# Portal für ETH World

**Diplomarbeit Kai Jauslin**
**März 2001**

**Departement Informatik, ETH Zürich**
**Institut für Informationssysteme**

**Prof. H.-J. Schek, Roger Weber**

# Inhaltsverzeichnis

# Zusammenfassung

Ein Portal steht an der Schnittstelle von zwei Welten. Es steuert und kontrolliert den Zugang von einer Welt in die andere. In dieser Arbeit wird das Konzept und ein Prototyp für ein Portal innerhalb von ETH World entwickelt. Hier ist dieses Portal Bindeglied zwischen Benutzern und der virtuell verfügbaren Information.

Als steuernde Middleware benötigt das Portal Flexibilität. Diese entsteht unter anderem durch die Trennung von Informationen und deren Präsentation. Unabhängige Informationsproduzenten (Applikationen) generieren deshalb eine Zwischenrepräsentation in XML–Syntax, welche mittels XSLT in das vom Benutzer benötigte Format transformiert wird.

Den Applikationen stehen Basisdienstleistungen des Portals zur Verfügung, welche die Aufgaben von Personalisierung, Transformation und Sicherheit übernehmen. Die Personalisierung ist als hierarchisches Modell mit Benutzer und Benutzergruppen realisiert. Transformation und Applikation können komponentenorientiert aufgebaut werden. Auf diese Weise entsteht eine Modularisierung und Entkopplung von Entwicklungsbereichen.

Im entwickelten Prototyp, dem *Portal Application Server*, sind die Applikationen separate Java–Klassen oder Java Server Pages (JSP), welche XML generieren. Dabei können Tag Libraries und Java Beans verwendet werden. Der Prototyp ist flexibel konfigurierbar und einfach erweiterbar.

Zukünftige Portale könnten ihre Dienstleistungen als horizontale Middleware allen Applikationen zur Verfügung stellen und auf diese Weise eine flexible Schnittstelle zum Benutzerinterface anbieten.

# Kapitel 1

# Einführung

In dieser Diplomarbeit wird ein Portal für ETH World entwickelt. Als Einstieg in das Thema soll zunächst der Begriff und die Idee des *Portals* von einem eher allgemeinen Standpunkt her betrachtet werden.

## 1.1 Portale

***Zugang***

Ein Portal ist der Eingang zu einem neuen Gebiet oder Ort. Diese *Pforte* ermöglicht einen kontrollierten Zugang zur dahinterliegenden Welt. Das Web–Portal kanalisiert also den Einstieg und Zugriff in und auf eine Datenwelt. Darüber hinaus dient es dem Benutzer als Ausgangspunkt für seine virtuelle Reise.

***Aktualität***

In den Anfängen des World Wide Web war das Portal meistens identisch mit der Startseite des jeweiligen Internet-Zugangsproviders. Dieser bot dem Benutzer eine Reihe von vordefinierten Hyperlinks in Kombination mit Informationen zum eigenen Angebot an. Bald darauf begannen andere Anbieter dem Benutzer interessantere Startorte anzubieten. Die sogenannten *Content–Provider* boten als Lockmittel *aktuellere* Inhalte, zum Beispiel Schlagzeilen von Nachrichten, an.

***Angebot***

Im nächsten Schritt wurden Inhalte aus verschieden Quellen auf einer einzelne Seite zusammengestellt. Der Benutzer konnte es sich so sparen, mehrere aktuelle Seiten nacheinander zu besuchen. Der Anreiz auf Anbieterseite besteht natürlich nicht in der Brauchbarkeit für den Benutzer. Vielmehr kann er durch hohe Zugriffszahlen den Werbeplatz auf seiner eigenen Seite besser verkaufen oder die Bekanntheit seines nicht–virtuellen Angebots fördern.

***Personalisierung***

Personalisierbare Portale ermöglichen es dem Benutzer, die Quellen der Informationen und die Art der Darstellung selbst festzulegen (*was* und *wie*). Durch die Verwendung von Benutzerkonten kann die eigene Personalisierung von überall her abgerufen werden und ist auch gegen Zugriff von aussen geschützt. In dieser Arbeit soll betrachtet werden, wie die Serverseite eines solches Portalsystems aussehen könnte. Dabei geht es insbesondere um die Trennung von Darstellung und Inhalt und die Personalisierungsmöglichkeiten.

## 1.2 Ausblick

***Usability*** Betrachtet man die Entwicklung der Portalsysteme, so fällt auf, dass sie von der Funktionalität her immer mehr von der heutigen Desktop–Metapher übernehmen. Dabei geht es nicht nur um den Zugriff auf Informationen, sondern vielmehr darum, dem Benutzer eine personalisierbare Schnittstelle zu den verfügbaren Informationen zu bieten. Aus diesem Grund ist zu erwarten, dass spätere Generationen von Portal–Systemen mit dem “traditionellen” Desktop verschmelzen. Diese Entwicklung hat Microsoft bereits vorweggenommen mit der Integration von Windows Explorer und Internet Explorer. Natürlich stiess diese *Innovation* bei der Konkurrenz nicht auf besonderen Anklang. Bei der Diskussion um Software–Innovation und Monopolverhalten kommen in der Regel die Interessen der Kunden zu kurz.

# Kapitel 2

# Anforderungen und Ziele

In diesem Kapitel werden, basierend auf der Aufgabenstellung zur Diplomarbeit (Anhang A), die Anforderungen präzisiert, die das Konzept und die Umsetzung des Portals erfüllen sollen. Am Beispiel eines bestehenden Portals an der ETH werden Probleme aufgezeigt und Verbesserungsvorschläge erarbeitet. Das Konzept für das *neue* Portal nimmt diese im Folgekapitel auf und definiert die konkrete technische Umsetzung und Details.

## 2.1 Umfeld

***Infostructure***

Einer der Kernpunkte der Zielumgebung ETH World besteht im Aufbau einer virtuellen Informationsinfrastruktur, einer sogenannten *infostructure*. Diese soll es allen Teilnehmern, also ETH Mitarbeiter, Studenten, Alumni etc. ermöglichen, immer und überall auf die zur Verfügung stehenden Informationen zuzugreifen. Die konkrete Ausarbeitung ist dem Gewinnerteam des ETH World Masterplan-Wettbewerbs übertragen worden.

***Heterogenität***

Bereits jetzt ist absehbar, dass diese virtuelle Infrastruktur technisch sehr heterogen zusammengesetzt sein wird. Die Informationen müssen auf den unterschiedlichsten Endgeräten und mit unterschiedlichsten Methoden abgerufen werden können. Die Benutzer befinden sich in räumlich verteilten Organisationen und kommen aus verschiedensten fachlichen Richtungen. Die ETH World Infrastruktur soll es jedem Benutzer ermöglichen, das System auf seine Bedürfnisse optimieren zu können.

***Schnittstelle***

Das ETH World Portal soll ein benutzerspezifischer Einstiegspunkt in die virtuelle Umgebung sein. Als solches kann es als zentrales Schnittstellenelement zwischen Infrastruktur und Benutzer angesehen werden. Die Infrastruktur besteht aus Diensten (Applikationen), welche über das Portal mit dem Benutzer kommunizieren. In diesem Sinne hat es zwei Facetten: gegenüber den Applikationen tritt es als Laufzeitumgebung (Framework) auf und gegenüber dem Benutzer als Präsentator.

Im Rahmen der Diplomarbeit wird ein Konzept für eine solche Portalschnittstelle als Prototyp entwickelt. Wenn möglich soll nicht das Rad neu erfunden, sondern schon existierende Entwicklungs- und Arbeitsmethoden integriert werden. Auf diese Weise wird der Einarbeitungsaufwand für zukünftige Entwickler möglichst klein gehalten. Für das Portal als Schnittstelle gibt es verschiedene Anforderungen:

- **Dynamische Darstellung.** Geeignete Auswahl und Präsentation der personalisierten Information auf unterschiedlichsten Endgeräten.
- **Personalisierung.** Ein einzelner Einstiegspunkt zur Authentifizierung, d.h. ein *Single Sign–On* für den Benutzer. Gegenüber den Applikationen wird eine transparente Möglichkeit zur benutzerspezifischen Anpassung geboten.
- **Framework.** Anbieten von Dienstleistungen und Bibliotheken zur einfachen Applikationsentwicklung. Die Einbindung externer Informationsquellen soll ebenfalls mit geringem Aufwand erreicht werden können.
- **Wartung und Entwicklung.** Das Portal soll eine klare Schnittstelle für Entwickler bieten und diese mittels geeigneten Tools unterstützen.
- **Sicherheit.** Benutzeridentifikation, Authentifizierung und Authorisierung werden durch die Schnittstelle vorgenommen.
- **Skalierbarkeit.** Das Konzept soll im Ansatz auch mit hohen Benutzerzahlen und hoher Last umgehen können.

## 2.2 Allgemein

***Bestehendes Portal***

Die Erfahrungen des Autors als Teil des Entwicklungs- und Betriebsteams des my.polyguide [1], einem bestehenden Web-Portal an der ETH Zürich, sollen an dieser Stelle einfliessen. Ziel ist es, die positiven Eigenschaften dieses Portalsystems zu übernehmen, die bestehenden konzeptionellen Probleme aber weitgehend zu beseitigen.

### 2.2.1 my.polyguide

Der my.polyguide (siehe Abbildung 2.1) wurde 1999 von einem Gründungsteam bestehend aus vier Informatikstudenten innerhalb von vier Monaten entwickelt. Mittlerweile sind etwa 1390 Benutzer eingetragen, wovon etwa 200 das Portal jeden Tag benutzen. Als technische Basis dient ein Apache Web-Server mit der Skriptsprache PHP und dem RDBMS PostgreSQL. Die Entwicklung basiert auf einem zentralen CVS Code Repository.

**Abbildung 2.1: my.polyguide Startseite mit Suchmodul (Mitte links)**

### 2.2.2 Seiten und Sitzungen

***Einzelseiten***

Die ursprüngliche Idee des Webs sieht vor, fixe statische Dokumente durch Hyperlinks miteinander zu verknüpfen. Eine solche Seite ist die Antwort eines Web–Servers auf eine bestimmte Anfrage. Kommuniziert wird dabei über das HTTP–Protokoll [2]. Dieses definiert die Form und den Ablauf von Anfrage und Antwort. HTTP ist ein statusloses Protokoll, das heisst ein Web–Server kann nicht ohne weiteres einen direkten Zusammenhang zwischen einzelnen Anfragen herstellen. Genau dies ist aber eine Grundlage für personalisierte Anwendungen — der Benutzer soll sich nicht für jede Seite neu anmelden müssen.

***Sitzungen***

Die Erhaltung der Zustandsinformationen auf der Seite des Servers kann auf verschiedene Arten durchgeführt werden. Zum Beispiel könnten alle Zustandsinformationen mit jeder Anfrage und Antwort durch das Herumschieben von Parametern zwischen Einzelseiten erreicht werden. Das bläht aber die Anzahl der übertragenen Daten auf. Diese müssten zudem auf Serverseite geeignet serialisiert werden. In der Regel werden deshalb *Benutzer-Sitzungen* verwaltet. Eine solche Sitzung ist definiert durch Einzelzugriffe, die innerhalb einer bestimmten Zeitspanne auftreten. Sie wird definiert durch eine *Session-ID*, welche jeweils als Parameter mitgeschickt wird. Das RFC 2109 [3] beschreibt einen weiteren Mechanismus, allgemein bekannt unter dem Stichwort *Cookie*. Dieser ist ebenfalls in allen Web–Browsern implementiert und stellt so eine elegante Alternative zur Parameter-Methode dar.

Im my.polyguide–Portal wird das Session Management von einer Basisbibliothek übernommen. Diese wird in jeder Seite per `include` eingebunden und ordnet dem Programm ein gültiges Session-Objekt zu. Die Einzelseite kann sich also auf ihre Aufgabe konzentrieren.

**Ziel** Die innerhalb des Portals laufenden Applikationen arbeiten implizit statusorientiert.

### 2.2.3 Modularisierung

Das my.polyguide–Portal besteht zunächst aus einer personalisierbaren Hauptseite und konfigurierbaren Einzelteilen, sogenannten *Modulen*. Diese sind für den Inhalt und dessen Darstellung verantwortlich. Die Module stützen sich auf die gemeinsame Basisbibliothek, welche neben der Aufgabe des Session–Managements auch die Personalisierung übernimmt. Das Beispiel 2.1 zeigt einen Ausschnitt aus dem Suchmodul. Es ermöglicht dem Benutzer direkten Zugriff auf mehrere Suchmaschinen. Die Auswahl der Suchmaschine wird dabei als Personalisierung abgespeichert.

***Externe Komponenten***

Das Beispiel 2.1 zeigt anhand von PHP–Code deutlich die Grundidee von Skriptsprachen für das Web. Die Steuerung der Ablauflogik einer Applikation befindet sich grundsätzlich innerhalb einer einzelnen Seite. In diesem Beispiel wurde ein grosser Teil des allgemeinen Codes in eine externe Bibliothek ausgelagert und in einer übergeordneten Hauptseite eingebunden. Die Lebensdauer des Codes ist aber auf die einzelne Seite bzw. die dazugehörige Anfrage beschränkt. Es wäre besser, wenn sich der Code in externe Komponenten auslagern liesse, deren Lebensdauer der Entwickler definieren kann. So können Komponenten von mehreren Anfragen gleichzeitig genutzt werden und auch Daten direkt und ohne Umwege austauschen.

***Wiederverwendbarkeit***

Eine Komponente könnte zum Beispiel mit Standard-Synchronisationsmethoden Meldungen zwischen Benutzersitzungen austauschen. Befindet sich die Programmlogik in Komponenten, dann kann sie an verschiedenen Orten gleichzeitig verwendet werden (Wiederverwendbarkeit) und unabhängig vom Rest der Seite einfach ausgetauscht werden (Update). Aus diesen Gründen sind gemeinsam genutzte Module in der Softwareentwicklung nicht wegzudenken. Im Umfeld des Web ist deren Verwendung aufgrund der stark seitenbezogenen Vergangenheit aber noch lange nicht Standard.

> **Ziel** Modularisierung der Programmlogik mittels gemeinsam genutzten Komponenten (seitenübergreifend).

### 2.2.4 Trennung von Einflussbereichen

***Zusammenarbeit***

Bei der Entwicklung von Webapplikationen braucht es die Zusammenarbeit verschiedenster Disziplinen. Unter anderem sind Marketingleute, Designer, Grafiker, Ergonomen, Programmierer und viele mehr am Werk. Jeder wird durch sein Spezialwissen einen Teil zum Gesamtwerk beisteuern. Dabei will und soll das einzelne Teammitglied nur diese Teile verändern können, welche ihn betreffen. Schaut man sich das Codebeispiel 2.1 aus dem my.polyguide an, so ist leicht ersichtlich, dass dort diese Anforderung nicht erfüllt ist. Die Layoutarbeit des Designers kommt in einen direkten Konflikt mit dem Programmierer. Es können beide sich gegenseitig stören. Eine Änderung am Code wird eine Änderung am Layout mit sich bringen und umgekehrt. Benötigt wird ein System, dass die Aufsplittung einer Applikation in verschiedene Einflussbereiche erlaubt.

**Beispiel 2.1 Ausschnitt aus dem Code des my.polyguide–Suchmoduls**

```
<form name="search" action="modules/Search/processSearch.php"
 target="searchResults" method="get">

 <select name="search.SearchEngine">
 <? $searchSelEngine = $User->GetParameter("defaultSearch");
     for ($i=0; $i < count($searchNames); $i++) {
        echo("<option value=\"".$i."\"");
        if ($searchSelEngine == $i) {
          echo(" selected");
        }
        echo(">".$searchNames[$i]."\n");
     }
  ?>
  </select>

  <input type="text" size=<?
      if ($ModuleContext["Search"] == 'S') echo("13");
      else echo("30");
   ?>
   name="searchKeyword" value="">
  <? if ($ModuleContext["Search"] == 'L')
       echo("  <input type=\"submit\" "+
            "name=\"submit\" value=\"Go!\">");
  ?>
</form>
```

Diese Trennung in Teilbereiche verträgt sich ausgezeichnet mit der Anforderung der Auslagerung der Programmlogik in Komponenten. Es braucht aber geeignete Schnittstellen um diese wieder zusammenzuführen. Wie eine Teilaufsplittung erreicht werden kann wird als nächstes besprochen.

