Oktober 2012

WENN XML ZUR FALLE WIRD

Wann XML verwendet werden sollte und wann besser nicht

Veröffentlicht im:

Dr. Jürgen Lampe
Agon Solutions

Abstract

Die breite Anwendbarkeit von XML und der darauf aufbauenden Techniken und Werkzeuge kann leicht die Sicht auf besser angepasste Lösungen für spezifische Einsatzfälle verstellen. Deshalb ist es wichtig, vor jedem Einsatz gründlich zu prüfen, ob XML wirklich eine passende und entwicklungsfähige Basis ist. Wird diese Prüfung unterlassen, kann es passieren, dass man sich nach einigen Jahren der Entwicklung in einer Situation befindet, in der XML die Ursache wesentlicher Probleme ist, aber mit vertretbarem Aufwand nicht mehr ersetzt werden kann - dass man in der XML-Falle sitzt.

Es ist unbestritten, dass der auf XML basierende Technologiestack stabil, etabliert und vielfach bewährt ist. Man darf aber nicht vergessen, dass universelle Werkzeuge im Einzelfall nie so effizient sind wie spezielle. Niemand würde ernsthaft vorschlagen, ein Schwein mit dem vielzitierten Schweizer Offiziersmesser zu zerlegen - außer in der Notsituation für die das Messer gedacht ist.

Hier geht es darum, wesentliche Anwendungseigenschaften herauszuheben, die für die kritische Bewertung eines XML-Einsatzes wichtig sind. Dabei können wir uns auf praktische Erfahrungen aus verschiedenen Projekten stützen, bei denen diese Technologie in einem fortgeschritten Einsatzstadium zu unerwarteten Problemen geführt hat. Trotz der Konzentration auf das Java-Umfeld sind die besprochenen Probleme und Überlegungen, bis auf wenige Ausnahmen, unabhängig von den jeweiligen Laufzeitumgebungen.

Der Ausgangpunkt

Seit 1998 existiert XML als Standard. In den seither vergangenen knapp 15 Jahren wurde praktisch die ganze IT durchdrungen. Eine Google-Suche lieferte vor einiger Zeit rund 950.000.000 Ergebnisse, mehr als dreimal so viel wie für die Datenbank-Abfragesprache SQL (299.000.000). Dabei kann sich das, was unter den drei Buchstaben XML verstanden wird, je nach Sicht deutlich unterscheiden. Angetrieben durch den Erfolg der Sprache HTML und des World Wide Web hat XML einen Hype erlebt, der noch nicht völlig abgeklungen ist.

Mittlerweile bleiben kritische Stimmen aber nicht mehr unbeachtet, so dass es an der Zeit ist, diese Entwicklung nüchtern zu analysieren und zu sehen, welche dauerhaften Folgen sie für die IT gebracht hat. Es zeigt sich schnell, dass XML ein sehr vielschichtiges und komplexes Phänomen ist, dessen umfassende Darstellung diesen Beitrag sprengen würde. Deshalb sollen lediglich einige Aspekte herausgegriffen werden, deren Diskussion als Denkanstoß dienen kann. Nicht verzichtet werden kann auf einen kurzen Rückblick, denn ohne den ganz konkreten historischen Kontext ist die Entwicklung von und rund um XML nicht zu verstehen.

Der Erfolg der Sprache HTML hatte den Auszeichnungssprachen eine breite Öffentlichkeit gebracht. HTML ist eine von SGML (Standard Generalized Markup Language), einem bis dahin nur wenigen Spezialisten bekannten Formalismus zur Dokumentenbeschreibung, abgeleitete Sprache, oder genauer eine Anwendung von SGML. SGML wurde 1986 von der ISO standardisiert, geht aber auf Entwicklungen zurück, die bei IBM bereits in den 1960er Jahren begannen.

SGML ist wesentlich durch den historischen Entwicklungskontext geprägt worden. Der war lange durch eine Trennung zwischen wissenschaftlicher und wirtschaftlicher Datenverarbeitung gekennzeichnet. Auch wenn diese Trennung inhaltlich nicht sonderlich scharf war, hatte sie einen erheblichen Einfluss, weil die unterschiedlichen Begriffswelten einen intensiven Austausch verhinderten und zu zahlreichen Parallelentwicklungen (unter verschiedenen Namen natürlich) führten.

