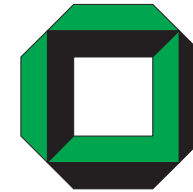

Software aus Komponenten



Prof. Dr. Gerhard Goos
Fakultät für Informatik
Universität Karlsruhe



Karlsruhe, Germany 2004

© Gerhard Goos, Dirk Heuzeroth,
Elke Pulvermüller, Volker Kuttruff
2004

<http://www.info.uni-karlsruhe.de/>



Organisation

- Vorlesung:
 - ▶ 1. Dienstag 14:00 – 15:30, -101, wöchentlich
 - ▶ 2. Mittwoch 14:00 – 15:30, -101, vierzehntägig
- Übungen:
 - ▶ Mittwoch 14:00 – 15:30, -101, vierzehntägig, erstmals 28.04.2004
- Übungsleiter: Dr. Dirk Heuzeroth, Mamdouh Abu-Sakran, Volker Kuttruff
- Adressen:
 - ▶ Allgemeines Verfügungsgebäude, 2. OG
 - ▶ ggoos@ipd.info.uni-karlsruhe.de
 - ▶ heuzer@ipd.info.uni-karlsruhe.de, msakran@ipd.info.uni-karlsruhe.de, kuttruff@fzi.de
- Sprechstunden:
 - ▶ Goos: Dienstags 16:00 – 17:00
 - ▶ Heuzeroth, Abu-Sakran, Kuttruff: Mittwochs 16:00 – 17:00
- Vorlesungsseite:
<http://www.info.uni-karlsruhe.de/lehre/2004SS/swk>

1. Einführung

- ▶ Motivation
- ▶ Definition,
- ▶ Konzepte für Komponenten (klassische, kommerzielle, akademische)

2. Industrielle Komponentensysteme der 1. Generation

2.1 CORBA

2.2 Enterprise JavaBeans

2.3 (D)COM, .Net

2.4 Webdienste, XML

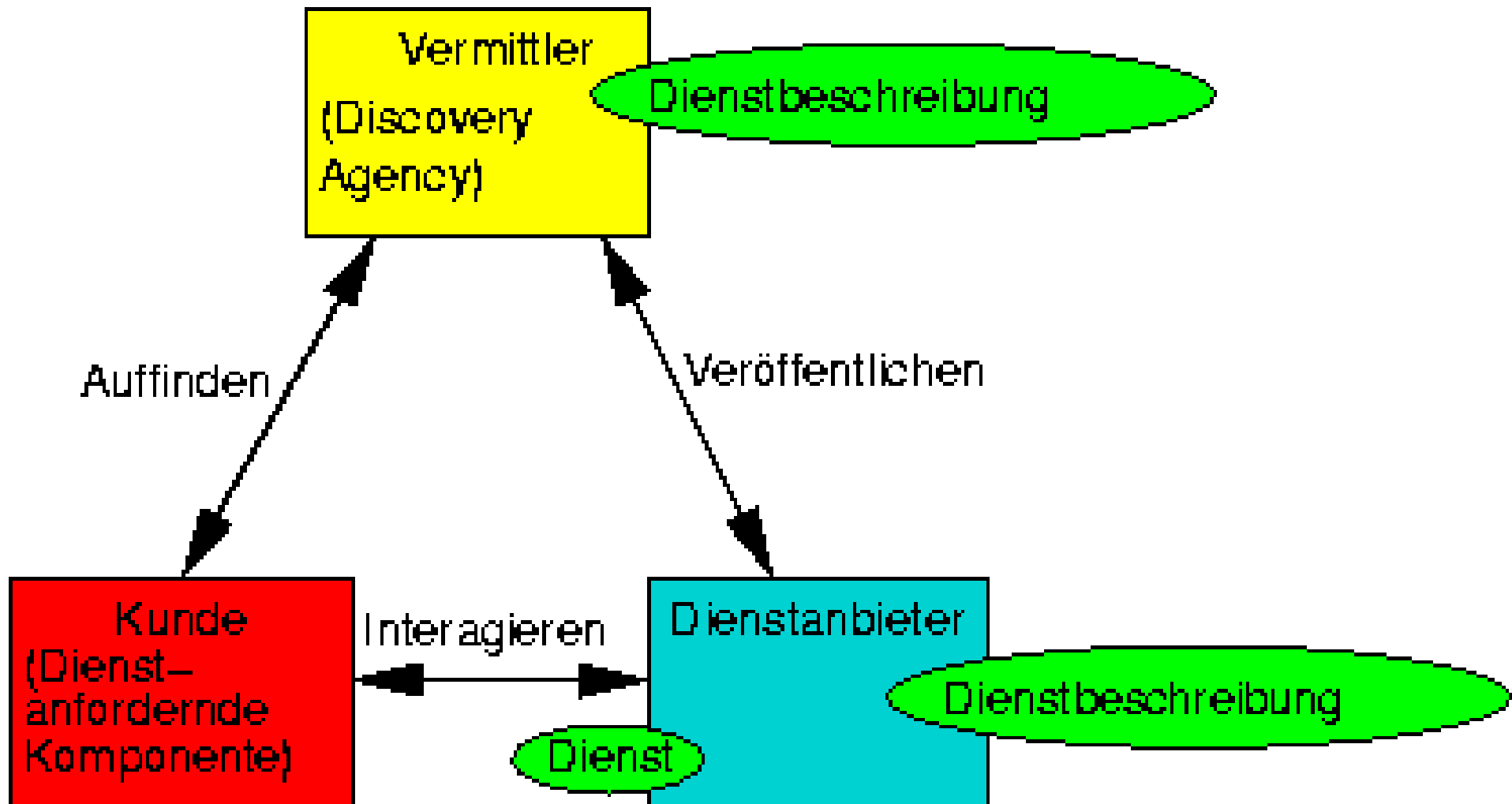
3. Akademische Ansätze

- ▶ Architektursysteme
- ▶ Aspektorientiertes Programmieren
- ▶ Metaprogrammierung und Invasive Komposition

2. Konkrete industrielle Systeme

- Pragmatisch motiviert
 - ▶ Wie baue ich große Systeme in einer heterogenen Welt?
- Wie baue ich große Systeme in einer heterogenen Welt?
- Man braucht Mechanismen zur Transparenz von
 - ▶ Sprachen (IDL)
 - ▶ Plattformen
 - ▶ Betriebssystemen
 - ▶ Ausführungsort:
 - Global eindeutige Referenz
 - Stellvertreterobjekte (*stub/skeleton*)
 - Wiederverwendbare Dienste zum Ermitteln von Komponenten und deren Eigenschaften (z.B. für dynamische Komposition):
 - Namensdienst
 - Makler
 - Persistenz (Objekte aus Datenbank holen)
 - Reflexion (Schnittstellen erfragen)
 - Objektverwaltung (Lebenszyklus)
- **Jetzt:** Ausprägung in
 - ▶ Corba (Mustergültig)
 - ▶ COM/DCOM/.NET (Plattformgebunden)
 - ▶ JavaBeans / Enterprise JavaBeans (Sprachgebunden)
 - ▶ Webdienste/XML