**Ziel** Klare Trennung von Einflussbereichen: Programmlogik, Information und Layout können von unterschiedlichen Personen definiert werden.

## 2.3 Dynamische Darstellung

Unter der dynamischen Darstellung versteht man die Selektion von Information, deren anschliessende Bearbeitung mit einem Programmcode und die Ausgabe des Endresultates. Im my.polyguide wird für alle drei Stufen die Skriptsprache PHP verwendet.

***Ausgabeformate***

Applikationen leiden oft an der Fixierung auf ein bestimmtes Ausgabeformat. So erzeugt eine my.polyguide–PHP Seite direkt HTML. Anfang 2000 bestand die Idee, eine WAP (Wireless Application Protocol)–Schnittstelle für das my.polyguide-Portal zu schreiben. Dadurch sollte das personalisierte Portal auch von unterwegs per Handy abrufbar sein. Nach einer gründlichen Evaluation mussten wir feststellen, dass ein komplettes Redesign der meisten Seiten nötig gewesen wäre. Die Erzeugung von Information und Darstellung läuft, wie auch im Beispiel 2.1 ersichtlich, nicht getrennt ab.

### 2.3.1 Zwischenrepräsentation

***Information***

Zur Lösung des oben geschilderten Problems, der Trennung von Information und Präsentation, betrachtet man die Gemeinsamkeiten von HTML und WML–Ausgabe: die Informationen; also die textuelle Beschreibung der Suchoptionen (im Beispiel 2.1). Diese wird dann verwendet um das gewünschte Ausgabeformat zu erzeugen. Die Information selbst muss aber auch irgendwie gespeichert werden. Da sie im Normalfall auf eine bestimmte Art strukturiert ist, empfiehlt sich auch ein strukturiertes Zwischenformat. Diese Struktur wird dann in der Transformationsstufe benutzt.

***Darstellung***

Wie das Zwischenformat effektiv aussieht, hängt primär vom Zusammenspiel zwischen Informationsproduzent (Applikation) und Transformationsstufe (Konsument strukturierter Information) ab. Die Information könnte in einem proprietären Binärformat abgelegt sein, verschlüsselt, in einer Datenbank oder rein textuell. Bei der Wahl eines geeigneten Zwischenformates gibt es verschiedene Faktoren zu berücksichtigen. Unter anderem braucht es entsprechende Tools und Arbeitsumgebungen um mit dieser Repräsentation zu arbeiten. Schaut man sich die am häufigsten auftretenden Ausgabeformate im Web–Bereich an, so sind dies alles Markup–Sprachen aus der Klasse XML/SGML, welche meist eine Baumstruktur aufweisen. Es liegt demnach nahe, der Zwischenrepräsentation ebenfalls eine XML–ähnliche, baumförmige Struktur zu geben.

Es stellt sich noch die Frage, ob es überhaupt einen gemeinsamen Nenner für die Zwischenrepräsentation gibt. Es könnte sinnvoll sein, die erzeugten Informationen direkt auf das Ausgabeformat abzustimmen. Unter Umständen muss hier ein Kompromiss getroffen werden.

### 2.3.2 Transformation

Aus der strukturierten Zwischenrepräsentation heraus entsteht die transformierte Ausgabe (siehe Abbildung 2.2). Eine flexible Transformationssprache ermöglicht die freie Selektion und Änderung der Informationen. Dem Implementationskonzept stehen dabei praktisch alle Möglichkeiten offen, von kleinen Programmstücken bis zur eigenen Sprache. Vom Implementationsstandpunkt her interessant ist sicher auch die Ausführungsgeschwindigkeit.

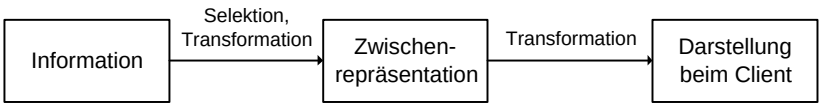


**Abbildung 2.2: Informationsfluss**

## 2.4 Personalisierung

*Einstellungen*

Unterschiedliche Benutzer stellen unterschiedliche Anforderungen an das System. Jeder hat seine persönlichen Präferenzen und passt die Arbeitsumgebung seinem Arbeitsverhalten an. Das Portal muss also Methoden zur Verfügung stellen, die es den laufenden Applikationen ermöglichen, mit den Einstellungen eines einzelnen Benutzers zu arbeiten; diese zu lesen und neu zu setzen. Die Einstellungen sollen persistent gespeichert sein, damit der Benutzer immer und überall darauf Zugriff hat.

### 2.4.1 Identifikation und Sicherheit

*Zweistufige Sicherheit*

Die Basis für die Arbeit mit personalisierten Webapplikationen ist die Identifikation und Authentifizierung des Benutzers. In der Arbeit mit dem my.polyguide hat sich aufgrund des Benutzerfeedbacks gezeigt, dass ein zweistufiges System gewünscht und sinnvoll ist:

1. **Identifikation des Benutzers.** Das Portal soll den Benutzer erkennen können, auch wenn er seine Sicherheits-Kredentialien (zum Beispiel Benutzername und Passwort) während der aktiven Sitzung noch nicht dem System präsentiert hat (noch keine *Authentifizierung*). Der grosse Vorteil dieser Identifikation ist der schnelle Zugriff auf das Portal. Unter Umständen können aber andere Personen auf persönliche Daten des einzelnen Benutzers zugreifen. Die einzelne Applikation kann deshalb entscheiden, ob eine Authentifizierung für den Zugriff nötig ist oder nicht. Diese könnte erwünscht sein, bevor der Benutzer seine persönlichen Einstellungen ändern kann.
2. **Authentifizierung des Benutzers.** Der Benutzer wird aufgefordert seine Sicherheits-Kredentialien dem System zu präsentieren (zum Beispiel Eingabe von Benutzername und Passwort — oder Zeigen eines Kerberos-Tickets). Die Authentifizierung gilt dann für den Rest der Sitzung. Diese kann aber – auf Wunsch des Benutzers – auch frühzeitig beendet werden und damit auf den Level Identifikation zurückgesetzt werden.

Wie sich das System verhalten soll, kann der Benutzer wiederum personalisieren. Er kann also die Identifikation für sich abschalten. Je nach verwendeten Sicherheits–Kredentialien erschwert das aber die Benutzung des Portals, da für jede Sitzung eine Authentifizierung erforderlich ist. Insgesamt muss sich der Benutzer für alle im Portal ablaufenden Applikationen, genau einmal pro Sitzung identifizieren oder authentifizieren (Single Sign–On).

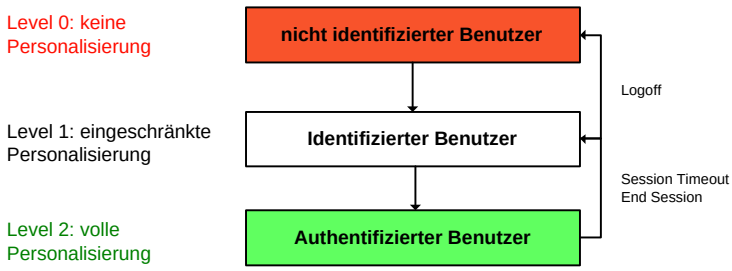


**Abbildung 2.3: Zweistufige Benutzererkennung**

Nach erfolgreicher Identifikation des Benutzers stehen den Applikationen sämtliche personalisierten Parameter änderbar zur Verfügung. Die Sicherheitsüberprüfungen für die Parameter selbst liegen bei der Applikation. Benötigt diese eine weitere Verfeinerung von Rechten, so kann sie eigene Ressourcen–IDs generieren und zur Laufzeit überprüfen lassen.

> **Ziel** Integrierte und flexible zweistufige Benutzererkennung durch das Portal. Weitergehende Sicherheitsprüfungen durch die Applikationen.

### 2.4.2 Benutzergruppen

***Hierarchie***

Innerhalb der ETH–Arbeitsumgebung lassen sich die Benutzer in Gruppen einteilen. Zum Beispiel wird sich ein Informatiker in Bezug auf die Arbeit mit dem Portal meistens anders verhalten als ein Biologe. Er hat andere Erfahrungen und Präferenzen. Dennoch gibt es auch Gemeinsamkeiten; es gehören nämlich beide zur ETH. Auf diese Weise ergibt sich eine hierarchische Gruppenstruktur, wie in Abbildung 2.4 dargestellt. Eine ETH–Schriftart könnte so für alle Portal–Benutzer voreingestellt werden. Auf Wunsch können die Informatiker als ganze Gruppe, oder ein Einzelbenutzer, diese ändern und für sich anpassen.

***Overriding***

Unter Umständen ist es nicht erwünscht, dass die Parametervorgabe einer höherliegenden Gruppe weiter personalisiert wird. Es soll deshalb die Möglichkeit bestehen, diese als fixe, nicht änderbare Vorgabe zu setzen. Das Überschreiben von existierenden Werten einer Gruppe wird später *overriding* genannt.

In dem vorliegenden Modell ist jeder Benutzer höchstens einer einzigen übergeordneten Gruppe zugeteilt. Für die Gruppe gilt wiederum dasselbe. Nun ist es natürlich denkbar, dass ein Doktorand gleichzeitig noch als Student Fächer belegt und daher eigentlich zu zwei verschiedenen Benutzergruppen gehörte. Diese Probleme sind lösbar: ein Benutzer könnte zum Beispiel mehrere wechselbare *Personalities* vertreten. Oder es werden einfach zwei verschiedene Accounts benutzt. Um das Portalkonzept jedoch einfach und verständlich zu halten, werden diese Fälle hier nicht weiter betrachtet.

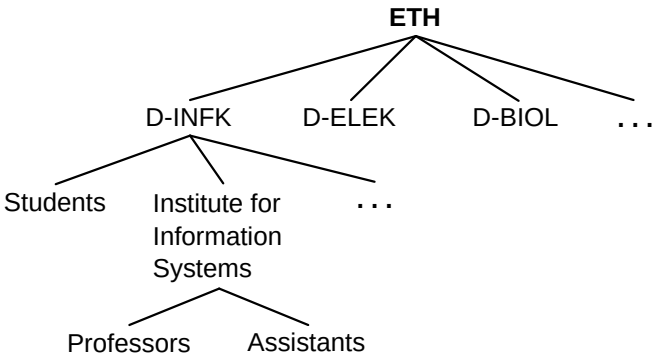


**Abbildung 2.4: Mögliche Gruppenhierarchie an der ETH**

### 2.4.3 Einstellungen

*Parameter*

Jede Applikation hat die Möglichkeit, Parameterwerte für einen Benutzer- oder eine Benutzergruppe zu erzeugen oder auf bestehende Werte zuzugreifen. Die Applikation braucht keine Kenntnisse über die effektive Speicher- und Zugriffsmethoden. Auf einen Parameter wird mittels eines Schlüsselwertes zugegriffen. Diese Methode entspricht ungefähr einer Windows Registry. Es gilt zu beachten, dass als Parameterwerte primär kleine Datenmengen abgelegt werden. Das Management grosser Binärdaten (BLOBs) soll die Applikation selbst übernehmen.

**Ziel** Benutzer- und gruppenorientierte Personalisierung mittels Parametern

## 2.5 Zusammenfassung

An dieser Stelle sollen die oben beschriebenen Wünsche und Anforderungen nochmals zentral zusammengefasst werden. Das Konzept des zu implementierenden Portals soll also die folgenden Eigenschaften aufweisen:

**Portal — Applikationsserver**

- Seitenübergreifendes Session–Management
- Auslagerung der Programmlogik in Komponenten
- Separation zwischen Information und deren Darstellung
- Identifikation und Authentifizierung des Benutzers (Single Sign-On)
- Hierarchie von Benutzergruppen
- Benutzer- und gruppenorientierte Personalisierung

### Allgemeine Anforderungen an das Konzept

- einfache Applikationsentwicklung
- Entwicklung und Wartung mittels bestehenden Tools
- Performance und Skalierbarkeit

# Kapitel 3

# Konzept

*Bausteine*

In diesem Kapitel wird ein technisches Gesamtkonzept entwickelt, welches die gestellten konzeptionellen Anforderungen aus Kapitel 2 möglichst gut erfüllt. Dabei geht es primär um die Definition und Auswahl von Bausteinen und der zu verwendenden Technologien – also um eine Bauanleitung auf Komponentenebene. Bei diesem Konzept sollen, soweit möglich, bereits existierende Technologien eingebunden werden; schliesslich sollte das System ja auch in einer produktiven Umgebung lauffähig sein. Damit ist auch die Wiederverwendung von entsprechendem Know–How möglich.

## 3.1 Übersicht des Gesamtsystems

*Anforderungen*

Wie im vorhergehenden Kapitel ausgeführt, werden an das Portal Anforderungen von ganz verschiedenen Seiten gestellt. Zunächst soll hier betrachtet werden, wie das neue System als Ganzes in eine bestehende Infrastruktur einzufügen ist, ohne dass grundsätzliche Abstriche bei den Anforderungen gemacht werden müssen.

Das Portal spricht Applikationen an, welche eine Zwischenrepräsentation generieren und diese dann mittels Transformation in ein passendes Ausgabeformat bringen. Dieses wird beim Client dargestellt. Das System besteht demnach wie in Abbildung 3.1 mindestens aus den Komponenten Client, Applikation und Transformation.

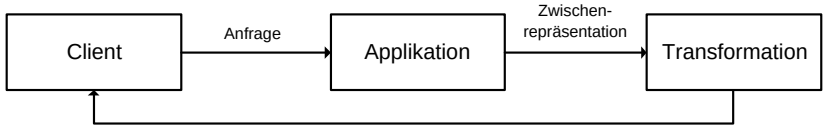


**Abbildung 3.1: Mindestkomponenten für das Portal**

### 3.1.1 Kommunikationsarchitektur

***HTTP Clients***

Bei der Wahl von geeigneten Kommunikationsmethoden geht es zunächst um die Achse Client–Applikation. Das Portal soll nahtlos mit bereits bestehenden Clients (Web–Browser) genutzt werden. Diese können in erster Linie HTML, XHTML oder XML darstellen und benutzen das HTTP–Protokoll [2]. Aufgrund der weiten Verbreitung dieser Clients wird hier das Kommunikationsprotokoll dieser Achse auf HTTP festgelegt. Auf diese Weise ist es auch möglich WAP-Applikationen anzubieten, da der WAP-Client nur über ein WAP-Gateway kommuniziert und dieses den gewünschten Inhalt ebenfalls per HTTP anfordert. Die Anbindung anderer Protokolle als HTTP ist grundsätzlich mittels Adaptern auf Serverseite realisierbar.

***Portal Application Server***

Auf der Basisebene besteht für das Portal demnach eine statuslose Client/Server Architektur mit HTTP als Protokoll. Der Client trägt nur die Verantwortung für die Darstellung und direkte Interaktion mit dem Benutzer. Alle anderen Aspekte, wie etwa Sicherheit, Transaktionen, Lastverteilung, Applikationsverwaltung und so weiter liegen beim Server. Ein solcher Server wird auch als Web–Applikationsserver bezeichnet. Das zu entwickelnde Portal wird deshalb sinngemäss *Portal Application Server* getauft.

> **Ergebnis.** Um die Kommunikation mit bestehenden Clients auf einfache Art zu ermöglichen, wird HTTP als Basisprotokoll für die Kommunikation mit dem Portal verwendet.