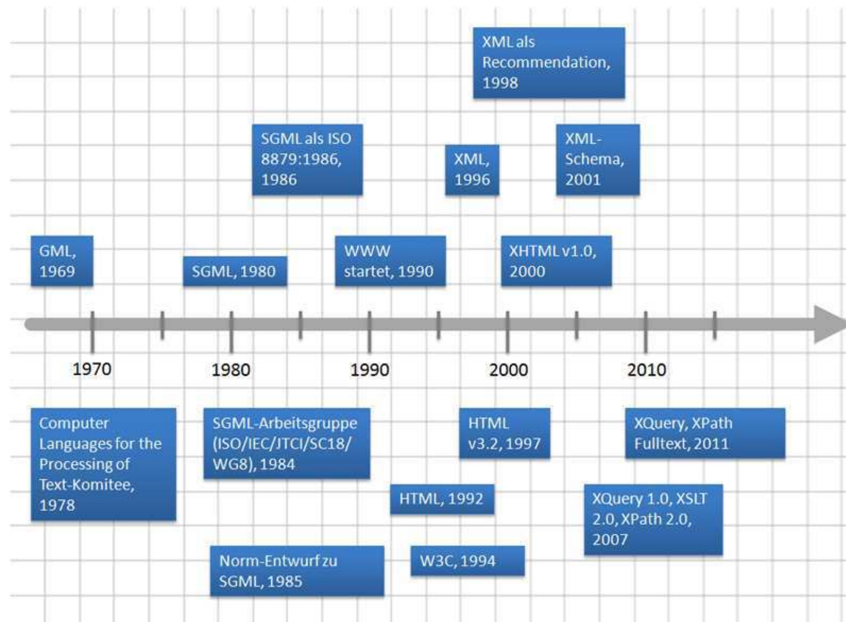

Abbildung 1: Zeitstrahl

Die Grundidee, nämlich die logische Struktur eines Dokuments durch Markups auszuzeichnen, wurde bereits Anfang der 1960er Jahre gelegt. Neben der Geräteunabhängigkeit der Darstellung spielten auch Fehlertoleranz (Paritätsbits waren damals selbstverständlich) und Langzeitspeicherung eine große Rolle. Automatische Verarbeitung im heutigen Sinn, insbesondere De-/ Serialisierung stand weniger im Fokus.
Man muss sich die Frage stellen, ob SGML wirklich eine gute Basis für die Entwicklung eines Formalismus mit derart breit gefächertem Anspruch war. Aus heutiger Sicht spricht sehr viel für eine negative Antwort.
Wenn es überhaupt Vorteile gab, dann für die Definition, Akzeptanz und Verbreitung. Es ist immer ein gutes Argument, wenn man bei der Präsentation einer Neuentwicklung darauf verweisen kann, dass man sich im Rahmen eines internationalen Standards bewegt. Und wahrscheinlich hat der Bezug auf die Vorlage auch manche Diskussionen unter den Protagonisten verkürzt oder ganz überflüssig gemacht.
Nachdem HTML so erfolgreich etabliert war, wurde bei Beginn der Entwicklung von XML – trotz der ganz anderen Zielsetzung - diese Basis nie ernsthaft in Frage gestellt
Abgesehen davon, dass XML, ähnlich wie das Internet, die Lösung fast aller Probleme sein sollte, gab es auch einige ambitionierte aber heute beinahe vergessene Ziele. Eine der Visionen, die zwar bemerkenswerte Ergebnisse hervorgebracht hat, selbst aber verfehlt wurde, war die Ablösung der unscharfen Trennung von Daten und Darstellung in HTML, durch die Kombination von XML und XSL. Von Anfang an zeigte sich dabei das Verwischen

der Grenzen zwischen dem, was XML ist, und der Ziele, die man damit erreichen kann. Letztlich ging es darum, eine möglichst allgemeine Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten zu definieren. Im Unterschied zu den Vorgängern SGML/HTML und als ein Zeichen der namensgebenden Erweiterbarkeit sollte es dabei keine vordefinierten Elementnamen geben.

Der XML-Mythos

Ohne den Mythos hätte XML schwerlich eine derart atemberaubende Entwicklung vollziehen können. Für die Entstehung eines Mythos ist es wichtig, dass fehlendes Wissen durch Glauben ersetzt wird. Dafür gab es keinen besseren Zeitpunkt als den WWW-Hype (bzw. Dotcom-Boom). Plötzlich wurden so viele Sachen möglich, die vorher undenkbar erschienen, dass es sogar kritischen Geistern schwer fiel, zuverlässig zwischen Wunsch und Wirklichkeit zu unterscheiden. Mit HTML konnten selbst IT-Laien einiges zuwege bringen. Es wurden aber auch Mängel und Grenzen dieser Auszeichnungssprache sichtbar. Gleichzeitig trat der Mangel an standardisierten Datenaustauschformaten immer deutlicher zutage. In dieser Situation wurde XML als willkommene Lösung präsentiert. Und begeistert empfangen.
Es gibt aus jener Zeit kaum einen Artikel über eine neue Web-Anwendungstechnik, der nicht wenigstens im letzten Absatz darauf verweist, dass sich in Zukunft durch den Einsatz von XML noch ganz andere Möglichkeiten ergeben würden.[1] Um die Jahrtausendwende war scheinbar jeder bemüht, sein Projekt irgendwie mit der magischen Abkürzung XML in Verbindung zu bringen. Die Frage war nur, ob es durch XML zu formalisieren ist (geht eigentlich immer) und nicht, ob das auch sinnvoll ist.

Die wichtigsten Zutaten für den Mythos waren dabei:

- der **richtige Zeitpunkt**, ein Faktor den man gar nicht überschätzen kann
- ein **geschickt** gewählter **Name**, etwas was eXtensible ist, muss einfach besser und flexibler sein
- die **einfache Lösung** für fast alle Probleme, ganz egal, ob Objektorientierung, Entwurfsmuster oder NoSQL - wir hoffen immer wieder, dass es doch eine einfache Lösung für die vielen hässlichen kleinen und großen Probleme gibt, mit denen wir uns jeden Tag herumschlagen
- genügend **unbestimmt**, da im Allgemeinen nicht so genau zwischen der Auszeichnungssprache und den darauf aufbauenden Technologien unterschieden wird, konnten auch der Technologie nicht so nah stehende Manager mitreden, ohne

[1] Dazu ein Zitat von M. Jeckle aus dem Jahr 2004:
Das Akronym XML ist derzeit in aller Munde... Es hat Einzug in fast jede Produktankündigung jüngerer Zeit gefunden, und keine neuere IT-Entwicklung, die nicht XML enabled wäre.
Es scheint, dass sich anhand XML dieselbe übersteigerte (und nicht notwendigerweise immer mit wahrheitsgemäßen Versprechungen arbeitende) Marketing-Euphorie (engl. hype) vollzieht, die bereits für das Schlagwort Objektorientierung zu beobachten war.
Die fachlichen Einschätzungen des Themas XML reichen von ASCII des 21 Jahrhunderts (H. S. Thomson) bis hin zur (berechtigten) kritischen Hinterfragung des Neuheitswertes, insbesondere im Vergleich zu bekannten Lösungen wie troff, LaTeX oder das Rich Text Format (RTF). [1]

Gefahr zu laufen sich zu blamieren (Umgekehrt hat das natürlich auch funktioniert. Der Autor erinnert sich deutlich an den Eindruck, den man mit dem Verweis auf XML-Erfahrungen in Bewerbungsgesprächen machen konnte.)

- **standardisiert**, zunächst konnte auf den seit 1986 vorliegenden ISO-Standard für SGML verwiesen werden, dann wurde XML selbst sehr schnell als ISO-Standard zertifiziert und - schon vorher und zumindest damals noch beeindruckender - zum Web-Standard erklärt
- bedient auch all die magischen Eigenschaften: **geräteunabhängig**, **herstellerunabhängig**, **lizenzfrei**, **unicode**-basiert (nicht, dass diese unwichtig wären, aber für ein ernst zu nehmendes Austauschformat sind das triviale Voraussetzungen)

XML in der Realität

Wenn man heute von XML spricht, ist häufig gar nicht die Auszeichnungssprache selbst gemeint, sondern ein ganzes Bündel von Sprachen und Technologien, die eng damit verknüpft und teilweise parallel entstanden sind. Wir wollen uns aber zunächst auf den Kern, die Sprache XML konzentrieren.

In vergleichsweise sehr kurzer Zeit wurde ein Standard erstellt. XML-Strukturen sind einfach zu verstehen und dank der Textrepräsentation und im Unterschied zu zahlreichen älteren Datenaustauschformaten relativ leicht zu lesen (solange man die Unicode-Unterstützung nicht extensiv ausnutzt).

Ein weiterer oft genannter Vorteil ist die strikte Syntax – Weglassungen sind nicht erlaubt – wodurch ein einfacheres Parsen möglich ist. Dieses Argument ist allerdings nur verständlich, wenn man es mit Bezug auf HTML sieht.

Außerdem kann man die Struktur zulässiger XML-Dokumente definieren, sodass in gewissem Rahmen eine Validierung möglich ist. Zunächst gab es dafür ausschließlich die Dokument-Typ-Definition (DTD), die allerdings selbst kein XML ist.

