

III.

Entwicklung von Informationssystemen

Kapitel 3:

Programmiertechniken und Beschreibungssprachen

- 1. Algorithmen, Datenstrukturen & Design Patterns

- 2. Beschreibungssprache XML
 - 3.1 HTML und XML – wozu?
 - 3.2 XML-Syntax
 - 3.3 XML Verarbeitung
 - 3.4 XML Beispielanwendungen aus der Praxis

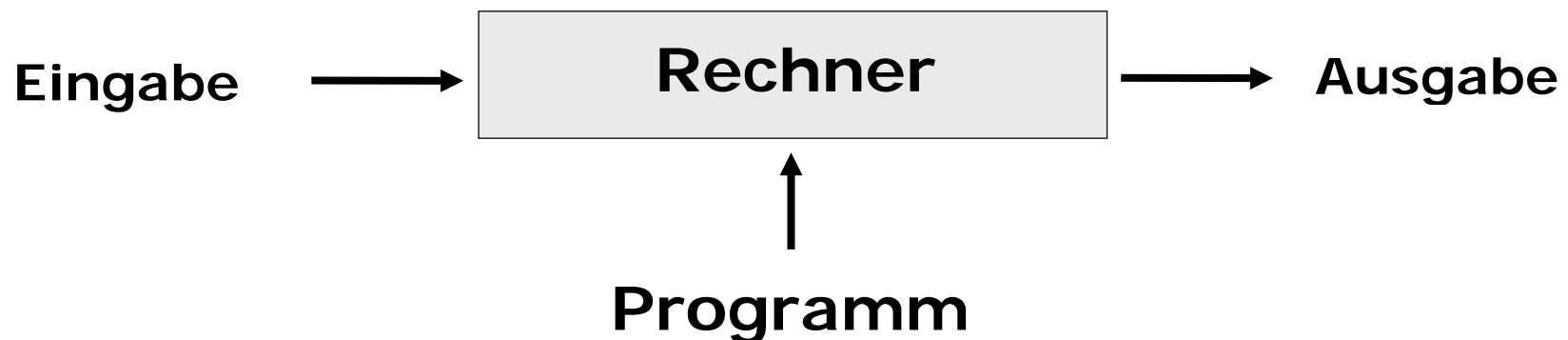
Algorithmus:

- Präzise formulierte Verarbeitungsvorschrift zur Lösung eines Problems (z.B. durch einen Computer)
- Gibt an, wie Eingabedaten schrittweise in Ausgabedaten umgewandelt werden
- Genaue und eindeutige Handlungsanweisung

- Ein Algorithmus heißt **korrekt**, wenn er genau die vorgegebene Spezifikation erfüllt, also auf alle Eingaben mit den gewünschten Ausgaben reagiert.
- **Verifikation**: Formaler Nachweis von Korrektheit eines Programmes (vgl. Testen eines Programmes für einzelne ausgewählte Eingaben)
- Tatsächlich ist der „**Beweis**“ der Richtigkeit von Programmen sehr aufwendig.
- Durch „**Testen**“ zeigt man, dass eine gewisse Art von Fehlern momentan nicht vorhanden ist.

Ablaufstruktur	Bedeutung
Sequenz	Aktivitäten werden nacheinander ausgeführt.
Alternative	Aktivitäten werden alternativ ausgeführt.
Wiederholung / Schleife	Eine Aktivität wird wiederholt ausgeführt.
Rekursion	Aktivität „ruft sich selbst auf“.

- Formulierung eines Algorithmus und der dazugehörigen Datenbereiche in einer Programmiersprache.
- Schema der Ausführung eines Programms:



Programmiersprachen

- Sprache zur Formulierung von Aktionen, die von einem Computer verstanden und ausgeführt werden können.
- Syntax und Semantik müssen eindeutig definiert sein, um die automatische Abarbeitung zu eröffnen:
 - **Syntax** legt die als Programm zulässigen Zeichenfolgen fest (Grammatik),
 - **Semantik** ist die Bedeutung der programmiersprachlichen Ausdrücke.

- Entwicklungsumgebung für Python
 - Diverse Python Installer sind auf www.python.org zu finden.
 - Aktuelle Release-Version ist 2.6.5
 - Direkter Link für Windows Clients (Stand 26.03.2010): <http://www.python.org/ftp/python/2.6.5/python-2.6.5.msi>
 - Der Installer installiert eine integrierte Entwicklungsumgebung (Interpreter & Editor) auf dem lokalen Rechner

- Alternativ:
 - Python Spielwiese der Professur für Information Systems Engineering, Uni Frankfurt
 - <http://www.ise.wiwi.uni-frankfurt.de/spielwiesen/> → Python Spielwiese

- Variablen sind benannte Platzhalter für Daten. Eine Variable wird in Python durch eine Zuweisung eingeführt:

```
a = 10          # Zuweisung; führt die Variable „a“ ein
b = 20          # Zuweisung; führt die Variable „b“ ein
c = a + b       # Zuweisung mit Ausdruck

print c        # Ausgabe von c; liefert den Wert „30“
```

- "Hello World" mit Variable:

```
text = "Hello World"
print text
```

```
1.   a = 2  
     b = 5  
     c = a * b  
     print c           # Ausgabe?
```

```
2.   b = b + 1  
     print b           # Ausgabe?
```

```
3.   a, b = 3, 9  
     a = a * b  
     print a           # Ausgabe?
```

- Python kennt drei grundsätzliche Datentypen:
 - **Strings** zur Speicherung von Zeichenfolgen
 - **Numbers** zur Speicherung von Zahlen
 - **Listen** zur Gruppierung von Werten

- Beispiele:

```
text = "Zeichenfolge"    # String
```

```
a = 7                    # Number: Integer → Ganzzahl
```

```
b = 2.0                  # Number: Float → Gleitkommazahl
```

```
L = [ 'abc', 'xy', 10, a ] # Liste von Werten
```

- Ein Array wird durch eckige Klammern definiert:

```
a = [ 1, 2, 3 ]
```

- Die Variable a enthält nun eine Liste mit den Elementen 1, 2 und 3 als Ganzzahlen.
- Der Zugriff auf die Elemente einer Liste erfolgt über Indizes. Die Elemente sind durchnummeriert, beginnend mit 0.

```
print a[0]      # gibt die Zahl 1 aus  
print a[1]      # gibt die Zahl 2 aus  
print a[2]      # gibt die Zahl 3 aus
```

- Möglich ist auch:

```
print a[-1]     # gibt ebenfalls die Zahl 3 aus
```

- Länge der Liste ermitteln

```
len (a)         # ergibt 3
```

- Listen können neben Strings und Numbers wiederum Listen enthalten:

```
a = [ ["Artikel0001", "V"], ["Artikel0002", "E"], ... ]
```

- Zugriff auf Listenelemente:

```
a[0]          # ["Artikel0001", "V"]
```

```
a[0][0]       # "Artikel0001"
```

```
a[1][1]       # "E"
```

```
a[0][0][0]    # "A" ( Strings werden als Liste von  
                Zeichen interpretiert )
```

- Boolesche Variablen sind nach dem englischen Mathematiker **George Boole** benannt, der den Grundstein für die formale Logik und die Rechentechnik legte.