Webdienste (Ortstransparenz)



- Offene Standards (W3C) als Schnittstelle zum Web
- Client/Server-Paradigma ist kein Muß, d. h. es muß keinen zentralen Datenbank-Server geben.
- Austausch strukturierter Dokumente zwischen den Diensten

```
<%@ WebService Language="C#" Class="TimeService" %>
```

```
using System.Web;
```

```
using System.Web.Services;
```

```
public class TimeService : WebService {
```

```
    [WebMethod(Description="Returns the current time.")]
```

```
    public string GetTime() {
```

```
        return System.DateTime.Now.ToLongTimeString();
```

```
    }
```

```
}
```

- Anbindung an .NET durch Rahmensystem/Werkzeuge unterstützt

- Ebenen laut Microsoft:

Ebenen	Standard
Datenformat	XML
Nachrichtenformat	SOAP
Dienstbeschreibung	WSDL
Auffindung von Diensten auf Servern	?
Auffindung von Servern	UDDI

- Problematische Unterscheidungen
 - ▶ Datenformat - Nachrichtenformat
 - ▶ Ebenen des Auffindens
- Funktionalität steht im Vordergrund, nicht Performanz.

- Cobol
 - ▶ Grundprinzip: alle Ausgaben als Text (Serialisierung)
 - ▶ Textelemente typisiert
- Unix
 - ▶ alle Ausgaben serielle (Text-)Dateien
 - ▶ typfrei: Typisierung der Systemdateien nicht explizit codiert
 - daher alle Systemdateien einheitlich mit Texteditor bearbeitbar, falls der Benutzer das Format (Typ) kennt
 - Unterschied zu Windows und neueren Ansätzen unter Linux
- SGML - Standard Generalized Markup Language
 - ▶ Ziel: einheitliche Beschreibungssprache für Textformatierung
 - ▶ Definieren eigener Unterformate (Typen) möglich
 - ▶ syntaktisch sehr flexibel
- HTML - Hypertext Markup Language
 - ▶ Ein Unterformat von SGML
 - ▶ Keine eigenen Unterformate möglich
 - ▶ Siegeszug mit dem WWW

- Problem:
 - ▶ Datenaustausch zwischen Programmen, nicht nur Textformatierung
 - ▶ aus SGML entwickelt, aber erweitertes Aufgabengebiet
 - ▶ HTML zu spezifisch, nur für Hypertext
- Lösung: Datenaustauschformat, allgemeine Sprachen zur
 - ▶ Beschreibung von attributierten Termen/Bäumen
 - ▶ Typdefinition für attributierte Terme/Bäume
 - ▶ Querverweise durch Attribute
 - ▶ Untermenge von SGML
 - ▶ Syntax an HTML angelehnt
- Umsetzung: XML
 - ▶ Standard des WWW Konsortiums (W3C): <http://w3c.org>
 - ▶ kritische Masse erreicht

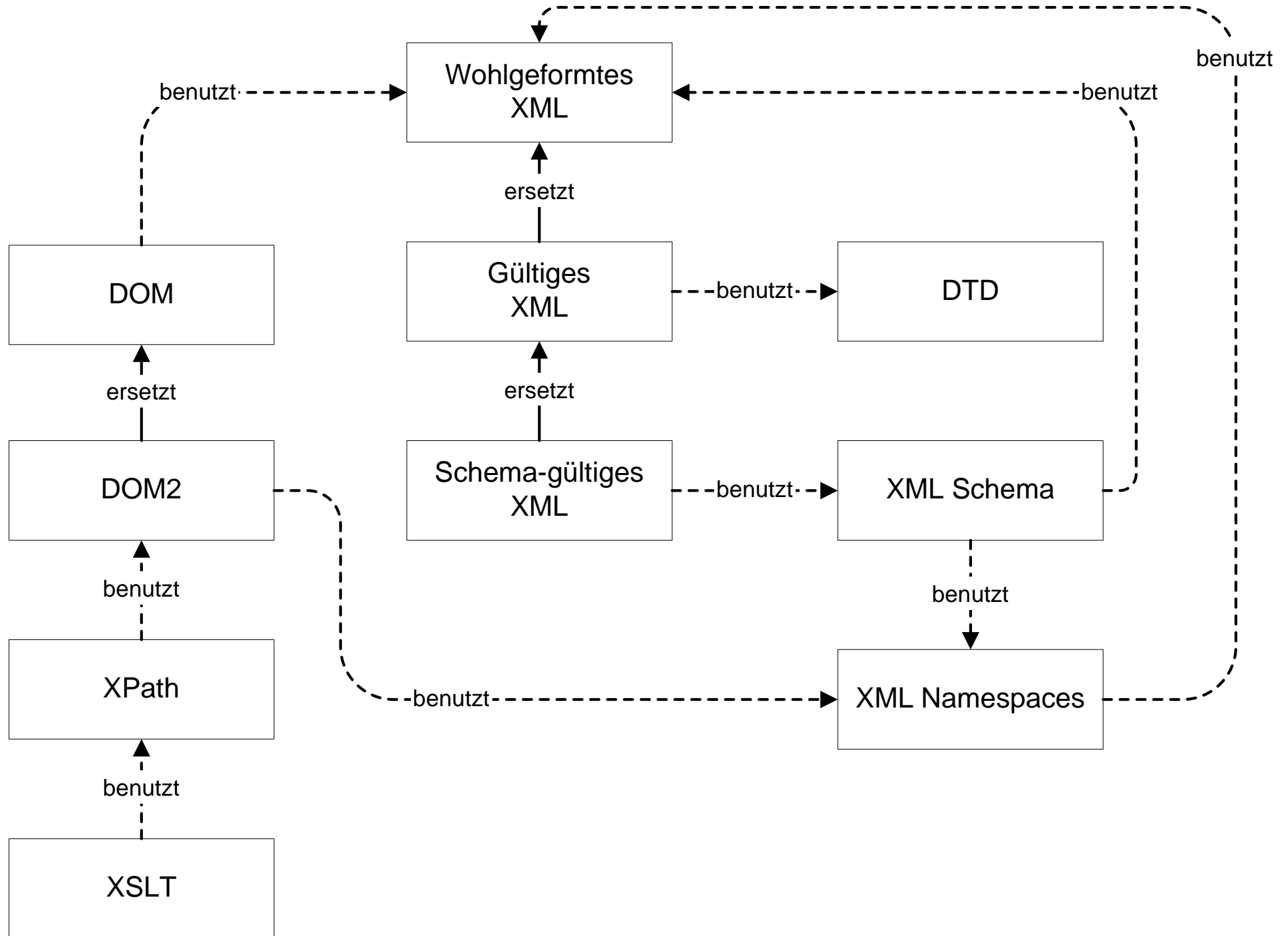
XML Beispiel

```
<treatment>
  <patient insurer="1577500"nr='0503760072' />
  <doctor city    ="HD"      nr='4321' />

  <service>
    <mkey>1234-A</mkey>
    <date>2001-01-30</date>
    <diagnosis>No complications.
  </diagnosis>
</service>

</treatment>
```