XML hat von SGML und dessen Vorgängern einen Ansatz übernommen, der Dokumente als eigenen Objektbereich sieht, ohne große Gemeinsamkeiten mit den Forschungsgegenständen der Informatik. Das ist sachlich falsch und hat bedauerlicherweise weitreichende Konsequenzen.

Tatsächlich handelt es sich um eine Anwendung formaler Sprachen, deren Theorie bereits in den 1960/70er Jahren umfassend etabliert wurde und sich in Compilern und Datenanalysesystemen bewährt hat. Es gab Untersuchungen zu effizienten Parsingmethoden und dazu, wie man Sprachen so definieren kann, dass sie durch solche effizienten Methoden analysierbar werden. Und es gab und gibt hinreichend Tools, mit deren Hilfe Parser zuverlässig und schnell aus formalisierten Grammatiken erzeugt werden können. Die XML-Protagonisten scheinen das zum großen Teil nicht gut gekannt oder aus schwer nachvollziehbaren Gründen bewusst ignoriert zu haben.

Der Rückgriff auf den vorhandenen Fundus an Wissen hätte nicht nur dazu geführt, dass hoffentlich einiges besser gemacht worden wäre, sondern vieles wäre wohl auch wesentlich schneller gegangen.

> **Anmerkung:** Darauf, dass sich an dieser Ignoranz leider wenig geändert hat, deutet das folgende Zitat aus einer Bewertung von HTML5 hin: *Anscheinend haben die Autoren der Spezifikation den Zusammenhang zwischen DTDs, formalen Sprachen und endlichen Automaten vergessen. In einem endlosen Kapitel 8 breiten sich die Parsing-Regeln ... aus. ... Hier wird das Rad neu erfunden.* [2]

So ist XML von seiner Entstehung her eine Dokumentenauszeichnungssprache. Primäres Ziel war die Beschreibung der logischen Struktur von Dokumenten, unabhängig von ihrer Darstellung.

Gleichzeitig bestand ein großer Bedarf an einer leicht austauschbaren Datenstrukturbeschreibung. Aus abstrakter Sicht besteht zwischen Dokumenten und Datenstrukturen kein wesentlicher Unterschied. Deshalb kann XML für beides eingesetzt werden.

Unberücksichtigt bleibt dabei aber der pragmatische Aspekt. Als Datenaustauschformat ist XML deutlich weniger gut geeignet.[2]

Man könnte sagen, XML ist für die Datenstrukturen das, was Assembler für die Programmabläufe ist: eine sehr allgemeine und damit mächtige, aber auch elementare und (durch Unübersichtlichkeit) schwierig zu handhabende Beschreibung. Es gibt jedoch einen sehr großen Unterschied: Assembler führen (i. d. R.) zu redundanzarmen kompakten Programmen, bei XML ist eher das Gegenteil der Fall.

Trotz der weiten Verbreitung hat XML nicht alle Ziele erreicht und Versprechen nicht gehalten oder nicht halten können. Wobei zur Ehrenrettung der wirklich Beteiligten sagen muss, dass manche publizierten Versprechen aus dafür gar nicht autorisierten Quellen kamen.

Unter den genannten Voraussetzungen ist es nicht überraschend, dass XML vorrangig die Forderungen erfüllt, die für die Dokumentenstrukturauszeichnung wesentlich sind. Die Eigenschaften, die von einem Austauschformat, das innerhalb automatischer Verarbeitungsprozesse eingesetzt wird, erwartet wird, sind hingegen deutlich weniger ausgeprägt. Im Folgenden werden einige wichtige Eigenschaften detaillierter betrachtet.

XML-Aspekte

Aspekt 1: Trennung von Inhalt und Darstellung

Ein Ziel der XML-Definition war die bereits in HTML rudimentär angelegt Trennung von Inhalt und Darstellung konsequent zu Ende zu führen. Ein XML-Dokument sollte so völlig

[2] Man kann das Verhältnis grob mit dem zwischen Datenbanken und Content-Management-Systemen vergleichen, die auf abstrakter Ebene ebenfalls sehr viel gemeinsam haben, konkret aber unterschiedlich implementiert werden.

unabhängig vom Darstellungsmedium (im weitesten Sinne) sein. Beispielsweise sollte eine Menge von Daten sowohl in einem Browser als Tabelle oder Grafik darstellbar sein, als auch einem automatischen Handelssystem als Entscheidungsgrundlage dienen können.
Dieser Ansatz hat sich grundsätzlich bewährt. Er funktioniert überall dort, wo es sich bei den Daten um einen semantisch hinreichend gut verstandenen Bereich handelt. Tatsächlich treten die größten Probleme dabei auf, ein gemeinsames Verständnis der verwendeten Begriffe zu erreichen. Das ist jedoch eine Frage, die vom verwendeten Formalismus weitgehend unabhängig ist. XML hat dazu beigetragen, dass sie stärker bewusst geworden ist, was sich u. A. in der Forschung in einem größeren Interesse an Ontologien oder den Arbeiten am sogenannten *Semantic Web* niederschlägt und in die Entwicklung von HTML5 Eingang gefunden hat.

Aspekt 2: Unabhängige Linkverwaltung

Unter den mit XML verbundenen Hype-Themen um 2000 war eines die kommende Möglichkeit, Verweise (Links) unabhängig von den betroffenen Dokumenten, was die Quelle, das Ziel oder beides betrifft, anlegen zu können. Zu diesem Zweck wurde XLink entwickelt und als Empfehlung des WWW-Consortiums (W3C) verabschiedet. Xlinks können in beliebige Elemente als Attribute des Namensraums *http://www.w3.org/1999/xlink* eingebaut werden.

Beispiel:

```
<seite xmlns:xlink="http://www.w3.org/1999/xlink"
          xlink:title="Inhaltsverzeichnis"
          xlink:label="inhalt"
          xlink:type="simple"
          xlink:href="http://beispiel.de/inhaltsvz"> …
</seite>
```

Dies definiert einen Link auf das Inhaltsverzeichnis, auf diesen Link kann über seinen Namen „inhalt“ dann in anderen Verweisen Bezug genommen werden.

Praktisch spielt diese durchaus interessante Technik bis heute aber keine große Rolle, was nicht zuletzt daran liegt, dass sie von den meisten Browsern gar nicht oder nur sehr eingeschränkt unterstützt wird.