- Ein boolescher Wert ist entweder **True** oder **False**.

- Konnektoren:

and, **or** und **not**

- Operatoren:

== ist gleich?

!= ist ungleich?

>(=) ist größer (gleich)?

<(=) ist kleiner (gleich)?

- In Python können zwei Schleifentypen genutzt werden:

while <Bedingung>:
 <zu wiederholende Sequenz>

for <element> **in** <liste>:
 <Sequenz pro Element>

- Beispiel:

```
a = 1
while a < 5:
    print a
    a = a + 1
```

- Berechnet eine Fibonacci-Folge mittels einer Schleife:

```
a, b = 0, 1
```

```
while b < 1000:
```

```
    print b
```

```
    a, b = b, a+b
```

```
# mehrfache Zuweisung
```

```
# Wiederholung
```

```
# Ausgabe
```

```
# Folge "weiterdenken"
```

- Ausgabe:

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

- Eine **FOR-Schleife** durchläuft eine Liste und führt für jedes Element einer Liste den eingeschlossenen Programmcode aus.

- Beispiel:

```
a = [ 98.90, 49.99, 225.50 ]      # Liste
for x in a:                       # Für jedes x in (der Liste) a
    print "Netto:", round(x, 2), # Ausgabe
    print "Brutto:", round(x * 1.16, 2)
```

- Ausgabe:

```
Netto:    98.9  Brutto: 114.72
Netto:    49.99 Brutto:  57.99
Netto:   225.5  Brutto: 261.58
```

- Problemstellung: Eine bestimmte Aktivität soll n-mal ausgeführt werden.
- Solche Zähl-Schleifen können in Python ebenfalls elegant durch eine FOR-Schleife umgesetzt werden.
- Die FOR-Schleife arbeitet nur auf Listen, folglich muss diese erst generiert werden.
- Dies erledigt die Funktion: `range()`
- Beispiel:

`range(10)` liefert folgende Sequenz als Liste:

0 1 2 3 4 5 6 7 8 9

```
for i in range(10):  
    # mach was
```

- `a = 0`
 while `a < 10`:
 `print a`
 `a = a + 1`
- **for** `a in range(10)`:
 `print a`
- **for** `a in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`:
 `print a`

- Über die Ausführung alternativer Programmcodes wird durch die Sprachkonstrukte IF, ELIF und ELSE entschieden. Dabei werden Bedingungen überprüft, die entweder wahr (True) oder falsch (False) sind.
- Beispiel:

x = 1

```
if x < 0:
    print "Negativ"
elif x == 0:
    print "Null"
else:
    print "Mindestens Eins"
```

The diagram illustrates the logical components of the code blocks:

- Bedingung**: Points to the condition `x < 0` in the `if` statement.
- Alternative mit Bedingung**: Points to the condition `x == 0` in the `elif` statement.
- Alternative**: Points to the `else` statement.

- Der Befehl "print" erzeugt eine Ausgabe. Beispiel:

```
print "Hallo Welt"           # Hallo Welt
```

- Mehrfache Ausgabe durch Kommata:

```
print "Hallo", "Welt"       # Hallo Welt
```

- Jedes Komma erzwingt ein Leerzeichen. Vermeidung der Leerzeichen durch *Konkatenation* möglich.

```
print "Hallo" + "Welt"     # HalloWelt
```

```
print 7 + " Zwerge"       # Fehler
```

```
print str(7) + " Zwerge"   # 7 Zwerge
```

```
def helloWorld(x):
```

```
    y = 1
```

```
    while y <= x:
```

```
        print "helloWorld"
```

```
        x = x - 1
```

Funktionskopf

Funktionsrumpf
Eingerückte Zeilen
gehören zur Funktion

- ```
def summe(x,y):
```

```
 return x+y
```

*Funktionsparameter*

*Rückgabewert der Funktion*

- ```
print summe(3,5)
```

Funktionsaufruf

Berechnung der Fakultät

▪ Iterativ

```
• def fakultaet(n):  
    ergebnis = 1  
    faktor = 1  
    while faktor <= n:  
        ergebnis = ergebnis * faktor  
        faktor = faktor + 1  
    print ergebnis
```

▪ Rekursiv

```
• def fakultaet(n):  
    if n <= 1:  
        return 1  
    else:  
        return n * fakultaet(n - 1) # Selbstreferenz
```

Vorteile der jeweiligen Ansätze

- Iterativ
 - Hohe Effizienz (hinsichtl. Rechnerleistung & Speicherbedarf)
- Rekursiv:
 - Kompaktere Darstellung des Algorithmus bei komplexeren Problemstellungen

- Formel:
 - $n > 1$: $fak(n) = n * fak(n-1)$
 - $n = 0, 1$: $fak(n) = 1$
- Beispiel für $fak(5)$

fak(5) =

$$5 * fak(4) =$$

$$4 * fak(3) =$$

$$3 * fak(2) =$$

$$2 * fak(1) =$$

$$1 * fak(0) =$$

120

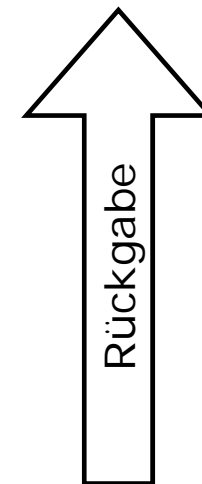
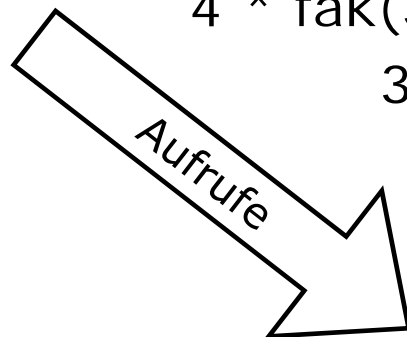
$$5 * 24$$

$$4 * 6$$

$$3 * 2$$

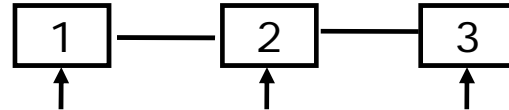
$$2 * 1$$

$$1 * 1$$



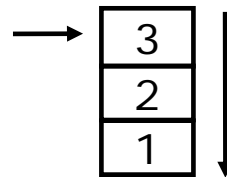
- Datenstrukturen dienen Speicherung, Zugriff sowie Manipulation von Daten innerhalb einer Anwendung
- Verschiedene grundlegende Datenstrukturen
 - List
 - o Liste beliebiger Elemente
 - o Jedes Element besitzt einen Verweis auf das nächste Element (Verkettung).
 - Stack
 - o Stapel beliebiger Elemente
 - o Der Zugriff auf die Elemente erfolgt nach dem LIFO Prinzip.
 - Queue
 - o Warteschlange, die beliebige Elemente aufnehmen kann
 - o Es gilt das FIFO-Prinzip.
 - Graph
 - o Graph, der beliebige Elemente aufnehmen kann
 - o Knoten können uni-direktional oder bi-direktional auf andere Knoten verweisen.
 - Tree
 - o Spezieller Graph (Baum) mit einem Wurzelknoten, mit dem weitere Knoten verkettet werden

