

AxKit - modernes Web-Publishing mit Perl und XML

Jörg Walter

7. deutscher Perl-Workshop
Medienkulturzentrum Pentacon, Dresden

9. - 11. Februar 2005

Web-Publishing ist geprägt von immer wiederkehrenden Problemen und häufig wechselnden Anforderungen an die visuelle Gestaltung, neuerdings gepaart mit Anforderungen für mobile Geräte, Mehrsprachigkeit und Barrierefreiheit. Außerdem ist die Zeit vorbei, wo der "Web-Designer" eine Person war: Design, Programmierung, Wartung und inhaltliche Pflege aus einer Hand ist heutzutage kaum noch möglich, zu groß sind die Erwartungen an eine gute Online-Präsenz.

Manch ein Privatmann gibt die Programmierung schon ganz auf, gibt sich mit einem allgemeinen Theme für seine Webseite ab und nutzt einfach ein fertiges Blog, Wiki oder Newssystem – Hauptsache wenig Arbeit und halbwegs annehmbares Design. Was im privaten Umfeld vielleicht noch geht, ist im gewerblichen Bereich aber undenkbar. Und auch manch Hobbyprogrammierer stellt höhere Ansprüche an sich.

Die "großen" Techniken um dynamische Webseiten zu erstellen kennt wohl jeder, zumindest Oberflächlich. Doch sie alle haben ihre Probleme, und so entstand vor fünf Jahren AxKit, damals noch unter dem Namen axdtk - Apache XML Development Toolkit. XML deswegen, weil es eine Lösung für viele dieser Probleme bietet. Von Anfang an wurde auf aktuelle Technologien wie XML und XSLT gesetzt, zusammen mit der Perl-typischen Flexibilität, um die Stärken der Online-Welt zu nutzen anstatt immer nur deren Schwächen zu umgehen.

Der Vortrag wird zeigen, wie man die alltäglichen Probleme der Entwicklung mit AxKit leicht meistert. Es soll vor allem aufgezeigt werden, wie man Inhalt, Präsentation und Programmlogik voneinander trennen kann - AxKit ist geradezu geschaffen dazu, das MVC-Paradigma auf Web-Applikationen anzuwenden.

Auf andere beliebte Entwicklungstechniken wird ebenso eingegangen: Eventbasierte Programmierung ist problemlos möglich, das allgegenwärtige Templating ist auch in AxKit ein zentraler Punkt, und wer von "Themes" oder "Customization" träumt, der wird sicher glücklich. Die Erstellung von Programm- und Template-Bibliotheken ist natürlich auch dabei, sowie andere Formen der Wiederverwendbarkeit.

Doch auch der "kleine" Web-Entwickler bleibt nicht auf der Strecke - CGI war so erfolgreich, weil man schnell eine kleine Funktion einbauen konnte. PHP ist populär, weil es sehr direkt und intuitiv funktioniert. AxKit steht dem in nichts nach - es wird darüber hinaus auch gezeigt, wie man aus einem schnell entworfenen Prototypen ein leistungsfähiges Live-System macht, ohne wieder bei Null anfangen zu müssen.

Der Vortrag hat neben Perl einen Schwerpunkt in XML. Es wird kritisch betrachtet, wo die Dynamik von Perl hinderlich ist, und welche Alternativen es gibt. Es werden daher ebenso Techniken erläutert, die rein auf XML/XSLT basieren wie auch die vielzähligen Möglichkeiten von AxKit aufgezeigt, die semi-statischen Inhalte von XML/XSLT mit dynamischen Elementen zu versehen, ohne die Vorteile der statischen Verarbeitung zu opfern.

Inhaltsverzeichnis

1. Ax... wer?	
AxKit als Ersatz für HTML, PHP, CGI, mod_perl, ASP, JSP, ...	7
1.1. Theorie...	7
1.2. ... und Praxis	7
1.3. Funktionsweise	9
1.3.1. Provider	9
1.3.2. Transformation	9
1.3.3. Cache	10
1.3.4. Plugins	10
1.4. Die ganze Wahrheit	10
1.5. Dynamik	12
2. aber warum?	
Probleme der modernen Web-Entwicklung mit AxKit lösen	15
2.1. Trennen von Kompetenzen	15
2.2. Outsourcing	16
2.3. Wiederverwendbarkeit	17
2.4. Wartbarkeit	18
2.5. Erweiterbarkeit	19
2.6. Migration	20
3. Nutze die Macht!	
AxKit sinnvoll ausreizen	22
3.1. Anforderungen	22
3.2. Grobentwurf	23
3.3. Die Details	26
3.3.1. XSLT: das globale Design	27
3.3.2. XML: Das Schema	28
3.3.3. XSLT: XHTML-Aufbereitung	30
3.3.4. XSLT: Umwandlung des alten Datenformats	32
3.3.5. Die Konfiguration	32
3.4. Zusammenfassung	33
4. Die Fäden verbinden...	
Technologiemix als Kernprinzip	34
4.1. AxKit und Apache	34

4.2. AxKit und Perl	35
4.3. AxKit und Sprachen	36
4.4. AxKit und das Betriebssystem	36
4.5. AxKit und externe Programme	37
4.5.1. Provider	39
4.6. AxKit und Datenbanken	40
4.7. AxKit und Benutzereingaben	41
4.8. AxKit und Netzwerkdienste	41
4.8.1. Taglibs	43
4.9. Zusammenfassung	44
5. ... und nicht den Faden verlieren	
Wartbarkeit mit AxKit ist kein Problem	45
A. Quellen für das Selbststudium	47

Über den Autor

Jörg Walter ist 27 Jahre alt und seit 2001 einer der Kernentwickler des AxKit-Frameworks. Er bestreitet seinen Lebensunterhalt als selbständiger Unternehmer, tätig in den Bereichen Aus- und Weiterbildung, Beratung und Programmierung. Nebenbei vervollständigt er zur Zeit sein Informatik-Diplom an der Fernuniversität Hagen und ist zudem ehrenamtlich als Prüfer der IHK für den Beruf des Fachinformatikers Fachrichtung Anwendungsentwicklung tätig. Er programmiert seit seinem 12. Lebensjahr mit Leib und Seele und zählt dadurch etliche Programmiersprachen zu seinem aktiven Repertoire. Perl ist die beste davon. Frühe Schocks wie COBOL mit 15 Jahren und die typische Programmier-Faulheit haben ihn zu einem Verfechter der semantischen Strukturierung von Inhalt und Logik gemacht. Außerdem ist er der Beauftragte für nigerianischen Spam von Dahut.pm :-)

Vorbemerkung

Es wird ein solides Grundwissen von Perl vorausgesetzt: man sollte neben allgemeiner Programmiererfahrung in Perl CPAN-Module nutzen können und die Objektorientierung in Perl kennen. Zudem wird nicht in die Funktionsweise von XML selbst eingegangen - wer noch nicht damit vertraut ist, sollte sich vorher eine halbe Stunde ein Tutorial im Web gönnen. Auch XSLT wird nicht ausführlich erklärt - es dürfte nicht schwer fallen, die wesentlichen Teile nebenbei aufzuschnappen, aber Vorwissen ist empfehlenswert. Wer sich vorbereiten möchte, kann eine Übersicht über viele Technologien aus dem Themenbereich unter <http://www.ebseiten.hab.ich.garni.ch/Docs/XML/> finden.

Die schriftliche Fassung unterscheidet sich vom Vortrag darin, daß auch die Grundlagen von Axkit nur Auszugweise wiederholt werden – der Kern des Vortrages ist die Strukturierung von Web-Anwendungen innerhalb des Frameworks, und so sei dem interessierten Leser das Buch „XML Publishing with AxKit“ von Kip Hampton, erschienen im O’Reilly Verlag empfohlen. Wer wie der Autor zu den Hardcore-Autodidakten gehört, sollte sich vor der Lektüre die Referenzen zu AxKit, XML und XSLT zurechtlegen.

In der Bibliographie sind zu allen Themen Quellen für das Selbststudium aus dem Internet aufgeführt.

Lizenz

Dieses Dokument wird unter einer CreativeCommons-Lizenz verbreitet.

Sie dürfen:

- den Inhalt vervielfältigen, verbreiten und öffentlich aufführen
- Bearbeitungen anfertigen
- den Inhalt kommerziell nutzen

Zu den folgenden Bedingungen:

- Namensnennung. Sie müssen den Namen des Autors/Rechtsinhabers nennen.
- Weitergabe unter gleichen Bedingungen. Wenn Sie diesen Inhalt bearbeiten oder in anderer Weise umgestalten, verändern oder als Grundlage für einen anderen Inhalt verwenden, dann dürfen Sie den neu entstandenen Inhalt nur unter Verwendung identischer Lizenzbedingungen weitergeben.

Im Falle einer Verbreitung müssen Sie anderen die Lizenzbedingungen, unter die dieser Inhalt fällt, mitteilen. Jede dieser Bedingungen kann nach schriftlicher Einwilligung des Rechtsinhabers aufgehoben werden.

Die gesetzlichen Schranken des Urheberrechts bleiben hiervon unberührt.

Die genauen Bedingungen sind unter

<http://creativecommons.org/licenses/by-sa/2.0/de/legalcode>

einsehbar.

1. Ax... wer?

AxKit als Ersatz für HTML, PHP, CGI, mod_perl, ASP, JSP, ...

1.1. Theorie...

AxKit ist ein Framework für Web-Entwicklung.

Aber was bedeutet das? An jeder Ecke wird man mit Buzzwords und vollmundigen Phrasen gelockt. Leider kann man AxKit nicht vollständig buzzword-frei beschreiben, doch wenn man hinter die Kulissen schaut, wird schnell klar, was das Ziel ist.

„Web-Entwicklung“ umfasst in diesem Fall alle Systeme, die Präsentation und/oder Interaktion über große Distanzen ermöglichen. Dazu gehören ist eine einfache Homepage, ein Online-Shop oder eine automatisierte Schnittstelle zwischen einem Nachrichtendienst und einer Suchmaschine.

„Framework“ ist in diesem Kontext ernst zu nehmen: Axkit bietet nur einen Rahmen, der geeignet ist, alle denkbaren Szenarien umzusetzen.

Dazu setzt AxKit auf das bewährte Perl-Prinzip: There is more than one way to do it. Der einzige feste Kern von AxKit ist ein Modul von ca. 1.700 Zeilen inklusive Dokumentation. Alle anderen Bestandteile sind wählbar bzw. austauschbar. Dabei baut sich das Wissen wie schon bei Perl selbst sukzessive auf. Anfangs wird man nur die mitgelieferten bzw. auf CPAN verfügbaren Transformations-Module wechseln, später schreibt man eigene Providerklassen, und wer viel Erfahrung hat, ist auch in der Lage, einen alternativen ConfigReader oder einen Cache für spezielle Einsatzgebiete zu schreiben.

1.2. ... und Praxis

Aber genug der Theorie - wenn hier AxKit mit anderen Technologien verglichen werden soll, dann wären ein paar Beispiele angebracht, wie AxKit-Dokumente aussehen. Wer noch nicht mit XML vertraut ist, den wird dieses Beispiel sicher überraschen:

```
<data>Hello World!</data>
```

Das ist ein sehr kleines, gültiges AxKit-Dokument. Es sieht nicht nur willkürlich aus, das ist es auch. Mit AxKit gibt es kaum Grenzen in der Wahl unseres Quellmaterials, nur XML sollte es - wenn möglich - sein.

Hier noch ein Dokument, das die XML-Syntax etwas ausführlicher demonstriert und das im weiteren Verlauf häufiger genutzt wird:

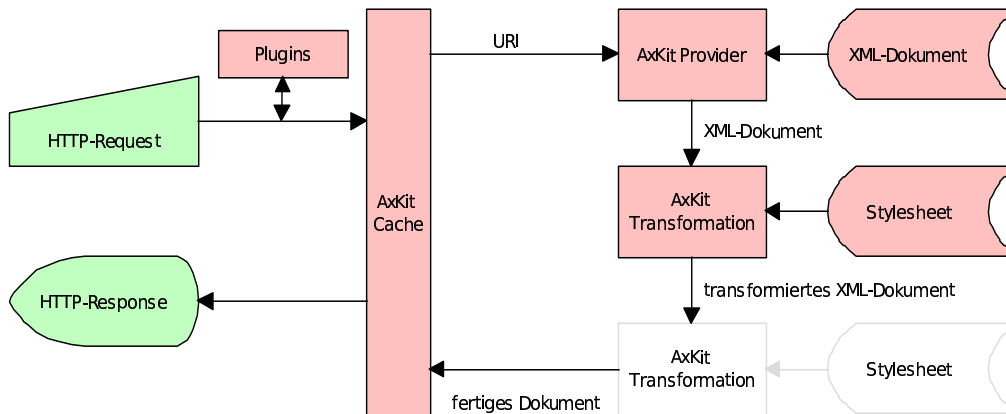
```
<?xml version='1.0' encoding='ISO-8859-15'?>
<article xmlns='http://garni.ch/2005/demo'>
  <title xml:lang='de'>Die Anatomie des Dahut</title>
  <authors>
    <author>Matt Sergeant</author>
    <author>Kip Hampton</author>
    <author>Robin Berjon</author>
    <author>Barrie Slaymaker</author>
    <author>Jörg Walter</author>
  </author>
  <original-language>en_US</original-language>
  <Wahrheitsgehalt>nada</Wahrheitsgehalt>
  <body xml:lang='de' xmlns='http://www.w3.org/1999/xhtml'>
    <p>Die Anatomie des Dahuts ähnelt der einer Ziege. Auffälligstes
    Unterscheidungsmerkmal sind die einseitig kürzeren Beine (siehe
    Skizze).</p><p><img src='dahut.png' /></p>
  </body>
</article>
```

Nahezu alle Quelldokumente und Stylesheets sind XML. Andere Datenformate sind problemlos verwendbar, intern werden sie jedoch übersetzt und als XML verarbeitet.