Aspekt 3: Dokumenten-Transformation

Die Idee, Daten als XML zu übertragen und dann im Browser mittels (unterschiedlicher) XSL-Dokumente für die Darstellung aufzubereiten, hat sich in dieser Form als Standardtechnik nicht durchgesetzt, obwohl beispielsweise der MS InternetExplorer nach wie vor einen XSLT-Prozessor enthält. Für diesen Misserfolg sind aus Sicht des Autors verschiedene Gründe verantwortlich:

- XSLT ist zwar ein leistungsfähiges Transformationssystem, ihm fehlen aber wichtige Funktionen, z. B. die zum Rechnen mit Zahlen, was nicht nur zum Skalieren eigentlich unverzichtbar ist. Man findet im Internet zwar Beispiele, wie man in XSLT sehr wohl „rechnen“ kann, aber praktisch ist das hoffnungslos umständlich.
- XSLT arbeitet regelbasiert, d.h. nicht prozedural. Alle Erfahrungen mit regelbasierten Programmiersprachen zeigen, dass es einem großen Teil der Anwender offenbar sehr schwer fällt, regelbasiert zu denken, bzw. sich von der erlernten und geübten vorgehensbasierten (prozeduralen) Arbeitsweise zu trennen. Bei vielen XSLT-Dokumenten handelt es sich dann auch um mehr oder weniger verkappte Anweisungsfolgen.
- Nicht zuletzt hatten auch die im Weiteren betrachteten Verarbeitungsprobleme ihre Wirkung.

Abgesehen davon ist der XSL-Technik durchaus Erfolg beschieden gewesen. Insbesondere wenn es darum geht, aus Daten Dokumente wie Rechnungen oder Berichte – auch in unterschiedlichen Formaten - zu erstellen, sind die auf XSL aufbauenden Technologien nicht mehr wegzudenken.

Aspekt 4: Objekt-Serialisierung

Für die Serialisierung/Deserialisierung von nicht nur Java-Objekten hat sich XML inzwischen als Standardformat durchgesetzt. Beispielsweise ist die JAXB-Bibliothek seit Version 6 Teil der Java-Distribution. Es gibt zudem stabile und bewährte alternative Implementationen für alle wichtigen Programmierumgebungen.
Für diesen Erfolg sind jedoch weniger die Qualitäten von XML verantwortlich als vielmehr das Fehlen ernsthafter Konkurrenten. Andere Werkzeuge verwenden proprietäre Formate und sind entweder stark an bestimmte Programmiersprachen gebunden, z. B. Pickle an Python, oder Teil kommerzieller Pakete. Über System- bzw. Programmiersprachengrenzen hinaus verwendbare Formate bieten JSON und YAML. Während JSON nicht alle möglichen Objektstrukturen darstellen kann und die Typisierung schwierig ist, fehlt YAML schlicht die Publizität.

Aspekt 5: Werkzeuge

Der Erfolg von XML ist eigentlich ein Erfolg der verfügbaren Werkzeuge. Ohne die schnelle Unterstützung durch zum großen Teil freie Software, wäre die Durchdringung der verschiedenen IT-Bereiche kaum möglich gewesen. Beim genaueren Hinsehen zeigt sich aber auch, dass oft Schnelligkeit vor Gründlichkeit gegangen ist. Für Programme ist das kein Problem, weil man Fehler und Ineffizienzen durch Updates beheben kann. Wie noch gezeigt wird, haben sich nachteilige Festlegungen jedoch auch in API eingeschlichen, wo man sie später wesentlich schwerer korrigieren kann.
Da auf der Basis der vorhandenen Werkzeuge relativ schnell und problemlos prototypische Lösungen realisiert werden können, tragen sie entscheidend zur Verbreitung dieser

Technologie bei. Bei allen kritischen Bemerkungen soll nicht bestritten werden, dass XML und diese Werkzeuge an vielen (aber nicht allen) Stellen helfen, Software effektiv und schnell zu produzieren.
Allerdings scheint es, als ob die dynamische Phase der Werkzeug-Entwicklung seit einigen Jahren vorbei ist. Die Technologie ist stabil und etabliert. Es gibt wenig neue Anwendungen oder über die obligatorische Pflege hinausgehende Innovationen. Das große Interesse der Web-Gemeinde ist weitergewandert zu HTML5.

Problembereiche

XML ist ohne Zweifel eine bewährte Standardtechnologie. Das heißt nicht, dass keine Mängel zu konstatieren wären. Im Folgenden sollen einige der Punkte genauer betrachtet werden, die für Probleme verantwortlich sein können. Dabei soll nicht vergessen werden, dass eine Einordnung als Vor- oder Nachteil immer von der jeweiligen Sicht abhängt. Entscheidend sind die Umstände des Einsatzes. In diesem Sinn handelt es sich eher um Denkanstöße als um Bewertungen – letztere sind dem Leser in seiner konkreten Situation vorbehalten.

Problem 1: Wahlfreiheit

Die Flexibilität von XML stellt den Anwender gleich zu Beginn vor Fragen, deren Konsequenzen nicht leicht überschaubar sind. Das beginnt mit der Unterscheidung zwischen wohlgeformten und gültigen Dokumenten. Ein XML-Dokument, das elementaren Regeln genügt, wird als wohlgeformt bezeichnet.
Ein wohlgeformtes Dokument ist gültig, wenn es darüber hinaus eine DTD (direkt oder als Referenz) enthält und den darin definierten Regeln entspricht. Für den Anwender gibt es keine festen Regeln, welche Form er benutzen sollte, nur Ratschläge wie diesen:

> *Müssen Sie immer ein gültiges XML-Dokument mit einer DTD schreiben oder reicht ein wohlgeformtes Dokument? Die Antwort ist uneindeutig: Es hängt von Ihren Bedürfnissen ab. … Dadurch, dass man auf die DTD verzichtet, bleibt der XML-Code schön übersichtlich. Und wenn Sie mal auf die Schnelle ein kleines XML-Dokument erstellen wollen, ist das wohlgeformte XML ohne DTD immer einfacher programmiert.* [3]

Was kaum beachtet wird, ist die Auswirkung auf die Verarbeitung. Die Prüfung auf Gültigkeit erfordert, dass nicht nur das XML-Dokument eingelesen und analysiert wird, sondern auch die DTD (bzw. XSD). Das schafft zwar eine größere Sicherheit gegen falsch strukturierte Dokumente und kann besonders in Editoren hilfreich sein, verursacht aber einen entsprechend größeren Verarbeitungsaufwand. Darüber hinaus sollte man nicht übersehen, dass die häufig verwendete Referenzierung externer DTD ein Tor für Angriffe durch Manipulation dieser DTD öffnet, wenn keine zusätzlichen Vorkehrungen getroffen werden.
Prinzipiell ist es auch möglich, aus der Strukturbeschreibung (DTD/XSD) dedizierte Parser zu generieren. Offensichtlich rechtfertigt der dadurch mögliche Performancegewinn den

Aufwand nicht. In Ansätzen findet sich diese Lösung lediglich beim Generieren von Web-Services.
Eine andere Frage, die immer wieder kontrovers diskutiert wird, ist die nach Regeln, wann etwas als Attribut und wann als Element definiert werden soll. Der häufig aufgeführte Vorschlag, dass Attribute primär für Meta-Daten und Elemente für die eigentlichen Nutz-Daten verwendet werden sollen, hat für Dokumente durchaus Sinn. Bei reinen Datenstrukturen, wie beispielsweise serialisierten Objekten, ist die Unterscheidung aber sehr sophistisch. In der Praxis führt das bisweilen dazu, dass ausschließlich Elemente verwendet werden, da diese das allgemeinere Konstrukt sind, während es für Attribute und deren mögliche Werte verschiedene Einschränkungen (aber auch genauere Wert-Spezifikationen) gibt.