- List

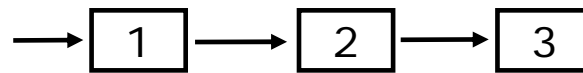


Die Pfeile zeigen die möglichen Zugriffspunkte auf die Elemente.

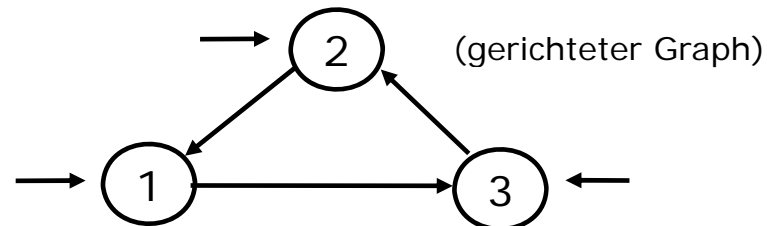
- Stack



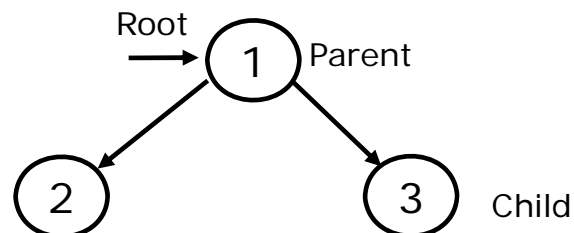
- Queue



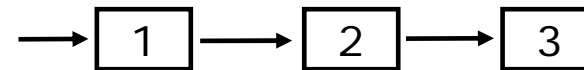
- Graph



- Tree



- Sequentielle Suche eines Elements in einer **Queue**

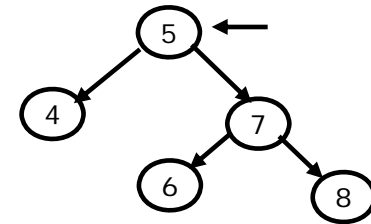


- x: Erstes Element der Queue
s: Gesuchter Wert eines Elements

- def search(x, s): # Funktionsdeklaration
 while x != null:
 if key(x) == s:
 print x # Ergebnis anzeigen
 else:
 x = next(x) # zum nächsten Element

- Die Funktion **next(x)** liefert das nächste Element in der Queue.
- Die Funktion **key(x)** liefert den Inhalt des Elements x.

- Binary-Search-Tree-Algorithmus
 - Baum, bei dem die Kind-Knoten links kleiner und rechts größer sind als der aktuelle Knoten
 - Suche nach einem bestimmten Knoten im Baum



- x : Wurzel des Search-Trees
 s : Gesuchter Wert eines Elements

- ```
def search(x, s):
```

# Funktionsdeklaration  

```
if x == null or s == key(x):
```

# Ergebnis-Knoten anzeigen  

```
print x
```

```
elif s < key(x):
```

# im linken Teilbaum weitersuchen  

```
search(left(x), s)
```

```
else:
```

# im rechten Teilbaum weitersuchen  

```
search(right(x), s)
```

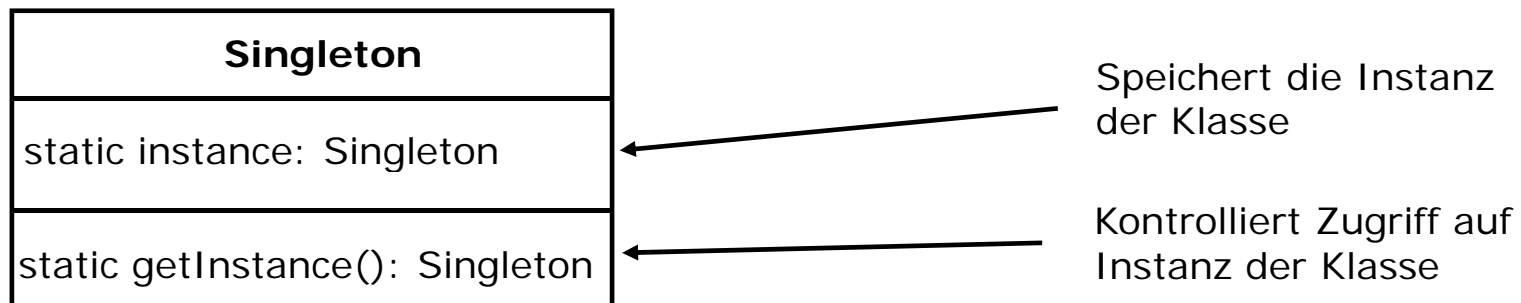
- Die Funktionen **left(x)** bzw. **right(x)** liefern jeweils das linke bzw. rechte Kind eines Knotens.
- Die Funktion **key(x)** liefert den Inhalt des Knoten  $x$ .

- Entwurfsmuster für die Lösung bekannter Probleme im Software Engineering
  - Im Software Engineering bewährtes Konzept
  - Wiederverwendbar für verwandte Probleme
  - Basieren ggf. auf anderen Entwurfsmustern
  - Unabhängig von einer Programmiersprache spezifiziert
  - **Idiome**, als programmierspezifische Implementierung von Entwurfsaspekten

- Architectural patterns
  - Entwurfsmuster für Architekturprobleme (z.B. Client-Server Konzept)
- Fundamental patterns
  - Grundlage für andere Entwurfsmuster (z.B. Kontrollstrukturen für den Zugriff auf Objekte)
- Creational pattern
  - Kontrollierte Erzeugung von Objekten (z.B. Sicherstellung, dass ein Objekt nur einmal erzeugt wird)
- Structural patterns
  - Umsetzung von Beziehungen zwischen Objekten (z.B. **Container**, zur Speicherung und Verwaltung von Objekten)
- Behavioral patterns
  - Ermöglicht eine gemeinsame Kommunikation zwischen Objekten (z.B. **Mediator**, für eine einheitliche Schnittstelle zu unterschiedlichen Objekten)
- Concurrency patterns
  - Steuerung gemeinsamem Zugriffs auf ein Objekt (z.B. **Read/Write Lock**, erlaubt gemeinsamen Lesezugriff, aber exklusiven Schreibzugriff auf ein Objekt)
- ...