### 3.1.2 Applikationsserver

***Steuerung***

Die Serverseite nimmt Anfragen vom Client entgegen und sendet Antworten zurück. Dazwischen müssen die gewünschten Applikationen aufgerufen, und deren Antwort in das benötigte Ausgabeformat transformiert werden. Es ist wünschenswert, dass die Applikationen sich auf die Aufgabe der Informationsproduktion konzentrieren können. Softwaretechnisch kann dies über eine gemeinsame zentrale Komponente erreicht werden. Diese steuert die Applikationen und bietet ihnen gleichzeitig eine Basisfunktionalität in Form einer Programmbibliothek an. Die Funktionalität lässt sich in ein Application Programming Interface (API) und eine Steuerungskomponente (Dispatcher) aufteilen. Das API bietet den Applikationen Dienste in den folgenden Bereichen:

- Personalisierung (Zugriff auf Parameter des Benutzers oder der Gruppe)
- Sicherheit (Überprüfung von Rechten)
- Aufruf von anderen Applikationen (Einbindung)
- Detailsteuerung der HTTP Antwort (Redirect, Cookies, ...)

Die Koordinationskomponente übernimmt die Steuerung des Gesamtablaufs, was folgende Aufgaben umfasst:

- HTTP Request Dispatching
- Session-Management
- Initialisierung und Reload von Applikationen
- Identifizierung und Authentifizierung des Benutzers
- Transformation der Zwischendarstellung in das Ausgabeformat
- Fehlerbehandlung

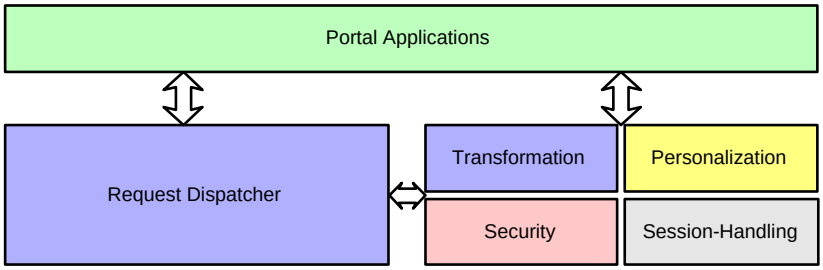


**Abbildung 3.2: Grobaufbau Applikationsserver**

## 3.2 Dynamische Darstellung

***Inhalt***

Die Programmlogik der Applikationen ist verantwortlich für den *Inhalt* der Ausgabe. Sie generiert dazu eine Zwischenrepräsentation mit diesem Inhalt, welche sinnvoll in ein Ausgabeformat transformiert werden kann. An dieser Stelle soll zunächst ein Überblick über bestehende Technologien im Web-Bereich gegeben werden.

### 3.2.1 Standards und Techniken

#### 3.2.1.1 XML

***Markup***

Die Extensible Markup Language [4] ist eine Sprache zur textuellen Beschreibung von Daten und deren Struktur. Aufgrund der einfachen Lesbarkeit wird sie insbesondere im Umfeld des Datenaustausches verwendet. Durch die breite Verfügbarkeit von XML–Parsern und Verarbeitungswerkzeugen wird die Einbindung in eigene Projekte vereinfacht. Die Struktur eines XML-Dokuments kann mittels einer Document Type Definition (DTD) [4] beschrieben und automatisiert überprüft werden. XML ist eine Untermenge der Structured Generalized Markup Language (SGML). Das *eXtensible* im Sprachnamen steht dafür, dass

es mittels der DTD möglich ist, selber zu definieren, wann ein XML–Dokument gültig ist.

***Tags*** Die Beschreibung der Struktur erfolgt in XML mittels sogenannter *Tags*. Das sind spezielle Markierungen, welche mit einem Namen und optionalen zusätzlichen Attributen ausgestattet sind. Jeweils zwei solcher Tags (Start- und Endtag) schliessen einen inhaltlichen Abschnitt ein. Dieser kann selbst weitere Tags enthalten. Gibt es zu jedem Starttag einen zugehörigen Endtag und sind diese in Baumform geschachtelt, so ist das XML–Dokument *wohlgeformt*.

#### 3.2.1.2 Statisches HTML

Die Hypertext Markup Language [5] ist eine XML–ähnliche Sprache, die aber weniger restriktive Anforderungen an das Dokument stellt (im Allgemeinen keine Wohlgeformtheit). Ein zufällig gewähltes HTML–Dokument aus dem Internet wird sich meistens nicht einmal an den vom W3C publizierten Standard halten (enthält demnach Syntaxfehler), sondern auf besondere Eigenschaften der Browser auf Client–Seite eingehen.

***Formatierung*** Das Standard–HTML erlaubt eine fixe Menge von Tag–Elementen, mit welchen die Information dargestellt werden kann. Die Tags dienen dabei nur Präsentationszwecken, also der Formatierung und Anordnung der Information. Aus diesem Grund enthalten solche Dokumente visuelle Abhängigkeiten und eignen sich somit nicht für eine Weiterverarbeitung oder Extraktion der Information.

Trotz diesem wesentlichen Nachteil ist HTML immer noch eines der weitverbreitesten Dokumentenformate und kann, mit Ausnahme der spezifischen Eigenheiten, von allen Web Browsern angezeigt werden. Für die Darstellung von Web-Seiten führt im Moment kein Weg darum herum. Der XHTML–Standard [6] als Ersatz für HTML sieht vor, HTML–Dokumente in wohlgeformter XML–Syntax abzulegen, damit diese von allen Programmen besser verarbeitet werden können. Einige der verfügbaren Web Browser können auch dieses darstellen.

#### 3.2.1.3 Statisches CSS

***Cascading Style*** Die Cascading Style Sheets [7] wurden eingeführt, um die Stilinformationen aus den HTML–Seiten zu verbannen. Ein sogenanntes *CSS–Stylesheet* ermöglicht es, zu jedem HTML–Element eine Layoutbeschreibung zu erfassen. Dabei kann diese für eine ganze Klasse von Elementen (z.B. alle Abschnitte) oder für ein Einzelnes verwendet werden. Das *Cascading* im Namen kommt daher, dass die Elemente eines Stylesheets ein anderes Stylesheet *überschreiben* können. Ein Benutzer könnte somit seine eigenen Vorstellungen über diejenigen des Webseiten–Designers stellen.

Die Definition des Layouts erfolgt in spezieller Syntax (Abbildung 3.3). CSS ist im Prinzip nicht auf HTML oder andere Ausgabemedien eingeschränkt, sondern kann auch für die Layoutbeschreibung eines XML-Dokuments verwendet werden.

Eine CSS–Definition gilt immer nur für ein einzelnes Element und dessen Inhalt. Die Definition von ganzen Tabellen oder anderen elementübergreifenden Darstellungen ist nicht möglich. CSS alleine ist für die komplette Beschreibung des Layouts einer Seite nicht ausreichend.

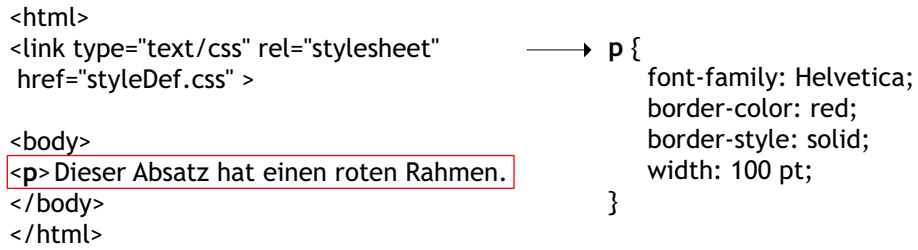


**Abbildung 3.3: Einbindung von Cascading Stylesheets (CSS)**

#### 3.2.1.4 Template in Skriptsprache

Eine Template definiert einen Rahmen, in dem Information dargestellt wird. Diese wird in einer Zwischenrepräsentation gehalten. Um die nötige Flexibilität zu gewährleisten, werden programmiersprachenähnliche Konstrukte angeboten, mit der die Information dargestellt werden kann. Eine `FOR` Schleife könnte so wie im Beispiel 3.1 dargestellt zur Ausgabe einer Tabelle führen.

***Eigene Syntax***

Das grosse Problem bei der Verwendung von Skriptsprachen ist aber, dass eine eigene Syntax verwendet wird und damit die Einbindung in entsprechende Arbeitsumgebungen erschwert wird. Ein Designer (der schliesslich die Template erstellen soll) muss zudem über Kenntnisse in dieser speziellen Sprache verfügen. WebMacro [8] ist ein Beispiel für eine solche Sprache. Während sich die Seite zwar mittels eines normalen HTML-Editors bearbeiten lässt, muss der Designer doch eine neue Syntax erlernen. Immerhin kann das WebMacro-Template (Beispiel 3.1) mit Standard HTML–Editoren bearbeitet werden.

**Beispiel 3.1 Template-gesteuerte Web-Seiten**

```
<html><body>
<table>
    #foreach $name in $Names {
        <tr><td>$name.firstName $name.lastName</td></tr>
    }
</table>
</body></html>
```

#### 3.2.1.5 Statisches XSLT

***Funktionale Sprache***

Die Extensible Stylesheet Language (XSL) [9] ist der Ansatz einer W3C–Arbeitsgruppe zur Transformation und Darstellung von Information im XML–Format. Die Transformation wird dabei mittels der Extensible Stylesheet Transformation Language (XSLT) [10] vollbracht. Im Ansatz ist XSLT eine funktionale Programmiersprache. Es wird ein Teilbaum des XML Quelldokuments als Input ausgewählt und ein veränderter Output generiert. Die Funktionen werden in XML–Syntax beschrieben. Die Beschreibung der XSLT–Transformationen selbst ist also wieder ein XML Dokument. Die Auswahl der Quellknoten im XML Baum erfolgt mittels XPath [11]

Der Einsatz von XSLT ist im Web–Umfeld sinnvoll, da es sich problemlos mit bestehenden Umgebungen und Werkzeugen versteht. Die Ausgabe eines XSLT–Prozessors ist wieder ein XML–Dokument, also zum Beispiel XHTML. Dieses liesse sich sehr einfach nach HTML konvertieren, um auf bestehenden Browsern angezeigt zu werden. Ein konkretes XSLT–Dokument kann wie in Beispiel 3.2 aussehen. Hier werden alle Knotenelemente mit dem Namen “adresse” in einen XHTML-Abschnitt transformiert, der den Inhalt (Adresse) fettgedruckt darstellt. Da die XSLT-Anweisungen in ein XML Dokument verpackt sind, lässt sich das gesamte *XSLT-Stylesheet* mit einem normalen XML–Editor bearbeiten.

**Beispiel 3.2 XSLT Stylesheet**

```
<?xml version="1.0">
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:template match="adresse">
    <b><xsl:value-of select="." /></b><br />
</xsl:template>

</xsl:stylesheet>
```

#### 3.2.1.6 XSL–FO

***Formatierung Inline***

Der zweite Teil des XSL–Ansatzes behandelt die sogenannten *Formatting Objects*. Diese definieren das genaue Erscheinungsbild eines Knotens. Die änderbaren Grundeigenschaften des Layouts sind in etwa dieselben wie bei CSS. Die Stärken von XSL–FO liegen in den erweiterten Möglichkeiten für Paginierung und Positionierung. Ein grundlegender Unterschied ist aber die Art der Beschreibung. Während die Idee von CSS darin besteht, das Aussehen von ganzen Elementklassen zu definieren, wird bei den Formatting Objects der Styling–Code direkt *inline*, zwischen Struktur und Information, geschrieben. Natürlich kann auch ein solches Vorgehen in XSLT modularisiert werden (mit einer eigenen Template für jedes Element), ist aber umständlicher.

Die Ausgabe einer Transformation mit XSL-FO muss aber trotzdem in der vom Client verstandenen Sprache dargestellt werden. Ist dies sowieso

HTML/CSS macht der Umweg über die FO kaum Sinn. Sehr nützlich hingegen können die Formatting Objects für die Generierung und Ausgabe von anderen Dokumentenarten wie etwa PDF sein. Im Moment scheint die Zukunft und Akzeptanz von XSL-FO nicht ganz klar zu sein. Während XSLT bereits den Status als W3C Recommendation hat, befindet sich XSL–FO noch im Entwurfsstadium (W3C Working Draft). Die Verfügbarkeit entsprechender Tools zur Bearbeitung von XSL (XSLT+XSL–FO) wird sicher eine entscheidende Rolle für die Durchsetzung spielen.

#### 3.2.1.7 Dynamisches XSLT und Include

***XSLT–Puzzle***

Wenn in verschiedenen XSLT–Stylesheets immer wieder die gleichen XML-Elemente vorkommen, so könnte man das XSLT dynamisch zur Laufzeit zusammensetzen, um nicht immer wieder die gleichen Transformationen neu zu definieren. Dies hat aber den Nachteil, dass das Handling wesentlich komplizierter wird, da kein separates Stylesheet als Dokument mehr vorliegt. Zusätzlich wird sicherlich auch die Ausführungsgeschwindigkeit leiden, weil das dynamische XSLT immer wieder neu geparst und optimiert werden muss.

Der XSLT–Standard sieht eine andere Möglichkeit zur Modularisierung vor: `include` und `import`. Hierbei ist es möglich, komplette XSLT–Funktionen in ein anderes XSLT-Stylesheet auszugliedern. Mit `import` können auch Funktionen aus dem eigenen Stylesheet überschrieben, d.h. neu definiert werden.

#### 3.2.1.8 SAX und DOM

***Zugriff auf XML***

Für den Zugriff auf eine XML–Quelle braucht es eine passende Schnittstelle. Als Standard vom W3C entwickelt wurde das Document Object Model (DOM) [12]. Ein *Document Builder* übernimmt zunächst das Parsen der Quelle und baut eine Baumrepräsentation des Gesamtdokuments im Speicher auf. Das DOM definiert ein API für den Zugriff auf diesen Baum. Es können unter anderem Teilbäume und Elemente selektiert werden, welche ein bestimmtes Kriterium erfüllen.

***Filter***

Das Simple API for XML (SAX) [13] arbeitet auf eine andere Weise: Anstatt das Dokument als Baum aufzubauen, wird während dem Parsen eine Serie von *Ereignissen* (Events) generiert. Die Applikation registriert einen *Event–Handler* für diese Ereignisse und kann so Informationen über das Dokument und seine Struktur erhalten und eine interne Repräsentation aufbauen. Der Parser generiert zum Beispiel Ereignisse beim Dokumentenanfang und -ende, allgemeinen Start- und Endtags, XML Processing Instructions und bei Start- bzw. Ende eines Textbereiches.

SAX eignet sich besonders dann, wenn nur ein Teil des XML–Quelldokumentes von der Applikation benötigt wird und kann so als Informationsfilter benutzt werden. Der Event–Handler muss aber aufgrund des ereignisorientierten Programmablaufs oft Zustandsinformationen zum Dokument, zum Beispiel Listen und Zähler, mitführen. Vorsicht ist insbesondere dann geboten, wenn mehrere Threads zur gleichen Zeit mit dem gleichen Event–Handler arbeiten.

***Transformation***

Die Arbeit mit dem DOM ist wesentlich einfacher. Es ist dann geeignet, wenn die Applikation direkt mit der internen Baumstruktur weiterarbeiten möchte (ohne eigene Datenstruktur) oder Transformationen angewendet werden sollen. Allerdings kann der Speicherverbrauch hoch sein, da immer der ganze Dokumentbaum im Speicher präsent sein muss.

Es gilt, je nach Situation die Vor– und Nachteile von SAX oder DOM miteinander zu vergleichen und das Bessere zu verwenden. Beide werden aktiv weiterentwickelt und sind für verschiedenste Sprachen und Plattformen verfügbar.