Statt relativ kurz

```
<property name="ID" value="22" />
```

muss dann

```
<property>
      <name>ID</name>
      <value>22</value>
</property>
```

geschrieben werden. Das ist wesentlich schlechter lesbar, insbesondere wenn man an eine ganze Liste solcher Properties denkt.
Derartige Überlegungen haben wohl auch dazu beigetragen, dass z. B. Tomcat ab der Version 5.5 in den Konfigurationsdateien von der reinen Elementstruktur auf die Verwendung von Attributen umgeschwenkt ist. Die Lesbarkeit dieser Dateien hat dadurch erheblich gewonnen und die Konfiguration ist (zumindest gefühlt) einfacher geworden, weil man auf einen Blick mehr Informationen erhält als vorher.
Das Attribut-Element-Dilemma ist letztlich Ausdruck der unterschiedlichen Anwendungskontexte. XML ist primär für die Strukturierung von Dokumenten entworfen worden. Als Datenaustauschformat ist es nur bedingt geeignet.
Die syntaktische Armut eröffnet gleichzeitig vielfältige Möglichkeiten, ein und denselben Sachverhalt auf viele verschiedene Weisen darzustellen.

Das erscheint zunächst als großer Vorteil, hat auf die Dauer aber u.a. die Nachteile:

- Es ist schwierig, *Coding-Standards* für XML-Strukturen zu etablieren. Die Tatsache, dass man XML benutzen kann, ohne vorher eine Struktur definieren und damit durchdenken zu müssen, begünstigt voreilige Entscheidungen, die oft nicht mehr korrigierbar sind.
- Es ist nicht möglich, den Inhalt von Elementen zu beschränken (Längenbeschränkung, zulässige Zeichen usw.). Das gilt auch bei Verwendung einer DTD, nur XSD bringt diesbezüglich eine Verbesserung.

Problem 2: Geschwätzigkeit (Verbosity)

Die Geschwätzigkeit von XML ist beabsichtigt. Sie soll die Lesbarkeit für Menschen verbessern. Gleichzeitig erhöht diese Redundanz die Robustheit gegen Datenfehler. Der letzte Punkt ist für die Langzeitspeicherung von Dokumenten wichtig und nur dafür gültig, denn die durch XML gegebene Redundanz ist auf die Strukturauszeichnung beschränkt. Für Text-Dokumente im weiteren Sinn, d. h. den ursprünglichen SGML-Objektbereich, ist diese Einschränkung akzeptabel, weil natürlich-sprachliche Texte selbst höchst redundant sind. Für die Daten einer serialisierten Objektrepräsentation gilt das nicht. Dort besteht eine deutliche Asymmetrie zwischen der redundanten Strukturinformation und der wenig redundanten inhaltlichen Information.
Dem scheinbaren Vorteil der Fehlertoleranz gegenüber wurde dem erhöhten Platz- bzw. Bandbreitenbedarf keine wesentliche Bedeutung zuerkannt, da einerseits Speicherplatz immer billiger wird und andererseits XML-Daten sehr gut komprimiert werden können.
XML-Dokumente sind deutlich umfangreicher als in anderen vergleichbar ausdrucksstarken Formaten angelegte Dokumente. Wenn Daten aber komprimiert, d. h. nicht direkt lesbar abgelegt werden, warum dann im XML-Format und nicht in einem besser angepassten Format? Wenn sie nicht komprimiert werden, entsteht ein Mehraufwand, der von der Anzahl der Tags und der Länge von Element- und Attributnamen abhängig ist und bis auf fast 50 Prozent steigen kann.

Niemand kommt auf die Idee, Programme als XML-Strukturen zu schreiben (Beispiel 1). Die Texte wären viel länger und entgegen den XML zugeschriebenen Eigenschaften auch schlechter lesbar. Das führt zur Frage: „Warum soll, was für Programme gilt, nicht auch für die Datenbeschreibung zutreffen?“
Gute Ingenieurarbeit zeichnet sich durch Sparsamkeit aus, und zwar nicht nur in Bezug auf die Kosten, sondern auf alle eingesetzten Mittel. Im Software-Engineering ist dieser Grundsatz etwas aus dem Fokus geraten. Daran ist sicher die schnelle Entwicklung der Hardware Schuld, die zu einer einseitigen Konzentration auf die Entwicklungskosten geführt hat.
Es ist absehbar, dass, wie in allen reiferen Technologien, das Streben nach sparsamen und einfachen Lösungen zunehmen wird. Vermutlich wird in Zukunft die Suche nach speicher- und Prozessorzyklen-sparenden Formaten und Methoden in der Softwareentwicklung eine ähnliche Rolle einnehmen, wie sie heute z. B. die Suche nach hochfesten und leichten Materialien im Fahrzeug- oder Flugzeugbau hat.
In der Unix-Programmierung hat das Prinzip Sparsamkeit schon lange eine große Bedeutung. In den letzten Jahren findet sich der Begriff auch wieder häufiger in Blogs oder Kolumnen, denn Sparsamkeit ist eng verknüpft mit Einfachheit. Je komplexer die Anforderungen werden, desto wichtiger wird Einfachheit als Designprinzip. [4] Das Thema wird uns nicht verlassen.

Beispiel 1 – Java in XML

Nichts spricht grundsätzlich dagegen, ein Java-Programm als XML-Dokument zu schreiben:

```
    <class scope="public" name="Beispiel" package="de.xml.bsp">
          <extends class="de.xml.bsp.Base" />
          <implementing>
                <implements name=java.util.List" />
          </implementing>
          <statics> ...
    </class>
```

wäre denkbar für

```
package de.xml.bsp;
import java.util.List;
class Beispiel extends Base implements List {
...
}
```

Wem das obige Beispiel noch nicht abschreckend genug erscheint, analoge Datenstrukturen werden auch so beschrieben:

```
<class>
<properties|>
          <property>
                <name>name</name>
                <value>Beispiel</value>
          </property>
          <property>
                <name>scope</name>
                <value>public</value>
          </property> ...
    </class>
```

Problem 3: Performance

Wenn es darum geht, XML als Austauschformat zu verwenden, wird auf die gute Komprimierbarkeit verwiesen. Die ist zwar tatsächlich gegeben und beweist die Redundanz, aber Komprimieren und Dekomprimieren sind nicht kostenlos. Beides erfordert Prozessorleistung. Trotz aller Fortschritte ist Prozessorleistung in großen Systemen unverändert eine knappe Ressource. Nebenbei, Prozessoroperationen verbrauchen Energie. Die Verarbeitung von XML-Dokumenten erfordert schon heute signifikante Energiemengen.
Es gibt einen zweiten Punkt der aufwendig ist: das Parsen und Schreiben der XML-Dokumente. Der Aufwand dafür ist mindestens linear von der Anzahl der Zeichen abhängig.