- Singleton-Pattern (Creational Pattern)
  - Kontrolliert, dass eine Instanz einer Klasse nur einmal erzeugt wird
  - Anwendungsszenario
    - Ein Programm soll nur einmal auf einem PC gestartet werden.

- UML Klassennotation



- Funktion
  - Nutzer einer bestimmten Klasse können von dieser Klasse keine Instanzen selbst erzeugen, sondern müssen über die Singleton-Klasse eine Referenz auf die Instanz anfordern.

- 1. Algorithmen, Datenstrukturen & Design Patterns
- 2. Beschreibungssprache XML
  - 3.1 HTML und XML – wozu?
  - 3.2 XML-Syntax
  - 3.3 XML Verarbeitung
  - 3.4 XML Beispielanwendungen aus der Praxis

- HTML ist eine sehr einfache Beschreibungssprache:
  - Kaum semantische Beschreibung der Inhalte (nur Unterteilung in Abschnitte, Überschriften, Listen, etc.)
  - Dies macht automatische Verarbeitung von Webinhalten oft unmöglich.
- Z.B. woran erkennt man automatisch die Postadresse auf einer Webseite?

<H1>Lehrstuhl für M-Commerce und Mehrseitige Sicherheit</H1>

<P>Gräfstraße 78</P>

<H2>60054 Frankfurt am Main</H2>

- Beim Erzeugen von HTML-Seiten aus Datenbanken ist erheblicher Aufwand nötig (Skripte, Formatierung, Darstellung).
- Dadurch redundante Datenhaltung, Gefahr von Inkonsistenzen
- HTML eignet sich nicht zur Datenhaltung, da die Sprache sich nicht eignet, Daten zu beschreiben.
- Bei HTML hat sich unsaubere Syntax etabliert – Elemente werden oft nicht abgeschlossen:
  - `<LI>Erta Ale is a shield volcano, part of the East African Rift system.</LI>`
- Es besteht also Bedarf an einer Beschreibung von Daten, welche diese Mängel beseitigt.

## Entwicklung von Metasprachen zur Datenbeschreibung



GML: Generalized Meta Language von IBM

SGML: Standard Generalized Meta Language als Norm ISO 8879 für Datenaustausch und -speicherung

HTML: Def. der Version 2 als SGML-Dialekt

XML: Verbindet HTML mit dem Anspruch von SGML:  
**Extensible Markup Language**

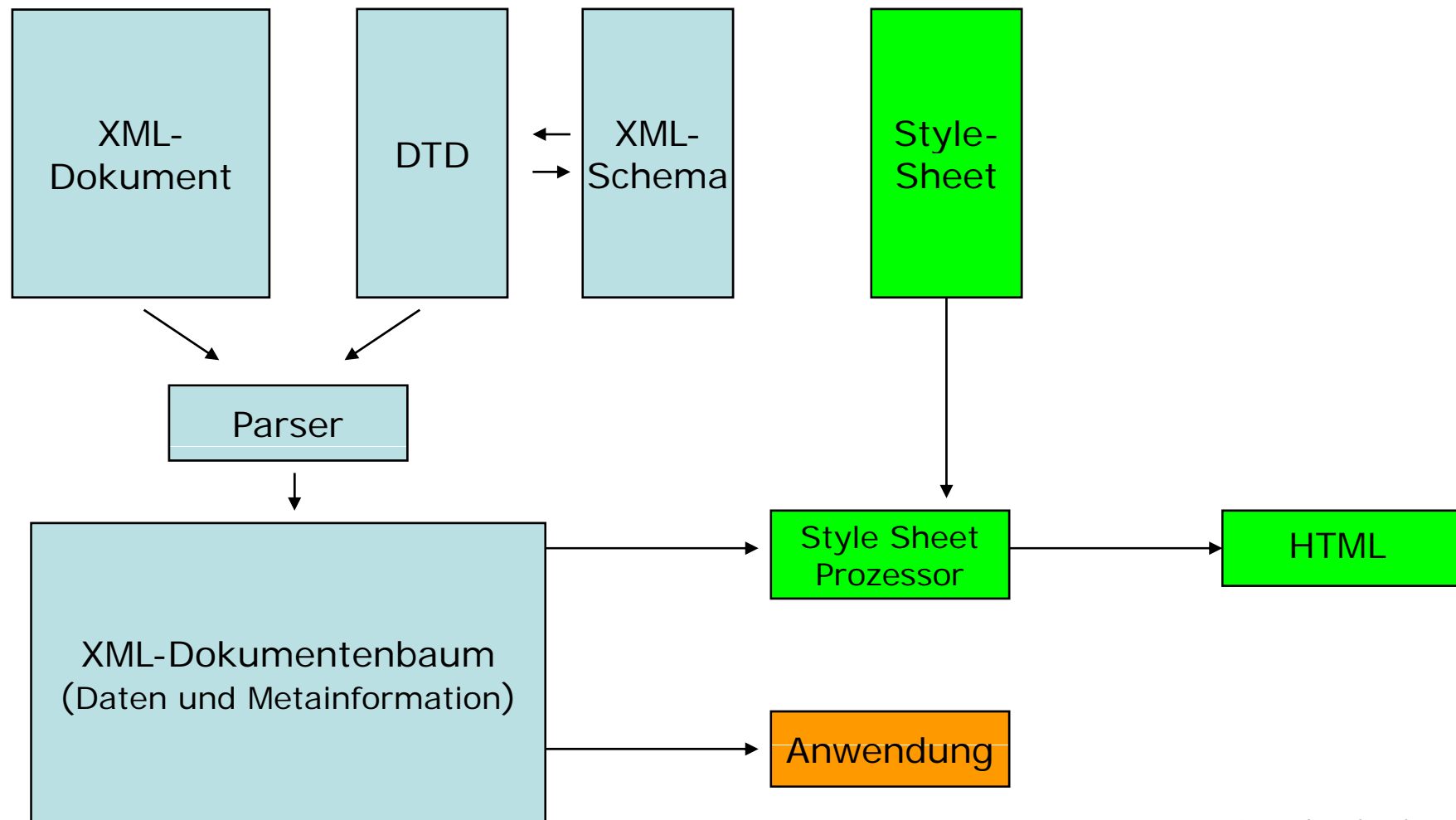
XHTML: HTML formuliert auf Basis von XML