XML Standards



- Leichter zerteilbar als allgemeine kontextfreie Sprachen
 - ▶ Elemente durch Separatoren "<", . . . , ">" ausgezeichnet
 - ▶ Explizite Schachtelung von Elementen (schließende Klammern)
 - ▶ Grammatik für Kinderelemente ist regulär
- Einschränkung
 - ▶ Format ist auf Unicode definiert
 - ▶ Konvertierung in Bytestrom (Serialisierung) nicht in Grammatik festgelegt
 - Unicode UTF-8 oder UTF-16
 - ISO-8859-1, -15, . . .
- Formen der Korrektheit
 - ▶ Wohlgeformtes (*well-formed*) XML:
Korrekte Syntax und Klammerung
 - ▶ Gültiges (*valid*) XML:
Wohlgeformt und konform zu einer DTD
 - ▶ Schema-gültiges (*schema-valid*) XML:
Wohlgeformt und konform zu einem XML Schema

Namensräume

- Problem: Heterogene Systeme
 - ▶ Wieviele Definitionen von z.B. Behandlung gibt es?
 - ▶ Wie vermeide ich Probleme beim Zusammenführen?
- Ansatz: Einführen von Namensräumen
 - ▶ Name ist Paar (Namensraum, lokaler Name)
 - ▶ international eindeutige Bezeichner für Namensräume (?)
- Umsetzung in XML
 - ▶ Lange Bezeichner für Namensräume, lokale Platzhalter im Dokument
 - ▶ URIs als Bezeichner eines Namensraums (aber nicht vorgeschrieben)
 - ▶ `<ns:ln xmlns:ns="http://www.noga.de/namespaces/vorlesung">`
 - `<ns:xyz/>`
 - `</ns:ln>`
- leider ein technisch nicht ausgereifter Standard:
 - ▶ Gültigkeitsbereich eines Namensraum-Bezeichners: XML-Element mit Kindern (wie Variable) einschl. Elementname, vorangehende Attribute!
 - ▶ Daher nicht LL(k) wie der Rest von XML
 - ▶ Pässe:
 - Zerteile wohlgeformtes XML, löse Namensräume auf, prüfe Typen

Document Type Description (DTD)

- Typbeschreibung für XML-Dokumente (kontextfreie Grammatik)
- Selbst kein XML-Dokument
- Basisdatentypen für Elemente
 - ▶ Zeichenkette (PCDATA)
- Typkonstruktoren für Elemente
 - ▶ Iteration
 - ▶ Sequenz
 - ▶ Alternative
 - ▶ Schachtelungen davon
- Basisdatentypen für Attribute
 - ▶ Zeichenkette (CDATA)
 - ▶ Schlüssel und Bezüge (ID, IDREF)
 - ▶ ...
- Typkonstruktoren für Attribute
 - ▶ Aufzählung

DTD Beispiel

```
<!ELEMENT treatment
  (patient (doctor|hospital) service+)>
<!ELEMENT patient EMPTY>
  <!ATTLIST patient  insurer CDATA '1'
                    nr CDATA #REQUIRED>
<!ELEMENT doctor EMPTY>
  <!ATTLIST doctor  city CDATA #REQUIRED
                    nr CDATA #REQUIRED>
<!ELEMENT hospital EMPTY>
  <!ATTLIST hospital city CDATA #REQUIRED
                    nr CDATA #REQUIRED>
<!ELEMENT service (mkey date diagnosis)>
<!ELEMENT mkey #PCDATA>
<!ELEMENT date #PCDATA>
<!ELEMENT diagnosis #PCDATA>
```

Typkonstruktoren und reguläre Ausdrücke

Datentyp	Ausdruck
ARRAY OF X	$(X)^*$
RECORD X, Y, ..., Z	(X, Y, \dots, Z)
UNION X, Y, ..., Z	$(X Y \dots Z)$

- selbst kein XML (Problem bei *boot-strapping*)
- sehr eingeschränktes Typsystem
 - ▶ keine Basistypen außer Zeichenketten
 - ▶ keine festen Reihungen
 - ▶ keine Vererbung
- mehrdeutige Ableitungsbäume
 - ▶ Ursache: Schachtelung von Konstruktoren
 - ▶ Beispiel:
 - DTD-Fragment `<!ELEMENT a (b?, b?)>`
 - Ableitung von `<a>` ist nicht eindeutig
- Namensräume nicht integriert
- langsame Verarbeitung wegen Interpretation

die (nachträglich erfundene) Verallgemeinerung von DTDs

- Zweck: Beschreibung der Struktur und Einschränkung des Inhalts von XML-Dokumenten
- XML Schema ist selbst XML-Dokument
- W3C Standard
- Typsystem
 - ▶ übliche Basistypen: String, Integer, ...
 - ▶ unübliche Basistypen: Date, Time
 - ▶ Einschränkungen darauf möglich
 - ▶ Feste Reihungen durch Einschränkung des Sequenzenkonstruktors
 - ▶ Kovariante und kontravariante Vererbung
- Namensräume integriert
- langsame Verarbeitung wegen Interpretation
- mehrdeutige Ableitungsbäume

Bestandteile von XML-Schema:

- Primäre Komponenten, die einen Namen haben können (Typdefinitionen) oder müssen (Element und Attributdeklarationen):
 - ▶ Einfache Typdefinitionen
 - ▶ Komplexe Typdefinitionen
 - ▶ Attributdeklarationen
 - ▶ Elementdeklarationen
- Sekundäre Komponenten, die Namen haben müssen:
 - ▶ Attributgruppendifinitionen
 - ▶ Identity-constraint-Definitionen
 - ▶ Modellgruppendifinitionen
 - ▶ Notationsdeklarationen
- Hilfskomponenten:
 - ▶ Annotationen
 - ▶ Modellgruppen
 - ▶ Partikel
 - ▶ Wildcards
 - ▶ Attributverwendungen

- Definition eines Elements:

- ▶ Zuordnung eines Typs zu einem Namen
- ▶ Gültigkeitsbereich ist der umgebende Typ

- Typ durch Attribut zuordnen:

```
<xs:element name="PurchaseOrder" type="PurchaseOrderType"/>
```

- Typ explizit angeben:

```
<xs:element name="gift">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="birthday" type="xs:date"/>  
      <xs:element ref="PurchaseOrder"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>
```

- DTD
 - ▶ Typ kein eigenständiges Konzept
 - ▶ Definition eines XML-Elements
 - definiert den zugehörigen Typ
 - mit globalem Gültigkeitsbereich
- XML Schema
 - ▶ Typ ist eigenständiges Konzept
 - Konstruktoren für benutzerdefinierte Typen

- Konstruktoren
 - ▶ Einschränkung
 - ▶ Vereinigung
 - ▶ Iteration

- Hier nur Einschränkung

```
<!-- Einschränkung des Typs integer auf eine maximale Stellenzahl -->  
<simpleType name='insurer' base='integer'>  
  <precision value='7' />  
</simpleType>
```

```
<!-- Einschränkung des Typs date mit Ober- und Untergrenzen -->  
<simpleType name='myDate' base='date'>  
  <minInclusive value='2001-01-01' />  
  <maxExclusive value='2001-04-01' />  
</simpleType>
```