Publizierte Benchmarks zeigen sogar einen deutlichen Leistungsrückgang, wenn die Dokumente größer werden, selbst wenn ihre Verschachtelungstiefe nicht zunimmt, und zwar unabhängig von den verwendeten Parsingmethoden. [5]
Die Bedeutung des Verarbeitungsaufwands wird auf absehbare Zeit noch zunehmen, da Leistungssteigerungen bei Prozessoren immer weniger durch erhöhte Taktraten erreicht werden. Die Beschleunigung durch superskalare und Mehrkern-CPU kommt beim Parsen jedoch kaum zum Tragen. Der relative Anteil der für die XML-Verarbeitung benötigten Zeit wird, bei sonst unveränderten Bedingungen, allein durch diese technische Entwicklung steigen.
Wie groß der Overhead wirklich ist, kann man sich leicht veranschaulichen durch den Vergleich der Leistung des Java-Compilers, der um Größenordnungen komplexere Operationen ausführt, mit dem Einlesen oder Transformieren eines XML-Dokuments. Natürlich ist dieser Vergleich nicht ganz fair, weil ein allgemeineres in Java codiertes Verfahren einem speziell optimierten gegenüber immer im Nachteil ist, aber die Unterschiede sind einfach zu groß, um sie allein damit begründen zu können.
Außer dem grundsätzlichen Problem gibt es in der XML-API für Java noch einige hausgemachte Performancekiller, für die jedoch wenigstens teilweise Abhilfe möglich ist. Listing 1 zeigt eine Methoden-Signatur aus Javas Standard-XML-Interface und die analoge Definition aus der Javolution-Bibliothek.[6]

Listing 1

```
// Javax-API
// javax.xml.stream.XMLStreamWriter:
public void writeStartElement(String localName) throws
                                                XMLStreamException;


// Javolution-API
// javolution.xml.stream.StreamWriter:
```

Der vermeintlich kleine Unterschied kann bei geschickter Programmierung deutliche Effizienzgewinne ermöglichen, nämlich dann, wenn die zu schreibende Zeichenkette selbst vorher zusammengesetzt wird. Listing 2 zeigt den Unterschied. Die zweite, nur mit Javolution mögliche Variante, vermeidet pro Aufruf die Erzeugung eines String-Objekts. [3]
Trotz aller erreichten Verbesserungen ist die Objekterzeugung eine relativ teure Operation geblieben, ganz abgesehen vom unnötig allokierten Speicherplatz. Dazu kommt, dass beim Anlegen eines Strings dessen Wert in einen internen Puffer kopiert wird, eine Operation mit längenabhängigen Kosten, die ebenfalls eingespart wird.
Analoge Effizienzgewinne sind beim Parsing möglich. Trotzdem ist die Javolution-Bibliothek zum Standardinterface weitgehend quellcode-kompatibel. Eine Neucompilierung mit den

[3] Das Interface CharSequence wurde erst viel zu spät mit Java 1.4 eingeführt und wird bisher in den Standardbibliotheken nur sehr zögerlich eingesetzt. Trotz der erheblichen Auswirkungen auf die Performance mangelt es der String-Behandlung insgesamt an der Stringenz, die z. B. das Collection-Framework auszeichnet.

geänderten Importen ist ausreichend. Um die gezeigten Vorteile konsequent zu nutzen, sind allerdings entsprechende Anpassungen erforderlich.

Listing 2

```
    StringBuilder stringBuilder= new StringBuilder();
    stringBuilder.append(...);
    ...
// javax u. Javolution:
    streamWriter.writeStartElement(stringBuilder.toString());

// nur mit Javolution möglich:
    streamWriter.writeStartElement(stringBuilder);
```

Problem 4: Datentypen

Mittels DTD können Strukturen für XML-Dokumente vorgegeben werden, es ist jedoch nicht möglich, Einschränkungen der zulässigen elementaren Werte zu definieren. (Das geht zu einem gewissen Grad nur bei den Attributen.) Letztlich wird alles als Zeichenkette (CDATA/PCDATA) betrachtet. Auch dieses Erbe der SGML-Abstammung ist bei entsprechenden Dokumenten akzeptabel. Wenn XML als Format für den Austausch von Datenstrukturen wie bspw. serialisierten Objekten verwendet wird, ist eine genauere Beschreibung jedoch zwingend. Um diesen Mangel zu beheben, wurde vom W3C die Beschreibungssprache XML Schema (auch XSD für **X**ML **S**chema **D**ocument) entwickelt.

Listing 3

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
	<xs:element name="doc">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="body" type="xs:string"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="head">
		<xs:complexType>
			<xs:sequence>
				<xs:element name="title" type="xs:string"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
</xs:schema>
```

Im Unterschied zu DTD werden XSD-Dokumente in XML formuliert. Im Ergebnis handelt man sich mit dieser eigentlich wünschenswerten Vereinheitlichung wieder die bereits diskutierten Nachteile, insbesondere die Geschwätzigkeit, ein.
Listing 3 zeigt eine einfache Schema-Definition in XSD –Schreibweise. Die abgesehen vom Namensraum gleichwertige DTD ist in Listing 4 zu sehen. Man kann deutlich erkennen, dass der größere Umfang der XSD-Beschreibung nur zu einem unwesentlichen Teil auf die genauere Typauszeichnung zurückzuführen ist.

Listing 4