Inzwischen sollte etwas auffallen: Man sieht keinen Programmcode, keine Template-Anweisungen, nur strukturierte Daten. Im Gegensatz zu anderen Systemen ist es leicht, ganz normale Dateien für die Quelldaten zu nutzen - viele (auch interaktive) Anwendungsfälle funktionieren ganz ohne Datenbank. Die eigentlichen Nutzdaten stehen stärker im Mittelpunkt, es wird mehr Augenmerk auf die Wahl des XML-Vokabulars und damit der Datenstruktur gelegt, während deren Aufbereitung sich dieser Struktur mit Leichtigkeit anpasst.

Diese Trennung von Inhalt und Präsentation ist ein Kernprinzip von AxKit. Man sieht nicht, was einmal mit dieser Datei passieren wird. Es kann sein, daß dieses Dokument für eine Webapplikation direkt ausgeliefert wird, oder in HTML für den Browser eines Besuchers aufbereitet wird. In typischen AxKit-Systemen ist es sogar wahrscheinlich, daß beides gleichzeitig zutrifft. Und nichts hindert uns daran, aus allen Dateien eines Verzeichnisses eine Inhaltsübersicht zu generieren.

Abbildung 1.1.: AxKit Funktionsweise



1.3. Funktionsweise

AxKit wird dieses Beispieldokument also auf verschiedene Arten verarbeiten. Um die nachfolgenden Beispiele dieser Verarbeitung besser zu verstehen, soll nun die grobe Funktionsweise von AxKit dargestellt und die im weiteren Verlauf verwendeten Fachbegriffe erklärt werden.

Abb. 1.1 zeigt die wesentliche Funktionsweise von AxKit. Intern ist der Ablauf wesentlich komplexer, doch als Anwender reicht es, die drei wichtigsten Bausteine zu kennen:

1.3.1. Provider

Provider liefern die Quelldaten an AxKit. Die Daten werden standardmäßig einfach aus XML-Dateien gelesen. Es können aber auch beliebige andere Dateien in XML umgewandelt werden, oder XML dynamisch aus einer Datenbankabfrage generiert werden - der Phantasie sind hier keine Grenzen gesetzt.

1.3.2. Transformation

Die XML-Daten werden von Transformationsmodulen mit der Hilfe von Stylesheets in andere XML-Daten umgewandelt. In der Praxis bedeutet dies z.B., daß die Daten in (X)HTML umgewandelt werden oder gemeinsame Seitenelemente hinzugefügt werden, Navigation, Banner usw. Dieser Teil ähnelt bekannten Template-Systemen. In der Dokumentation heißen diese Module auch Language-Module. Wie schon in Abb. 1.1 angedeutet, kann es mehrere Transformationsschritte geben. Der letzte Transformationsschritt kann dabei beliebige Daten erzeugen, das Resultat kann also auch PDF oder HTML sein.

1.3.3. Cache

Der Cache ist eine überraschend elementare Komponente. Im Vergleich zu vielen dynamischen Web-Entwicklungsumgebungen ist die Transformation oftmals nur auf den ersten Blick dynamisch. Viele Teile einer Website sind in Wirklichkeit statisch, und AxKit erlaubt es, diesen Umstand auszunutzen. Der Cache ist hocheffektiv und ermöglicht es, Daten fast so schnell wie reine HTML-Dateien auszuliefern. Dabei kümmert sich der Cache aber um die Konsistenz - wird eine Quelldatei oder ein Stylesheet verändert, so wird die Seite automatisch regeneriert. Hierbei werden alle Dateien erfasst und überprüft, die zum Aufbau einer Seite genutzt wurden, egal wie komplex die Dateien und Stylesheets andere Dateien einbinden. Bei Seiten, die ohnehin dynamisch sind, reicht der Cache die Daten nur durch. Dieses Prinzip vereint die Vorteile von Offline-Template-Lösungen und voll dynamischen Umgebungen.

1.3.4. Plugins

Bevor AxKit mit der Abarbeitung einer Anfrage beginnt, erhalten noch Plugins die Möglichkeit, Aspekte der Verarbeitung zu ändern. Plugins sind dabei ganz schlichte Perl-Module, die z.B. die Transformationspipeline beeinflussen, das Cache-Verhalten steuern oder Daten im Request-Objekt verändern. So gibt es z.B. mehrere Sitzungs- und Authentifikationsmodule, die über den Plugin-Mechanismus arbeiten, was zu einer leichteren Konfiguration gegenüber den Apache/mod_perl-Direktiven führt.

1.4. Die ganze Wahrheit

Wenn also aus dem Beispiel auf Seite 8 ein vorzeigbares Dokument werden soll, dann wird nun noch ein Transformationsmodul und ein Stylesheet benötigt.

Hier wird wieder die Vielseitigkeit von AxKit offenbar, denn man ist nicht an eine einzelne Stylesheetsprache gebunden. Die im XML-Umfeld wohl am häufigsten genutzte Sprache ist XSLT. Als W3C-Standard hat sie den Vorteil, daß es viel Literatur und inzwischen auch Entwicklerwerkzeuge gibt.

XSLT könnte man wohl am besten als semi-statische Transformation bezeichnen. Sie hat den Vorteil, daß sie in den üblichen Anwendungsfällen vollkommen deterministisch ist: ein Quelldokument und ein Stylesheet ergeben stets das gleiche Ergebnis, da ansonsten keine Eingaben von außen genutzt werden (können). Das ermöglicht korrektes und effizientes caching, und doch stehen viele Möglichkeiten zur Verfügung, die typisch für dynamische Webseiten sind.

Ein einfaches Stylesheet, das obiges Dokument in HTML umwandelt, könnte so aussehen:

```
<xsl:stylesheet version='1.0'  
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
```

```

xmlns:art='http://garni.ch/2005/demo'
xmlns:html='http://www.w3.org/1999/xhtml' >

<xsl:template match='/'>
  <html>
    <head>
      <title><xsl:value-of select='/article/title'/></title>
    </head>
    <body><xsl:apply-templates/></body>
  </html>
</xsl:template>

<xsl:template match='art:title'>
  <h1><xsl:apply-templates/></h1>
</xsl:template>

<xsl:template match='art:authors'>
  <p>Autoren:
    <xsl:for-each select='art:author'>
      <xsl:value-of select='.'/>
      <xsl:if test='position() != last()'>, </xsl:if>
    </xsl:for-each>
  </p>
</xsl:template>

<xsl:template match='art:Wahrheitsgehalt'>
  <p>Der Wahrheitsgehalt dieser Geschichte ist
    <xsl:value-of select='.'/>.</p>
</xsl:template>

<xsl:template match='art:original-language' />

<xsl:template match='html:body'><xsl:apply-templates/></xsl:template>

<xsl:template match='html:*'>
  <xsl:copy select='.'>
    <xsl:copy-of select='@*' />
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>

```

Dieses Stylesheet erzeugt aus der Beispieldatei auf Seite 8 das folgende Dokument:

```

<html xmlns='http://www.w3.org/1999/xhtml'>
  <head>
    <title>Die Anatomie des Dahut</title>
  </head>
  <body>
    <h1>Die Anatomie des Dahut</h1>
    <p>Autoren: Matt Sergeant, Kip Hampton, Robin Berjon,
      Barrie Slaymaker, Jörg Walter</p>
    <p>Der Wahrheitsgehalt dieser Geschichte ist nada.</p>
    <p>Die Anatomie des Dahuts ähnelt der einer Ziege. Auffälligstes
      Unterscheidungsmerkmal sind die einseitig kürzeren Beine (siehe
      Skizze).</p><p><img src='dahut.png' /></p>
  </body>
</html>

```

Streng genommen ist dies nicht das exakte Resultat. Einige der weniger wichtigen Eigenschaften von XML und XSLT sind hier jedoch der Lesbarkeit zuliebe geopfert worden.

Um alle Teile nun zusammenzuführen, muß AxKit installiert und konfiguriert werden. Als Konfiguration reichen wenige Direktiven in der Datei HTTPD.CONF¹:

```

PerlModule AxKit
AddHandler axkit .xml
AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT
AxAddProcessor text/xsl file:///example/path/stylesheet.xml

```

Mit dieser Konfiguration werden alle Dateien, die in .XML enden, von AxKit verarbeitet und mit dem Beispielstylesheet in HTML transformiert.

1.5. Dynamik

Um echte interaktive Anwendungen zu schreiben, will man aber Formulare nutzen, die Tageszeit berücksichtigen und den Besucher namentlich grüßen. Man braucht also mehr Eingabedaten. Der Vorteil von XSLT, die Isoliertheit, ist hier ein Nachteil.

XSLT würde sogar ausreichen - es ist Turing-Vollständig, und es gibt durchaus Mittel und Wege, zusätzliche Eingabedaten in die Verarbeitung einfließen zu lassen. Die Verfügbarkeit von Debuggern und Interpretern zeigt, daß die Programmierung in XSLT auch real existiert. Doch zum einen ist XSLT für allgemeine Programmierung sehr unhandlich, und zum anderen hat AxKit viel mehr zu bieten. AxKit erlaubt es uns, nach dem Prinzip „The right tool for the right job“ zu handeln.

¹Bis auf die Zeile PERLMODULE AXKIT können alle Direktiven auch an allen anderen Orten der Apache-Konfiguration erscheinen, z.B. in .HTACCESS-Dateien.

Das einfachste dieser Werkzeuge für die nächste Aufgabe ist wohl XSP - eXtensible Server Pages. Wie schon die Namensverwandtheit andeutet, funktioniert es ähnlich wie ASP oder PHP - man kann ganz einfach Code und XML mischen.

So könnte eine XSP-Datei aussehen:

```
<xsp:page language='Perl'
          xmlns:xsp='http://www.apache.org/1999/XSP/Core'>
<html>
  <head>
    <title>XSP Demo</title>
  </head>
  <body>
    <p>Aktuelle Uhrzeit: <xsp:expr>localtime</xsp:expr></p>
    <dl>
      <xsp:logic>
        foreach my $key (keys %ENV) {
          <dt><xsp:expr>$key</xsp:expr></dt>
          <dd><xsp:expr>$ENV{$key}</xsp:expr></dd>
        }
      </xsp:logic>
    </dl>
    <p>
      1 ist
      <xsp:expr>(1 &lt; 0?'kleiner':'nicht kleiner')</xsp:expr>
      als 0
    </p>
  </body>
</html>
</xsp:page>
```

Man sieht sehr schön den Perl-Code und die Daten miteinander vermengt. XSP hat zwei Besonderheiten gegenüber vergleichbaren Systemen: Zum einen muß das gesamte Dokument gültiges XML sein, was bedeutet, daß z.B. der numerische Operator „kleiner als“ als < geschrieben werden muß². Das führt auf der anderen Seite aber dazu, daß XML-Tags stets als Daten erkannt werden, selbst wenn sie wie oben in der FOREACH-Schleife mitten in Perl-Code stehen, es ist kein PRINT-Befehl o.ä. nötig.

Aus dieser Datei könnte beispielsweise folgender HTML-Code entstehen:

```
<html>
```

²Wer denkt, man könne stattdessen gleich den Perl-Operator „LT“ verwenden, vergißt daß dies nicht das selbe wie „<“ ist. Allerdings kann man statt dem ebenso problematischen „&&“ bequem „AND“ verwenden.

```
<head>
  <title>XSP Demo</title>
</head>
<body>
  <p>Aktuelle Uhrzeit: Sat Dec 24 18:59:59 2005</p>
  <dl>
    <dt>HTTP_HOST</dt><dd>www.example.com</dd>
  <dl>
    <p>1 ist nicht kleiner als 0</p>
  </body>
</html>
```

Auch hier genügen wenige Direktiven in der Datei HTTPD.CONF:

```
PerlModule AxKit
AddHandler axkit .xsp
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP
AxAddUriProcessor application/x-xsp NULL \.xsp$
```

Mit dieser Konfiguration werden alle Dateien, die in .XSP enden, von AxKit verarbeitet. Im Gegensatz zum vorigen Beispiel gibt es kein separates Stylesheet, Code und Daten sind miteinander vermischt. Das macht den Einstieg leicht und ist die Parallele zu PHP, ASP u.ä., erzeugt aber andere Probleme, die ja gerade mit AxKit gelöst werden sollen. XSP eignet sich dennoch sehr schön zum Prototyping, und kann mit der richtigen Strategie auch in Produktionssystemen sinnvoll eingesetzt werden.

2. aber warum?

Probleme der modernen Web-Entwicklung mit AxKit lösen

Die grundlegenden Funktionsprinzipien von AxKit wurden im vorigen Abschnitt gezeigt. Doch das bisher gezeigte ist wohl mit allen aktuellen Web-Entwicklungsumgebungen machbar. Es drängt sich also die zentrale Frage dieses Vortrags auf: Was hat AxKit, was andere nicht haben?