- Konstruktoren
 - ▶ Einschränkung
 - ▶ Erweiterung
- Hier nur Einschränkung

```
<!-- Einschränkung des Urtyps (implizit) -->  
<complexType name='treatment'>  
  <element name='patient' type='patient' />  
  <choice>  
    <element ref='doctor' />  
    <element ref='hospital' />  
  </choice>  
  <element ref='service'  
    maxOccurs='unbounded' />  
</complexType>
```

- Attribute werden innerhalb komplexer Typen definiert
- Definition bindet einfachen Typ an Namen
- Unterschiedliche Syntax: Bezug auf Typ, Inline-Definition

```
<complexType name='patient' content='empty'>  
  <attribute ref ='insurer' use='required' />
```

```
  <attribute name='nr' use='required'>  
    <simpleType base='integer'>  
      <precision value='10' />  
    </simpleType>  
  </attribute>
```

```
  <attribute name='since' type='myDate' />  
</complexType>
```

Programmiersprachen

- Basisdatentypen
 - ▶ unterschiedliche Integers, Floats, usw.
- Reihungen
 - ▶ statisch
 - ▶ dynamisch
 - ▶ flexibel

XML Schema

- Basisdatentypen
 - ▶ erweitert um URL, Datum, Zeit, etc.
 - ▶ Standardtypen mit beliebigen Einschränkungen auf Wertebereiche und Literale
- Reihungen durch Attribute
 - ▶ dynamische Grenzen (min=max → statisch)
`minOccurs='X' maxOccurs='Y'`
 - ▶ dynamische und flexible Reihungen werden nicht unterschieden
`maxOccurs='unbounded'`

Programmiersprachen

- Verbunde
 - ▶ Zugriff über Namen
- Verbunde mit Varianten
 - ▶ Verbunde mit/ohne Typkennung
 - ▶ Zugriff auf Variante, die in Typkennung codiert ist
 - ▶ typsicher, wenn kein Variantenwechsel bei ein und demselben Objekt möglich ist

XML Schema

- Verbunde
 - ▶ Struktur von XML (Schema ist XML)
 - ▶ Zugriff über Namen von Unter-elementen und/oder Position
- Verbunde mit Varianten durch Elemente mit Auswahl
 - ▶ `<choice> Auswahl </choice>`
 - ▶ aktuelle Variante i. a. nicht erkennbar, da nicht eindeutig
 - ▶ nicht typsicher
 - ▶ typsicher, wenn Auswahl nur unter Elementen (siehe Beispiel auf nächster Folie)

Beispiel: Verbunde mit Varianten

```
<complexType name="C">
  <choice>
    <sequence maxOccurs
      ="unbounded">
      <element ref="a"/>
      <element ref="b"/>
    </sequence>

    <choice maxOccurs
      ="unbounded">
      <element ref="a"/>
      <element ref="b"/>
    </choice>
  </choice>
</complexType>
```

Ableitung für <a/> ?

```
<complexType name="C1">
  <sequence maxOccurs="unbounded">
    <element ref="a"/>
    <element ref="b"/>
  </sequence>
</complexType>
```

```
<complexType name="C2">
  <!-- analog -->
</complexType>
```

```
<complexType name="C">
  <choice>
    <element name="c1" type="C1"/>
    <element name="c2" type="C2"/>
  </choice>
</complexType>
```

hier sicher, z.B. <c1><a/></c1>

Programmiersprachen

- Klassen
 - ▶ Verbunde mit Daten und Funktionen
 - ▶ Vererbung
 - ▶ Polymorphie

XML Schema

- Klassen
 - ▶ Verbunde wie besprochen
 - ▶ Referentielle Transparenz: keine Unterscheidung zwischen 0-stelligen Funktionen und Daten
 - ▶ Keine Entsprechung für allgemeine Funktionen
- Beschreibung von Funktionen erfordert Erweiterungen
 - ▶ Standard: WSDL

Programmiersprachen

- Klassenvererbung
 - ▶ Spezialisierung (Kovarianz)
 - ▶ Konformität (Kontravarianz)
- Referenzen
 - ▶ typisiert oder untypisiert
 - ▶ auf beliebige Werteobjekte

XML Schema

- Ableitung von Verbundtypen
 - ▶ Einschränkung entspricht Spezialisierung
 - ▶ Erweiterung entspricht Konformität
- Referenzen durch Attribute
 - ▶ ID und IDREF Typen
 - ▶ Untypisiert
 - ▶ Identifikation von Objekten mit Schlüsselwerten
 - ▶ Keine Referenzen auf WSDL-Dienste

XML erlaubt das Spezifizieren von Daten und das Parametrisieren von Diensten in Schnittstellen.

- Vorteile
 - ▶ standardisierte Syntax
 - ▶ standardisierte Transformation
- Nachteile
 - ▶ nur Wertesemantik
 - ▶ nur Verbunde, keine Klassen
 - ▶ kein Vererbungskonzept
- Es besteht also noch Verbesserungsbedarf, um Komponenten speziell objektorientierte mit XML zu spezifizieren.

Webdienste: Simple Object Access Protocol (SOAP)

- XML-basiertes Nachrichtenformat
- Nachricht ist Umschlag mit Kopf und Rumpf
 - ▶ Kopf enthält Adressaten und Verarbeitungsinformation
 - ▶ Rumpf enthält Nutzdaten oder Fehlercodes
- Wirre Mechanismen zur Typdefinition
 - ▶ Reihungen
 - ▶ sonst Rückgriff auf Namensräume (implizit also Schemata)
- Transport ist transparent, vordefinierte Kanäle
 - ▶ HTTP (mit Rückkanal)
 - ▶ SMTP (Simple Mail Transport Protokoll)
 - ▶ TCP (mit Rückkanal)
- Problem: Beliebigkeit wegen zu vieler nicht standardisierter Eigenschaften
 - ▶ Typen der Nutzdaten a priori unbekannt
 - ▶ Festlegen von Typen in der Nachricht
 - ▶ Interpretation nötig, daher ineffizient

Webdienste: WSDL

- XML-basierte Beschreibungssprache für Webdienste
- WSDL (Web Services Description Language) beantwortet folgende Fragen:
 - ▶ Welche Dienste bieten welche Methoden an?
 - ▶ Über welche Tore (*ports*), Protokolle und Nachrichten können die Methoden aufgerufen werden?
 - ▶ Welche Namen und welche Parameter hat eine Nachricht?
 - ▶ Wie sehen die verwendeten Datentypen aus?
- Ansatz
 - ▶ Schnittstellen sind Mengen von Signaturen
 - ▶ Typsystem transparent (XML Schema)
 - ▶ Nachrichtenformat/Abbildung auf konkrete Schnittstellen:
 - SOAP
 - MIME (*Multi-Purpose Internet Mail Extensions*) für SMTP
- Probleme
 - ▶ Strukturiertes Programmieren, aber keine Objektorientierung (keine Vererbung, keine Polymorphie)
 - ▶ Typsystem nicht standardisiert
 - ▶ XML Schema kennt nur Werttypen
 - ▶ Gewünscht: Typisierte WSDL Referenzen
 - ▶ Referenzen (auf Dienste) nicht explizit unterstützt