```
<!ELEMENT doc (head, body)>
<!ELEMENT head (title)>
<!ELEMENT title (#PCDATA)>
```

XSD stellt atomare Datentypen bereit und bietet Konstruktoren, mit deren Hilfe zusammengesetzte Typen definiert werden können. Auch Typerweiterungen oder -einschränkungen sind möglich. Die Beschreibung ist hinreichend genau, um daraus Codes generieren zu können (und umgekehrt), wovon im Web-Service-Umfeld Gebrauch gemacht wird.

Aktuell ist die Version XSD 1.1 (April 2012). Die XSD 1.0 Spezifikation wurde 2001 publiziert. Es ist nicht gelungen, XSD als einheitliche XML-Schemasprache zu etablieren. Der Einsatz konzentriert sich auf Bereiche, in denen die exakte Typisierung der Elemente essentiell ist. Für Dokumente im weiteren Sinne sind nach wie vor DTD die Standardbeschreibung. Die aktuelle Einführung in die Java-XML-Verarbeitung enthält immer noch die Aussage: *Because there are very few XML parsers supporting the XML Schema specifications, most of the sample programs use the first set of XML documents, based on DTD*. [7]
Es gibt mit RELAX NG (REgular LAnguage for XML Next Generation) [8] noch eine weitere Schemasprache, die neben dem XML-Format eine wesentlich kürzere sogenannte Compact Syntax besitzt. Diese Sprache wurde 2003 als Teil eines ISO-Standards normiert, trotzdem hat sie vergleichsweise wenig Verbreitung gefunden.
Obwohl die benötigten Technologien existieren, werden aus Schema-Definitionen praktisch nie Parser generiert und als Komponenten verfügbar gemacht. Auch das ist ein ungenutztes Optimierungspotential.

Pro und Contra

There really is no single “better”—it depends on what you want. [9]

Das Herstellen von Software ist zu großen Teilen eine Ingenieurleistung oder sollte es zumindest sein, und Ingenieurarbeit ist nicht zuletzt das Finden von sinnvollen Kompromissen (Trade-offs). Wo der richtige Punkt liegt, ist immer vom konkreten Fall und den konkreten Umständen abhängig. Deshalb kann niemand ernsthaft Empfehlungen für oder gegen eine Technologie oder ein Framework im Allgemeinen geben. Vom trivialen Fall, dass etwas gar nicht funktioniert, abgesehen. Umso wichtiger ist es aber, die Kriterien zu kennen und sich überhaupt bewusst zu machen, dass man einen Kompromiss schließt und zu welchen Bedingungen.

Für die Verwendung von XML sprechen viele Gründe. Die meisten wurden bereits erwähnt:

- Es existiert ein Standard.
- Es gibt zahlreiche Tools und viele Programme unterstützten XML.
- XML ist bei Entwicklern bekannt, Erfahrungen sind vorhanden.
- Kurz, XML ist allgemein akzeptiert.

Keine Diskussion gibt es auch dann, wenn XML als Schnittstellenformat gefordert wird. Dann bleibt höchstens zu überlegen, ob für interne Prozesse eine Alternative in Betracht kommt.

Es gibt noch einige weitere Überlegungen, die zu einer Entscheidung pro XML führen können:

- Wenn es sich um (große) Dokumente handelt, für die z. B. kein wirklich effizienteres Format existiert – also um den ursprünglichen SGML-Aufgabenbereich

- Wenn bestimmte Software eingesetzt werden soll oder muss, z. B. XSL-FO-Prozessoren
- Wenn die Anwendung nicht performance-kritisch ist und keine großen Datenmengen entstehen (geringe Leistungsanforderungen, Prototypen)
- Wenn – ganz allgemein – die aus der Verwendung vorgefertigter Bausteine resultierenden Vorteile (geringerer Entwicklungs- und Testaufwand) größer sind als die daraus resultierenden Nachteile.

Gegen die Verwendung von XML sprechen die aufgeführten Problembereiche. Ganz besonders kritisch sind dabei Kernkomponenten zu betrachten, die für die Gesamtleistung entscheidend sind. Die Gefahr, dass man sich Leistungsprobleme einhandelt, ist besonders groß, wenn sehr viele kleine XML-Dokumente zu erzeugen und zu parsen sind. Zum ohnehin vorhandenen Overhead kommt dann noch der Aufwand für die jeweils erforderliche Initialisierung hinzu. Ein typisches Beispiel dafür ist die in verschiedenen Applikationsservern übliche Serialisierung/Deserialisierung mittels XML-Format.
Es lohnt sich, die Argumente genauer zu prüfen, die für die XML-Verwendung sprechen. Nur wenn es einen konkreten Nutzen für das vorliegende Projekt gibt, sollten sie akzeptiert werden. Allzu oft findet man Begründungen, die auf eher unbestimmte Vorteile durch mögliche Nachnutzungen oder leichte Austauschbarkeit von einzelnen Komponenten verweisen. Das kann zutreffen. Man muss aber vorher kritisch fragen:

- *Wie oft habe ich es erlebt, dass eine entsprechende Komponente wirklich ausgetauscht oder nachgenutzt wurde?*
- *Wie wahrscheinlich ist es, dass das diesmal geschieht?*

Nur wenn es eine hinreichend realistische Chance gibt, dass dieser Fall eintritt, sollten solche Vorteile gewertet werden. Es ist klar, dass diese Bewertung für ein Software-Produkt völlig anders ausfallen wird als für ein kundenspezifisches Projekt.

Suche von Alternativen

Alternativen gibt es mehr als ein flüchtiger Blick vermuten lässt, und es gäbe sicherlich noch mehr, wenn öfter danach gefragt würde. Getreu dem Motto *„Eines schickt sich nicht für alle“* können hier nur einige Denkanstöße gegeben werden.
Eine wichtige Voraussetzung für die erfolgreiche Suche ist, dass man sich die in den vorangegangenen Abschnitten diskutierten Punkte klar gemacht hat, dass man weiß, wonach zu suchen ist.

Im Folgenden werden einige Beispiele für alternative Lösungen kurz diskutiert:

- Konfigurationsdateien
  In vielen Fällen können XML-Konfigurationsdateien problemlos durch ganz normale Property-Dateien ersetzt werden. Nachdem eine Zeit lang der umgekehrte Weg vorherrschend war, findet man das jetzt wieder häufiger. Dabei muss es nicht beim standardmäßig vorgegebenen Format bleiben, Erweiterungen können mit vergleichsweise einfachen Mitteln implementiert werden. Es existieren diverse Bibliotheken.

- Druckdokumente
  Als Alternative für die Erzeugung hochwertiger Druckdokumente durch das ressourcenintensive XSL/FO-Paar kommt beispielsweise LaTeX in Frage. Ähnlich wie bei XML werden Texte dabei mit einer logischen Auszeichnung versehen. Es gibt eine große Anzahl von stabilen Implementierungen für fast alle Rechnerplattformen und Betriebssysteme. Besonders hervorzuheben ist die Eignung für umfangreiche Dokumente. Größere Erfahrungen im Umgang sind außerhalb des Hochschulbereichs nicht so stark verbreitet. Andererseits hat ein erheblicher Teil der Absolventen seine Abschlussarbeit in diesem Format erstellt, sodass es durchaus einiges an Basiskenntnissen gibt, die sich relativ einfach aktivieren lassen.