Um zu sehen, wo AxKit Lösungen bietet, sollte man sich im Klaren darüber sein, was eigentlich die Probleme der üblichen Web-Entwicklung sind. Es gibt mehr als man zunächst denken mag - wer nur an Ein-Mann-Projekten sitzt, wird einiges vielleicht bisher nicht so zur Kenntnis genommen haben, doch betreffen diese Probleme praktisch jeden Entwickler früher oder später. In diesem Kapitel wird die Theorie beleuchtet, ein umfassendes, praktisches Beispiel wird es im nächsten Kapitel geben.

2.1. Trennen von Kompetenzen

Das Problem:

Ein guter Programmierer ist ein schlechter Designer, und umgekehrt. Für diese Verallgemeinerung entschuldige ich mich bei denen, die die Ausnahme von der Regel sind, doch für die Masse der Programmierer (und Designer) trifft dies zu. Das ist völlig normal, denn die Denkweisen für beide Berufen sind eben unterschiedlich.

Dazu kommt, daß weder Programmierer noch Designer später einmal die Inhalte einer Online-Präsenz pflegen werden. Egal ob günstige Aushilfen zur Datenerfassung eingestellt werden, oder die hausinterne Presseabteilung die redaktionelle Betreuung übernimmt, die Chancen stehen gut, daß es weder der Programmierer noch der Designer sein wird. In Ein-Mann-Projekten möchte oftmals der Auftraggeber selber Texte und Daten erfassen und pflegen, um Kosten zu sparen, während in großen Firmenpräsenzen vielleicht Personen aus allen Abteilungen an der Pflege beteiligt sein sollen.

Ein Programm aus einem Guß, in dem Inhalt, Präsentation und Layout über alle beteiligten Dateien verstreut sind, erfüllt diese Anforderungen nicht, und doch sehen viele Online-Präsenzen hinter den Kulissen so aus. Und wenn eine solche Trennung erst separat programmiert werden muß, ist das auch nicht das Gelbe vom Ei.

Die Lösung:

Wie im vorigen Kapitel schon gezeigt wurde, ist strenge Trennung von Inhalten und Daten ein Grundprinzip in AxKit. Dadurch, daß mit XSLT ein W3C-Standard unterstützt wird, ist die Zusammenarbeit mit professionelleren Designern inzwischen leicht. Etablierte Arbeitsweisen sind trotzdem möglich - es muß also nicht alles auf einmal geändert werden.

AxKit hilft auch, Daten von Code zu trennen. Das ist bisher noch nicht sehr deutlich geworden, doch gibt es mehrere Varianten, um Programmierern klassische objektorientierte Entwicklung zu ermöglichen und die Anwendung dann in AxKit einbinden zu können. Das auf dem Perl-Workshop vom letzten Jahr vorgestellte SAWA-System läßt sich beispielsweise als Provider in die AxKit-Transformationspipeline einbinden, und selbst für XSP gibt es Möglichkeiten, den eigentlichen Programmcode getrennt zu erstellen.

2.2. Outsourcing

Das Problem:

Zu dem vorigen Problem gesellt sich noch ein ähnliches: Was ist, wenn Design oder Inhalte extern beauftragt werden? Was wenn Programmfunktionen von anderen Online-Plattformen bereit gestellt werden?

Die verbreitete Praxis, externe Dienstleistungen in einem Frame aufzurufen führt zu optischen und inhaltlichen Inkonsequenzen, zudem ist man für alle Änderungen auf diese Drittfirma angewiesen - selbst kleinste Änderungen können nicht eigenständig durchgeführt werden.

Beim Design ist in der Praxis noch viel mehr im Argen: Mal werden Screenshots ausgetauscht, mal FrontPage-Dateien, oder viele andere Dinge - als Programmierer muss man sich auf alles einrichten, und selten kommen die Daten programmgerecht an - es fehlt an standardisierten Möglichkeiten, die Schnittstelle zwischen Design und Programm zu beschreiben. Das Sandwich-System (Inhalt zwischen zwei Template-Hälften) ist oftmals der einzige Ausweg.

Selbst die Zusammenarbeit zwischen einzelnen Abteilungen einer Firma kann hier schon Probleme bereiten. Je mehr Abhängigkeiten zwischen den Teilen einer Webpräsenz bestehen, desto schwieriger lassen sich bewährte und etablierte Verfahren wie z.B. (S)FTP und Unix-Zugriffsrechte oder die aktuelle standardisierte Lösung WebDAV zur kontrollierten Arbeit mit Zugangskontrolle nutzen.

Die Lösung:

In AxKit besteht standardmäßig eine 1:1 Zuordnung zwischen URLs und Dateien. Wie schon gezeigt wurde, enthalten die Dateien auch tatsächlich nur die Nutzdaten bzw. die

Designinformationen. Das macht es leicht, direkten Zugriff auf diese Dateien zu erlauben. Eine Vielzahl von Transformationsmodulen ermöglicht eine hohe Flexibilität im Umgang mit extern gelieferten Design-Daten.

Daten können über das Netz von anderen Dienstleistern abgerufen werden und direkt als Quelldaten in den Transformationsprozeß einfließen, denn Interoperabilität ist ja gerade ein Grundprinzip von XML. Werden die benötigten Daten über ein schon existierendes Programm in einer Datenbank verwaltet, z.B. eine Produktdatenbank des Warenwirtschaftssystems, dann kann über einen entsprechenden Provider diese Datenbank direkt abgefragt werden. Das schon erwähnte Caching sorgt dabei stets für gute Zugriffszeiten selbst bei vergleichsweise langsamen Abfragen quer über das Internet.

2.3. Wiederverwendbarkeit

Das Problem:

Eine der großen Errungenschaften der Objektorientierung in der klassischen Programmierung war die Fähigkeit, generische Komponenten zu erstellen und diese dann nur noch für den jeweiligen Einsatzzweck zu spezialisieren, ohne den Quelltext der generischen Komponente modifizieren oder auch nur einsehen zu müssen.

Während die Wiederverwendung von Teilen der reinen Programmlogik in praktisch allen Web-Umgebungen genauso funktioniert wie in der klassischen Anwendungsprogrammierung gibt es sehr unterschiedliche Konzepte für die Präsentationsebene. In den meisten Fällen läuft es darauf hinaus, daß man explizit bestimmte Komponenten in die Quelldateien einbinden muß. Das ist hinderlich, da die einzelnen Seiten aus logischer Sicht die höchste Spezialisierung darstellen, im Internet jedoch möglichst generisch behandelt werden sollen - schließlich will man das Aussehen und die Darreichungsform, ja sogar die Funktionsweise je nach Endgerät variieren. Will man also aus der selben Programmlogik unterschiedliche Datenformate erzeugen, so muß man dies in jedem Quelltext machen, was wieder mehr Arbeit und kompliziertere Strukturen bedeutet.

Die Lösung:

Für den Programmcode werden die üblichen Mechanismen von Perl verwendet. Die Präsentationsebene wird wie im vorigen Kapitel über die Apache-Konfiguration gesteuert - es ist kein Eingreifen in Datendateien oder Stylesheets nötig.

Die Apache-Konfiguration bietet mit `<FILES>`-, `<LOCATION>`-, oder `<DIRECTORY>`-Blöcken und `.HTACCESS`-Dateien schon wohlbekannte Methoden, Klassen von Dokumenten eine einheitliche Behandlung zukommen zu lassen. Über dies hinaus lässt sich die Transformation über verschachtelte Konfigurationsanweisungen, Plugins, virtuelle Pfade oder die Programmlogik weiter variieren, ohne die herkömmliche Konfiguration komplett aufgeben zu müssen.

Daß die Daten überhaupt durch ein simples Austauschen der Transformationspipeline wiederverwendet werden können, liegt wieder einmal an der Natur von XML selbst. Aus dem gleichen Grund ist das Kombinieren oder automatische Einfügen von Dateien kein Problem.

2.4. Wartbarkeit

Das Problem:

PHP ist weit verbreitet, und das verleitet viele Auftraggeber, PHP in die Auftragspezifikation aufzunehmen. Der Gedanke dahinter ist, daß es leicht ist, qualifiziertes Personal für Wartung und Erweiterung zu finden. Das ist leider (oder in unserem Fall, zum Glück) nur die halbe Wahrheit. Ein System, das Ad-Hoc Lösungen leicht macht und darüber hinaus von Haus aus wenige Möglichkeiten zur Strukturierung mitbringt, fördert den guten alten Spaghetticode und damit das Problem, das eigentlich gerade umgangen werden sollte.

Auf der anderen Seite ist ein System wie JSP, das auf klassische Objektorientierung setzt, nur mit sorgfältiger Planung zu entwerfen, und dann schwer zu erweitern - typische Onlinpräsenzen haben eine wesentlich losere Bindung zwischen den einzelnen Komponenten als typische Standalone-Applikationen.

Im Angesicht der vorangegangenen Anforderungen darf auch die Überprüfung von geänderten Daten und Programmteilen nicht vernachlässigt werden. kleine Syntaxfehler in PHP können drastische, aber zunächst unbemerkte Auswirkungen haben. JSP ist ein Albtraum für die Generierung von strikt standardkonformem HTML. Praktisch keine Entwicklungsumgebung kann korrektes Verschachteln und Schließen von Elementen über Dateigrenzen hinweg sicherstellen.

Die Lösung:

Perl ist eine bekannte Sprache, und sauberes Anwendungsdesign erzeugt auch sauberen Code - auch wenn dies nicht dem üblichen Ruf entspricht. Dabei ist Perl nicht schwieriger zu erlernen als PHP - es bedarf aufgrund der hohen Freiheit einer gewissen Disziplin, doch die braucht ein Projekt, das wartbar sein soll, ohnehin. Im Gegensatz zu PHP gibt es aber fertige Hilfsmittel für nahezu alle Programmierparadigmen, z.B. die mitgelieferten Direktiven für Objektorientierung oder SAWA für endliche Automaten und ereignisbasierte Verarbeitung. Die Freiheit der Sprache verhindert, daß alles in ein Objektmodell gezwängt werden muß wie in Java - wer das will, kann das natürlich trotzdem tun, doch man kann für jedes Problem die richtige Technik verwenden.

Auf der Datenseite sorgt die strikte Überprüfung und Erhaltung der XML-Konformität dafür, daß typische kleine Syntaxfehler im HTML wie vergessene Anführungszeichen oder falsch gesetzte spitze Klammern gar nicht auftreten können. Der Umstand, daß auch

Stylesheets XML-Dateien sind, sorgt dafür, daß falsch geschachtelte Elemente unmöglich sind. Die Robustheit der XML-Verarbeitung allgemein bewirkt, daß eine einfache Überprüfung auf Wohlgeformtheit in den meisten Fällen ausreichend genug ist. Wenn höhere Ansprüche an die Überprüfung von Daten und Design gestellt werden, bietet die XML-Welt DTD- oder Schema-basierte Validierung.

2.5. Erweiterbarkeit

Das Problem:

Sehr ähnlich ist das Problem der Erweiterbarkeit. Einer der großen Vorteile der Web-Entwicklung ist immer die lose Kopplung der einzelnen Seiten gewesen - Änderungen in einem Bereich der Präsenz wirken sich nur wenig auf andere Bereiche aus. Andererseits müssen manche Änderungen über alle Seiten gleich wirken. Die Probleme der Wartbarkeit treffen also auch hier unmittelbar zu.

Das Refaktorisieren von ganzen Datenbankmodellen ist ohnehin katastrophal, aber statische Sprachen bekommen schon Probleme beim Wechsel eines Datentyps. Die Umstellung auf ein neues Datenformat trifft die meisten Anwendungen unverhofft - es wurde bei der ursprünglichen Planung keine Infrastruktur zur Verwaltung mehrerer Formate bzw. Versionen geschaffen. Paradebeispiel ist die nachträgliche Erweiterung auf mehrere Sprachen - aus Verzweigung und Zeitdruck wird dann einfach die komplette Präsenz dupliziert, mit allen verbundenen Problemen der Synchronisierung.

Die Lösung:

Anwendungsweite Änderungen sind im allgemeinen optische Änderungen. Ein zusätzliches XSLT-Stylesheet kann eine solche Änderung durchführen, ohne die folgenden bzw. vorherigen Transformationsstufen zu beeinträchtigen. Gleichzeitig sind ja die Datendateien autonom, also kann eine Inhaltsänderung niemals Auswirkungen auf andere Daten haben.

Muß das Datenformat geändert werden, so funktionieren die Stylesheets trotzdem weiter: hinzugefügte Felder werden ignoriert, entfallene Felder bleiben leer. Der Datentyp von XML ist sowieso Text, also gibt es auch da keine Probleme. Bei einschneidenden Änderungen, z.B. wenn die Warenwirtschaft die Datenbankstruktur gewechselt hat, muß nur der Provider angepasst werden, der die Datenbankdaten in XML aufbereitet. Eine einfache Behandlung von Mehrsprachigkeit ist mit dem `XML:LANG`-Attribut sogar direkt in XML verankert, falls man sonst keine Infrastruktur geschaffen hat: ein einfaches zusätzliches Stylesheet in der Transformationspipeline reicht aus.