Eine WSDL besteht aus 5 hierarchisch aufeinander aufbauenden Abschnitten:

```
<definitions>
  <types>          <!-- beschreibt die verwendeten Datentypen in XML -->
    <xsd:schema>
  </types>
  <message>        <!-- beschreibt eine Nachricht (Name und Parameter) -->
    <part>         <!-- "<message>" kann mehrmals vorkommen. -->
  </message>
  <portType>       <!-- beschreibt für jedes Protokoll die aufrufbaren Nachrichten, -->
    <operation>    <!-- die eine Operation bilden (Eingangs- und Ausgangsnachricht). -->
      <input>      <!-- Dadurch werden auch alle Operationen beschrieben, -->
      <output>     <!-- die über diesen "Port" erreichbar sind. -->
    </operation>
  </portType>
  <binding>        <!-- bindet ein Protokoll an einen Port und beschreibt, wie die -->
    Protokollinfo  <!-- Daten codiert und serialisiert werden (z.B. SOAP, rpc/encoded). -->
    <operation>
  </binding>
  <service>        <!-- beschreibt den Dienstnamen und über welche URI und -->
    <port>         <!-- Anbindung dieser Dienst erreichbar ist. -->
  </service>
</definitions>
```

- abstrakter Teil: <types>, <message>, <portType>
- konkreter Teil: <binding>, <service>

Beispiel WSDL

```
<wsdl:definitions>
  <!-- missing schema for med:patient import -->
  <wsdl:message name="startTreatmentIn">
    <!-- only one parameter -->
    <part name="body" element="patient" type="med:patient"/>
  </wsdl:message>
  <wsdl:message name="startTreatmentOut">
    <part name="body" element="accepted" type="xsd:boolean"/>
  </wsdl:message>
  <wsdl:portType name="treatmentPort">
    <wsdl:operation name="startTreatment">
      <wsdl:input message="startTreatmentIn"/>
      <wsdl:output message="startTreatmentOut"/>
    </wsdl:operation>
  </wsdl:portType>

  <!-- missing binding & service definitions -->
</wsdl:definitions>
```

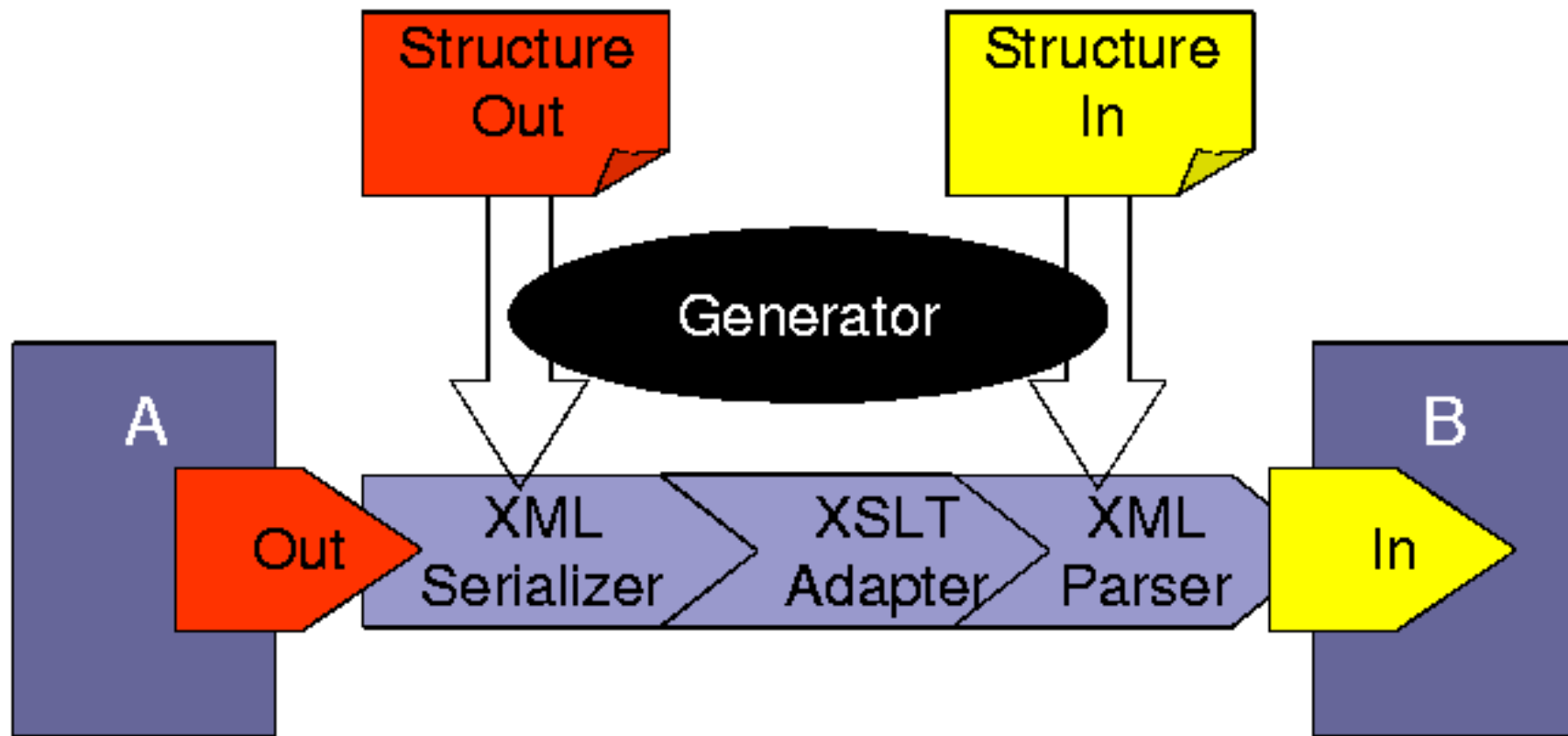
- XML-basierte Beschreibungssprache für Webdienste
- Gegenstand: Mengen von Funktionen
 - ▶ Strukturiertes Programmieren
 - ▶ Keine Objektorientierung - keine Vererbung
- Modellierung von Parametern
 - ▶ XML Schema oder beliebige andere Beschreibungssprachen
 - ▶ Referenzen (auf Dienste) nicht explizit unterstützt
- Abbildung auf konkrete Schnittstellen
 - ▶ SOAP
 - ▶ MIME (*Multi-Purpose Internet Mail Extensions*) für SMTP
- Probleme: Beliebigkeit, fehlende Objektorientierung

UDDI (Uniform Description, Discovery and Integration)

- Beschreibt Dienste auf Geschäftsebene
- Registrierung ist XML Deskriptor
 - ▶ *White Page*: Adresse, Erreichbarkeit
 - ▶ *Yellow Page*: Semantik (basiert auf Standard-Taxonomie)
 - ▶ *Green Page*: Technische Spezifikation (URL, WSDL, etc.)
- Logisch zentralisierte, physisch verteilte Datenbank
- UDDI ist kein Makler oder Marktplatz
 - ▶ Keine geographischen Anfragen
 - ▶ Keine Preisanfragen
 - ▶ Keine Zeitanfragen
- Also: Wie DNS oder JINI, nur komplexere Felder