- Grundidee aller SGML-Sprachen
  - Ermöglichen die Computerverarbeitung von Dokumenten durch Meta-Information über deren Struktur und Inhalt
  - System- und herstellerunabhängiger Standard
  - Trennung von Struktur, Inhalt und Darstellung eines Dokuments
  
- Weitere bekannte SGML-Dialekte
  - LaTeX
  - Postscript

## Merkmale von XML

- Einfache und menschenlesbare Syntax (nicht binär)
- Standardisiert
- Selbstbeschreibend durch enthaltene Meta-Beschreibung
- Erweiterbar durch neue Elementbeschreibungen -> anwendungsspezifische Datenmodelle
- Eignet sich zur Datenhaltung

- **DTD**  
Document Type Definition – beschreibt die formale Struktur eines Dokuments
- **XML Schema**  
Alternativer Ansatz mit Erweiterungen der DTD
- **Parser**  
Übersetzt XML-Dokument in einen Dokumentenbaum
- **Style Sheet**  
Layout-Vorgaben zur Ausgabe von Dokumenten
- **Style-Sheet-Prozessor**  
Setzt die Style-Vorgaben um und generiert die Ausgabeseiten

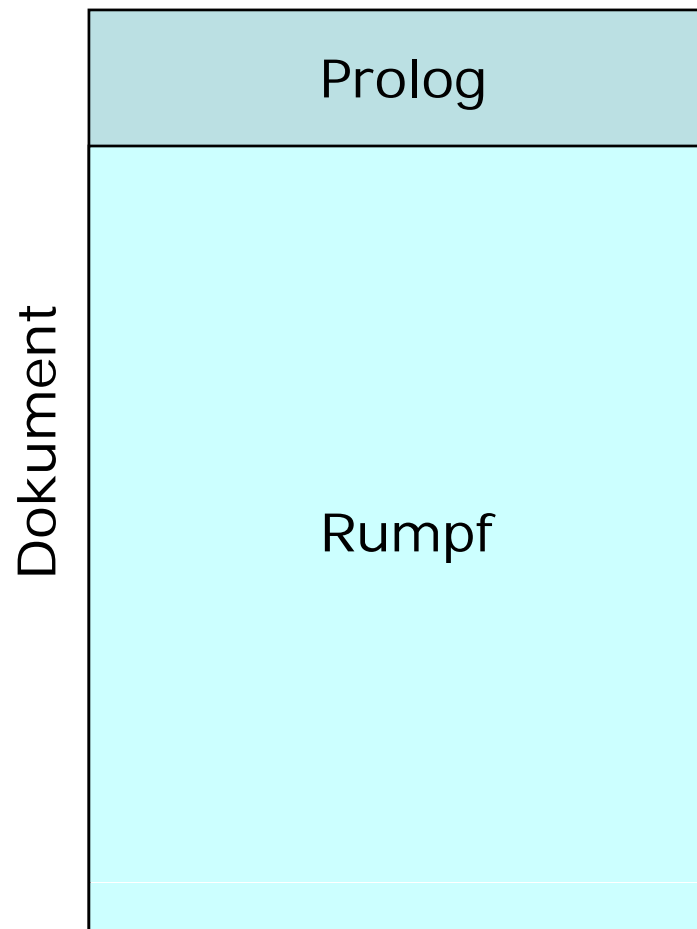


Legende auf nächster Seite

## XML-Sprachen für Branchen und Anwendungen:

- OFX, OFE: Open Financial Exchange für Finanzinformationen ([www.ifxforum.org](http://www.ifxforum.org))
- MathML: Mathematische Formelbeschreibung per XML ([www.w3.org/Math](http://www.w3.org/Math))
- SAML: Security Assertion Markup Language for exchanging authentication and authorization information ([www.oasis-open.org](http://www.oasis-open.org))
- EPAL: Enterprise Privacy Authorization Language is a formal language to specify fine-grained enterprise privacy policies (<http://www.zurich.ibm.com/security/enterprise-privacy/epal/>)

- 1. Algorithmen, Datenstrukturen & Design Patterns
- 2. Beschreibungssprache XML
  - 3.1 HTML und XML – wozu?
  - 3.2 XML-Syntax
  - 3.3 XML Verarbeitung
  - 3.4 XML Beispielanwendungen aus der Praxis



Prolog enthält XML-Version und Angaben zum verwendeten Zeichensatz.

Rumpf enthält die Daten.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

Prolog

```
<flirt>
 <name>Daisy</name>
 <mobilfon>+436508469249</mobilfon>
 <email>daisy@m-lehrstuhl.de</email>
 <stadt>Innsbruck</stadt>
 <erstkontakt>20.6.2006</erstkontakt>
 <letztkontakt>14.7.2007</letztkontakt>
 <geburtstag>12.7.1978</geburtstag>
 <vegetarier>nein</vegetarier>
 <gebunden>nein</gebunden>
</flirt>
```

Rumpf

- XML erwartet abgeschlossene Elemente!
  - `<name>` ist ein „tag“ (engl. für Marke, Markierung)
  - Syntax: `<StartTag>Inhalt</EndTag>`
  - `<name>Daisy</name>`
  
- Attribute werden im Starttag angegeben:
  - `<stadt wohnsitz=„erster“>Innsbruck</stadt>`
  
- Ein „Datensatz“ wie der auf der Vorseite mit `<stadt>` markierte heißt „Document element“.
  - Elemente können geschachtelt auftreten.
  - Elemente dürfen sich nicht überlappen!