2.6. Migration

Das Problem:

Vergleichbar mit den vorigen beiden Problemen ist das der Migration. Heutzutage existieren üblicherweise schon Vorgängersysteme, die erweitert oder ersetzt werden sollen. Ob man versuchen will, den Programcode teilweise oder ganz wiederzuverwerten, sei dahingestellt, da das neue Horrorszenarien der Wartbarkeit erzeugt. Die Daten möchte man jedoch tunlichst nicht erneut eingeben müssen.

Solange die Daten aus einer Datenbank stammen, ist dies für alle modernen Systeme kein allzugroßes Problem. Allerdings sind die Datenbankabfragen üblicherweise quer über alle Quelldateien verstreut, was eine Anpassung von schon existierendem Code aufwendiger macht. Auch die Koexistenz von übernommenen Daten und neuen Daten kann problematisch werden, wenn sich die Struktur der Nutzdaten wesentlich geändert hat.

Am Ende könnte man sich entscheiden, alle vorhandenen Daten einmalig in die neue Repräsentation umzuwandeln, sie also in das neue System zu importieren. Das allerdings macht es schwer, ein neues System schrittweise ohne Unterbrechungen in Betrieb zu nehmen - das alte System wird an einem Stichtag abgeschaltet, dann wird konvertiert und das neue System aktiviert. Viel besser wäre es, wenn man beide Systeme parallel laufen lassen könnte, um das neue System im realen Einsatz von ausgewählten Mitarbeitern testen zu können. Besteht dringender Bedarf für eine bestimmte neue Funktionalität, so könnte dieser Teil sogar schon der Allgemeinheit zugänglich gemacht werden. Doch je stärker die Abhängigkeiten der einzelnen Bestandteile eines Systems, desto schwieriger wird die isolierte Inbetriebnahme von Funktionen.

Die Lösung:

Die Antwort hierauf liegt gar nicht so sehr in AxKit verborgen als in der richtigen Strukturierung der zu erstellenden Web-Anwendung. Allerdings bietet das Provider-Konzept die Fähigkeit, live zwischen unterschiedlichen Datenformaten oder Datenbankstrukturen vermitteln zu können. Ein gut geplanter Datenbankprovider enthält alle nötigen Abfragen gesammelt an einem Ort, so daß die Anpassung an eine andere Datenquelle eine Leichtigkeit wird.

Die Freiheit in der Gestaltung und die lose Kopplung von Komponenten in AxKit-basierenden Anwendungen ermöglicht es, die Strukturen eines neuen Systems so aufzubauen, daß ein Teil der Anwendung noch mit altem Code läuft, während problematische oder fehlerhafte Teile schon mit neuem Code laufen. Ein solcher Migrationsvorgang kann sich ohne Probleme über Monate oder Jahre hinziehen, indem immer nur ein Teil ausgetauscht wird, an dem ohnehin Wartungsbedarf besteht.

Gleichzeitig soll AxKit in naher Zukunft technologieübergreifend arbeiten können: Die Zusammenarbeit mit mod_perl-basierten Systemen wie HTML::Mason ist schon Reali-

tät. Mit Apache 2 wird es zusätzlich möglich sein, daß die Ausgabe von PHP als Quelldaten in AxKit einfließen. Das eröffnet ganz neue Wege in der schrittweisen Migration.

3. Nutze die Macht!

AxKit sinnvoll ausreizen

Nach so viel Theorie und Problemgewühle soll nun demonstriert werden, daß AxKit in der Tat adäquate Lösungen bietet. Aufbauend auf dem Beispiel des ersten Kapitels wird hier nun eine Website entworfen, die ein wissenschaftliches Literatur- und Informationssystem darstellen soll.

In diesem Szenario will eine Gruppe Kryptozoologen¹ ihre Informationen katalogisieren und einheitlich aufbereiten. Der Einfachheit halber sei angenommen, daß die bisherigen Daten schon als XML vorliegen, jedoch in dem wenig durchdachten Vokabular aus Kapitel 1. Ein wichtiger Bestandteil von typischen Webanwendungen wird jedoch ausgelassen: Es werden über die Webpräsenz keine Änderungen an den Nutzdaten vorgenommen. Dieses Thema füllt mit Leichtigkeit ein ganzes Tutorial für sich, daher wird auf Standardschnittstellen zurückgegriffen. Es sei nur am Rande erwähnt, daß es Zusatzmodule für AxKit gibt, die hierbei wertvolle Dienste leisten.

3.1. Anforderungen

1. Auf lange Sicht soll statt dem Format auf Seite 8 ein neues Vokabular für bessere Strukturierung der Daten sorgen. Es wird erwartet, daß einzelne Fachbereiche das Vokabular erweitern wollen, um Informationen strukturiert erfassen zu können, die nur in deren Bereich entstehen, beispielsweise statistische Daten über die Lafrichtung bei Dahuts.
2. Ein einfaches Design der Seiten wird zunächst von den Programmierern erstellt und dient als Platzhalter für die im späteren Verlauf von einem externen Dienstleister gelieferten Gestaltung.
3. Jeder Wissenschaftler soll Dateien erstellen können und seine eigenen Dateien bearbeiten können. Dabei reicht eine einfache Benutzer-/Gruppenbasierte Zugangskontrolle aus.
4. Es soll vier Versionen der Website geben: Eine mit einem normalen Browser benutzbare HTML-Fassung mit aufwendigem Design, eine für PDAs optimierte Version,

¹Kryptozoologie (wörtlich: Studien verborgene Tiere betreffend) befasst sich mit Tieren, die in Geschichten und Kulturen überliefert sind, jedoch bisher nicht wissenschaftlich nachgewiesen konnten. Bekannte Beispiele sind der Yeti, das Monster von Loch Ness und der Wolpertinger.

eine Variante für sehbehinderte und blinde Menschen, die gleichzeitig als Druckversion fungieren soll, sowie eine Schnittstelle für den automatisierten Datenaustausch mit angeschlossenen Instituten.

5. Es ist zu erwarten, daß sich desöfteren wissenschaftliche Hilfskräfte an der Wartung und Weiterentwicklung beteiligen werden. Die leichte Einarbeitung soll beim Entwurf berücksichtigt werden.
6. Die Zusammenarbeit mit anderen Instituten wird unweigerlich dazu führen, daß das System regelmäßig um neue Abfragemechanismen oder Datenformate erweitert werden muß.
7. Mehrsprachigkeit ist zunächst nicht nötig, da nicht klar ist, wie internationale Zusammenarbeit ablaufen wird, aber die erfassten Daten sollen auf jeden Fall schon Informationen enthalten, in welcher Sprache sie erstellt wurden und ggf. von welcher Sprache sie übersetzt wurden.

3.2. Grobentwurf

Die Anforderungen wurden so gewählt, daß sie alle Themenbereiche des vorigen Kapitels abdecken und real existierende Anforderungen widerspiegeln. Dabei sind schon einige nichttriviale Punkte enthalten, die jedoch bei richtiger Planung kein Problem darstellen.

AxKit ist ja im Prinzip recht einfach gestrickt. Für jede der Anforderungen muß nur der richtige Ansatzpunkt gefunden werden, und da gibt es nicht viel Auswahl:

- Provider dienen der Bereitstellung und ggf. der erstmaligen Aufbereitung von Daten. Außerdem ist es sinnvoll, umfangreichere Programmlogik hier anzusiedeln.
- Die Quelldokumente enthalten die Nutzdaten. Bei kleineren Aufgaben kann dank XSP auch Programmlogik hier untergebracht werden. Dies ist besonders vorteilhaft, wenn eine hohe Unabhängigkeit der Komponenten gewünscht wird.
- Die Transformationen erledigen die Aufbereitung der Daten für die Auslieferung, insbesondere auch das Umwandeln und Filtern von Informationen. Wenn die Stylesheets und die Verarbeitungspipeline intelligent entworfen werden, wird der Programmierer zu Anfang das komplette Gerüst aufstellen, das dann später nicht oder nur noch minimal verändert werden muß.
- Der Cache arbeitet automatisch. Er ist nur insofern zu beachten, daß nicht aus versehen statische Inhalte den Cache umgehen, weil man einen voll dynamischen Verarbeitungsschritt eingebunden hat.
- Plugins steuern die Verarbeitung. Über sie kann man leicht das gewünschte Datenformat, die Sprache oder die Darstellung wählen. Dabei setzen sie nur bestimmte Informationen im Request-Objekt, die dann von Provider, Transformatoren und Cache ausgewertet werden.

Die 7 Anforderungen aus Kapitel 3.1 finden nach den obigen Regeln folgende Ansatzpunkte:

1. Das Datenformat wird vorab entworfen und existiert zunächst nur in der Dokumentation. Ein existierender, bekannter oder sogar standardisierter XML-Dialekt ist natürlich vorzuziehen, sofern er anwendbar ist. Keinesfalls sollte man irgendeinen Standard wählen und dann seine Daten dort hineinzwängen. Beispielsweise sollte man sich in diesem Beispiel davor hüten, die Daten in XHTML oder DocBook zu repräsentieren. Das wäre zwar die Aufbereitung leicht machen, aber man müsste die in den Standards klar definierten Tags sinnentfremden. Das ist eine der XML-Todsünden, denn es geht ja gerade um die Freiheit, die Tags passend zur Bedeutung des Inhalts wählen zu können.

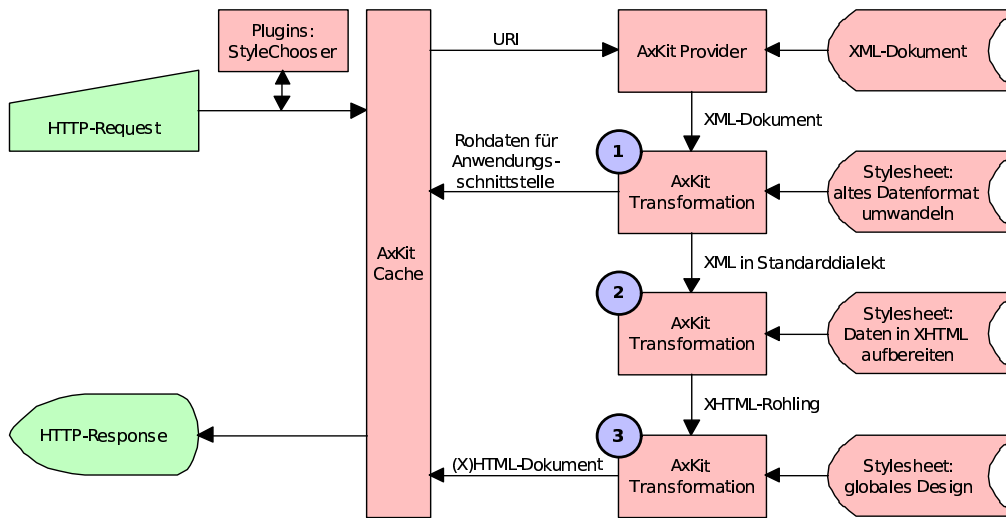
Die schon existierenden Daten werden durch ein kleines XSLT-Stylesheet am Anfang der Verarbeitungskette in den gewählten neuen Dialekt umgewandelt. Fachbereichsspezifische Informationen werden in einem nächsten Schritt in ein menschenlesbares Format aufbereitet.

2. Das Design ist ganz klar Teil der Transformationspipeline. Das globale Seitendesign ist dabei am zweckmäßigsten der letzte Verarbeitungsschritt. Wichtig für die Zusammenarbeit mit einem externen Dienstleister ist die klare Definition der XML-Quelldaten, die das Design-Stylesheet als Eingabe bekommt. Die Erstellung eines Dummy-Layouts hilft bei der Konsolidierung dieser Schnittstelle.
3. Da es sich um ein In-House-System handelt, wird auf eine integrierte Weboberfläche verzichtet und (S)FTP oder WebDAV als Zugangsmechanismus verwendet. Die ohnehin vorhandene Zugriffskontrolle des Betriebssystems wird hierbei zur Zugangssteuerung verwendet. Die Dateien sollen mit einem Texteditor oder einer auf die Datenstruktur angepassten Standardsoftware bearbeitet werden und werden dann serverseitig nur noch per DTD- oder Schema-Validierung geprüft. Vorzugsweise soll aber die Client-Software auch schon diese Überprüfung vornehmen.
4. Das Browser-Design wurde ja schon behandelt. Zwei alternative letzte Transformationsschritte stellen Mobilgeräte- und Behinderten-Version bereit. Das zu Punkt 2 angesprochene Dummy-Design könnte tatsächlich sogar eines dieser beiden Designs sein. Alternativ könnte der externe Dienstleister auch die Mobilversion liefern. Die Version für automatisierten Datenaustausch wird durch das direkte Ausliefern der XML-Rohdaten realisiert.

Alle vier Varianten werden über ein Plugin gewählt. Es wird dabei über die URL bestimmt, welche Variante gewünscht wurde.

5. Eine Dokumentation des Systems ist selbstverständlich. Daß sich ungeübte Kräfte in akzeptabler Zeit einarbeiten können demonstriert dieser Vortrag. Die konsequente Verwendung von XML sorgt dafür, daß auch Programmierer mit anderen Schwerpunkten als Perl selten mit Perl-Code in Berührung kommen. Ein robustes

Abbildung 3.1.: Transformationspipeline für das Informationssystem



Grundwissen in XML und ggf. XSLT reichen für viele Wartungsarbeiten aus, und Beispieldokumente werden den Einstieg weiter erleichtern.