- Objektserialisierung
  In diesem Bereich war offenbar der *„Leidensdruck“* besonders hoch, da hier Alternativen inzwischen Fuß gefasst haben. Am bekanntesten ist JSON (**J**ava**S**cript **O**bject **N**otation), ein kompaktes für Mensch und Maschine lesbares Datenformat. Der Hauptvorteil ist die Kompaktheit – JSON-Dokumente sind deutlich kürzer als vergleichbare XML-Texte. Die Vorteile bei der Verarbeitung und Übertragung sind so wesentlich, dass für Web-Technologien wie REST und Ajax JSON fast schon das Standardformat ist.
  Eng verwandt mit JSON ist YAML (**Y**AML **A**in't **M**arkup **L**anguage)[10], das einen etwas allgemeineren Ansatz verfolgt. In der aktuellen Version YAML 1.2 ist jedes JSON-Dokument auch ein gültiges YAML-Dokument. Für beide Formate existieren Bibliotheken, die den Einsatz, auch mit unterschiedlichen Programmiersprachen, ohne großen Aufwand ermöglichen. Falls erforderlich, ist eine Umwandlung in XML möglich, für Teilbereiche existieren sogar entsprechende Tools.
- Fachsprachen (DSL – **D**omain **S**pecific **L**anguages)
  In vielen Bereichen existieren bereits mehr oder weniger formalisierte Beschreibungen. Solche Fachsprachen (DSL) sind eine hervorragende Basis nicht nur für die Algorithmenbeschreibung – wo sie seit einiger Zeit wieder verstärkt beachtet werden – sondern auch für die Datenstrukturen. In ihnen manifestiert sich jahrelange Erfahrung, die sie für den betrachteten Bereich zu einer nahezu optimalen Lösung machen. In der Praxis sollte man sich nicht davon abschrecken lassen, dass diese Formalismen zu den Rändern hin manchmal etwas „ausfransen“, d. h. weniger streng oder eindeutig sind. Das ist eine notwendige Bedingung für ihre lebendige Weiterentwicklung. Es gibt immer einen Kern, auf den die implementierte Fachsprache beschränkt werden kann. Diese Einschränkung darf und sollte durchaus auch im Hinblick auf die einfache Implementierbarkeit erfolgen. Wichtig ist, dass wie bei XML die Texte lesbar sind. Erstellt werden sie im Allgemeinen durch Programme oder mit Hilfe von Programmen, wodurch das Einhalten der Beschränkungen leicht garantiert werden kann.

Zusammenfassung

XML ist ein Bündel von bewährten Technologien, das sich in vielen Bereichen als effektiv erwiesen hat. Allerdings ist die Verwendung im laufenden Betrieb oft nicht sehr effizient. Entscheider stehen deshalb vor der schwierig zu beantwortenden Frage, ob die Effizienz der Erstellung oder die Effizienz im Betrieb höher zu bewerten ist. Die Antwort hängt immer von den konkreten Umständen ab. Im Artikel wurden die wichtigsten Einflussfaktoren dargestellt und diskutiert. Dabei wollten wir XML nicht verdammen, sondern das Gespür dafür schärfen, dass es sich um eine Technologie handelt, die wie alle ihre Vor- und Nachteile hat, und die deshalb vor jeder Adaption ebenso gründlich bewertet werden muss. Es gibt oft Alternativen, nur sind sie meist weniger bekannt. Nicht zu vergessen, mit Bewährtem geht man zwar kein oder nur ein geringes Risiko ein, aber auf neuen Wegen sind die größeren Erfolge möglich.

Literaturverzeichnis

[1] Martin Jeckle: Skriptum zur Vorlesung XML,
http://www.jeckle.de/vorlesung/xml/script.html, 2004

[2] Stefan Mintert: Freie Bahn - Neue Elemente, weniger Einigkeit, iX Sonderheft "Webdesign" 2/2012

[3] Michael Seeboerger-Weichselbaum*: XML - Das Einsteigerseminar, zit. nach*

http:// www.xmlguru.de/html/_d/04buch/buch10_d_kap03-3.html

[4] Jørn Ølmheim*:* Beauty Is in Simplicity*,*
http:///programmer.97things.oreilly.com/wiki/index.php/Beauty_Is_in_Simplicity

[5] Matthias Farwick, Michael Hafner: XML Parser Benchmarks,
http://www.xml.com/lpt/a/1702

[6] Javolution Homepage, http://www.javolution.org

[7] Thierry Violleau: Java Technology and XML,
http://java.sun.com/developer/technicalArticles/xml/JavaTechandXML/ (Stand: 31.5.2012)

[8] RELAX NG Homepage, http://relaxng.org/

[9] Pragmatic Programmers: http://pragprog.com/articles/high-fidelity

[10] YAML Homepage, http://www.yaml.org

Der Autor

Jürgen Lampe ist IT-Berater bei der Agon Solutions in Frankfurt. Vor seiner Tätigkeit als Berater wirkte er als Hochschullehrer an der Technischen Universität Dresden. Seit mehr als 15 Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen im Bankenumfeld. An Fachsprachen (DSL) und Werkzeugen für deren Implementierung ist er seit seiner Studienzeit interessiert.

Email: juergen.lampe@agon-solutions.de

Agon Solutions

Die Agon Solutions 2004 gegründet, ist ein unabhängiges IT-Dienstleistungsunternehmen mit Firmensitz in Eschborn bei Frankfurt und weiteren Standorten in Hamburg und Berlin. Das branchenübergreifende Dienstleistungsportfolio von Agon umfasst das „Agon-proven IT-Consulting", eine bewährte, herstellerneutrale IT-Beratung; sowie die „Agon-tailored IT-Solutions", zu denen maßgeschneiderte, individuelle Softwareentwicklung, passgenaue Softwareintegration und effiziente Business Intelligence Lösungen gehören. Agon stellt sich mit professionellem Projektmanagement, proaktivem Anforderungsmanagement und änderbaren Softwarearchitekturen auf die individuellen Bedürfnisse seiner Kunden ein. In ausgewählten Branchen wie Banken, Versicherungen, Aviation und Health Care bietet Agon gemeinsam mit seinen Partnern Lösungen, die auf fundiertem Geschäftsprozess-Know-how beruhen und speziell auf die jeweilige Branche zugeschnitten sind: die „Agon-tailored Business Solutions". Die plattformübergreifende technologische Kompetenz bei Agon reicht von klassischen Mainframe-Architekturen bis hin zu modernen Java/JEE Web- und Portal-Architekturen. Zu den Referenzkunden von Agon gehören unter anderen die AOK Berlin-Brandenburg, die Commerzbank, die Deutsche Bank, die Deutsche Bank Bauspar, die Deutsche Börse, die Finanz Informatik und die Deutsche Lufthansa.

Copyright:

A:gon Solutions GmbH

Frankfurter Strasse 71-75
D-65760 Eschborn
Telefon : +49 6196 80269 0
Telefax : +49 6196 80269 11
http://www.agon-solutions.de

Handelsregister Frankfurt HRB 58185
St.-Nr. 4022826171
Geschäftsführer: Udo Peters