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```
<flirt>
```

```
 <name>Daisy</name>
```

```
 <telefon>
```

```
 <mobilfon> +436508469249</mobilfon>
```

```
 <zuhaus> +431324087356</zuhaus>
```

```
 <arbeit> +498974448696</arbeit>
```

```
 </telefon>
```

```
 <email>daisy@m-lehrstuhl.de</email>
```

```
 <stadt>Innsbruck</stadt>
```

```
 <kontakte>
```

```
 <erstkontakt anlass="Peters Party">20.6.2006</erstkontakt>
```

```
 <letztkontakt anlass="Valentinstag">14.7.2007</letztkontakt>
```

```
 </kontakte>
```

```
 <geburtstag>12.7.1978</geburtstag>
```

```
 <vegetarier>nein</vegetarier>
```

```
 <gebunden>nein</gebunden>
```

```
</flirt>
```

Schachtelung

- Unerlaubte Schachtelung:

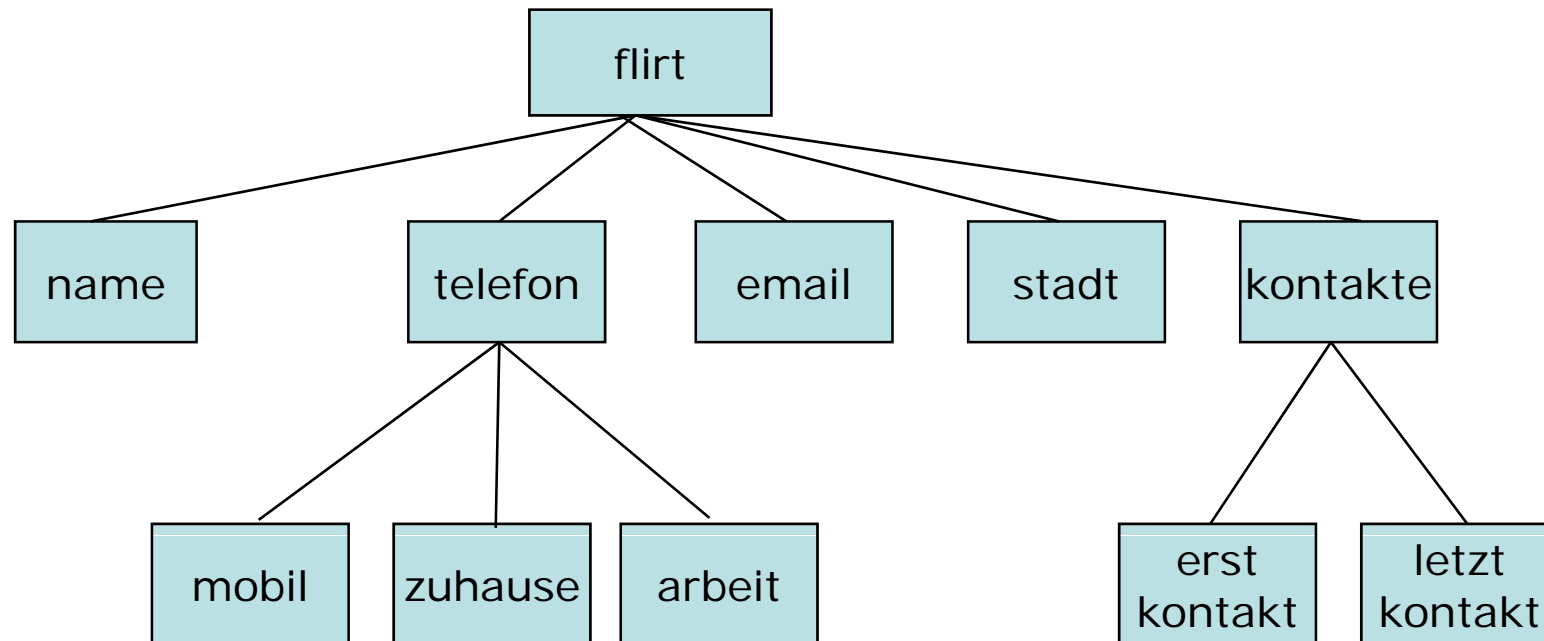
```
<telefon>
 <mobil> +4916008154711
 <zuhause> +497213848855
 </mobil> </zuhause>
</telefon>
```

- Hier ist nicht mehr erkennbar, was zu <mobil> und was zu <zuhause> gehört!

- Es gibt wie bei HTML einige Zeichen, die zur Syntaxfeststellung genutzt werden:

Zeichen	Schreibweise
<	&lt;
>	&gt;
&	&amp;
,	&apos;
„	&quot;

- Man kann XML-Dokumente auch als "Objekte" einer objektorientierten Datenbank oder als Baum betrachten:



- Daher der Name "document tree".

- Wegen der eindeutigen, baumartigen Struktur und der Ähnlichkeit zu objektorientierten Systemen erkennt ein Rechner beim Einlesen von XML die Datenstruktur eindeutig.
- Ein „**wohlgeformtes Dokument**“ in XML ist ein syntaktisch korrektes Dokument mit abgeschlossenen, nicht überlappenden Tags und einheitlicher Groß- und Kleinschreibung.

- Eine Document Type Definition (DTD) beschreibt Struktur und Grammatik von XML-Dokumenten.
- Ist vergleichbar mit einer Variablen- / Typendeklaration in einer Programmiersprache
- Es wird definiert, welche Werte in Elementen vorkommen dürfen, damit ein „**gültiges**“ **XML-Dokument** entsteht.
- DTD „übersetzt“ also ein XML-Dokument in Datentypen für Datenbanken und erzeugt Regeln für Elemente.

- Element-Content:

EMPTY	leeres Element
ANY	beliebiger Inhalt
	Auswahlliste
,	Sequenz
()	Gruppierung
(#PCDATA)	Zeichen- oder Stringdaten

- Kardinalität:

	leer: genau ein Wert nötig
+	mindestens ein Wert
?	Null oder ein Wert
*	Null oder mehr Werte

- Deklaration der Elementregeln in einer DTD:

<!ELEMENT flirt	(name,telefon,email,stadt,kontakte, geburtstag,gebunden) >	
<!ELEMENT name	(&#PCDATA) >	← Text
<!ELEMENT telefon	(mobil   zuhause   arbeit) + >	
<!ELEMENT mobil	(&#PCDATA) >	↙ Auswahlliste
<!ELEMENT zuhause	(&#PCDATA) >	
<!ELEMENT email	(&#PCDATA) >	
<!ELEMENT stadt	(&#PCDATA) >	
<!ELEMENT kontakte	(&#PCDATA) >	
<!ELEMENT geburtstag	(&#PCDATA) >	
<!ELEMENT gebunden	(&#PCDATA) >	

- **Neues Problem**

Wie vermeidet man Verwechslungen bei Daten aus mehreren Quellen, die gleichlautende Tag-Namen haben?

```
<BUCH>
 <TITEL>Computernetzwerke</TITEL>
 ...
</BUCH>

<Autor>
 <TITEL>Professor</TITEL>
 ...
</Autor>
```

- **Lösung**

Man definiert sich einen Namespace mit einem Universal Resource Identifier (URI), welcher mittels eines globalen Pfades eine Identifikation des Elementes erlaubt. Dazu erzeugt man einen Präfix vor dem Element-Namen:

```
<BUCH
 xmlns:book="http://www.amazonen.de/namespaces/books"
 xmlns:aut="http://www.amazonen.de/namespaces/authors"
>
 <book:TITEL>Computernetzwerke</book:TITEL>
</BUCH>
```

- URIs werden z.B. auch in Java zur global eindeutigen Klassenidentifikation genutzt.

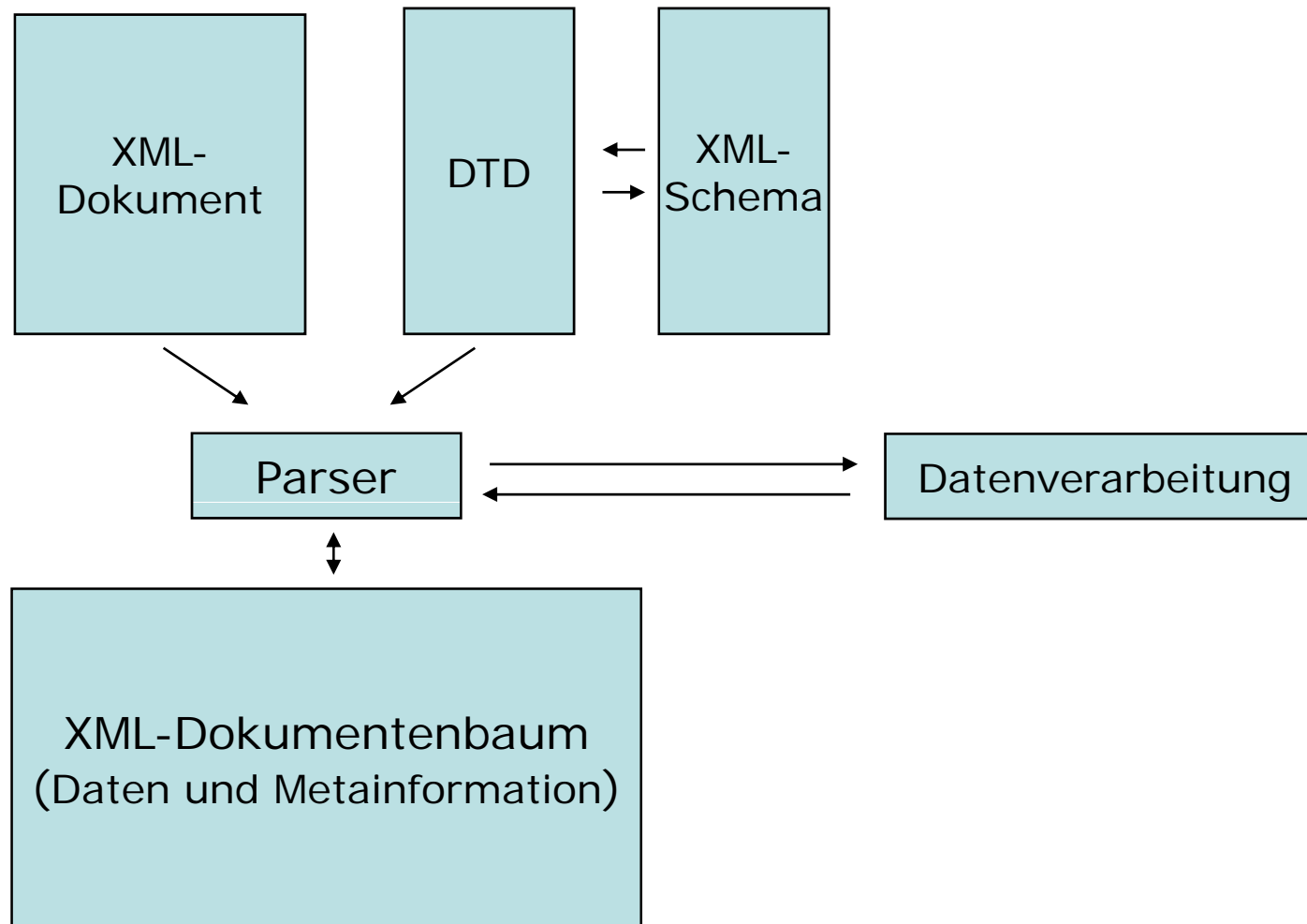
- Vorsicht: Eine URI ist oftmals nur eine eindeutige Herkunftsangabe – unter der Adresse gibt es in der Regel kein Dokument!
- Man kann einen Default Namespace definieren:  
`<BUCH xmlns="http://www.amazonen.de/">`

- Alternativ zu DTD gibt es „XML Schema“.
- Schema beseitigt einige Schwächen der DTD:
  - Bessere Content-Modellierung zur Syntaxüberprüfung
  - Reihenfolge und Schachtelung festlegbar
  - Festlegbare Wertegrenzen von Werten
  - Prüfung der Datentypen der Elementinhalte
  - Bessere Definition der Kardinalitäten mit Min. und Max.
  - Ausführlichere Datentypenauswahl analog zu Programmiersprachen und Datenbanken (z.B. boolean, number, double, datetime, ...)

- Named Types:
  - Abgeleitete Datentypen basierend auf vorhandenen Typen.
  - Können beispielsweise Restriktionen definieren
  
- Ziel der Schemas:
  - Bessere Feststellung der „Gültigkeit“ eines Dokuments

- 1. Algorithmen, Datenstrukturen & Design Patterns
- 2. Beschreibungssprache XML
  - 3.1 HTML und XML – wozu?
  - 3.2 XML-Syntax
  - 3.3 XML Verarbeitung
  - 3.4 XML Beispielanwendungen aus der Praxis

- Zur Verarbeitung benötigt man einen Parser.
- Ein Parser ist eine Software, welche DTDs, Schemas und XML-Dokumente einlesen kann, um dann einer Anwendung Zugriffe auf alle Elemente zu ermöglichen.
- Übliches Vorgehen:
  - Anwendung öffnet XML-Datensatz.
  - Parser liest XML-Dokument und nötige DTDs, Schemas.
  - Parser bietet Anwendung Schnittstellen mit Funktionen wie "ElementeAuflisten()".
  - Anwendung durchsucht mittels der Schnittstellen das Dokument und bearbeitet die Elemente.
  - Anwendung speichert den überarbeiteten XML-Datensatz ab.
- Ergebnis  
Ein syntaktisch fehlerfreies und gegen eine DTD geprüftes XML-Dokument wird als "gültig" bzw. "valid" bezeichnet.



- Es gibt zwei verschiedene Arten von Parsern:
  - Document Object Model (DOM)
  - Simple API for XML (SAX)
  
- DOM-Parser laden alle Elemente in den Speicher und erzeugen eine Baum-Datenstruktur, auf der dann gearbeitet wird.
  