6. Provider stellen die allgemeine Form von Abfragemechanismen dar, also sollten direkte Zugriffe auf Datenbestände anderer Institute über diese realisiert werden. Für den Fall, daß fremde Daten einfach importiert werden sollen, kann wie schon bei Punkt 1 ein XSLT-Stylesheet die Umwandlung vornehmen. Das hat zudem den Vorteil, daß sich diese fremden Datenbestände regelmäßig synchronisieren lassen, da die Quelldaten unberührt bleiben.
7. Der in Punkt 1 gewählte Dialekt wird Sprachkennzeichnungen enthalten. Zusätzlich werden alle Texte (auch in Stylesheets) per `xml:lang` gekennzeichnet. Genutzt werden diese Informationen jedoch vorerst nicht, sie werden nur an relevanten Stellen angezeigt.

Die Transformationspipeline wird daher in etwa wie in Abb. 3.1 aussehen. An den mit 1-3 gekennzeichneten Stellen können später weitere Stylesheets für zusätzliche Funktionen eingefügt werden: Position (1) ist für Datenkonverter für Daten anderer Institute, Position (2) ist für die Aufbereitung fachbereichsspezifischer Informationen, und Position (3) kennzeichnet das globale Stylesheet, das je nach gewünschter Fassung ausgewechselt wird.

Das praktische an XSLT ist, daß durch die Verwendung von Namespaces die Umwandlung von Daten nur dann stattfindet, wenn sie auch tatsächlich in dem benötigten Eingabeformat vorliegen. Mit ein paar kleinen Vorkehrungen kann das XML-Quelldokument sogar ein XHTML-Rohling sein (beispielsweise für Hilfeseiten, die Startseite u.ä.) und passiert die ersten Stufen der Transformation unverändert.

Folgende Teile werden insgesamt benötigt:

- Drei Stylesheets für das globale Design
- Ein Stylesheet für die Konvertierung des alten Datenformats
- Ein Stylesheet, das die Daten in schlichtes XHTML umwandelt; hierbei müssen aber noch ein paar semantische Informationen verbleiben, um den Designern ihre Arbeit zu ermöglichen
- Ein Beispiel-Stylesheet zur Aufbereitung von fachbereichsspezifischen Informationen in XHTML
- Ein Schema für das neue Datenformat als DTD oder XML-Schema
- Ein Plugin für die Wahl der Darreichungsform
- Eine Dokumentation des gesamten Systems
- Eigene Providerklassen zum Zugriff auf externe Datenbestände (zunächst nicht nötig)
- Ein Programm zum Bearbeiten der Quelldaten nebst FTP/WebDAV-Zugriffsmöglichkeit und Validierung, dazu ggf. ein Skript zur serverseitigen Validierung
- Die Konfiguration, die alle Teile verbindet

3.3. Die Details

Unter all den benötigten Bestandteilen gibt es einige, die schon existieren. Texteditoren mit (S)FTP-Fähigkeiten und XML-Funktionalitäten sind reichlich vorhanden. Wer ein wenig sucht, wird Editoren finden, die aus XML-Schemata Eingabemasken erzeugen und teilweise WYSIWYG-Fähigkeiten haben. Die XML-Exportfähigkeiten der Microsoft-Office-Suite, speziell Access als Datenbank, kann hier auch sehr nützliche Dienste leisten. Die Wahl und ggf. Anpassung der Client-Software wird aufgrund der reichhaltigen Alternativen hier nicht näher behandelt.

Die benötigten Plugins zur Steuerung der Transformation sind im CPAN-Archiv unter den Stichwort „StyleChooser“ zu finden, und die Dokumentation existiert in Form dieses Dokuments.

Was also effektiv benötigt wird, sind einige XSLT-Stylesheets und ein paar Quelldateien mit einem gemeinsamen Schema, sowie die nötigen Einstellungen in der Apache-Konfiguration. Im großen und ganzen ergibt das ein Informationssystem mit fortgeschrittenen Funktionen, und doch liegt der Schwerpunkt im den Kern der Aufgabe: dem Sammeln, Organisieren und Veröffentlichen von Daten. Es ist in der jetzigen Projektphase nicht eine Zeile Perl im Spiel.

3.3.1. XSLT: das globale Design

Das Dummy-Design soll schlicht aussehen: Der eigentliche Textinhalt in der Mitte, darüber das Logo des Instituts, links ein Menü mit Verknüpfungen zur Startseite und zum Katalog. Als Eingabedaten bekommt es einen XHTML-Rohling, also einen schlichten Artikel im XHTML-Format ohne jegliche gestalterischen Elemente. Folgendes Stylesheet erfüllt den gewünschten Zweck ganz hervorragend:

```
<xsl:stylesheet version='1.0'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
  xmlns:html='http://www.w3.org/1999/xhtml'
  xmlns='http://www.w3.org/1999/xhtml' >

<xsl:output method="html" encoding="UTF-8"
  doctype-public="-//W3C//DTD HTML 4.01 Strict//EN"
  doctype-system="http://www.w3.org/TR/html4/strict.dtd"/>

<xsl:template match='/'>
  <html>
    <head>
      <title>
        Kryptozoologie online:
        <xsl:value-of select='/html:html/html:title'/>
      </title>
      <link rel='stylesheet' href='/main.css'/>
    </head>
    <body>
      <div class='head'>
        <img src='/logo.png' alt='Kryptozoologie-Institut Bielefeld'/>
      </div>
      <div class='menu'>
        <a href='/'>Startseite</a>
        <a href='/katalog/'>Katalog</a>
      </div>
      <div class='body'>
        <h1><xsl:value-of select='/html:html/html:title'/></h1>
```

```

        <xsl:apply-templates select='/html:html/html:body' />
    </div>
</body>
</html>
</xsl:template>

<xsl:template match='html:body'><xsl:apply-templates /></xsl:template>

</xsl:stylesheet>

```

Selbst wer sich mit XSLT noch nicht so intensiv auseinandergesetzt hat, wird mit Leichtigkeit verstehen, was hier passiert. Es ist wohl erwähnenswert, daß CSS eingesetzt wird: es ist das richtige Werkzeug für den gewünschten Zweck. Reine CSS-basierte Layouts sind heutzutage in der Praxis realisierbar - es sind nur wenige Workarounds nötig, und das Resultat funktioniert browserübergreifend: Mozilla und Konqueror haben hervorragende CSS2-Kompatibilität, und die wenigen störenden Lücken beim Internet Explorer lassen sich durch dessen proprietäre JavaScript-Ausdrücke umgehen.

Dieses XSLT-Stylesheet bringt eigentlich alle Voraussetzungen mit, um als Mobil- oder Behindertenversion genutzt zu werden, das zugehörige CSS-Stylesheet kann Dinge wie Layout und Schriftgrößen passend festlegen. Daher wird auf diese beiden Stylesheets verzichtet - sie können genauso aussehen wie dieses minimalistische Stylesheet.

3.3.2. XML: Das Schema

Um die Nutzdaten in XHTML umwandeln zu können, wird nun das neue XML-Vokabular für unsere Nutzdaten benötigt. Wie schon erwähnt sollte man die Interoperabilität eines Vokabulars gegenüber der Ausdrucksfähigkeit und der Gefahr von semantischen Inkorrektheiten abwägen. In diesem fiktiven Fall gibt es eine andere fiktive Forschergruppe, die schon kryptozoologische Daten gesammelt und dazu einen XML-Dialekt² entworfen hat. Das ist besser als nichts, und es passt zu den Anforderungen, also ist das die beste Wahl.

Da die Pflege der Daten über externe Programme erfolgen soll, wird die Validierung nicht weiter erläutert. Daher bleibt die genaue Spezifikation und das Schreiben eines Schemas dem fiktiven Entwicklerteam vorbehalten, ein Beispiel soll hier zur Verdeutlichung ausreichen:

```

<?xml version='1.0'?>
<cryptids xmlns='http://garni.ch/2005/demo'>
  <species xml:lang='de'>
    <name>Dahut</name>
  </species>
</cryptids>

```

²Der nicht fiktiv ist, sondern aus dem AxKit-Buch (S. 28) stammt.

```

<habitat>französische Alpen</habitat>
<description>
  <para>
    Das Dahut als scheuer Vertreter des alpinen Rotwilds hat sich
    auf einzigartige Weise den Herausforderungen des Berglebens
    angepasst, indem seine Beine auf einer Seite deutlich länger
    wachsen als auf der anderen. Dieser asymmetrische Körperbau
    erlaubt leichtes Grasen an steilen Hängen, verhindert jedoch,
    daß sich das unglückliche Geschöpf umdrehen kann. Jäger nutzen
    diesen Umstand aus, indem sie sich von hinten an ein Dahut
    anschleichen und entweder leise flüsternd oder laut "Dahut!"
    rufen. Wenn das verwirrte Tier sich dann umdreht, um den
    Angreifer zu sehen, enden die langen Beine auf der falschen
    Seite und es stürzt hinab in sein Schicksal.
  </para>
</description>
</species>
</cryptids>

```

Das fiktive Forscherteam hat noch einen zweiten Dokumententyp entworfen, der in einer 1:n relation zu den SPECIES-Einträgen steht. Es handelt sich hierbei um Sightungen solcher Kryptiden:

```

<?xml version='1.0'?>
<sightings xmlns='http://garni.ch/2005/demo'>
  <sighting xml:lang='de'>
    <name>Jersey-Teufel</name>
    <location>Bordentown, New Jersey, USA</location>
    <date>Herbst 1816</date>
    <description>
      <para>
        Ein Jersey-Teufel wurde berichten zufolge von Joseph Bonaparte,
        ehemaliger König von Spanien und Bruder Napoleons, während einer
        Jagd in den Wäldern nahe Bordentown, New Jersey, USA gesehen.
      </para>
    </description>
    <witnesses>
      <name>Joseph Bonaparte</name>
    </witnesses>
  </sighting>
</sightings>

```

Es gibt also sogar ein einfaches Relationsmodell der Daten. Solange die Relationen zwischen Datenobjekten nicht zu komplex werden bzw. keine zu komplexen Abfragen über

den Datenstamm benötigt werden, kann eine Sammlung von XML-Dateien eine relationale Datenbank problemlos ersetzen - und im Bereich Web-Publishing sind die Abfragen üblicherweise trivial.

3.3.3. XSLT: XHTML-Aufbereitung

Obwohl zwei unterschiedlichen Dokumentenarten behandelt werden müssen, reicht hierfür ein XSLT-Stylesheet aus. Dies ist sogar ratsam, da dadurch die Gleichbehandlung gleicher Tags leichter wird. Aus diesem Grund ist auch die exakte Definition der Bedeutung eines Tags wichtig: der <NAME>-Tag z.B. wird immer den Namen eines Kryptiden kennzeichnen. Würde dieser Tag auch nur leicht sinnentfremdet verwendet werden, dann wären alle Vorteile der Gleichbehandlung verloren. Gleiches gilt für den Fall, wenn man einen anderen XML-Dialekt gewählt hätte und dort Tags sinnentfremdet hätte.

Eine Eigenschaft dieser Dateien wird hier wichtig: Dieses Dateiformat erlaubt das Sammeln mehrerer Einträge in einer Datei. Das geplante System geht bisher von einer Datei pro Kryptiden aus, doch sollen alle Sichtungen in einer einzigen Datei namens SIGHTINGS.XML gesammelt werden. Warum das so ist, wird das nun folgende Stylesheet zeigen, das einen Kryptiden zusammen mit seinen Sichtungen aufbereitet:

```
<xsl:stylesheet version='1.0'
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
  xmlns:cryptid='http://garni.ch/2005/demo'
  xmlns='http://www.w3.org/1999/xhtml' >

<xsl:template match='cryptid:cryptids'>
  <html>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>

<xsl:template match='cryptid:species'>
  <div class='cryptid'>
    <h1>
      <xsl:value-of select='cryptid:name'/>
      <span class='habitat'>
        (<xsl:value-of select='cryptid:habitat'/>)
      </span>
    </h1>
    <h2>Beschreibung</h2>
    <xsl:apply-templates select='cryptid:description'/>
  <xsl:apply-templates
```

```

select='document("sightings.xml",/)/*/sighting[cryptid:name=current()/cryptid:name]'/>
  </div>
</xsl:template>

<xsl:template match='cryptid:sighting'>
  <div class='sighting'>
    <h2>Augenzeugen-Bericht</h2>
    <p class='witness'>
      von
      <xsl:for-each select='cryptid:witnesses/*'>
        <xsl:value-of select='.'/>
        <xsl:choose>
          <xsl:when test='position() = last()'/>
          <xsl:when test='position() = last()-1'> und </xsl:when>
          <xsl:otherwise>, </xsl:otherwise>
        </xsl:choose>
      </xsl:for-each>,
      <xsl:value-of select='cryptid:date'/> in
      <xsl:value-of select='cryptid:location'/>
    </p>
    <xsl:apply-templates select='cryptid:description'/>
  </div>
</xsl:template>

<xsl:template match='cryptid:para'>
  <p class='description'><xsl:apply-templates/></p>
</xsl:template>
<xsl:template match='cryptid:*'><xsl:apply-templates/></xsl:template>
<xsl:template match="*">
  <xsl:copy select=".">
    <xsl:copy-of select="@*|namespace::*"/>
    <xsl:apply-templates/>
  </xsl:copy>
</xsl:template>

</xsl:stylesheet>

```

Hier wird eine einfache Verknüpfung der Daten aus dem angefragten Dokument und den gesammelten Sichtungen gebildet. Man kann hieran auch einige Arbeitsprinzipien von XSLT zeigen, jedoch führt das zu weit vom Thema weg. XSLT ist ein hervorragendes Werkzeug mit vielen Möglichkeiten, im Anhang sind ein paar Links zu frei erhältlichen Informationen hierzu gesammelt.