### 3.2.2 Zwischenrepräsentation

***Strukturierte Information***

Die Zwischenrepräsentation ist die Ausgabe der Applikationslogik. Sie enthält eine strukturierte Form der auszugebenden Information, aber ohne Präsentationselemente. Diese Form wird als Eingabe für die Transformationsstufe verwendet, wo die Selektion und Transformation stattfindet und die Information für die Präsentation auf dem Client aufbereitet wird. Bei der Wahl eines geeigneten Repräsentationsformates ist es also wichtig, diese beiden Seiten als Entscheidungsfaktoren einzubeziehen.

In Web–Portalen gibt es verschiedene Gründe, die für die Verwendung von XML oder einer XML-ähnlichen Form als Zwischenrepräsentation sprechen:

1. **Flexibilität.** XML ist eine Syntaxbeschreibung für semi–strukturierte Information. Die Grösse und Codierung des Inhalts kann deshalb beliebig variieren. Mittels entsprechender DTD wird die Gültigkeit der Struktur automatisch überprüft. Bei Änderungen an dieser Struktur sind meist nur geringfügige Anpassungen an den beteiligten Anwendungen oder Dokumenten nötig.

2. **Plattform- und Sprachunabhängigkeit.** XML ist weit verbreitet und nicht an ein bestimmtes System oder Programmiersprache gebunden. Die Information wird in textueller Form codiert, wofür internationale Standards existieren. Für die Verarbeitung von XML existieren zudem eine grosse Menge von Softwarebibliotheken.

3. **Einbindung von Fremdprodukten.** Die in Portalen dargestellten Informationen können aus ganz unterschiedlichen Quellen stammen. Die XML–Syntax scheint sich als generische Schnittstelle für den Datenaustausch durchzusetzen.

4. **Verständlichkeit.** Aufgrund der weiten Verbreitung die XML bis jetzt schon gefunden hat, sind entsprechende Kenntnisse auf Seite der Applikationsprogrammierer vorhanden. Die textuelle Darstellung ermöglich zudem auch manuelle Überprüfungen und hilft beim Verständnis von Strukturen.

Es wäre zwar grundsätzlich vorstellbar, eine komplexe Zwischenrepräsentation zu entwickeln. Dabei könnten Hashtabellen und andere Datenstrukturen

Verwendung finden. Es ist aber zu wünschen, dass die Kopplung zwischen Applikationen und steuerndem Server möglichst klein ist. Deshalb empfiehlt es sich ein Format zu verwenden, welches bekannt ist und weite Verbreitung hat. Bei der Implementierung wird sich zudem herausstellen, dass XML nahtlos in bestehende Web-Konzepte integriert werden kann (siehe JSP).

***Performance***

Allerdings kann es sein, dass durch die Verwendung von XML-Dokumenten, die vor der Weiterverwendung nochmals geparst werden müssen, die Performance sinkt. Optimierte Applikationen könnten dann aber dem Applikationsserver direkt einen XML DOM Baum weiterreichen. Alles in allem ist XML sicher eine vernünftiges Format für die Zwischenrepräsentation und findet deshalb in diesem Portal Verwendung.

**Ergebnis.** Die Zwischenrepräsentation wird in XML abgelegt.

### 3.2.3 Transformation

Im Gegensatz zur Zwischenrepräsentation ist die Transformationsstufe nicht nur für den Applikationsentwickler wichtig, sondern beeinflusst auch direkt den Web–Designer. Es gilt zu berücksichtigen, dass ein solcher im Allgemeinen keine oder nur wenige Programmierkenntnisse mitbringt und ziemlich sicher mit visuellen Editoren arbeiten möchte.

***Markup–Sprachen***

Wie bereits bei der Kommunikationsarchitektur erwähnt wurde, verstehen nahezu alle Clients eine Markup–Sprache. Es macht daher Sinn, wenn die Transformation als Ausgabe ebenfalls eine solche generiert. Da sich XML als *Prototyp* der Markup–Sprachen verstehen lässt, genügt es, wenn die Transformation eine Ausgabe in XML–Syntax generiert. Im Falle der Zielsprache HTML wird XHTML generiert, welches mittels definierter Regeln auf HTML abgebildet werden kann.

Soll als Ausgabe ein PDF Dokument erzeugt werden ist etwas mehr Arbeit nötig. In diesem Fall kann aber die Transformation zunächst eine Beschreibung der PDF-Datei (inklusive Formatierungsoptionen in XSL–FO) generieren. Diese wird mittels einem seperaten *Renderer* dann im Binärformat ausgegeben. Eine solche Komponente [14] existiert bereits und ist frei verfügbar.

***Transformation Serializer***

Auf konzeptioneller Ebene wird der Transformationsprozess also zweigeteilt in die eigentliche Transformation und einen Renderer. Der Renderer, die letzte Stufe, lässt sich auch als *Serializer* bezeichnen, der das Endresultat in einen Stream verpackt und an den Client schickt.

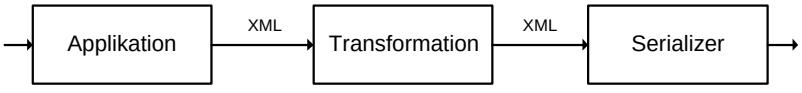


**Abbildung 3.4: Von der Zwischenrepräsentation zur Ausgabe**

Abbildung 3.4 zeigt das Blockdiagramm der Transformation auf Komponentenebene. Nun geht es darum die Komponenten geeignet zu wählen. Die erweiterten Anforderungen dabei sind:

1. XML—XML Transformation

2. Sprache muss Selektion und Änderung der Quellinformationen erlauben und möglichst einfach und verständlich für Web–Designer sein

***Design*** Die Transformation kann zum Beispiel mittels dem Aufruf von DOM–Funktionen aus einem Programmstück erfolgen. Dann aber wäre es für den Designer recht schwierig, ein schönes Layout zu generieren. Natürlich würde sich dieser gerne nur die Darstellung kümmern, ohne für die Auswahl der Informationen verantwortlich zu sein. Als Lösung könnte sich XSLT herausstellen. Damit ist es nämlich *im Prinzip* möglich, mit einem visuellen Editor wie etwa Dreamweaver [15] zu arbeiten. Das Problem liegt also grundsätzlich beim Design der Ausgabe. Ohne die Verfügbarkeit entsprechender Editoren ist die Aufgabe für einen Web-Designer schwieriger.

***Erweiterbarkeit*** An dieser Stelle soll noch auf eine weitere Eigenschaft von XSLT hingewiesen werden: die Erweiterbarkeit. Die meisten XSLT-Implementierungen erlauben es, externe Programmkomponenten, wie etwa Java-Beans oder Javascript-Code, aufzurufen. Dadurch könnte der Programmierer komplexere Layouting–Probleme (zum Beispiel dynamische Bilder) auslagern und der Designer müsste nur noch die entsprechenden Aufrufe vornehmen. Dieses Vorgehen erinnert stark an die in den Anforderungen erwähnten Tag Libraries. Diese werden in Bezug auf die Applikationen im nächsten Abschnitt genauer unter die Lupe genommen. Das bis jetzt erarbeitete Blockdiagramm der Transformation ist in Abbildung 3.5 dargestellt.

**Ergebnis.** Die Transformationsstufe nimmt das XML der Zwischenrepräsentation und erzeugt wiederum XML. Als Transformationssprache wird XSLT verwendet.

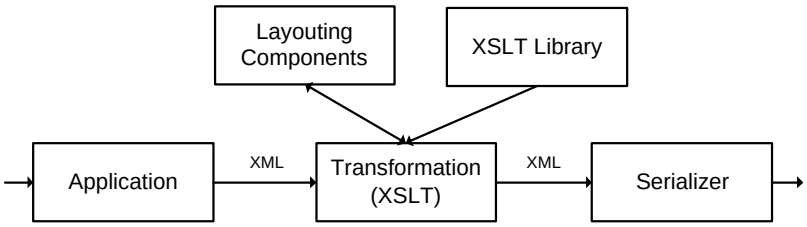


**Abbildung 3.5: Transformation mit XSLT**

## 3.3 Applikationen

***Informations–Produktion***

Die Aufgabe des Informationsproduzenten ist es, die Programmlogik auszuführen und XML für die Zwischenrepräsentation zu generieren. Dazu könnte man sich eine Programmkomponente vorstellen, die per `write("<begintag>")` entsprechende Dokumentteile in den Ausgabestrom schreibt. Eine andere Möglichkeit wäre der direkte Aufbau eines XML/DOM-Baumes. Die erste Variante ist nicht sehr elegant, weil die Syntax des zu erzeugenden XML-Dokumentes irgendwo im Programmcode untergebracht ist. Sowohl eine automatisierte Validierung als auch die Wartung und Erweiterbarkeit werden dadurch erschwert. Die zweite Möglichkeit führt zwar zu (wie eine Implementierung gezeigt hat) "sauberem" Code, aber dieser bläht sich durch die Komplexität des DOM unnötig auf und wird nur noch schwer verständlich.

Eine Lösung zur Trennung von Code und Dokumentensyntax ist die Verwendung sogenannter Tag Libraries. Dabei gehören die externen Tags zu einem erweiterten Namensraum des XML–Dokumentes welches die Applikation beinhaltet. Solcherart generierte Applikationen arbeiten dokumenten- bzw. seitenorientiert, was bei der Implementierung berücksichtigt werden muss.

Für die Implementierung wird es empfehlenswert sein, verschiedene Möglichkeiten der Applikationsanbindung anzubieten (etwa CORBA/RMI, XML Tag Libraries, Statisches XML, direkter Code). Der Entwickler sollte sich der Dualität von XSLT–Transformationskomponenten und Applikationskomponenten bewusst sein.

### 3.3.1 Tag Libraries

In sogenannten *Markup-Sprachen* wird der Text durch *Tags* strukturiert. Die "Sprache des Webs", HTML, [5] gehört neben XML [4] und XHTML [6] zu dieser Sprachklasse. Die Grundidee der Tag Libraries besteht nun darin, die bestehende Syntax für die Einbindung von Codekomponenten auszunutzen. Dadurch erhält der Quellcode der Seite eine einheitliche Struktur die auch mit Standardeditoren bearbeitet werden kann.

***Komponenten***

Diese Art der Codeeinbindung zwingt die Entwickler zur Aufteilung in wiederverwertbare Komponenten und bringt eine saubere Einbindung in das Seitenlayout. Allerdings muss gesagt werden, dass auch damit neue Probleme entstehen. So könnte der Output einer Tagklasse, weitere neue Tags beinhalten. Quellcode 3.1 zeigt ein Beispiel für die Verwendung von Tag Libraries innerhalb einer JSP–Seite welche XML generiert.

**Ergebnis.** Applikationen werden soweit möglich in Komponenten zerlegt und seitenorientiert unter Verwendung von Tag Libraries erstellt.

**Quellcode 3.1 Verwendung einer Tag Library**

```
<?xml version="1.0" ?>
<jsp:root
 xmlns:jsp="http://java.sun.com/jsp_1_2"
 xmlns:user="http://www.vis.ethz.ch/~kai/da/taglib/user">

<jspdemo>
Mein Name ist: <user:getParameter group="Style" name="myName" />
</jspdemo>

</jsp:root>
```

## 3.4 Personalisierung

***Angebot*** Wie die Applikation mit den Personalisierungsmöglichkeiten umgeht ist weitgehend freigestellt. Das API stellt nur die Möglichkeiten zum einfachen Zugriff zur Verfügung. Sämtliche personalisierte Daten müssen aber persistent gespeichert und für die Applikation immer verfügbar und änderbar sein.

### 3.4.1 Parameter

***Zeichenketten*** Aus Gründen der Einfachheit sollen alle Parameter als Zeichenketten abgelegt werden. Unter Umständen muss die Applikation entsprechende Konvertierungen vornehmen.

Für den Zugriff auf einen einzelnen Parameter werden folgende Werte benötigt:

- Applikationskennung
- Benutzerkennung
- Parametergruppe
- Parameterschlüssel

***Flache Struktur*** Wird kein Parameterschlüssel angegeben, kann direkt auf alle Werte einer bestimmten Parametergruppe zugegriffen werden. Das API liefert die Parameter als flache Struktur zurück. Das heisst die Informationen der Gruppenhierarchie des Benutzers gehen verloren. Dies hat den Vorteil, dass sich die Applikation nicht mehr um das Parameter–Overriding kümmern muss (und auch gar nicht kann). Alle zurückgelieferten Parameter sind dabei für den gewünschten Benutzer die richtigen und gültigen (siehe dazu Abbildung 3.6).

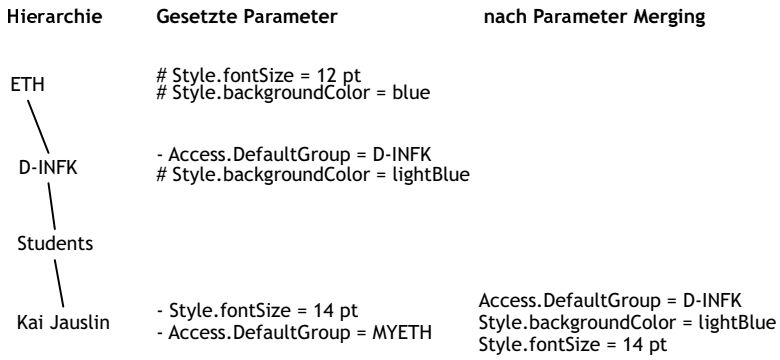


**Abbildung 3.6: Parameter Merging**

### 3.4.2 Parametergruppen

***Parameter–Gruppen***

Wird der Ausgabeprozess wie oben vorgeschlagen in Informationsproduktion und Informationstransformation aufgeteilt, so müssen die Parameter auf irgendeine Weise auch dem Transformator zur Verfügung gestellt werden. Damit das System zwischen Stil und den anderen Parametern unterscheiden kann, werden *Parametergruppen* mit eindeutigen Namen eingeführt. So ist der Wert eines Parameters mit Gruppe `Style` per Definition ein Styling-Parameter.

### 3.4.3 Transaktionen

***Konflikte***

Es könnte vorkommen, dass ein Parameterwert vor dem Zurückschreiben der einen Applikation durch eine andere gelesen wird. Es wäre zum Beispiel ein Szenario vorstellbar in dem ein Benutzer zwei parallele Reloads startet (in zwei verschiedenen Browserfenstern). Die Implementierung hat dafür zu sorgen, dass die Bearbeitung von Parameterwerten durch eine Applikation immer innerhalb einer Transaktion stattfindet. Es darf nie vorkommen, dass der Wert eines Parameters, welche eine Applikation gerade bearbeitet, gleichzeitig noch an einem anderen Ort geändert wird. Spezielle Sorgfalt ist zudem geboten, wenn zusätzlich noch Caching–Strategien angewendet werden. Eine einzelne Anfrage an das Portal sollte deshalb innerhalb einer eigenen Transaktion ablaufen.

***Humans in the loop***

Entscheidungen auf Client–Seite können dauern. Da keine direkte Kontrolle über den Zustand der Clients von Serverseite aus besteht, wird ein Request in eine einzelne Transaktion auf Serverseite umgesetzt. Nach der Beendigung des Request ist die Transaktion abgeschlossen. Ist die Applikation auf Daten des Clients angewiesen und könnten diese in Konflikt mit anderen Daten auf Serverseite stehen, so muss die Applikation vor dem Ändern diese erneut überprüfen. Eine weitergehende Betrachtung des Themas findet sich im Client–Server Survival Guide [16].

## 3.5 Styling

***Site Style***

Beim Styling geht es um das Problem, wie Stilparameter, welche meistens an verschiedenen Orten benötigt werden, sinnvoll in das bisherige Gesamtkonzept eingebunden werden können. Bezüglich der verfügbaren Techniken steht CSS oder XSL–FO zur Auswahl. Bezüglich der Akzeptanz und Zukunft von XSL–FO besteht noch Unklarheit. CSS ist weitherum akzeptiert und es existieren eine Reihe von leistungsfähigen Editoren. Aus diesen Gründen fällt hier die Wahl auf CSS. Es steht der Implementierung aber frei, auch XSL–FO anzubieten, zum Beispiel zur Ausgabe von PDF–Dokumenten.