Anpassungen: Transformation von XML-Dokumenten

- Schnittstellen und Daten durch XML-Dokumente beschrieben
- ⇒ Transformation der XML-Dokumente zur Beseitigung von Inkompatibilitäten zwischen Komponenten und damit Anpassung der Schnittstellen oder Nachrichten (inkl. Prozeduraufrufen)



- bislang:
 - ▶ Daten beschrieben mit Typbeschreibung
 - ▶ Interne Darstellung der Daten
- nun:
 - ▶ Transformation eines XML-Dokuments in ein anderes oder in Nicht-XML-Form
- Vorgehen:
 - ▶ Transformator ausprogrammieren
 - ▶ XSLT (entsprechender XML Standard dafür) benutzen
 - ▶ deskriptiver Ansatz
 - beschreibe Ergebnis der Transformation
 - nicht die nötigen Operationen!
 - ▶ Term- bzw. Graphersetzer (Techniken aus dem Übersetzerbau)

- Zerteiler liest Dokument und DTD/Schema
- Dokument wird gegen Grammatik validiert
- DOM wird aufgebaut
- Transformation/Filterung auf dem DOM
 - ▶ Transformation definiert durch XSLT Spezifikation
 - ▶ Ausgabe nicht notwendig XML

Verarbeiten von XML-Dokumenten

Verarbeitung von XML-Dokumenten: Zugriff auf Baumstruktur notwendig

- *Document Object Model* (DOM)
 - ▶ Ziel: Navigieren in einem XML-Baum ermöglichen (zur Implementierung von (Datenbank-)Abfragen)
 - ▶ Baumstruktur implizit durch Navigation gegeben.
 - ▶ W3C Standard
 - ▶ Elemente und Attribute
 - alle gleichen Typs
 - sind explizite Objekte (unterschieden durch Namen)
 - ▶ Zugriffe
 - Iteratoren über Kindelemente und Attribute
 - *i*-tes Kindelement, erstes Kindelement mit Namen *x*
 - Attributwert über Attributnamen
 - ▶ alles explizit, daher langsam, umständlich, kein Stromformat
 - ▶ Typinformation geht verloren
- SAX: push- statt pull-Modus.
- DOM 2:
 - ▶ integriert Namensräume
 - ▶ typisierte Syntaxbäume (generiert aus DTD oder Schema)
 - ▶ Schlüsselzugriffe z.B. für ID und IDREF
 - ▶ Persistente Formate (Anbindung an Datenbanken, Persistent DOM)
 - ▶ Binäres Kodierungsformat (Wahlfreier Zugriff)

- *eXtensible Stylesheet Language Transformation* (XSLT)
- W3C Standard
- XML Syntax
 - ▶ XML Schema für XSLT
 - ▶ DTD Beschreibung der Syntax nicht möglich wegen beliebiger Ausgabe
- Navigation auf DOM
 - ▶ Mischung zwischen deklarativer Notation und imperativen Anweisungen
- Mustererkennung auf DOM
 - ▶ Menge von Pfadausdrücken (XPath)
 - ▶ Achtung: Keine Baum- oder Graphmuster!
- Erzeugung von XML-Dokumenten oder anderen Texten

```
<xsl:template match="treatment">
  <html><head/> <body>
    <xsl:apply-templates/>
  </body></html>
</xsl:template>
```

```
<xsl:template match="patient">
  <h1>Patient Nummer: <xsl:value-of select="nr"/></h1>
  <xsl:apply-templates/>
</xsl:template>
```

...

Beispiel - Ausgabe

```
<?xml version="1.0" encoding="iso-8859-1"?>
<html xmlns="http://www.w3.org/TR/xhtml1/strict">
<head/>
<body>
  <h1>Patient Nummer: 0503760072</h1>
  ...
</body>
</html>
```

Pfadausdrücke (XPATH)

- Pfad bildet (DOM-) Eckenmengen zur Weiterverarbeitung
 - ▶ Folge von Schritten `schritt1/schritt2/.../schrittN`
- Schritt bildet neue Eckenmenge aus einer bestehenden
 - ▶ Folge: `achse::auswahl[prädikat1][prädikat2]...[prädikatM]`
- Achsen bilden neue Eckenmenge aus einer bestehenden
 - ▶ Sprünge zu Eltern, Kinder, Nachbarn (Vor / Nachfolger in Pre-Ordnung)
 - ▶ Entsprechende Zugriffe im DOM zur Berechnung der Kontextknoten
 - ▶ Beispiel: `ancestors::`
- Auswahl filtert bestehende Eckenmenge
 - ▶ Joker für Elementnamen: `*`
 - ▶ Joker für Attributnamen: `@*`
 - ▶ Wurzel des Dokuments: `/`
 - ▶ Text, Kommentar, Verarbeitungsanweisungen

- Prädikate filtern bestehende Eckenmenge
 - ▶ Ausdruckssyntax
- Beispiele:
 - ▶ Ist Attribut definiert?
`element[@attribut]`
 - ▶ Hat Attribut bestimmten Wert (immer String)?
`element[@attribut=wert]`
 - ▶ Steht element an Position x (immer int)?
`element[position()=x]`

- XSLT ist Menge von Schablonen bestehend aus
 - ▶ Muster (XPATH Ausdruck)
 - ▶ Anweisungen (Ausgaben, rekursive Aufrufe etc.)
- initial enthält die Kontextknotenliste den Wurzelknoten
- Auswahl des passenden Musters
 - ▶ Ausführung der Anweisungen in der Schablone
 - ▶ bei Rekursion (apply-templates Anweisung):
Aufbau neuer Kontextmengen
- Einschränkungen mittels
 - ▶ Select (XPATH Ausdruck)
 - ▶ Regelmodi und -namen