Das letzte Template dient übrigens dazu, daß alle Tags, die nicht erkannt wurden,

unverändert durchgereicht werden. Dies ist immer ratsam, wenn man mehrere XSL-Transformationen nacheinander durchführt.

3.3.4. XSLT: Umwandlung des alten Datenformats

Dieses Stylesheet wird für eine Demonstration nicht benötigt und enthält gegenüber den schon vorgestellten Stylesheets keine Besonderheiten, daher wird hier auf einen Abdruck verzichtet. Wichtig ist aber, wie man eine solche Funktionalität in AxKit konfiguriert, daher bleibt es Teil des Informationssystems.

3.3.5. Die Konfiguration

Nun sind alle Teile vorhanden, um die Grundfunktionen in Betrieb zu nehmen. Folgender Block soll der Webserver-Konfiguration hinzugefügt werden:

```
PerlModule AxKit
AddHandler axkit .xml .xsp .xhtml .dkb
AxAddStyleMap text/xsl Apache::AxKit::Language::LibXSLT
AxAddStyleMap application/x-xsp Apache::AxKit::Language::XSP

AxAddUriProcessor application/x-xsp NULL \.xsp$
AxAddProcessor text/xsl file:///home/example/styles/datenformat.xml
AxAddProcessor text/xsl file:///home/example/styles/cryptid.xml
AxAddProcessor text/xsl file:///home/example/styles/global.xml
```

Das reicht aus, um die XML-Dokumente aufbereitet im Standardlayout auszuliefern.

Was in der ersten Stufe des Informationssystems nun noch fehlt, ist die Wahlmöglichkeit zwischen mehreren Designs. Dies ist mit dem mitgelieferten Plugin `APACHE::AXKIT::STYLECHOOSER` leicht möglich. Der letzte Block muss entfernt und durch folgende Zeilen ausgetauscht werden:

```
AxAddPlugin Apache::AxKit::StyleChooser::QueryString

<AxStyleName "#global">
  AxAddUriProcessor application/x-xsp NULL \.xsp$
</AxStyleName>

<AxStyleName "#default">
  AxAddProcessor text/xsl file:///home/example/styles/cryptid.xml
  AxAddProcessor text/xsl file:///home/example/styles/global.xml
</AxStyleName>
```

```

<AxStyleName "mobile">
  AxAddProcessor text/xsl file:///home/example/styles/cryptid.xsl
  AxAddProcessor text/xsl file:///home/example/styles/global-mobile.xsl
</AxStyleName>

<AxStyleName "print">
  AxAddProcessor text/xsl file:///home/example/styles/cryptid.xsl
  AxAddProcessor text/xsl file:///home/example/styles/global-print.xsl
</AxStyleName>

<AxStyleName "raw">
</AxStyleName>

```

Hier werden die Stylesheets mit Namen versehen. Der Name `#GLOBAL` kennzeichnet Stylesheets, die auf jeden Fall angewendet werden sollen. Der Name `#DEFAULT` kennzeichnet die Stylesheets, die angewendet werden sollen, wenn kein oder ein nichtexistenter Name ausgewählt wurde. Das Plugin erledigt den Rest: Die Auswahl geschieht z.B. über den Zusatz `?STYLE=PRINT` in der URL.

3.4. Zusammenfassung

Die Anforderungen an das Informationssystem wurden nun erfüllt. Das Programm zum Bearbeiten der Daten und der Uploadmechanismus wurde dabei in diesem Vortrag ausgelassen, wäre aber bei Wahl einer Standardsoftware inklusive der Erstellung eines Schemas zur Validierung in kurzer Zeit einsatzbereit. Die gewünschte Zugangskontrolle würde durch den (S)FTP-Server erfolgen, z.B. durch klassische Unix-Dateirechte. Erwähnenswert wäre in diesem Zusammenhang, daß AxKit auch unter Apache auf Windows-Systemen läuft, was weitere Optionen der Interaktion mit Betriebssystemdiensten ermöglicht.

Das System ist natürlich noch unvollständig - was demonstriert wurde, war der Kern, der die eigentliche Arbeit erledigt. Die Webpräsenz um das Informationssystem herum ist davon nicht erfasst. Es ist natürlich ratsam, diese auch mit AxKit aufzubauen, um Arbeit zu sparen, doch bleibt hier die freie Wahl. Es kann auch eine existierende, statische HTML-Webpräsenz weiter genutzt werden, nur erweitert um ein paar Links zur Navigation. Vorausschauend enthält die Konfiguration aber schon die Behandlung von Dateien der Endung `.XHTML`, und wie schon zuvor erwähnt stört das Kryptiden-Stylesheet die Verarbeitung von anderen Datenformaten überhaupt nicht. Die Nutzung von AxKit für statische Bestandteile ist also ohne weitere Vorkehrungen möglich.

4. Die Fäden verbinden...

Technologiemix als Kernprinzip

Die Konfigurationsdirektiven aus dem vorigen Kapitel zeigen schon, daß die Technologien, die zur Transformation genutzt werden, nicht festgelegt sind. Die Modularität von AxKit hilft nicht nur intern bei der Wahl einer Stylesheet-Sprache, sondern auch bei der Integration in existierende Systeme. AxKit lebt in einer Welt voller existierender Praktiken, Standards und Programmen, daher ist jeder Einsatz einer Software im Prinzip die Integration in ein existierendes System. Die Welt außerhalb von AxKit ist Thema dieses Kapitels, gleichzeitig wird gezeigt, wo AxKit geeignete Mechanismen bereitstellt.

Es fallen in dem Informationssystem aus dem letzten Kapitel schnell mehrere Kritikpunkte auf. Folgende Probleme dienen zur Demonstration hierzu:

- Es gibt keinen automatischen Index, kein Inhaltsverzeichnis aller Kryptiden
- Die alternativen Ausgabeformate können nur recht aufwendig verlinkt werden: Alle Links müssen den URI-Zusatz enthalten, damit man nicht wieder ungewollt bei der Standardversion landet.
- Eine Suchfunktion wurde in den Anforderungen nicht erwähnt, ist jedoch eine übliche Funktion.

4.1. AxKit und Apache

Apache bringt von Haus aus schon viele Funktionen mit, um Webseiten zu strukturieren. Neben dem Umstand, daß z.B. <FILES>-Sektionen mit allen AxKit-Direktiven zusammenarbeiten gibt es aber auch einige Module, die für AxKit sehr nützlich sind.

Für einen manuell erstellten Index der Kryptiden bietet sich folgendes an:

```
DirectoryIndex index.xhtml
```

Damit werden nun auch XHTML-Dateien als Kandidaten für Index-Dokumente herangezogen. Wie man einen automatischen Index erstellen kann, wird gleich gezeigt.

Sehr interessant für die XML-Verarbeitung ist außerdem `MOD_REWRITE`, das Universalwerkzeug zur URL-Manipulation. Es ermöglicht z.B. die einfache Erstellung von URLs

ohne Dateitypenerweiterungen, wie es sogar vom W3C empfohlen wird. Eine andere sehr nützliche Funktion hilft aber nun, um die Linkstruktur des Informationssystems zu verbessern: Statt URLs wie `HTTP://WWW.EXAMPLE.COM/CRYPTIDS/DAHUT.XML?STYLE=PRINT` wäre es praktisch, virtuelle URLs wie `HTTP://WWW.EXAMPLE.COM/PRINT/CRYPTIDS/DAHUT.XML` zu haben. Das würde die Verlinkung wesentlich erleichtern, da ganz normale relative Links das gewählte Ausgabeformat beibehalten würden:

```
RewriteEngine On
RewriteRule ^/print(/.*)$ $1?style=print
RewriteRule ^/mobile(/.*)$ $1?style=mobile
RewriteRule ^/raw(/.*)$ $1?style=raw
```

4.2. AxKit und Perl

Jeder, der Apache etwas näher kennt, kann ohne Perl-Kenntnisse etliches erreichen. Wer hingegen Perl besser als Apache kennt, kann folgende Vorgehensweise für den gleichen Effekt nehmen:

1. Symlinks mit den Namen `PRINT`, `MOBILE` und `RAW` liegen im Dokumentenroot und zeigen nach „.“, also auf das selbe Verzeichnis.
2. Ein alternativer `StyleChooser` wird geschrieben (und in den Suchpfad für Module eingebunden), der den ersten Teil der URL auswertet:

```
package Apache::AxKit::StyleChooser::VirtualDir;
use strict;
sub handler {
    my $r = shift;
    my $style = $r->uri;
    $style =~ s{/+}{};
    $style =~ s{/.*}{};
    $r->notes('preferred_style', $style);
    return Apache::Constants::OK;
}
```

3. In der Apache-Konfiguration wird der `StyleChooser` durch den neu geschriebenen ersetzt.

Generell funktionieren alle Plugins so: Sie definieren eine Funktion `HANDLER`, die direkt vor der Verarbeitung durch AxKit aufgerufen wird, bekommen das Request-Objekt (üblicherweise als `$R` bekannt) als einzigen Parameter übergeben, dürfen dies nach gutdünken verändern und müssen als Zeichen der erfolgreichen Verarbeitung `APACHE::CONSTANTS::OK` zurückgeben.

4.3. AxKit und Sprachen

AxKit als Mittel zum Web-Publishing muß potentiell mit Besuchern und Programmierern aus der ganzen Welt arbeiten können. Neben der schon angedeuteten Sprachkennzeichnung in XML enthält der Standard auch saubere und stabile Handhabung von Zeichenkodierungen. Unicode als Zeichensatz¹ sorgt intern dafür, daß es keine Probleme oder Fehler bei der Verarbeitung von Texten geht. Es gibt jedoch Quellen, deren Zeichenkodierung unbekannt ist. In URLs ist es beispielsweise noch nicht überall selbstverständlich, daß die Kodierung in jedem Fall UTF-8 sein soll - auch wenn dies von den relevanten RFCs schon länger empfohlen wird. Ähnliches gilt für Formulardaten. Aber auch hausgemachte Probleme warten auf den unvorsichtigen Webmaster: Was passiert, wenn der Name eines Dokuments Umlaute in der ISO-8859-15 (Westeuropäisch mit Eurozeichen) Zeichenkodierung enthält? Die URL muss dann ja zwangsläufig auch ISO-8859-15-kodiert sein - und was ist dann mit Formulardaten in der URL?

Man sieht also, hier lauern etliche Probleme gerade für europäische Nutzer. Die ideale Welt, in der alles UTF-8 ist, ist leider weit entfernt. Dennoch avanciert diese Kodierung zu einem Quasi-Standard: XML ohne weitere Angaben ist in UTF-8, URLs sollten so kodiert werden, und auch Perl nutzt es intern - das auf Microsoft-Plattformen gebräuchliche UTF-16 findet im Internet wenig Anklang.

UTF-8 hat dabei eine interessante und sehr vorteilhafte Eigenschaft: Es ist leicht zu erkennen. Ein in einer anderen Kodierung geschriebener Text ist nahezu unmöglich gültiges UTF-8, und ein UTF-8-Text macht in einer anderen Kodierung keinen Sinn. Diese Eigenschaft nutzt AxKit aus, um das Zeichensatzproblem zu lösen. Mit

```
AxExternalEncoding ISO-8859-15
```

in der Apache-Konfiguration wird AxKit zunächst alle Texte, deren Kodierung nicht bekannt ist, als UTF-8 interpretieren. Schlägt das fehl, dann wird der betroffene Text stattdessen mit der angegebenen Kodierung in UTF-8 umgewandelt. Somit ist es egal, wie genau der Browser die Daten sendet. Der Nachteil ist, daß nur eine Zeichenkodierung angegeben werden kann. Dieses Verfahren kann man also nur anwenden, wenn man weiß, woher die Besucher voraussichtlich kommen werden.

4.4. AxKit und das Betriebssystem

Um ein automatisches Verzeichnis zu erstellen, müsste man ja eigentlich nur die Liste der Dateien in einem Verzeichnis in irgendeinem XML-Format haben, dann könnte man diese Liste mit XSLT in ein verlinktes Inhaltsverzeichnis umwandeln. Wie schon in Kapitel 1.3 auf Seite 9 erklärt wurde, dienen Provider dazu, die Nutzdaten als XML an

¹Man beachte den feinen Unterschied zwischen dem Zeichensatz, der Menge verfügbarer Zeichen, und der Kodierung, also der Bytefolgen, die einem Zeichen zugeordnet sind.