### 3.5.1 Separates Stylesheet

***Problem Personalisierung***

Die Frage ist, wie das CSS zum Benutzer kommt und wie es auf Serverseite definiert wird. Es wäre wünschenswert, wenn dieses als seperate Datei auf dem Server abgelegt ist und im Enddokument dazugelinkt wird. Dabei wird allerdings das Problem nicht berücksichtigt, dass es auch personalisierte Stilinformationen gibt. Diese müssten während der Auführung der Applikation im Stylesheet gesetzt werden. Danach kann das neu entstandene Stylesheet als temporäre Datei verfügbar gemacht, oder direkt im Enddokument eingebettet werden.

Stellt man sich ein Portal mit mehreren Applikationen auf einer einzelnen Seite vor, so müssten sämtliche Applikationsstylesheets mit den Benutzereinstellungen gemergt werden, weil nicht im Voraus bekannt ist, welche Stileinstellungen überhaupt benötigt werden. Dieses Problem könnte über eine Erweiterungskomponente für XSLT gelöst werden, die sich zum einen merkt, welche Einstellungen benötigt werden um das Stylesheet zu generieren, und zum anderen die Parameter überhaupt in XSLT zur Verfügung stellt.

Es ist absehbar, dass auf diese Weise die Stilinformationen über ein komplexes Modul eingebunden werden. Mit ein paar Einschränkungen geht es aber einfacher.

### 3.5.2 Parametrisiertes Stylesheet

In der oben genannten Variante wurde das Stylesheet während der Bearbeitung des XSLT–Stylesheets erstellt. XSLT bietet aber bereits einen ähnlichen, voll integrierten Mechanismus: die Parametrisierung (siehe Beispiel 3.3). Dabei können vor dem Aufruf des XSLT-Prozessors beliebige Parameterwerte gesetzt werden, welche dann mit Namen im XSLT zur Verfügung stehen. Weiterhin ist es möglich, Default–Werte (d.h. Applikations–Defaultwerte) für diese Parameter direkt in XSLT zu setzen.

Im Zusammenhang mit `include` kann so extern definierte Stilinformation eingebunden werden. Es gelten aber gegenüber dem externen Stylesheet-Verfahren ein paar Einschränkungen:

1. Vor Aufruf müssen alle Parameter die verwendet werden sollen, bekannt sein und übergeben werden.

**Beispiel 3.3 Parametrisiertes XSL**

```
<?xml version="1.0">
<xsl:stylesheet
 xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

<xsl:param name="Style.fgColor"
           select="string('rgb(80,240,140);'" />

<xsl:template match="adresse">
    <span>
        <xsl:attribute name="style" value="$Style.fgColor" />
        <xsl:value-of select="." /></b><br />
    </span>
</xsl:template>
```

2. Die mittels `include` eingebundene Stilinformation hat nicht exakt dasselbe Format wie eine normale CSS-Datei (aber fast).

Da alle Stilinformationen bereits in einer Parametergruppe “Style” erfasst sind, werden einfach alle (auch die nicht benötigten) Parameter an den XSLT–Prozessor übergeben. Das Setzen eines Parameters kostet praktisch keine Rechenzeit. Dadurch spielt die erste Einschränkung keine Rolle mehr. Die Zweite insofern auch nicht, als dass ein bestehendes (z.B. mit einem externen Editor erstelltes) CSS-Stylesheet sich praktisch 1:1 übernehmen lässt. Einzig die personalisierten Parameter und XML–Basistags sind noch einzufügen. Hiermit gibt es also für XSLT eine einfache, flexible und integrierte Lösung, die in der Implementierung verwendet werden soll.

**Ergebnis.** Die personalisierten Stilinformationen werden in der Parametergruppe *Style* an den XSLT-Prozessor übergeben, wo diese mit einem parametrisierten Stylesheet verwendet werden können.

## 3.6 Zusammenfassung

Anhand von Abbildung 3.7, welche das Portalkonzept als Ganzes auf Komponentenebene zeigt, soll der geplante Ablauf des Konzepts erläutert werden. Es wird der Weg einer einzelnen Anfrage des Clients und die möglichen Ausführungswege im Portal verfolgt.

1. Ein Client sendet per HTTP eine Anfrage an die Steuerungskomponente, den *Dispatcher*. Dieser entscheidet, welche Applikationen aufgerufen werden müssen und leitet die Anfrage entsprechend weiter.

2. Die einzelne Applikation führt die Programmlogik aus. Diese kann andere

Komponenten und Tag–Libraries ansprechen oder die Ausführung delegieren. Über die Personalisierungsschnittstelle des Portals hat die Applikation vollen Zugriff auf die Parameter des Benutzers (lesen, schreiben, ändern und löschen). Die Ausgabe der Applikation ist die Zwischenrepräsentation im XML–Format, welche zurück an den Dispatcher geliefert wird.

3. Die Ausgabe wird nun vom Dispatcher an den XSLT Transformator geschickt. Das primäre XSLT Stylesheet kann andere Stylesheets aus der Bibliothek mittels `import` und `include` einbinden. Sämtliche Benutzerparameter der Parametergruppe `Style` werden als Stylesheet–Parameter zugänglich gemacht. Zusätzlichen können Teilbäume des Quelldokumentes (also der Zwischenrepräsentation) zur komplexeren Verarbeitung an externe Transformationskomponenten weitergegeben werden. Auf diese Weise könnte zum Beispiel ein dynamisches Bild gerendert werden. Die transformierte Darstellung wird wiederum im XML–Format an den Dispatcher zurückgegeben.

4. Sind alle Applikationen aufgerufen wird die endgültige Ausgabe zum Serializer geschickt. Dieser nimmt die letzten Umwandlungen vor (zum Beispiel XHTML nach HTML) und schreibt das Resultat direkt zum Client.

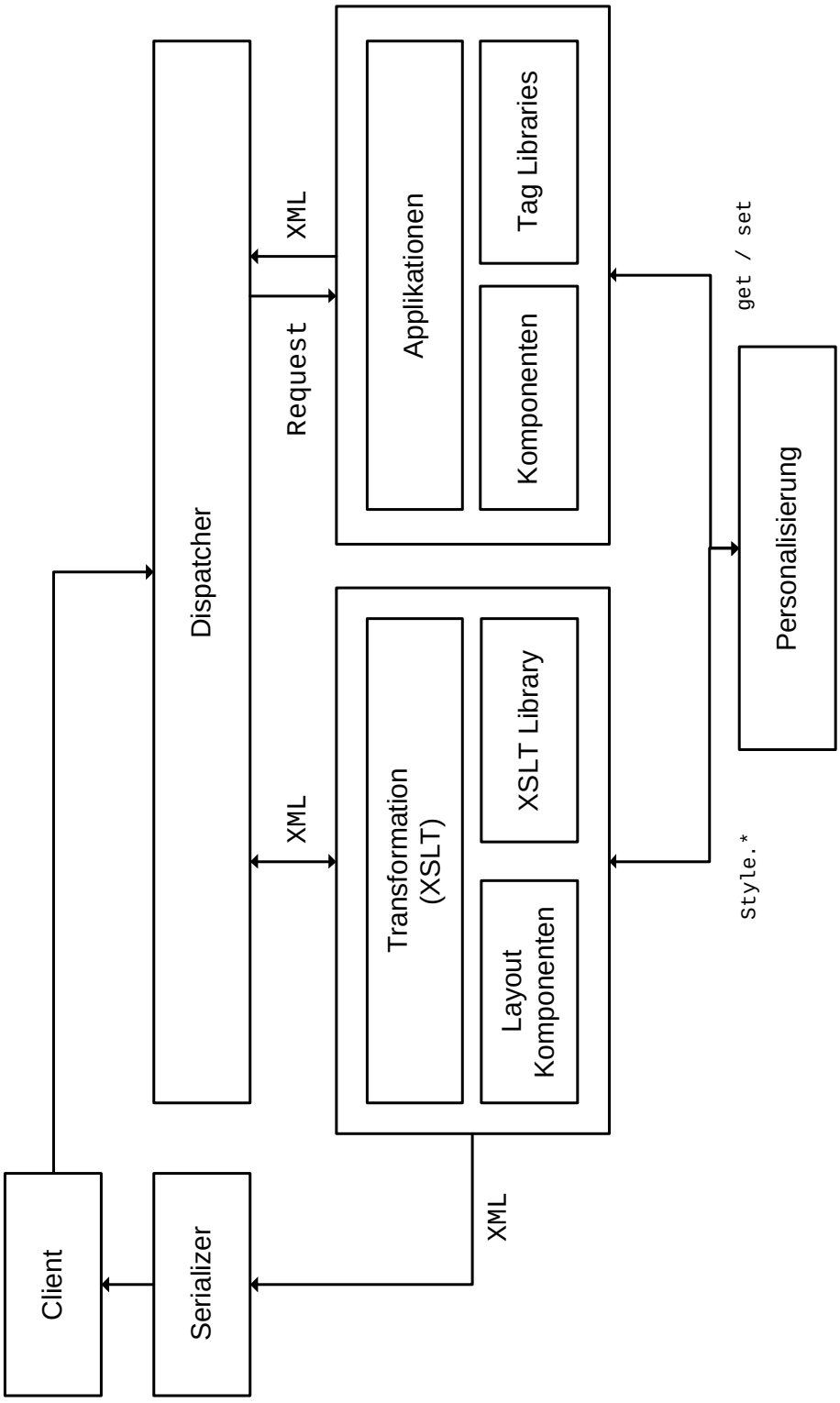


**Abbildung 3.7: Portalkonzept auf Komponentenebene**

# Kapitel 4

# Implementierung

Dieses Kapitel beschreibt die Implementierung des entwickelten Portal–Applikationsservers. Dabei wird der Schwerpunkt auf das objektorientierte Design, die Abläufe und den Programmfluss innerhalb des Servers gelegt. Desweiteren dient dieser Text als Einführung und Nachschlageort für Entwickler, die sich mit dem Portal auseinandersetzen möchten.

***Prototyp***

Es sei vorweggenommen, dass die vorliegende Implementierung des Konzeptes einen Prototyp darstellt, welcher der Veranschaulichung der entwickelten Konzepte und zum Sammeln von Erfahrungen dient. Die Aspekte Sicherheit, Stabilität und Performance sind nur am Rande behandelt. Somit ist der Prototyp in keiner Weise für produktive Umgebungen geeignet. Im letzten Kapitel der Arbeit finden sich aber Hinweise in Bezug auf eine zukünftige, produktbezogene Entwicklung.

Für die Entwicklung des Prototyps waren folgende Ziele massgebend:

- Veranschaulichung der Konzepte
- Verständlichkeit, Lesbarkeit des Codes
- Erweiterbarkeit
- konsequente Objektorientierung (siehe dazu [17])

## 4.1 Architektur und Design

***Java Servlet***

Der Prototyp ist komplett in der Sprache Java geschrieben. Der Entscheid für diese Plattform begründet sich mit der grossen und freien Verfügbarkeit von Bibliotheken und Komponenten im Web–Bereich. Die Implementierung ist so auf allen Plattformen lauffähig für die es eine Java–Unterstützung gibt.

Das Portal benutzt die Java Servlet–Dienste als Basis. Die ganze Ebene der Kommunikation per HTTP wird demnach vom Servlet Container übernommen.

Dieser kümmert sich auch um die Aspekte Parameter–Handling, Logging, Multithreading, Initialisierung, sichere Übertragung per SSL (optional) und verschiedene weitere Details. Durch diese solide Basis kann das Portal auf einer relativ hohen Abstraktionsebene entwickelt werden.

### 4.1.1 Komponenten

Für die Umsetzung des Konzeptes wurden verschiedene Fremdkomponenten eingesetzt, die im Folgenden kurz beschrieben werden.

**Tomcat 4** Ein Open Source Java Servlet Container, welcher von der Apache Software Foundation [18] entwickelt wird. Die Version 4 implementiert das Java Servlet API 2.3 und die Java Server Pages 1.2 [19]. Wie oben beschrieben, übernimmt er innerhalb des Portals die untere Basisschicht und gibt bei entsprechender Konfiguration Anfragen von ausserhalb einzig an das Portal–Servlet weiter. Für Tomcat existiert zudem ein WebDAV-Filter [20] der das "Distributed Authoring and Versioning" von Web–Applikationen und Dokumenten ermöglicht.

**Xerces-J 1.3** Der bekannteste Java XML–Parser, ebenfalls entwickelt von und verfügbar über die Apache Software Foundation [21]. Implementiert sind DOM Level 2 und SAX 2. Innerhalb des Portals wird nur die DOM–Funktionalität verwendet, einerseits um Konfigurationsdateien zu lesen, andererseits um die XML–Ausgabe von Applikationen zu integrieren.

**Xalan-J 2.0** Eine stabile und erweiterbare XSLT–Implementierung [22]. Xalan ist einer der ersten XSLT–Prozessoren, welcher mit dem Transformationsinterface von JAXP kompatibel ist. Mit Hilfe des Xalan XSLT–Prozessors wird die Zwischenrepräsentation transformiert und das Serializing durchgeführt.

**JAXP 1.1** Das Java API for XML Processing [23] ist eine Abstraktionsschnittstelle, welche die Kompatibilität unterschiedlicher XML–Parser und XML–Transformatoren gegenüber dem Applikationsprogramm gewährleistet. Das API gliedert sich in die Teile Parsing und Transformation. Auf diese Weise lassen sich im Prototyp des Portals beliebige XML–Transformatoren einbauen, sofern sie sich JAXP–konform verhalten.

**JSP 1.2** Die Java Server Pages Spezifikation [19] ermöglicht es, dynamische Seiten (Tags und Java-Code gemischt) zu erstellen. Diese werden bei einer Anfrage geparst und in ein Java Servlet compiliert, an welches der Request vom JSP Prozessor weitergeleitet wird. JSP enthält als Kernthemen die Trennung von Code und Layout mittels Java Enterprise Beans [24] und Java Tag Libraries. Innerhalb der Portals werden JSP zur seitenorientierten Applikationsentwicklung genutzt (JSP generiert dann XML).

**Jakarta Regexp** Um das korrekte Matching von der URL auf die zuständige Applikation vorzunehmen, verwendet der Dispatcher (Package portal.framework) diese Regular Expression–Bibliothek für Java [25]. Zwar müssten hierfür nicht zwingend Regular Expressions verwendet werden, diese bieten aber die Möglichkeit für ein zukünftig komplexeres Matching.

**Zusammenspiel**

Abbildung 4.1 gibt einen groben Überblick, wie die Komponenten zusammenarbeiten. Das Portal–Servlet wird von Tomcat aufgerufen und leitet den HTTP–Request gemäss Portalkonfiguration an die gewünschte Applikation weiter. Diese kann weitere Applikationen einbinden. Die Ausgabe (Zwischenrepräsentation) der Applikation wird an das Hauptservlet zurückgeliefert, ein passendes Stylesheet ausgewählt und mittels Xalan transformiert. Diese transformierte Ausgabe wird in den Serializer (diese Aufgabe wird ebenfalls von Xalan übernommen) gefüttert und das Endresultat via Tomcat an den HTTP–Client zurückgegeben. Im Falle einer JSP–Applikation wird zunächst eine Wrapper–Applikation aufgerufen, die ein Forwarding auf die gewünschte Seite durchführt. Die XML-Ausgabe dieser Seite wird automatisch mit Xerces geparst und als DOM–Baum an das Portal zurückgeliefert.