Beispiel Kopierer

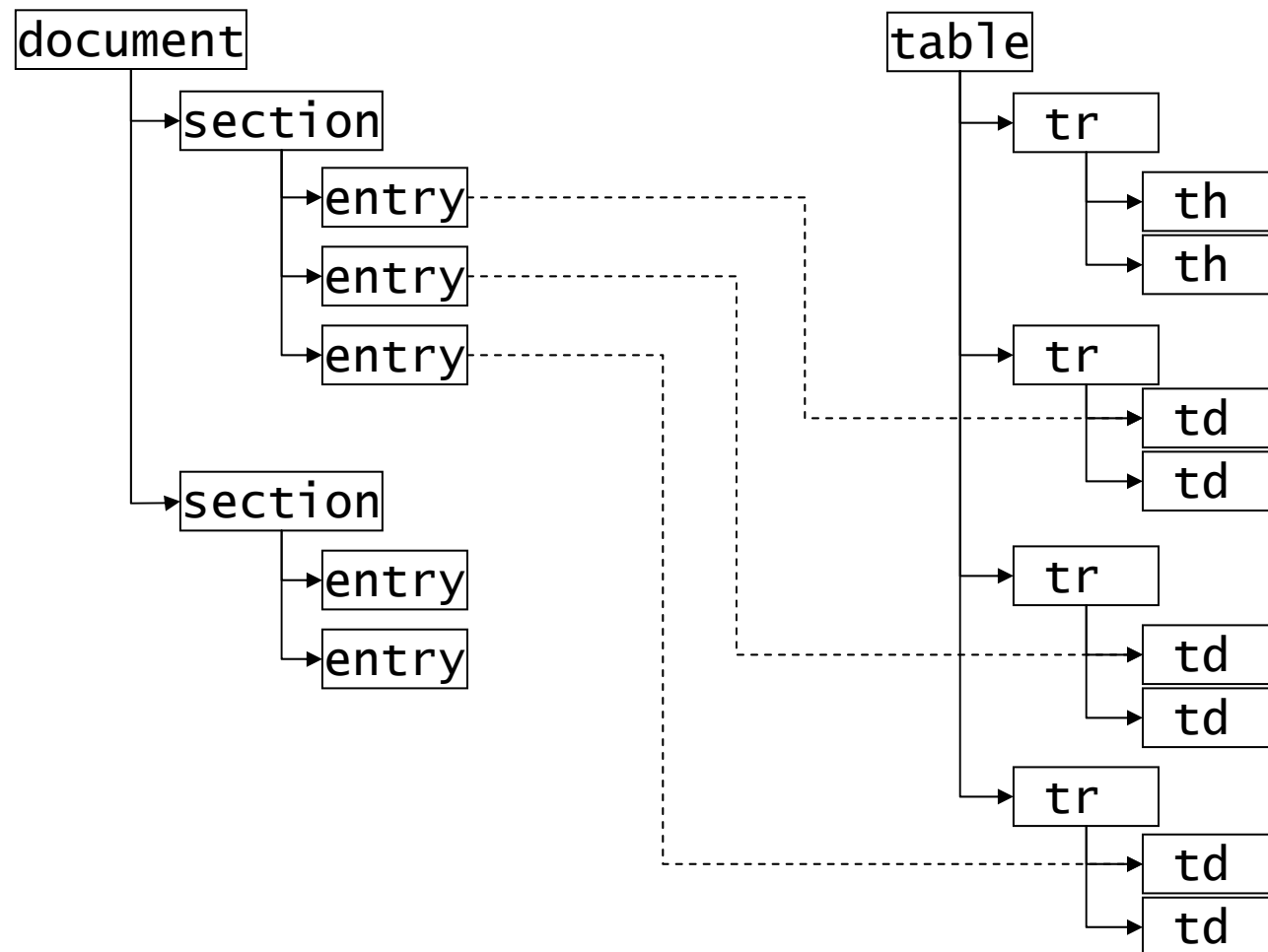
```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:strip-space elements="*" />
<xsl:output ... />

<!-- Regel gilt für alle Elemente, Attribute, Kommentare, PIs, Texte -->
<xsl:template match="*|@*|comment()|processing-instruction()|text()">
  <!-- Aktuellen Knoten kopieren -->
  <xsl:copy>
    <!-- Bekannte Regeln auf alle Kinder anwenden
      (Elemente, Attribute, Kommentare, PIs, Texte) -->
    <xsl:apply-templates
      select="*|@*|comment()|processing-instruction()|text()" />
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

- Klassische Funktionsaufrufe mit Parametern
- Rekursion
- Fallunterscheidung
- Variable mit statischer Einmalzuweisung
- Schnitte von Eckenmengen
- Anbindung an Skriptsprachen
 - ▶ möglicher Indikator für mangelnde Mächtigkeit von XSLT
 - ▶ bisher kaum genutzt

Beispiel mit Rekursion: XML → HTML



Beispiel

```
<xsl:template match="document">
  <html>
    <head> </head>
    <body>
      <table>
        <tr>
          <xsl:apply-templates select="section"/>
        </tr>|\\
        <xsl:call-template name="rowCounter">
          <xsl:with-param name="N" select="1"/>
        </xsl:call-template>
      </table>
    </body>
  </html>
</xsl:template>
```

Beispiel fortgesetzt