AxKit zu liefern. Man könnte also einen Provider schreiben, der das Verzeichnis ausliest und eine XML-Liste erzeugt. Der Standardprovider macht genau das sogar schon. Der XPath-Ausdruck `DOCUMENT('.',/)` liefert beispielsweise das Verzeichnis, in dem das XML-Dokument liegt, in einem vom AxKit-Team (genauer gesagt, dem Autor dieses Vortrags) entworfenen XML-Dialekt zurück, inklusive aller relevanten Metainformationen (Größe, Zugriffsrechte usw.).

Die `DOCUMENT`-Funktion ist für die Verwendung in XSL, wenn man also schon ein anderes Dokument transformiert. Das ist z.B. nützlich für ein Seitenmenü. Wenn das Verzeichnis selbst aber zu einem alleinstehenden Index umgewandelt werden soll, dann muß man ein paar Vorkehrungen treffen:

1. `MOD_AUTOINDEX` und `MOD_DIR` müssen deaktiviert oder passend konfiguriert werden, weil sie Verzeichnis-URLs schon umschreiben, bevor Axkit überhaupt zum Zuge kommt.
2. AxKit muß mitgeteilt werden, daß Verzeichnisse verarbeitet werden dürfen (was aus Sicherheitsgründen standardmäßig deaktiviert ist):

```
AxHandleDirs On
```

Dann erhält man für Verzeichnis-URLs ein Dokument wie z.B. dieses:

```
<?xml version="1.0" encoding="UTF-8"?>
<filelist xmlns="http://axkit.org/2002/filelist">
  <directory size="512"
    atime="1107512030" mtime="1107511077" ctime="1107511077"
    readable="1" writable="1" executable="1">.</directory>
  <directory size="512"
    atime="1107512030" mtime="1107511077" ctime="1107511077"
    readable="1" writable="1" executable="1">..</directory>
  <file size="995"
    atime="1107512030" mtime="1107511077" ctime="1107511077"
    readable="1">dahut.xml</file>
</filelist>
```

Dies kann dann wieder mit den schon bekannten Mechanismen in ein ganz normales HTML-Dokument umgewandelt werden. Die Zeichenkodierung von Dateinamen wird übrigens wie im vorigen Abschnitt umgewandelt.

4.5. AxKit und externe Programme

Es ist in Apache oft problematisch, aber hin und wieder kommt man um den Aufruf externer Programme nicht herum. Das PDF-Transformationsmodul `APACHE::AXKIT::LANGUAGE::HTMLDOC`

ist so ein Beispiel. Aber auch für selbstgeschriebene Programme gibt es gelegentlich gute Gründe für den Aufruf externer Software.

Wenn man zum Beispiel eine Volltextsuche über alle Dateien machen will, könnte man einen naiven Prototypen mit dem Unix-Befehl `grep` implementieren. Das XSP-Dokument, das diese Funktion bereitstellt, könnte z.B. so aussehen:

```
<xsp:page language='Perl'
          xmlns:xsp='http://www.apache.org/1999/XSP/Core'
          xmlns='http://www.w3.org/1999/xhtml'>
<html>
  <head>
    <title>Suche</title>
  </head>
  <body>
    <h1>Suche</h1>
    <p>Suchbegriff: (ein Wort, keine Sonderzeichen)</p>
    <form>
      <input type='text' name='query' value='{$cgi->param("query")}'/>
      <input type='submit' value='Suchen'/>
    </form>
    <xsp:logic>
      my $query = $cgi->param('query');
      if (defined $query and $query !~ m/^(\\w|_|\\.|-)+$/) {
        <p class='error'>
          Bitte nur ein Wort eingeben und keine Sonderzeichen verwenden!
        </p>
      } elsif (defined $cgi->param('query')) {
        $query =~ s/\\.\\.\\.\\/g;
        <h2>Suchergebnisse</h2>
        foreach (qx(grep -i -l '$query' *.xml)) {
          <a href='{$_}'><xsp:expr>$_</xsp:expr></a>
        }
      }
    </xsp:logic>
  </body>
</html>
</xsp:page>
```

Dieses Verfahren hat natürlich gravierende Mängel (speziell die mangelnde Flexibilität und die zweifelhafte Geschwindigkeit), dennoch ist es schnell geschrieben und kann sofort ausprobiert werden - genau das, was ein Prototyp leisten soll. Besonders bestechend ist der Umstand, daß hierzu keine besonderen Module und keine zusätzliche Software nötig ist - man benutzt wohlbekanntere Komponenten, die ohnehin vorhanden sind, und die Änderungen am gesamten System beschränken sich auf eine einzige, autonome Datei.

4.5.1. Provider

Sollte diese Evaluierung ergeben, daß GREP tatsächlich das richtige Werkzeug für diese Aufgabe ist², könnte man dieses XSP so umschreiben, daß es eine abstrakte XML-Dateiliste erzeugt - vielleicht genauso wie AxKit's Dateilisten - und die visuelle Aufbereitung einem nachfolgenden XSLT überlassen. Wie in Kapitel 1.1 schon gezeigt wurde, dienen aber eigentlich die Provider der Bereitstellung der Quelldaten. Nichts leichter als das:

```
package Apache::AxKit::Provider::Grep;
use strict;
use AxKit;
use base 'Apache::AxKit::Provider';
sub init {
    my $self = shift;
    my $r = $self->apache_request;
    my $query = AxKit::ToUTF8($r->uri);
    $query =~ s{.*/}{};
    $self->{query} = $query;
    my $files = $r->dir_config('AxGrepFilemask');
    if (defined $query && $query !~ m/^(\\w|_|\\.|-)+$/) {
        $self->{data} = '<error/>';
        $self->{query} = '';
    } elsif (defined $query) {
        $self->{data} = "<filelist xmlns='http://axkit.org/2002/filelist' query='$query'>";
        foreach (qx(grep -i -l '$query' $files)) {
            $self->{data} .= '<file>'.AxKit::ToUTF8($_).</file>';
        }
        $self->{data} .= '</filelist>';
    } else {
        $self->{data} = '<no-query/>';
    }
}
}
use constant process => 1;
use constant exists => 1;
use constant has_changed => 1;
sub get_strref {
    my $self = shift;
    return \$self->{data};
}
}
sub key {
```

²Das klingt unwahrscheinlich, aber man darf nie die Projektvoraussetzungen vergessen. GREP ist die schnellste Methode, nicht indizierte Texte zu durchsuchen, und das könnte die Nachteile überwiegen - es kommt eben darauf an, was gefragt ist.

```

    my $self = shift;
    return __PACKAGE__.' ' . $self->{query};
}
1;

```

Dieser Provider durchsucht die Dateien, die per PERLSETVAR AXGREPFILEMASK ausgewählt wurden, nach dem einzelnen Wort, das als PATH_INFO in der URL steht und erzeugt eine Dateiliste ähnlich den AxKit-Verzeichnissen. Eine Beispielkonfiguration:

```

<Location /query/>
AxContentProvider Apache::AxKit::Provider::Grep
PerlSetVar AxGrepFilemask /home/example/htdocs/criptids/*.xml
</Location>

```

Hiermit würde z.B. die Anfrage nach HTTP://WWW.EXAMPLE.COM/QUERY/DAHUT eine Suche nach dem Wort Dahut auslösen. Das Verzeichnis für /QUERY/ sollte allerdings existieren.

Ohne allzusehr in die Details einzugehen (die Dokumentation von AxKit oder das AxKit-Buch erklärt die Funktion von Providern genau) wird in der INIT-Methode die eigentliche Arbeit erledigt, GET_STRREF wird aufgerufen, wenn die eigentlichen Daten benötigt werden (durchaus mehrmals pro Request), während die anderen Methoden die Zuständigkeit und das Caching steuern. Setzt man HAS_CHANGED konstant auf 0, wird der AxKit-Cache alle Seiten, die aus den Daten dieses Providers erzeugt werden, für immer zwischenspeichern.

4.6. AxKit und Datenbanken

Um eine effizientere Suchfunktion anzubieten, liegt natürlich ein datenbankbasierter Index nahe. Der Zugriff auf Datenbanken generell kann an verschiedenen Stellen passieren. Neben den naheliegenden XSP und Providern wie im vorigen Abschnitt demonstriert gibt es noch eine Transformationssprache, die XPathScript getauft wurde und XML mit Hilfe von Perl-Anweisungen transformiert.

Aus Mangel an praktischer Erfahrung des Autors muss hier leider auf ein Beispiel für XPathScript verzichtet werden, doch gehört diese Sprache zum Lieferumfang von AxKit, daher gebührt ihr wenigstens eine Erwähnung.

Für XSP gibt es die ESQL-Erweiterung, die jedoch ursprünglich aus der Java-Welt stammt und für eingefleischte Perl-Anwender eher unhandlich wirkt.

Ein Datenbank-Provider ist analog zu dem Grep-Beispiel leicht erstellt, man nutzt dazu ganz herkömmlich die DBI-API. Es gibt vom Autor dieses Vortrags auch den fertigen Provider APACHE::AXKIT::PROVIDER::SQL, der per PERLSETVAR mit den SQL-Abfragen

konfiguriert werden kann und so ohne weitere Perl-Programmierung einsatzbereit ist. Dies ist die beste Methode, Dokumente aus Datenbankinhalten zu generieren, zumal dieser Provider auch korrektes Caching ermöglicht, ein Punkt der niemals vergessen werden sollte, denn der AxKit-Cache kann - sofern die Inhalte tatsächlich für den Cache geeignet sind - die von einem System erreichbaren Zugriffszahlen leicht verzehnfachen. Der eigens optimierte Zugriffspfad auf den Cache stellt alle anderen Servertechniken in den Schatten und erreicht ca. 70% der Geschwindigkeit von statischem HTML. Die 30% Verlust entstehen dabei durch das umfassende Überprüfen aller Abhängigkeiten, wie schon in der Einführung auf Seite 1.3 beschrieben.

Wer besonders modern sein möchte, kann auch zu einer XML-Datenbank greifen. Das ist in gewisser Weise die natürliche Kombination für AxKit, auch wenn frühe Implementationen solcher Datenbanken um Größenordnungen langsamer waren als klassische relationale Datenbanken und proprietäre Abfragemechanismen hatten. Mit der XML:DB-Spezifikation gibt es jedoch inzwischen eine sprachübergreifende und datenbankunabhängige API, für die auch eine Perl-Implementation existiert. Von Sleepycat (bekannt durch Berkeley DB) gibt es eine leistungsfähige Serversoftware, und über den Autor dieses Vortrags gibt es auch einen experimentellen Provider für die XML:DB-API.

4.7. AxKit und Benutzereingaben

Benutzereingaben, üblicherweise per HTML-Formular, können auf vielfache Weise verarbeitet werden. Allein für XSP gibt es etliche Lösungen. Neben einigen fortgeschrittenen Schema-basierten Varianten (XForms und XML-Schema wurden leider noch nicht gesichtet, vom Autor dieses Vortrags gibt es aber eine Datenbeschreibungssprache, die RSSDDL getauft wurde und vom Funktionsumfang dicht bei XForms liegt) ist das etablierteste Hilfsmittel `AXKIT::XSP::PERFORM`. Ohne Hilfsmittel ist dies jedoch auch nicht schwer: Oben gab es schon eine Suchfunktion mit Formular, und im nächsten Abschnitt gibt es noch ein einfaches Beispiel zur Demonstration, daß Formulare auch mit wenig Code schon ansprechend funktionieren.

Zweifelsohne bieten die Formular-Hilfsmittel wesentlich komfortablere Methoden, bessere Validierung und mehr Wiederverwendbarkeit, doch sorgt genau das auch zu vielen Formalitäten und einem hohen Startaufwand. Ein vollständiges und vorbildliches Beispiel sprengt daher leider den Rahmen dieses Vortrags.

4.8. AxKit und Netzwerkdienste

Um auf Netzwerkressourcen zuzugreifen, können beispielsweise in XSLT schlicht HTTP-URLs verwendet werden. Dies macht Anfragen natürlich langsam und praktisch unmöglich zu Cachen, ist aber bei einer guten Netzanbindung und geringen Besucherzahlen eine stabile und einfache Lösung. Manche Systeme setzen dieses Verfahren ein, um Daten von einem zweiten, lokal angebundenen Rechner abzurufen, beispielsweise von einem

Datenbankserver. Dieser liefert nur ein XML-Dokument, während AxKit dann die visuelle Aufbereitung erledigt. Auch Anwendungs- und Präsentationsebene lassen sich so elegant trennen und auf mehrere Rechner verteilen. Diese Art des RPC ist auch als REST bekannt (wenn auch REST etwas mehr ist als nur dieser Fall).

Will man Daten senden, per eMail zum Beispiel, oder genauere Kontrolle über HTTP-Anfragen haben, dann bleiben einem wieder die üblichen Stellen für Perl-Code. Geht es um RPC, dann besteht das „senden“ von Daten ja oftmals in der Antwort auf einen RPC-Aufruf. AxKit hat keine eingebauten XML-RPC oder SOAP-Fähigkeiten, hierzu müssen die üblichen Perl-Module genutzt werden. Einzig REST ist ein Verfahren, was AxKit ohne weiteres Zutun beherrscht. In dem fiktiven Informationssystem ist die raw-Variante der Webseiten nichts weiter als ein REST-basiertes RPC-Interface.