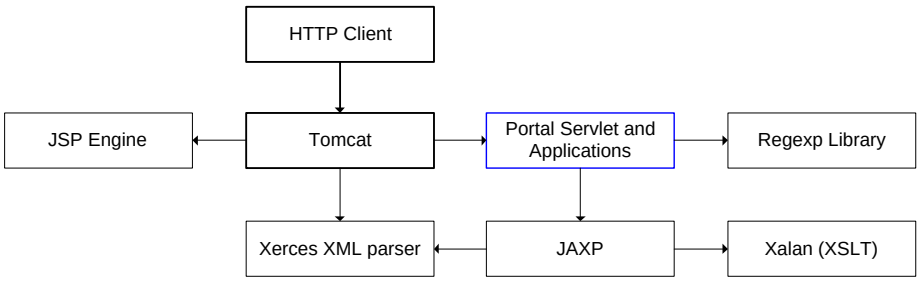


**Abbildung 4.1: Fremdkomponenten für das Portal**

Der *Portal Application Server* besteht aus über dreissig Klassen und Interfaces, aufgeteilt in 7 Packages. Die Hauptsteuerung für einen Request übernimmt dabei das Servlet mit der Klasse `Dispatcher` im Package *framework*. Im Anhang B ist das komplette Klassendiagramm als Referenz in der Übersicht dargestellt.

## 4.2 Übersicht

**Dispatcher Servlet**

Das Klassendiagramm in Abbildung 4.2 zeigt das Verhältnis des Portals zum Servlet Container (Tomcat). Die Klasse `Dispatcher` implementiert die Servlet–Funktionalität und übernimmt die globale Steuerung des Portals. Der von einem Client gestartete Request nimmt folgenden Weg:

1. Der Servlet–Container (Tomcat) nimmt den Request entgegen und wählt gemäss Konfiguration das gewünschte Servlet.
2. Die Methode `service` des Servlets wird aufgerufen.
3. Im Portal wird diese von der Klasse `HttpServlet` implementiert und ruft `doGet`, `doPost` oder eine andere Methode entsprechend dem HTTP–Protokoll auf.
4. Die Klasse `Dispatcher` übernimmt die Portal–Funktionalität durch Überschreiben von `doGet` und `doPost`.

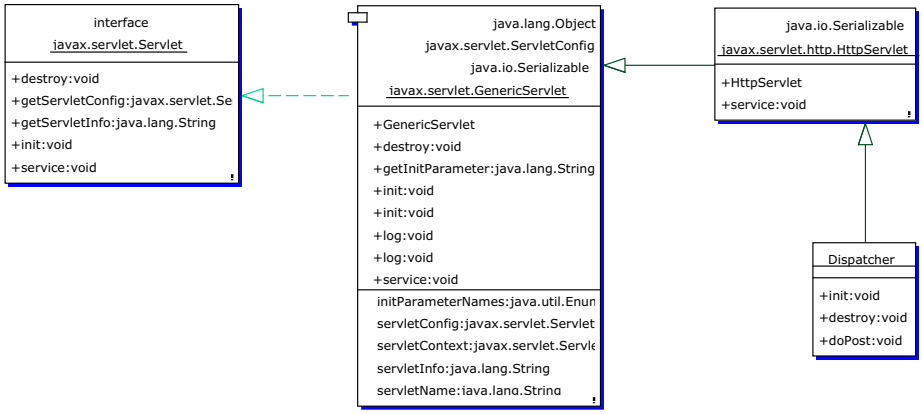


**Abbildung 4.2: Portal–Dispatcher als Servlet**

***Packages*** Die weitere Verarbeitung des Requests ist weiter unten beschrieben. Zunächst soll hier die Packagestruktur vorgestellt werden (siehe dazu Abbildung 4.3. Die sieben Packages gliedern sich in die Bereiche:

**Framework.** Steuerung des Portals, Kommunikation mit dem Servlet Container und der Datenbank. Wrapper für Request- und Responseklassen.

**Session.** Erzeugung und Verwaltung von Session–Objekten.

**Application.** Basisklasse für die im Portal ablaufenden Applikationen. Stylesheet–Verwaltung, Default–Applikationen (Login, Authentication, Logout, Logoff, ErrorHandler, JSPWrapperApplication). Exception–Klassen für Fehlermanagement.

**Security.** Überprüfen von Sicherheitsberechtigungen und Authentifizierungs–Kredentialien.

**Personalization.** Verwaltung von Benutzern und Benutzergruppen. Implementierung eines hierarchischen Speichers für Benutzerparameter.

**Util.** Utility–Klassen für Logging, String Handling, Base64 Encoding und Konvertierung von Tag–Arrays.

**Taglib.** Tag Library Komponenten für die Verwendung in JSP–Applikationen innerhalb des Portals.

## 4.3 Klassen

Einige ausgesuchte Klassen sollen im Folgenden näher angeschaut werden. Damit kann ein grober Eindruck vermittelt werden, wo welche Funktionalität implementiert ist. Die genauen Details können im Code nachgelesen werden.

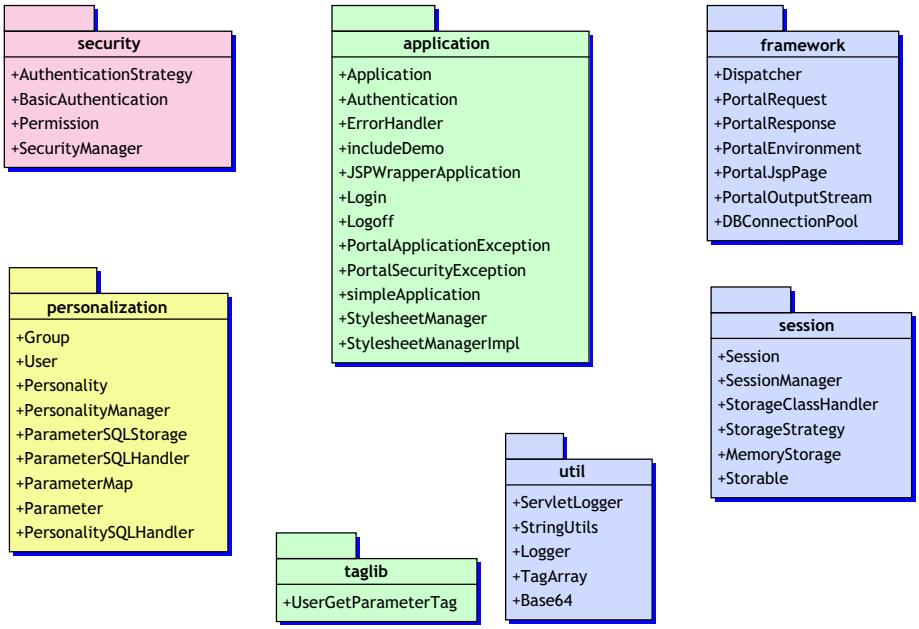


**Abbildung 4.3: Package–Struktur des Portals**

### 4.3.1 framework.Dispatcher

*Steuerung*

Der `Dispatcher`, implementiert als Servlet, stellt die Hauptklasse des Portals dar und ist für den Client von aussen per HTTP ansprechbar. Er übernimmt die Gesamtsteuerung des Portals und der Applikationen. Im Detail sind dies die folgende Aufgaben:

**Initialisierung.** Durch Aufruf der Methode `init` ermöglicht der Servlet–Container dem Portal die eigene Initialisierung. Als erstes wird dort die Konfiguration des Portals aus der Datei `configuration.xml` gelesen und geparst.

Danach wird der Datenbank–Pool, die Verbindung zur DB, aufgebaut und die konfigurierten Transformer- bzw. Serializer Factories initialisiert. Die Klasse `PortalEnvironment`, welche danach erstellt wird, enthält Funktionalität, die von allen Applikationen und dem Framework gemeinsam genutzt werden können (zum Beispiel XML Parser und DB–Poolinstanz).

Im nächsten Schritt werden die Applikationsklassen per Java Classloader geladen und jeweils eine einzelne Instanz erzeugt. Dieser Instanz wird ein Stylesheet–Manager zur Seite gestellt, um die gewünschten Stylesheets der Applikation zu verwalten. Desweiteren werden Regular Expressions compiliert um das Mapping von der URL zur Applikation durchführen zu können. Von den restlichen Teilkomponenten (Session Manager, Security Manager, Personality Manager) wird eine Instanz im Dispatcher gespeichert.

**Request Mapping und Routing.** Wenn der Dispatcher einen Request vom Servlet–Container erhält, wird er die passende Applikation bestimmen und den Request weiterreichen. Die Applikation ihrerseits kann wiederum den Dispatcher aufrufen, um weitere Applikationen einzubinden.

**Benutzeridentifikation- und Authentifizierung.** Bevor der Request die Applikation erreicht, wird überprüft ob diese eine Identifikation oder Authentifizierung verlangt. Ist dies der Fall und ist der Benutzer auf der falschen Stufe, so wird anstatt der gewählten Applikation das Login oder die Authentifizierung aufgerufen. Diese ihrerseits sind als normale Applikationen realisiert.

**Transformation und Serialisierung.** Die Ausgabe einer Applikation muss mit dem passenden Stylesheet transformiert werden. Der Dispatcher bestimmt dieses anhand dem *Content Type* des `PortalResponse` Objektes und ruft den entsprechenden Transformer auf. Vor der Ausgabe an den Client wird der Serializer aufgerufen, welcher aus dem intern verwendeten DOM–Baum einen Text erstellt und in den Ausgabestrom schreibt. Desweiteren werden Konvertierungen nach bestimmten Regeln vorgenommen (z.B. Kodierung der Zeichen, XHTML nach HTML).

### 4.3.2 SessionManager

Die Verwaltung der Benutzersitzungen übernimmt die Singleton–Klasse `SessionManager`. Er bietet Funktionen zum Erzeugen, Laden, Speichern und Löschen von `Session` Objekten und zur Bestimmung der Session–ID. *Wie* die Persistenz implementiert ist, bestimmt die `StorageStrategy`, welche in der vorliegenden Version nur einen `MemoryStorage` anbietet. Dieser speichert alle Objekte in einer HashMap im Speicher.

### 4.3.3 personalization.PersonalityManager

Die Singleton–Klasse des `PersonalityManager` übernimmt die Erzeugung von Personality–Objekten, also `User` oder `Group`. Dabei wird mittels DB die Gültigkeit überprüft und die entsprechenden Gruppen gesetzt.

### 4.3.4 framework.PortalRequest

***Request Wrapper***

Bei jedem Applikationsaufruf innerhalb des Portals wird die Klasse `PortalRequest` übergeben. Es handelt sich dabei um einen Wrapper für die Klasse HttpServletRequest. Die Applikation hat dadurch Zugriff auf die per HTTP übergebenen Parameter und den HTTP Header. An dieser Stelle ist ein Wrapper nötig, weil ein HttpServletRequest nur vom Servlet–Container erzeugt und verändert werden kann. Im Portal ist es aber an verschiedenen Stellen nötig, neue Requests zu erzeugen, zum Beispiel beim rekursiven Include von Applikationen oder wenn der Request über einen Applikations–Wrapper wieder an den Container zurückgegeben wird (siehe Klasse `JSPWrapperApplication`).

Der PortalRequest ist eine Subklasse des `HttpServletRequestWrapper`. Dieser ist erst ab dem Servlet API 2.3 verfügbar und ermöglicht überhaupt diese Form des Wrappings. Mit dem Servlet API 2.2 ist ein Request–Wrapping grundsätzlich *nicht möglich*.

### 4.3.5 framework.PortalResponse

Wie der PortalRequest ist der `PortalResponse` ebenfalls ein Wrapper, aber um die Klasse des `HttpServletResponse`. In Servlets kann der Output–Stream im Allgemeinen nicht umgeleitet werden. Diese Funktionalität wird aber im Portal benötigt, damit die Ausgabe in einen XML–Parser gefüttert werden kann. Erst die Klasse `HttpServletResponseWrapper` aus dem Servlet API 2.3 ermöglicht das.

### 4.3.6 session.Session

Eine Instanz der Klasse `Session` enthält alle Informationen über eine einzelne Benutzersitzung. Es lassen sich Attribute lesen, schreiben und die Benutzerzuordnung setzen. Um eine spätere Integration des Session–Managements von Portal und Servlet–Container zu erleichtern, ist `Session` von der Klasse `HttpSession` abgeleitet.

### 4.3.7 personalization.Personality

***Benutzer und Gruppen***

Eine `Personality` ist entweder ein Benutzer (`User`) oder eine Benutzergruppe (`Group`). Ein Benutzer hat immer eine eigene ID und ist einer einzelnen Gruppe zugeordnet. Die Unterscheidung in zwei Subklassen ermöglicht es, die *Gruppenhierarchie* unterschiedlich zu verwalten. Die Funktion `getAllIds()` liefert eine Liste mit allen Personalities, die ein Benutzer oder eine Gruppe vertreten kann.

## 4.4 Ablaufbeispiel

Der Dispatcher ist als Servletbasis dafür verantwortlich, dass die richtigen Applikationen aufgerufen, die entsprechenden Berechtigungen existieren und eine korrekte Ausgabe generiert wird. Nachdem die Initialisierung des Servlets schon im vorigen Abschnitt besprochen wurde, soll jetzt der Weg eines einzelnen Requests genauer angeschaut werden.

### 4.4.1 Allgemeiner Applikationsaufruf

Die Vorgänge in der Methode `doGet` des Dispatchers sind im UML Sequenzdiagramm im Anhang B dargestellt. Der Ablauf wird hier, mit ein paar Details angereichert, kommentiert.

1. **Datenbank–Verbindung.** Gleich zu Beginn wird aus dem applikationsglobalen Datenbankpool eine (bereits geöffnete) Verbindung zur Datenbank angefordert und damit eine neue Transaktion eröffnet. Diese dient dem konsistenten Lesen und Schreiben von Benutzerparametern. Jeder Request meint deshalb, alleine auf der DB zu arbeiten. Da die Applikationen seriell abgearbeitet werden, gilt dies auch für die einzelne Applikation.

2. **Session.** Aus den verfügbaren HTTP Parametern wird versucht, eine gültige Session–ID zu extrahieren. Gelingt dies, wird die Session vom SessionManager aktiviert und als Session–Objekt an den Dispatcher zurückgegeben. Der SessionManager berücksichtigt HTTP GET und POST Parameter und die Werte von Cookies mit dem Namen "portalSession". Falls noch keine Session existiert oder die Session nicht mehr gültig ist, wird mittels `newSession` eine neue Session erzeugt.

3. **User.** In diesem Schritt soll der Benutzer identifiziert und sein User–Objekt erzeugt oder gefunden werden. Ist ein User schon mit einer Session verknüpft, so ist die Suche erledigt. Andernfalls wird der `PersonalityManager` angefragt, die Benutzerkennung "portalUser" aus den Requestparametern herauszuholen. Gelingt dies nicht, ist der Benutzer unbekannt, was der Stufe 0 in der Abbildung 2.3 auf Seite 10 entspricht. Das `user` Objekt hat dann den Wert `null`.

4. **Mapping.** Die konfigurierten Applikationsmappings (siehe Kapitel 5, Konfiguration) werden benutzt, um die richtige Applikation aus der URL des Requests zu finden.

5. **Saved Request.** Wird ein Request aufgrund mangelnder Benutzeridentifikation oder -authentifizierung intern auf die `Login` oder `Authentifikation` Applikation umgeleitet, so ist es nötig, den Originalrequest zwischenzuspeichern. Dadurch wird die Anfrage des Benutzers nur unterbrochen und muss nicht neu gestellt werden. Bei erfolgreichem Login wird zudem die URL des gespeicherten Requests benutzt, um eine geeignete HTTP Redirection vorzunehmen.

6. **Permissions.** Die Berechtigungen des Benutzers für alle Resourcen werden vom SecurityManager angefordert.