- SAX-Parser navigieren sich lesend durch ein Dokument, ohne es komplett im Speicher abzulegen.

## Vergleich DOM- und SAX-Parser

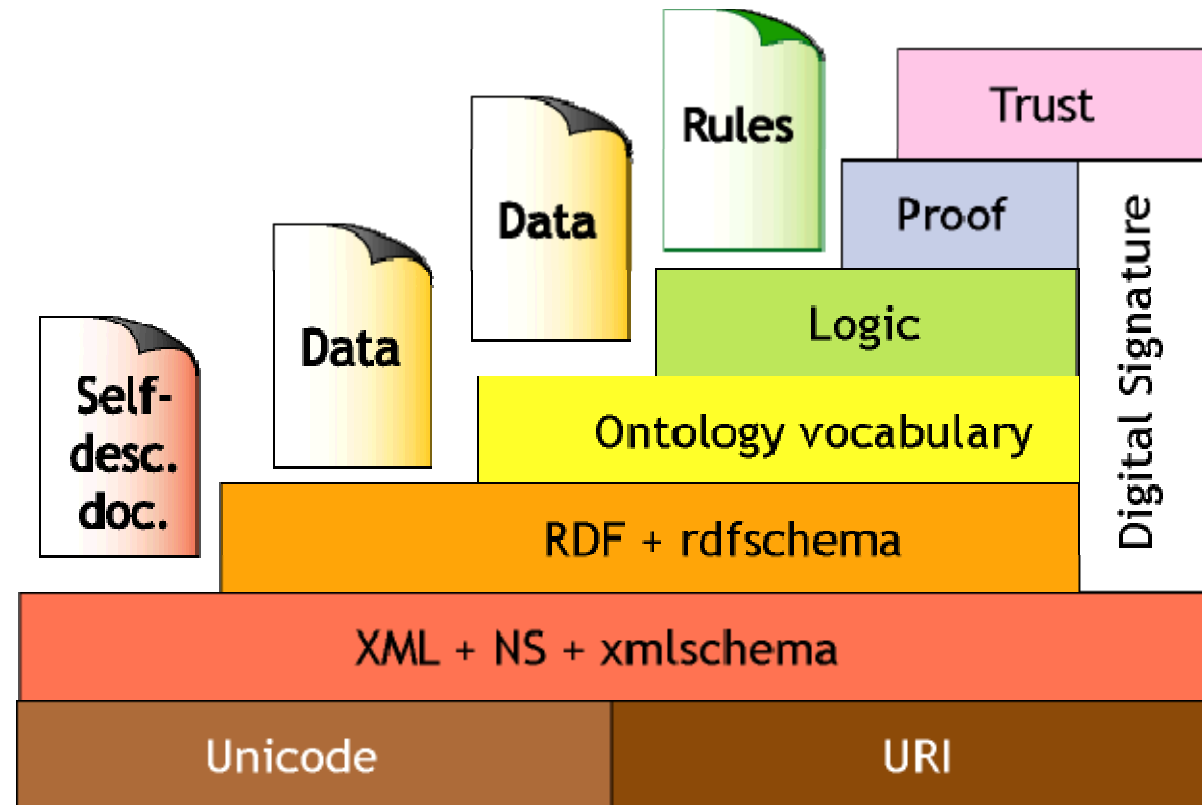
- SAX kann Dateien beliebiger Größe parsen.
- SAX ist effizient, wenn nur bestimmte Teile interessant sind.
- SAX hat einfaches Handling.
  
- DOM erlaubt freie Zugriffe und Änderungen am Dokument.
- DOM erzeugt eine vollständige Abbildung des Dokuments.

## Einsatzszenarien von DOM und SAX

- DOM-Parser eignen sich gut beim Bearbeiten gesamter Dokumente, z.B. zur Bearbeitung eines gegliederten Textes in einer Textverarbeitung.
- SAX-Parser eignen sich zum schnellen Auffinden von Datensätzen, z.B. der Adressen in einer XML-basierten Kundendatenbank.

- 1. Algorithmen, Datenstrukturen & Design Patterns
- 2. Beschreibungssprache XML
  - 3.1 HTML und XML – wozu?
  - 3.2 XML-Syntax
  - 3.3 XML Verarbeitung
  - 3.4 XML Beispielanwendungen aus der Praxis

- Semantic Web Stack

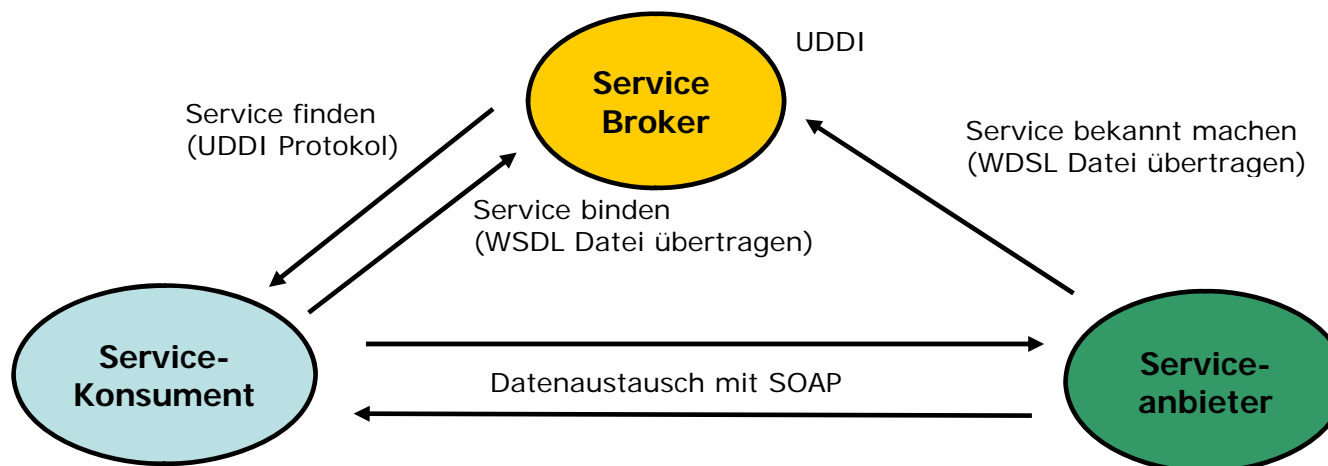


Quelle: Tim Berners-Lee (W3C), <http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html>

- The term *Web services* describes a standardized way of integrating Web-based applications using the XML, SOAP, WSDL and UDDI open standards over an Internet protocol backbone.

- XML is used to tag the data,
- SOAP is used to transfer the data,
- WSDL is used for describing the services available and
- UDDI is used for listing what services are available.

Quelle:  
[www.webopedia.com/TERM/W/Web\\_services.html](http://www.webopedia.com/TERM/W/Web_services.html)



- OpenDocument-Format
  - Entwickelt auf Basis des OpenOffice/StarOffice Dateiformats
  - OpenDocument 1.0 als ISO Standard verabschiedet
  
- Open XML
  - Von Microsoft entwickelt
  - European Computer Manufacturers Association Standard
  - ISO Standard ISO/IEC 29500:2008
  - Anwendung in Microsoft Office 2007

- Programmiersprache Python,  
Internet: [www.python.org](http://www.python.org)
- Tim Berners-Lee (2000), W3C Talk,  
Internet: [www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html](http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html)
- Webopedia,  
Internet: [www.webopedia.com/TERM/W/Web\\_services.html](http://www.webopedia.com/TERM/W/Web_services.html)