```
<xsl:template name="rowCounter">
  <xsl:param name="N" />
  <xsl:if test="section/entry[ \ $N ]">
    <tr>
      <xsl:for-each select="section">
        <td>
          <xsl:apply-templates select="entry[ \ $N ]"/>
          <xsl:text> </xsl:text>
        </td>
      </xsl:for-each>
    </tr>
    <xsl:call-template name="rowCounter">
      <xsl:with-param name="N" select="\ $N + 1"/>
    </xsl:call-template>
  </xsl:if>
</xsl:template>
<xsl:template match="section">
  <th><xsl:value-of select="@name"/></th>
</xsl:template>
```

- XSLT ist turingmächtig
 - ▶ Bedingungen
 - ▶ Rekursionen
- deskriptiv in den Pfadausdrücken
- Rest ist funktionales Programm
- Standardisierung ohne Verständnis der Standardtechnologie:
 - ▶ Mustererkennungsgeneratoren
 - ▶ Termersetzungsgeneratoren
 - ▶ Graphersetzungsgeneratoren

- unschön weil nicht deskriptiv
- Ineffiziente Implementierung
 - ▶ Standardwerkzeuge interpretieren Skripte online
- inhärent ineffizient
 - ▶ Kontextinformation zur Anwendbarkeit einer Transformation muß u.U. immer wieder neu berechnet werden
 - ▶ Aufwand $O(n^2)$, obwohl Problem eigentlich $O(n)$...

Alternativen

- Nachteil aller Alternativen: Transformationsspezifikation ist kein XML-Dokument.
- Dafür: Saubere, mächtige Spezifikations- und Transformationskonzepte:
 - ▶ Termersetzungssysteme (TES)
 - Eingabe: (XML)Term, Muster, Ersetzung
 - Ausgabe: minimaler Fixpunkt nach iterativem Ersetzen
 - ▶ Graphersetzungssysteme (GES)
 - Eingabe: (XML)Term (über Namen und Referenzen eigentlich ein Graph), Match, Ersetzung
 - Ausgabe: minimaler Fixpunkt nach iterativem Ersetzen
 - Systeme: Optimix (IPD) oder Progress (RWTH Aachen)
 - ▶ Durchmustern kann durch Prioritäten gesteuert werden
 - ▶ Generatoransatz:
 - erzeugt Term- oder Graphersetzungssystem aus Term-(Graph) Typ (DTD oder XML Schema) und Match-, Ersetzungsregeln
 - eigentliche Ersetzung durch generiertes Programm

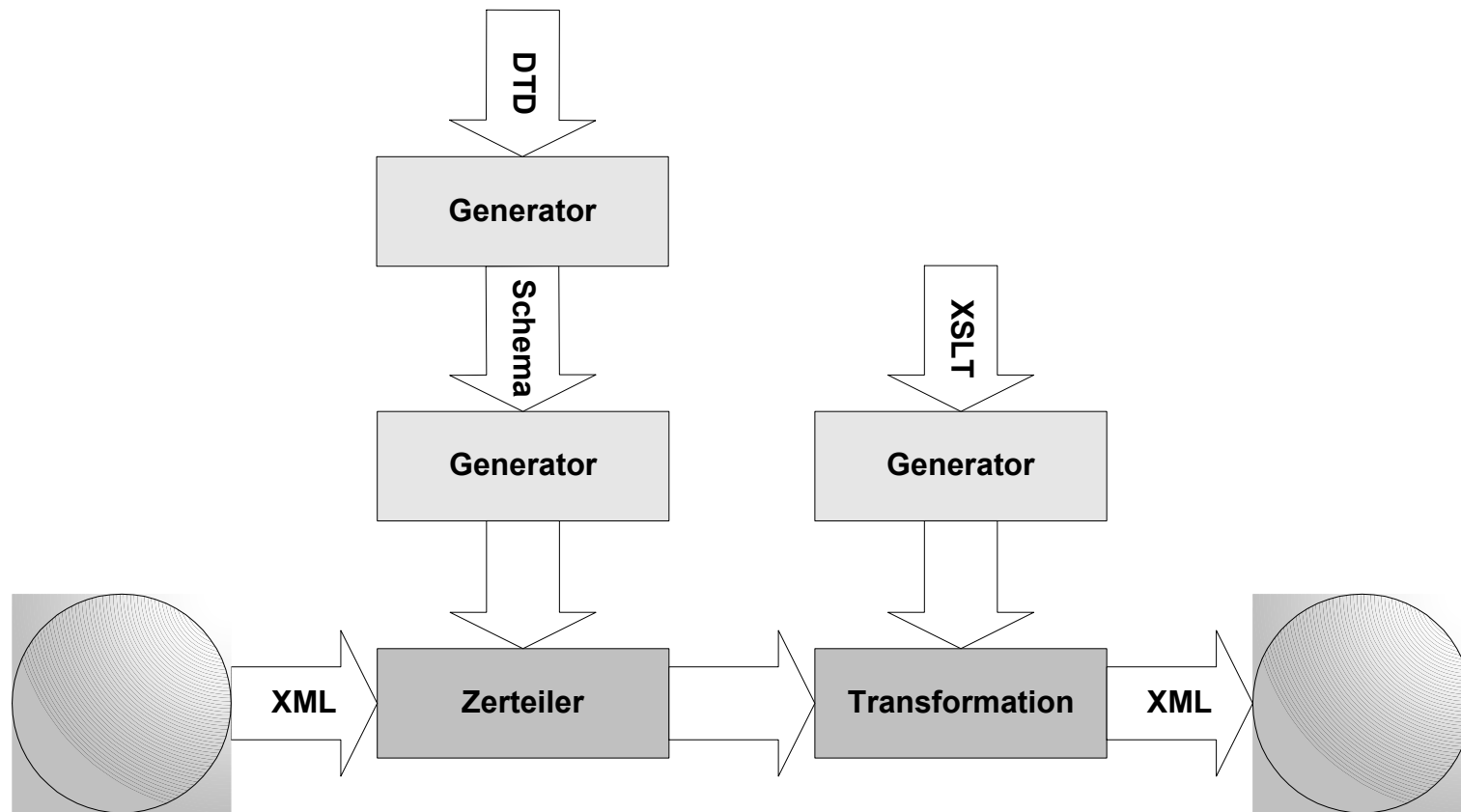
Beispiel Termersetzung für XML → HTML

```
PATTERNDEF { document < sections:section* > } MyDoc;
PATTERNDEF { section < entries:entry* > } Section;
PATTERNDEF { tr } TableRow;
PATTERNDEF { td } TableData;
PATTERNDEF { html < head body <t:table> > } HTMLRoot;
PROCEDURE transform
{
  WITH d <- MyDoc IN source DO {
    maxEntries = 0;
    WITH sec <- Section IN d.sections DO {
      maxEntries = max(maxEntries, length(sec.entries));
    }
    hroot = CREATE HTMLRoot;
    i = 0; WHILE (i < maxEntries) {
      trow = CREATE TableRow;
      APPEND trow TO hroot.t;
      WITH sec <- Section IN d.sections DO {
        APPEND CREATE TableData TO trow;
      }
      i = i + 1;
    } // while
    REPLACE d WITH hroot ADJOIN hroot.t;
  } // document iteration
}
```

- Generatoren brauchen explizite Typ- und Strukturinformation in den AST-Ecken
 - ▶ DOM hat beides nicht
 - DOM unterstellt, daß der komplette XML-Baum (alle Ecken) aufgebaut wird
 - Transformation filtert Ecken, solche Ecken müssten gar nicht erst erzeugt werden
 - ▶ Standard Übersetzerbautechnik
 - ▶ Übergang vom konkreten zum abstrakten Strukturbaum (AST)
 - XSLT ist an DOM aufgehängt, nicht an XML-Schema
- ⇒
- ▶ Übertragung der Probleme von DOM
 - ▶ Umständliche Spezifikation der Transformationen und deren Steuerung.

- Anpassungen fallen in der Entwurfsphase auf
 - ▶ DTDs/Schemata bekannt
 - ▶ \implies Transformation kann ausprogrammiert werden
- sowohl das vorhandene Format als auch das gewünschte Format sind dann fest
 - ▶ in unterschiedlichen Kontexten der Komponente unterschiedlich
 - ▶ ändern sich mit der Evolution der Komponente oder des Komponentenkontexts
 - ▶ \implies Transformation muß automatisch generiert werden

Generierte Transformationen



- aXMLerate Werkzeugkasten
 - ▶ Zerteilergeneratoren für C und Java Zerteiler (DTD und XML Schema)
 - ▶ Transformationsgenerator für Java (XSL-T)
 - ▶ Transformationsgenerator für C (Graphgrammatik)
- Laufzeiten der generierten Zerteiler in C
 - ▶ $3-4 \times 10^6$ Zeilen pro Sekunde
 - ▶ 12 MB / sec auf Intel Pentium III mit 500 MHz
 - ▶ 15 MB / sec auf Athlon mit 800 MHz

- Komponentenbauer beschreibt gelieferte Datenformate in IDL
- Automatische Transformation in XML Schema mit Hilfe von MOF
- Komponenten-Deployer beschreibt
 - ▶ geforderte Formate in IDL oder XML Schema
 - ▶ Transformation in XSLT oder durch GES-Beschreibung
- Deploymentphase
 - ▶ Erzeugung von Parsern, Zwischenstrukturen, Transformatoren
 - ▶ Optimierungen möglich
 - ▶ Entsprechender Code wird in die Stubs/Proxies/RemoteInterfaces eingewoben

- Funktionalität
- Kommunikation
- Synchronisation
- Protokolle
- Globale Korrektheit

- Sprachtransparenz: Ja
- Plattformtransparenz: Ja
- Ortstransparenz: Ja
- Reflexion: Nein
- Dienste: Wachsend, z.B. XAML (XML Transaction Authority)
- Anpassungen: Für Datenformate, sonst keine.