7. **Call application.** Über den Aufruf der privaten Methode `processApplication` wird nun die eigentliche Applikation aufgerufen. Beim Aufruf von `handleRequest` werden ein passendes `PortalRequest` und `PortalResponse` Objekt mitgeliefert. Die Applikation kann über die Methode `includeApplication` (Klasse PortalEnvironment) die transformierte, aber noch nicht serialisierte, Ausgabe von weiteren Applikationen einbinden.

8. **Transformation.** Die von der Applikation erzeugte Zwischenrepräsentation im XML–Format wird als DOM–Teilbaum (`DocumentFragment`) an den Dispatcher zurückgeliefert. Dieser erzeugt gemäss JAXP 1.1 zunächst einen neuen Transformer für das definierte Stylesheet. Danach werden für diese Transformer–Instanz alle Parameter mit der Gruppe `Style` gesetzt. Als letzter Schritt wird die DOM–DOM Transformation gestartet.

9. **Commit.** Die im User–Objekt gecachten Benutzerparameter werden in die DB zurückgeschrieben (falls eine Änderung stattgefunden hat). Anschliessend wird ein Commit auf der DB–Verbindung ausgeführt und diese wieder freigegeben.

10. **Serializing.** Falls keine HTTP–Redirection gewünscht ist (HTTP Headerfeld "Location"), kann die fertige Ausgabe an den Client gesendet werden. Der Serializer ist dabei ein normaler Transformer, der seine Ausgabe aber direkt in einen Stream schreibt. Was der Client in diesem Stream erwartet (XML, XHTML, HTML, etc.) wird vorher über die *Output Properties* gesetzt.

11. **Abschluss.** Der Session Manager wird nun angewiesen die aktive Session zu versorgen. Dieser benutzt dazu seine `SessionStorageStrategy` (siehe Klassendiagramm im Anhang B).

### 4.4.2 Aufruf von JSP–Seiten

***JSP Wrapper***

Für den Aufruf von *Java Server Pages* ist nicht der Dispatcher verantwortlich, sondern die Applikation `JSPWrapperApplication`. Der Request gelangt wie oben beschrieben zu dieser Applikation. Dort wird er aber weitergeleitet, wie im UML Sequenzdiagramm (Anhang B) ersichtlich ist. Im Detail werden die folgenden Schritte ausgeführt:

1. Anforderung eines geeigneten Request Dispatchers, um den Request auf die Tomcat–interne JSP–Engine weiterleiten zu können.

2. Erstellung eines neuen `PortalRequest` und `PortalResponse` Objektes, damit der Servlet Container weiss, wo sich die Seite befindet. Die Default–Seite heisst *index.jsp*. Im neuen Request werden Session und User–Attribute gesetzt, welche von der Basisklasse `PortalJspPage` wieder ausgelesen werden und als implizite Objekte auf der Seite verfügbar sind.

3. Der Request wird nun per `forward` an den Servlet–Container zurückgegeben, welcher ihn an die JSP Engine weitergibt. Diese wiederum ist verantwortlich, dass die Seite compiliert wird und der Request beim der JSP–Seite entsprechenden Servlet landet.

4. Wieder zurück, wird die Ausgabe der Java Server Page ausgelesen und in einen XML–Parser gefüttert. Dieser liefert ein vollständiges DOM–Dokument, welches anschliessend in ein `DocumentFragment` importiert und an den Dispatcher zurückgeliefert wird.

## 4.5 Details zur Implementierung

### 4.5.1 Datenbank

#### 4.5.1.1 Schnittstelle

***Pooling*** Verbindungen zur Datenbank sind teure Ressourcen. Zum einen kann es lange dauern, eine *neue* Verbindung zu öffnen, zum anderen ist die Verfügbarkeit von Verbindungen insgesamt begrenzt (Speicher, Performance, Lizenzen). Aus diesem Grund verwendet das Portal einen sogenannten Verbindungspool (`DBConnectionPool`). Die Verbindungen werden bei der Initialisierung einmal geöffnet, können dann angefordert und nach Benutzung wieder in den Pool zurückgegeben werden.

***Transaktionen*** Da eine einzelne Verbindung unter Benutzung der Java Database Connectivity (JDBC) maximal für genau eine Transaktion stehen kann, wird für jeden Request mindestens eine Verbindung zum Lesen und Schreiben der Personalisierungsparameter benötigt. Da der Login–Prozess als Applikation implementiert ist, muss er um seine eigene Verbindung besorgt sein (Überprüfung von Login und Passwort). Diese holt er ebenfalls aus dem Pool. Für das korrekte Funktionieren des Portals sind demnach mindestens *zwei* offene Verbindungen zur Datenbank nötig.

***Threads*** Der Verbindungspool kann von mehreren Threads gleichzeitig genutzt werden (Servlets sind multi–threaded). Aus diesem Grund schliessen sich die Methoden `getConnection` und `releaseConnection` gegenseitig aus (`synchronized`). Die Threads kommunizieren gegenseitig mittels *wait / notify*. Ein einzelner Thread hat beim Warten auf eine Verbindung eine Timeout Zeit von 10 Sekunden. Danach bricht die Bearbeitung ab und der Benutzer erhält eine Fehlermeldung.

#### 4.5.1.2 Aufbau

***DB–Schema*** Die (relationale) Datenbank dient als persistenter Speicher für Benutzerdaten, wie personalisierte Parameter, Metadaten zum Benutzer- und den Benutzergruppen, Sicherheitsmerkmale. Abbildung 4.4 zeigt das Schema. Der Einfachheit halber wurde das Portal mit Hilfe von Microsoft Access entwickelt, welches leider keine echten Transaktionen erlaubt. Es lässt sich aber jede beliebige JDBC–Datenbank mit diesem Schema konfigurieren (siehe Abschnitt Konfiguration im nächsten Kapitel).

### 4.5.2 Session–Management und Personalisierung

***Grundbaustein*** Session–Management ist einer der Grundbausteine des Portals. Die meisten Servlet Container bieten fertige Session Funktionalität an. Wenn man diese verwendet, ist man im Prinzip auf das in der Java Servlet Spezifikation angegebene Interface beschränkt, es sei denn man benutzt proprietäre Erweiterungen des jeweiligen Servlet–Containers.

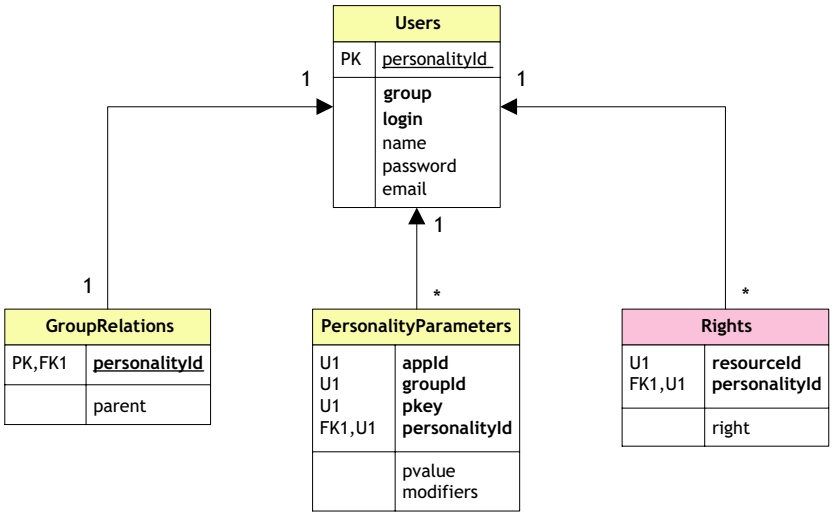


**Abbildung 4.4: Datenbank–Schema**

***Persistenz***

Die spezifizierte Session–Schnittstelle bietet keine Möglichkeit zur freien Wahl des Persistenzmechanismus (Memory oder DB), was Nachteile für die Skalierbarkeit bringen könnte. Desweiteren kann die Transportmethode der Session–ID (URL oder Cookie) nicht dynamisch gewählt und somit nicht personalisiert werden.

***Integration***

Aus diesen Gründen wurde für die vorliegende Implementierung ein eigenes Session–Management entwickelt (`portal.session.Session`). Bei der Verwendung von JSP wird der Servlet–Container mangels Integration aber immer noch sein eigenes Session–Handling anwenden. Dieses Problem ist nur durch das Schreiben von eigenen Session Klassen für den jeweilig verwendeten Container lösbar.

### 4.5.3 Sicherheit

Die Sicherheit wurde bei dieser Implementierung bewusst einfach gehalten. Die Passwörter werden im Plain–Text abgelegt (anstatt als Hash–Wert) und die einzige Möglichkeit zur Authentifizierung ist mittels Benutzername und Passwort. Die Sicherheitsmöglichkeiten des Servlet–Containers wurden nicht berücksichtigt. Für die Authentifizierung wäre es denkbar, weitere Mechanismen anzubieten, welche etwa HTTP Basic Authentication benutzen oder das SSL Client Zertifikat auswerten.

### 4.5.4 Probleme bei der Entwicklung

***Abhängigkeiten***

Eines der Grundprobleme bei der Entwicklung des Portals war die Abhängigkeit von den externen Komponenten. Seit Entwicklungsbeginn sind mehrmals neue

Releases von Xerces und Xalan erschienen, welche aufgrund neuer Funktionalität laufend integriert werden mussten. Insbesondere die Integration mit dem Java API for XML processing (JAXP) war zwischenzeitlich extrem aufwendig. Desweiteren hat sich mit dem Servlet API 2.3 erstmals die Möglichkeit ergeben, neue Requests zu erzeugen und Request–Forwarding zu implementieren (benötigt für die JSP–Funktionalität).

***Versionierung***

Ein weiteres Problem stellt die fehlende Versionierung von Java–Klassen dar. Tomcat benötigt einen eigenen XML–Parser um die Konfigurationsfiles zu lesen. Dieser verträgt sich unter Umständen nicht mit dem im Serlvet selbst verwendeten. Dieser Konflikt tritt als sogenannte *Sealing Exception* auf. Glücklicherweise war es möglich, Tomcat die neuen Parser–Versionen unterzuschieben (siehe dazu auch den Abschnitt über Konfiguration im nächsten Kapitel).

#### 4.5.4.1 Known Bugs

Der Prototyp hat in der vorliegenden Form noch eine bekannte Schwäche:

**Neuladen von Stylesheets.** Vor jeder Benutzung des gecachten Stylesheets wird geprüft, ob eine neue Version auf Disk vorliegt. Dieser Check schlägt fehl, wenn das Stylesheet weitere Stylesheets per `include` oder `import` einbindet und ein solches geändert wurde.

# Kapitel 5

# Applikationsentwicklung

Dieses Kapitel beschäftigt sich mit der Entwicklung von Applikationen, welche die Portal–Funktionalität nutzen. Dabei wird auch die Konfiguration und Installation des Prototyps beschrieben. Es werden vom Leser Kenntnisse von Java, Java Servlets, Java Server Pages (JSP) und dem Kapitel 4 (Implementation des Portals) vorausgesetzt.

## 5.1 Einrichtung der Arbeitsumgebung

Dieser Abschnitt beschreibt die nötigen Schritte, um mit dem Portal–Prototyp zu arbeiten und weiter daran zu entwickeln.

### 5.1.1 Installation des Prototyps

Der Portal–Prototyp benötigt zunächst eine korrekte Installation des Servlet Containers *Tomcat 4* [18]. Desweiteren werden folgende Fremdbibliotheken (als JAR–File ohne Sourcecode) in der angegebenen Version benötigt (für andere Versionen kann die Lauffähigkeit nicht garantiert werden):

- Xerces 1.3.0 [21]
- Xalan-J 2.0.0 [22]
- JAXP 1.1 [23]
- Regexp 1.2 [25]

Weil Tomcat selbst auch einen XML–Parser benötigt müssen die JAR–Files von Xerces und Xalan (welches den Parsing–Teil von JAXP 1.1 beinhaltet) in das `$TOMCATHOME/lib` Verzeichnis kopiert werden. Die existierenden Dateien `crimson.jar` und `jaxp.jar` aus der Tomcat Distribution müssen aus diesem Verzeichnis entfernt werden.

Die Dateien des Portals werden in ein seperates Verzeichnis entpackt. Die compilierten Java Classfiles befinden sich dabei in `WEB-INF/classes` oder `WEB-INF/lib` (für JARs). Im `lib`–Verzeichnis muss sich auch die Datei `jakarta-regexp-1.2.jar` befinden. Nun muss noch das Verzeichnis des Portal–Servlets in Tomcat konfiguriert werden. Dazu sind in der Datei `$TOMCATHOME/conf/server.xml` innerhalb eines `Engine` Abschnitts folgende Zeilen einzufügen (die `docBase` ist entsprechend anzupassen).

```
<!-- Tomcat Portal Root Context -->
<Context path=""
         docBase="E:/myDevelopment/myPortal"
         debug="1"
         reloadable="true" />
```

Aus Sicherheitsgründen kann es empfehlenswert sein, in der Datei `web.xml` zudem das Mapping der Endung `jsp` auf das Servlet `jsp` auszuschalten. Die `JSPWrapperApplication` ruft das Servlet nur mit dem Namen auf. Danach ist die Installation des Portal–Servlets abgeschlossen und Tomcat kann normal gestartet werden.

### 5.1.2 Konfiguration des Portals

***Zentrale Konfiguration***

Die Datei `configuration.xml` dient als zentrale Konfigurationsstelle für das Portal. Sie befindet sich im *Document Root* des Servlets. In ihr sind unter anderem sämtliche Stylesheets und Applikationen definiert, welche im Folgenden einzeln angeschaut werden.

**Base.** Basiseinstellungen, ohne die das System ganz sicher nicht funktionsfähig ist; darunter fallen:

```
<dbconnection applications="dispatcher"
              driver="sun.jdbc.odbc.JdbcOdbcDriver"
              init="jdbc:odbc:portal;User=portal;PW=portal"
              connectionCount="2" />
```

*Verbindung zur Portal–Datenbank*

| | |
|---|---|
| `applications` | Eine Liste von Applikationen, welche auf die Verbindung via DB–Pool zugreifen dürfen (*nicht berücksichtigt im Prototyp*). |
| `driver` | Java Klassenname eines passenden JDBC–Treibers. |
| `init` | Initialisierungsstring für die Datenbank (hier: ODBC Datenquelle und Benutzername, Passwort). |
| `connectionCount` | Anzahl Verbindungen die im Pool zur Verfügung stehen, also gleichzeitig geöffnet sind. |

```
<docroot path="E:/Development/da/portal/applications"
         relative="/applications" />
```

*Root–Verzeichnis der Applikationen*

| | |
|---|---|
| `path` | Absoluter Pfad zum Verzeichnis mit den Applikationen und Stylesheets. |
| `relative` | Relativer Pfad vom *Servlet–Root* aus gesehen (wo `configuration.xml` ist). |

**Transformers, Serializers.** Definition aller zu verwendenden JAXP Transformer- und Serializer.

```
<transformer name="xslt"
 class="org.apache.xalan.processor.TransformerFactoryImpl" />
```

*Definition eines einzelnen JAXP Transformers*

| | |
|---|---|
| `name` | Eindeutige Bezeichnung, die bei der Stylesheet–Definition angegeben wird. |
| `class` | Vollständiger Klassenname der Implementation der *Transformer–Factory*. |

```
<serializer type="text/html"
 properties="method=html,indent-amount=2"
 class="org.apache.xalan.processor.TransformerFactoryImpl" />
```

*Verbindung zur Portal–Datenbank*

| | |
|---|---|
| `type` | MIME–Type der Ausgabe (eindeutig) für welche dieser Serializer verwendet werden soll. |
| `properties` | Ausgabeeigenschaften, welche zur Laufzeit gesetzt werden sollen (*Output properties*). |
| `class` | Vollständiger Klassenname der Factory (wie beim Transformer). |