Als ein Beispiel einer ausgewogenen, sauberen XSP-Seite sei hier nun also einmal ein Formular gezeigt, über das Besucher des Informationssystems eine Zugangskennung beantragen können. So wie empfohlen wird nur eine (frei erfundene) Datenstruktur erzeugt, die in einem späteren XSLT-Schritt in ein HTML-Dokument gewandelt werden muß. Wird dieses XML nicht weiter umgewandelt, haben wir wiederum ein RPC-Interface zu unserem Informationssystem geschaffen, denn die Bedingungen sind ja einfach: Alles wird über eine URL - einen einzelnen Aufruf - gesteuert und die Antwort ist maschinenlesbar³.

```
<xsp:page language='Perl'
  xmlns:xsp='http://www.apache.org/1999/XSP/Core'
  xmlns:mail='http://axkit.org/NS/xsp/sendmail/v1'
  xmlns:my='http://garni.ch/2005/demo-taglib'>
<result>
  <mail:send-mail>
    <mail:to>admin@example.com</mail:to>
    <mail:from><param:email/></mail:from>
    <mail:subject>Neue Anmeldung</mail:subject>
    <mail:body>
      Neue Anmeldung:
      =====

      Name: <param:name/>
      Abteilung: <param:abteilung/>
      eMail: <param:email/>

      Datum: <my:datum/>
    </mail:body>
  </mail:sendmail>
  Mail wurde versendet.
```

³REST stellt eigentlich noch mehr Anforderungen, doch soll dies zur Einführung reichen. Im Anhang gibt es Verweise auf mehr Informationen.

```
</result>
</xsp:page>
```

4.8.1. Taglibs

Dieses Beispiel dient auch als Demonstration eines weiteren AxKit-Konzeptes: XSP-Taglibs ermöglichen es, beliebige Funktionen für XSP-Seiten als XML-Tags bereitzustellen. Statt `<PARAM:NAME/>` wird also beispielsweise bei der Ausführung der Seite der Formularwert mit dem Namen „name“ eingesetzt. Dahinter steckt reiner Perl-Code, und Taglibs können leicht selber erstellt werden. Hier gibt es mehrere Hilfsmodule, die die Erstellung von Taglibs erleichtern, allen voran `APACHE::AXKIT::LANGUAGE::XSP::SIMPLETAGLIB`, welches eine Unmenge an Funktionen bereitstellt, aber trotzdem sehr schlichten Perl-Code ermöglicht.

Im obigen Dokument kommen die Prefixe `MAIL` und `MY` vor. Ersterer ist durch den Namespace an das via CPAN erhältliche Modul `AXKIT::XSP::SENDMAIL` gebunden, letzterer soll nun durch eine kleine Taglib definiert werden:

```
PerlModule AxKit::XSP::Demo;
use strict;
our $NS = 'http://garni.ch/2005/demo-taglib';
use Apache::AxKit::Language::XSP::SimpleTaglib;
sub datum : XSP_expr { return scalar localtime }
```

Die `$NS`-Definition bestimmt, unter welchem Namespace diese Taglib verfügbar ist. In `SimpleTaglib` dienen Perl-Attribute, die mit `XSP` anfangen, als Kennzeichnungen für die Art der Verarbeitung. `XSP_EXPR` ist sehr üblich und erzeugt aus dem Rückgabewert einfachen Text. Weitere Funktionen sind der ausführlichen Dokumentation zu entnehmen, generell werden aber alle Funktionen, die eines der `XSP`-Attribute tragen, automatisch als Tags bereitgestellt. Durch diesen Mechanismus eignet sich `XSP` auch zur Implementierung größerer Anwendungen, das `XSP` an sich ist dann eher als Datenstruktur zu verstehen, denn der Programmcode lebt ja in herkömmlichen Perl-Modulen.

Um `XSP`-Taglibs zu nutzen, müssen sie natürlich noch `AxKit` bekannt gemacht werden:

```
AxAddXSPTaglib AxKit::XSP::Sendmail
AxAddXSPTaglib AxKit::XSP::Demo
```

Es gibt bereits etliche Taglibs, wobei nicht alle wirklich eine Arbeitserleichterung sind - `AXKIT::XSP::ESQL` beispielsweise entstammt der Cocoon-Welt und ist wesentlich aufwendiger als ein Zugriff per `DBI`. Unter diesen auf CPAN verfügbaren Taglibs befinden sich Module für Sitzungsverwaltung, Benutzerkontrolle, applikations-globale Variablen, eMail, SQL, Formulare, Exception-Handling, sowie Hilfsmittel für Request-Daten und Umleitungen. Richtig angewendet können diese `XSP`-Code wesentlich klarer und übersichtlicher machen.

4.9. Zusammenfassung

In diesem Kapitel wurden etliche Techniken vorgestellt, die AxKit bereitstellt. Die Hauptarbeit wird in einer dokumentenzentrischen Webpräsenz so wie im vorherigen Kapitel aussehen, doch die hier gezeigten Möglichkeiten geben jeder Website die dynamische Würze.

Wenn eher eine Online-Applikation erstellt wird, dann wird wohl die Hauptarbeit wie in dem Beispiel der Suchfunktion liegen. Hier gibt es mehrere Ansatzpunkte. Auf lange Sicht rächt sich der unbedarfte Einsatz von XSP, aber gut geplant und vernünftig abstrahiert bietet XSP auch unschätzbare Vorteile, z.B. die geringen Abhängigkeiten zwischen den einzelnen Funktionen. Das Provider-Verfahren kann sehr unflexibel und aufwendig zu konfigurieren sein, wenn viele getrennte Funktionen so eine Online-Applikation bereitstellen soll.

Für umfangreichere Anwendungen bietet sich aber noch eine elegante Möglichkeit, die unbedingt erwähnt werden sollte: SAWA. Von einem SAWA- und gleichzeitig auch AxKit-Autor wurde ein Provider geschrieben, der das SAWA-Framework in AxKit direkt verfügbar macht, mit allen Vorteilen, die Provider bieten können und der hohen Flexibilität von SAWA. Grob zusammengefasst bietet SAWA die Möglichkeit, Webapplikationen als endliche Automaten zu modellieren. Das zustandslose Modell von HTTP-Anfragen wird dabei elegant auf Zustandsübergänge abgebildet. Näheres hierzu, inklusive viel Beispielcode, kann der SAWA-Homepage entnommen werden.

Zweifelsohne gibt es im CPAN-Archiv noch viele andere Erweiterungen speziell für AxKit, und wie demonstriert ist es leicht, fertige Funktionen, Programme oder eben auch Perl-Module über Provider und XSP einzubinden, so daß die ganze Macht von Perl zur Verfügung steht. Dieses Kapitel konnte also unmöglich alle Varianten zeigen, jedoch wurden alle wichtigen Themen einmal angesprochen bzw. demonstriert.

5. ... und nicht den Faden verlieren Wartbarkeit mit AxKit ist kein Problem

Nachdem das Informationssystem die Reise quer durch AxKit abgeschlossen ist, kann man denken, daß eine solche Fülle von Möglichkeiten das Wartbarkeits-Horrorszenario schlechthin darstellt. Doch erzeugt Freiheit nicht automatisch unübersichtlichkeit. Selbst in PHP ist es nicht schwer für einen ungeübten Programmierer, das pure Chaos zu programmieren, dabei gibt es nur eine Standard-API, und Erweiterungsmodule sind eher unüblich. Der Autor durfte diese Erfahrung schon am eigenen Leib machen. Im Laufe seiner Dozenten- und Prüfertätigkeit sind ihm dann auch in praktisch allen anderen Sprachen kleine Meisterwerke eines teuflischen Genius begegnet, sei es das ach so strikte und saubere Java oder hip'ere .NET-Code - oder eben Perl. Das Schicksal von planloser Programmierung ist in allen Sprachen gleich: Jobsicherheit für den Programmierer.

Wenn es aber ein Projektleiter auf Code abgesehen hat, den er selber nach einem Jahr Betrieb (und Entwicklungspause) noch in vertretbarer Zeit erweitern kann, oder in den sich ein neues Teammitglied schnell einarbeiten kann, dann hat AxKit einiges zu bieten, was andere nicht haben. Streng genommen ist XSLT auch in allen anderen Sprachen verfügbar, und das Konzept von Taglibs ist auch nicht neu. SAWA-ähnliches findet man sicherlich auch in Java und .NET. Doch in AxKit ist all dies nicht aufgesetzt. Die Möglichkeiten der Trennung von Programmierung, Datenmodell und Layout ist so naheliegend, daß man unweigerlich versucht ist, ein sauberes MVC-Modell zu erstellen.

Hier liegt der Kern des Ganzen: Man kann es nicht nur, es ist so einfach, daß man Spaß dabei hat! Und selbst wenn man am Ziel vorbeischießt und sich doch der Bequemlichkeit und Einfachheit von XSP hingibt (das mythische „Projekt ohne Deadline“ ist leider noch nicht gesichtet worden), hat man trotzdem ein spürbares Mehr an Entkopplung erreicht.

Hinzu kommt, daß für viele Aufgaben des Online-Publishing gar keine Programmierung nötig ist. Man führe sich vor Augen, wie in Kapitel 3 ein vollständiges Informationssystem inklusive RPC-Schnittstelle geboren wurde, ohne eine Zeile Perl zu schreiben. XSLT in der geringen Komplexität wie hier vorgestellt kann man kaum als Programmierung bezeichnen - es wurde als deklarative Stylesheetsprache verwendet.

Doch auch das im MVC-Modell schwer anzusiedelnde XSP hat große Vorteile für die Wartbarkeit: Verändert man eine XSP-Seite, so sind die möglichen Nebenwirkungen auf andere Dokumente oder XSP-Seiten nahezu ausgeschlossen. Das ist ein großes Plus, das es gerade neuen Mitarbeitern ermöglicht, schnell einzusteigen. Ein wenig grundlegendes Perl genügt, denn XSP an sich ist fast selbsterklärend, da es nicht viel zu erklären gibt.

Teilt man die Funktionen einer Web-Anwendung in sinnvoll benannte Taglibs auf, dann werden sich einfache Wartungsarbeiten entweder in reinem XML-Territorium oder in reinem Perl-Code bewegen. Die Komplexität, die mit der Vermengung von umfangreichen Programmfunktionen und Ausgabedaten unweigerlich verbunden ist, wird elegant versteckt. Grundregel ist, daß eine XSP-Seite auch nur eine Funktion ausführen sollte. Wer ein Formular anzeigen will, die Benutzereingaben validieren will, daraufhin eine eMail verschickt und einen Datenbankeintrag macht, und zuletzt noch eine Bestätigung ausgeben möchte, der mag versucht sein, dies in einer XSP-Datei zu tun. Doch gab uns der Apache die interne und externe Weiterleitung, mit der sich solche Funktionen leicht entkoppeln lassen. Das leere Formular wird plötzlich zu einer Cachebaren statischen Seite, die Bestätigung ebenfalls, und dazwischen steht dann die eigentliche Transaktion.

Nicht zu vergessen ist aber auch, daß durch die große Freiheit das geeignetste Stylesheet-Modell gewählt werden kann. Ein Projektleiter muß nicht XSLT und XSP verwenden - alles, was Aussicht auf gute Zusammenarbeit hat, kann eingebunden werden. Es gibt im CPAN-Archiv alternative Stylesheet- bzw. Template-Systeme für AxKit, und wer Programmcode lieber mit HTML::MASON verwirklicht, kann dies tun (ein Blick in die Dokumentation zum „Filter“-Provider ist für Interessenten empfehlenswert).

Am Ende entscheidet der geistige Vater einer Anwendung durch seine Wahl der Hilfsmittel, wie sich ein System in Zukunft pflegen lässt. AxKit lässt ihm freie Hand. Wer keine Wahl trifft, für den gibt es eine ausgewogene Mischung an Techniken und Standards sofort verfügbar.

A. Quellen für das Selbststudium

Die wichtigste Quelle für alle Themen rund um AxKit ist die XML-Referenzkarte, die vom Autor des Vortrags erstellt wurde. Sie sammelt alle relevanten Standards, auch einige die nicht direkt zu XML gehören, mit den jeweiligen offiziellen Spezifikationen sowie Tutorials und Referenzen, sofern verfügbar.

<http://www.ebseiten.hab.ich.garni.ch/Docs/XML/>

(Das Original befand sich im AxKit-Dokumentationswiki, doch das hatte zeitweise ernste Probleme mit Pornospammern, was als Nebeneffekt zu einem Verlust des POD-Quelltextes geführt hat. Die obige URL ist die in Zukunft gepflegte Fassung.)

Das AxKit-Buch, „XML Publishing with AxKit“ von Kip Hampton, erschienen im O'Reilly-Verlag, ISBN 0-596-00216-5, ist eine hervorragende Einführung und Referenz zur Benutzung und besonders auch zur Erweiterung von AxKit. Einige Themen, die hier angesprochen worden sind, werden im Buch wesentlich ausführlicher erklärt.

Die PERLDOC-Dokumentation zu AxKit ist auch informativ und erstaunlich vollständig und aktuell. Nahezu jedes Modul hat seine Dokumentation, es handelt sich jedoch eher um Referenzen, nicht um Tutorial-ähnliche Dokumente.

Unter <http://www.axkit.org> sind noch weitere Dokumente zu AxKit zu finden, darunter Konfigurationsbeispiele und kleine Tutorials.