-File: Abbildung WSDL auf Java-Interface.

Clients (1): Static Stubs



ConverterClient:

```
ConverterService service = new ConverterServiceLocator();
Converter converter = service.getConverter();
double rate = converter.rateOfExchange("USD", "EUR");
```

Clients (2): Dynamic Proxies

- Dynamic Proxy: Stub-Code wird zur Laufzeit erzeugt.
- Der Proxy-Code wird aus dem WSDL-Dokument abgeleitet.

```
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(
    new URL("http://.../service/Converter?wsdl"),
    new QName("http://.../myservices", "ConverterService"));
Converter converter =
    (Converter)service.getPort(Converter.class);
converter.rateOfExchange("USD", "EUR");
```

Clients (3): Dynamic Invocation Interface (DII)

- Bei Verwendung von DII ist kein WSDL-Dokument notwendig.
- Typinformation muss explizit angegeben werden.

```
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(
    new QName("http://.../myservices", "ConverterService"));

Call currRateCall = service.createCall(
    new QName("http://.../myservices", "Converter"),
    new QName("http://.../myservices", "rateOfExchange"));

currRateCall.addParameter("fromCurr", XMLType.XSD_STRING,
    String.class, ParameterMode.IN);
currRateCall.addParameter("toCurr", XMLType.XSD_STRING,
    String.class, ParameterMode.IN);
currRateCall.setReturnType(XMLType.XSD_DOUBLE, Double.class);
currRateCall.setTargetEndpointAddress(serviceAddr);

Double rate = (Double)currRateCall.invoke(
    new Object[] { "EUR", "USD" });
```

Unterstützte Datentypen (1)

- Einfache Typen: *boolean, byte, short, int, long, float, double*.
- Wrapper-Klassen: *Short, Integer, Long, Float, Double, ...*
- Andere Klassen: *String, Date, Calendar, BigDecimal, BigInteger*.
- Arrays: Elementtyp muss unterstützt sein.
- *Value Classes*
 - Für komplexe (aggregierte) Typen,
 - nur Properties (Setter+Getter) werden serialisiert.

```
class Person {  
    private String name;  
    public void setName(  
        String n) { ... }  
    public void getName() { ... }  
}
```

```
interface PersonAdmin {  
    Person getPerson();  
    void update(Person p);  
}
```

Unterstützte Datentypen (2)

- *Holder Classes*

- Für Referenzparameter bei Methodenaufrufen.
- *IntHolder, DoubleHolder, CalendarHolder, ...*

```
class IntHolder {
    public int value;
    public IntHolder(int v) {
        value = v;
    }
}
```

```
interface Calculator {
    void twice(IntHolder n);
}
```

```
class Client {
    public static void main(String[] args) {
        Calculator calc = ...;
        IntHolder n =
            new IntHolder(5);
        calc.twice(n);
        System.out.println(n.value); // → 10
    }
}
```

Einschränkungen von Web-Services

- JAX-RPC Programmiermodell ist ähnlich zu jenem von RMI und CORBA.
- Wesentliche Einschränkung:
 - Entfernte Objekte leben nur für die Dauer eines Requests (Objekte sind *zustandslos*).
 - Web-Services können keine Referenzen auf ein entfernte Objekte übergeben.
 - Parametertypen dürfen nicht *java.rmi.Remote* implementieren.
 - Serialisierung von Methodenaufrufen erfolgt nicht mit dem Standard-Serialisierungsmechanismus von Java.
 - Nur Properties – definiert durch Setter- und Getter-Methoden – werden serialisiert.
 - Datenkomponenten werden nicht übertragen.

Abschließende Betrachtungen

- *Breite Unterstützung* der Software-Industrie
 - Für viele Plattformen und Programmiersprachen verfügbar (Java, .NET, C++, Script-Sprachen, PDAs, Mobiltelefone).
- Einfache *A2A-Kommunikation*.
 - einfaches APIs (z.B. JAX-RPC) für von Services und Clients.
- *Verbindungsloses* Programmiermodell
 - durch verbindungslose Transportprotokolle, wie HTTP,
 - geeignet für unsichere Netzverbindungen (WLAN).
- Web-Services sind *zustandslos*.
- Zugriff auf Web-Services ist *langsam*.
 - Parsen der SOAP-Nachrichten ist rechenzeit-intensiv (besonders bei Smart Clients),
 - SOAP-Nachrichten enthalten redundante Daten.