**Applications.** Definition aller verfügbaren Applikationen. Diese können in *built–in* und zusätzlich unterteilt werden.

```
<!-- login application -->
<application name="login"
             class="portal.Login"
             path="login"
             authRequired="no"
             loginRequired="no" >
   ...
</application>
```

*Beschreibung der einzelnen Applikation*

| | |
|---|---|
| `name` | Eindeutige Kennung, welche beim Mapping verwendet wird. |
| `class` | Vollständiger Klassenname der Applikation. |
| `path` | Relativer Pfad zum Applikationsverzeichnis, welches die Stylesheets und sonstige Applikationsdaten enthält. |
| `authRequired = [yes\|no]` | Benutzerauthentifikation (Stufe 2) nötig oder nicht. |
| `loginRequired = [yes\|no]` | Benutzeridentifikation (Stufe 1) nötig oder nicht. |

**Styles.** Jeder Applikation können beliebig viele Stylesheets zugeordnet werden. Ein Einzelnes wird wie folgt definiert:

```
<style name="loginStyle"
       transformer="xslt" target="text/html" media="screen"
       file="login.xsl"
       default="yes" />
```

*Verknüpfung mit einem Stylesheet*

| | |
|---|---|
| `name` | Eindeutige Kennung, welche von der Applikation verwendet werden kann. |
| `transformer` | ID der Transformer–Factory (siehe oben). |
| `target` | MIME–Type der Ausgabe und ID der Serializer–Factory. |
| `media` | Ausgabemedium für welches das Stylesheet geschrieben wurde (*nicht berücksichtigt im Prototyp*). |
| `file` | Datei, welches dieses Stylesheet enthält. |
| `default = [yes\|no]` | Für jedes Ausgabeformat kann ein Default–Stylesheet definiert werden. |

**Mappings.** Definiert die Abbildung von URL auf Applikation.

```
<mapping pattern="/login*" path="/login" application="login" />
```

*Einzelne Abbildung*

| | |
|---|---|
| `pattern` | Regular Expression mit welcher die URL geprüft wird. |
| `path` | Absoluter Pfad des Mappings, vom Servlet–Root aus gesehen. |
| `application` | ID der Applikation, welche aufgerufen wird. |

### 5.1.3 Weiterentwicklung am Framework

Für das Compilieren des ganzen Portal–Paketes müssen sich die richtigen Bibliotheken im Classpath des Java Compilers befinden. Einfacher ist die Entwicklung

mit Hilfe von Borlands *JBuilder 4* [26] oder *Together 4.2* [27], für welche schon Projektinformationen angelegt wurden. Wichtig ist, dass die Klassen direkt in das `WEB-INF/classes`–Verzeichnis geschrieben werden. So kann Tomcat direkt die neue Version erkennen und das Servlet im laufenden Betrieb neu laden. Voraussetzung hierfür ist die Option `reloadable="true"` bei der Konfiguration des Servlets in der Datei `server.xml` von Tomcat.

## 5.2 Applikation als Java–Klasse

***Aufruf***

Die im Prototyp ablaufenden Applikationen sind grundsätzlich Java–Klassen, welche von `portal.application.Application` abgeleitet sind. Diese abstrakte Oberklasse definiert zum einen die Schnittstelle für den Aufruf, zum anderen enthält sie die Konfigurationsinformationen und Stylesheets der Applikation. Der Dispatcher ruft von aussen die Methode

```
public DocumentFragment handleRequest( PortalRequest req,
                                       PortalResponse resp )
       throws PortalApplicationException
```

auf, welche von der konkreten Applikation überschrieben wird. Wie schon beschrieben, sind `PortalRequest` und `PortalResponse` Wrapper für den Servlet–Request und –Response, welcher der Dispatcher erhalten hat. Die Rückgabe der Applikation ist ein `DocumentFragment`, also ein XML–Teilbaum gemäss DOM. Der Dispatcher wird das erhaltene `DocumentFragment` mittels einem, für den gewählten Ausgabetyp passenden, Stylesheet transformieren und zum Client senden.

***Hello World***

Anhand eines einfachen Beispiels, einer *Hello World*–Ausgabe, soll die Applikationsentwicklung mit Java–Klassen illustriert werden. Die Ausgabe wird dabei HTML sein. Es wird wie folgt vorgegangen:

1. Schreiben einer Applikation, welche *Hello World* als `DocumentFragment` zurückliefert. Der Text wird als Inhalt eines Strukturelementes (`hellotext`) platziert.

2. Schreiben eines XSLT–Stylesheets, welches das Strukturelement selektiert und eine komplette HTML–Ausgabe generiert.

3. Konfiguration der Applikation in der Portalkonfiguration (`configuration.xml`) und Neustart von Tomcat.

Der Neustart des Servers ist beim Prototyp zwingend nötig. Spätere Versionen könnten ein automatisches Überprüfen und Neuladen der Konfiguration und der entsprechenden Applikationsklassen implementieren.

Der Quellcode dieser Hello World–Applikation ist im Anhang C dargestellt und dem Code des Application Servers beigelegt. Hier sei nur die Methode `handleRequest` gezeigt.

**Quellcode 5.1 Hello World – Applikation (Auszug)**

```
public  DocumentFragment
        handleRequest(PortalRequest req, PortalResponse resp) {

        Document doc = resp.getDocument();
        DocumentFragment docPart = doc.createDocumentFragment();
        Element docRoot = doc.createElement("hellotext");
        Text t = doc.createTextNode("Hello World!");
        docRoot.appendChild(t);
        docPart.appendChild(docRoot);

        return docPart;
}
```

Wie im Quellcode 5.1 ersichtlich, wird im ersten Schritt das Globaldokument für diesen Request angefordert. Mit diesem lassen sich das entsprechende `DocumentFragment`, das `Element` und der `Text` generieren. Der Text wird dann als Kind–Element an das `helloText`–Element angehängt und dieses wiederum an das `DocumentFragment`.

***DOM–Komplex*** Die Arbeit mit dem DOM ist, wie dieses Beispiel zeigt, nicht immer einfach. Als Alternative bietet sich das JDOM [28] an, ein besser an die Sprache Java angepasstes DOM API. Die Entscheidung, welches API verwendet wird, liegt beim Applikationsprogrammierer. Da es bessere Varianten zur Applikationsentwicklung gibt (siehe JSP), wird hier auch kein Schwergewicht gelegt.

***Stylesheet*** Das passende XSLT–Stylesheet ist im Quellcode 5.2 dargestellt. Zu Illustrationszwecken sind hier gleich zwei Templates definiert worden (anstatt nur eines für `hellotext`). Das Erste generiert den Rahmen des HTML–Dokumentes, das Zweite die Ausgabe des Inhalts des `hellotext`–Elements (`value-of`). Das Stylesheet wird als Datei mit Namen `HelloWorld.xsl` im Verzeichnis `hello` des Portal Application Root abgelegt.

***Konfiguration*** Nun wird noch die Konfiguration des Portals angepasst. Im Abschnitt `applications` und bei den `mappings` müssen die Zeilen

```
<!-- hello world demo application -->
<application name="HelloWorld"
             class="portal.demo.HelloWorld"
             path="/hello"
             authRequired="no"
             loginRequired="no" >
    <style name="helloStyle"
           transformer="xslt" target="text/html" media="screen"
           file="HelloWorld.xsl"
           default="yes" />
</application>

<mapping pattern="/hello*" path="/hello" application="HelloWorld" />
```

**Quellcode 5.2 Hello World – XSLT Stylesheet**

```
<?xml version="1.0" ?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="/">
<html><head><title>Hello Sample</title></head>
<body>
    <xsl:apply-templates select="hellotext" />
</body></html>
</xsl:template>

<xsl:template match="hellotext">
    <h1><xsl:value-of select="." /></h1>
</xsl:template>

</xsl:stylesheet>
```

eingefügt werden. Damit wird die Applikation und das Stylesheet beim Start des Portals geladen. Die Applikation muss vorgängig compiliert worden sein. Die Klasse befindet sich dann im Verzeichnis `WEB-INF/classes/demo`. Das `HelloWorld`–Beispiel lässt sich über `http://my.portal/hello` ansprechen.

### 5.2.1 Wrapper–Applikationen

***Generische Wrapper***

Mittels Wrapper–Applikationen können externe Informationsquellen in das Portal eingebunden werden. Diese erhalten als HTTP–Parameter oder zusätzlichen Pfad die URL der Information. Sie sorgen sich um das korrekte Mapping, das Laden dieser Information und das Parsing in einen entsprechenden DOM–Baum. Solche generischen Wrapper können auch mehrmals innerhalb verschiedener Applikationen verwendet werden. Als Beispiel sei hier die `JSPWrapperApplication` oder der `XMLWrapperApplication` genannt. Letzterer holt ein XML–Dokument von einem Pfad innerhalb des Portal–Dokumentenverzeichnisses. Eine mögliche Applikationsdefinition sieht dabei wie folgt aus:

```
<!-- hello world demo application -->
<application name="xmlfile"
             wrapper="portal.demo.XMLWrapperApplication"
             path="/"
             authRequired="no"
             loginRequired="no" >
</application>

<mapping pattern="/xml*" path="/" application="xmlfile" />
```

Der Dispatcher erzeugt bei der Initialisierung des Portals und der Applikation eine spezielle Applikationsinstanz (`GenericWrapper`), die einen Verweis auf den Applikationscode erhält. Dadurch können die Stylesheets und Pfadangaben der richtigen Applikation zugeordnet werden. Wird ein Dokument über die URL `http://my.portal/xml/data/hello.xml` angefordert, so wird die Applikation mit Namen `xmlfile`, also der `GenericWrapper`, den Request an die XMLWrapperApplication weiterleiten, welche das Dokument `data/hello.xml` liest und als DOM–Baum zurückliefert.

### 5.2.2 Personalisierung

Die Applikation kann mit der Methode `getUser` das aktuelle `User` und das `Session`–Objekt vom `PortalRequest` anfordern. Die für die Personalisierung relevanten Methoden des User–Objekts sind:

```
public synchronized Parameter getParameter
        (Object app, Object group, Object key);
public synchronized HashMap getParameters
        (Object app, Object group);
public synchronized void setParameter
        (Object app, Object group, Object key, Object value);
public synchronized void deleteParameter
        (Object app, Object group, Object key);
public Group getGroup();
```

***Caching*** Eine Beschreibung der Methoden befindet sich in der Javadoc–Dokumentation, welche dem Quellcode des Portals beiliegt. Es sei hier erwähnt, dass immer ganze Parametergruppen gelesen werden. Diese werden intern gecacht. Der mehrmalige Aufruf von `getParameter` mit derselben Parametergruppe führt also keine grossen Performanceeinbussen mit sich. Parameter für Benutzergruppen können gesetzt werden, indem zunächst via `getGroup` die richtige Gruppe gefunden wird, und dann die anderen vier Methoden benutzt werden. Diese sind in der Klasse `Personality`, der gemeinsamen Oberklasse von `User` und `Group` definiert.

## 5.3 JSP Applikationen

Bis zu diesem Punkt wurde nur die Entwicklung mit Java–Klassen besprochen. Das entwickelte Portal–Konzept sieht aber zudem die *Trennung von Code und Dokumentensyntax* und die Verwendung von *Tag Libraries* vor (siehe 3.3 auf Seite 23 und 3.3.1 auf Seite 23). Hervorragend hierfür geeignet sind die Java Server Pages (JSP). Diese sind vergleichbar mit den Microsoft Active Server Pages [29], können aber auch als XML–Dokument dargestellt werden.

***Jasper*** JSP–Seiten werden über einen JSP–Prozessor (in Tomcat wird dieser *Jasper* genannt) ausgeführt. Dieser liest und interpretiert das Ursprungsdokument und generiert daraus ein compiliertes Java–Servlet. Der ursprüngliche Request für

dieses Dokument wird dann vom JSP–Prozessor an dieses Servlet mittels der Methode `forward` aus dem Servlet–API weitergegeben. Die Ausgabe des dynamisch erzeugten JSP–Servlets wird wie gewohnt über die `ServletResponse` direkt in den Ausgabestrom geschrieben.

### 5.3.1 JSP im Portal

***JSP und XML***

Im Portal generiert nun die JSP–Aplikation die Zwischenrepräsentation als Ausgabe in XML Syntax. Diese Ausgabe wird aber von der `JSPWrapperApplication` abgefangen und an einen XML Parser gesendet, welcher einen DOM–Baum zurückliefert, der dann an den Dispatcher zur weiteren Verarbeitung und Transformation zurückgegeben wird. Auch Sun Microsystems [30] beschreibt eine ähnliche Methode.

Anhand eines einfachen *HelloWorld* Beispiels soll die Entwicklung einer JSP Portalseite illustriert werden. Der Quellcode 5.3 zeigt eine JSP–Seite, welche dieselbe Ausgabe generiert wie die Applikation 5.1. Es wird dasselbe XSLT Stylesheet 5.2 verwendet. Folgende Punkte sollten beachtet werden:

1. Als Ausgabe wird XML generiert und der XML Header gehört zu dieser Ausgabe. Die JSP–Direktive `contentType` gibt den entsprechenden MIME–Type an. Diese Direktive muss für Portal JSPs nicht zwingend verwendet werden, da sowieso nur XML erlaubt ist. Sie ist hier der Vollständigkeit halber aufgeführt.

2. Innerhalb des Portals verwendete JSP sollten von der Klasse `portal.framework.PortalJspPage` abgeleitet werden. Diese erlaubt die implizite Verwendung auf `User`–, `Session`– und anderen Kontextobjekten innerhalb der Seite.

3. Der Quellcode in 5.3 entspricht (im Gegensatz zur Ausgabe) nicht der XML–Syntax. Die JSP Spezifikation 1.2 [19] sieht dies aber vor. Damit wird auch automatische Syntaxprüfung per DTD möglich. Wie das gleiche Servlet in XML–Syntax aussieht zeigt der Quellcode in 5.4. Auch hier müssen die DOCTYPE und Ausgabetyp–Elemente nicht unbedingt angegeben werden.

**Quellcode 5.3 Hello World als JSP**

```
<?xml version="1.0"?>
<%@ page extends="portal.framework.PortalJspPage" %>
<%@ page contentType="text/xml" %>

<hellotext>Hello World</hellotext>
```

**Quellcode 5.4 Hello World als JSP in XML–Syntax**

```
<?xml version="1.0"?>
<?xml version="1.0" ?>

<jsp:root xmlns:jsp="http://java.sun.com/dtd/jsp_1_2">

<jsp:directive.page extends="portal.framework.PortalJspPage" />
<jsp:directive.page contentType="text/xml" />

<hellotext>Hello World</hellotext>

</jsp:root>
```

### 5.3.2 Programmlogik

Die User– und Session–Objekte stehen innerhalb der Seite implizit zur Verfügung und können über `portalUser` und `portalSession` angesprochen werden. Der Code kann *im Prinzip* innerhalb der Seite eingebunden werden, wie der Quellcode 5.5 zeigt.

**Quellcode 5.5 Einbettung von Java Code in JSP**

```
<colour><%
    Parameter p = portalUser.getParameter("myApp", "Style",
                                          "favouriteColor");
    if (p != null)
        write("Your favourite color: " + p.getValue());
%></colour>
```

***Komponenten*** Für wenig umfangreiche Codeabschnitte funktioniert dies einwandfrei. Das XML der Ausgabe wird vermutlich von keinem Designer benutzt und nicht oft geändert. Es sollte aber trotzdem ein Ziel des Entwicklers sein, möglichst viel der Programmlogik in externe Komponenten auszulagern. Für JSP werden zwei grundsätzliche Möglichkeiten angeboten: Java Beans und Tag Libraries, die nachfolgend beschrieben werden.

### 5.3.3 Java Beans

Java Beans sind unabhängige Java Klassen, welche innerhalb eines *Containers* als *Komponenten* angesprochen werden und bestimmte Anforderungen erfüllen. Sie sind Teil der Java 2 Enterprise Edition und in der Enterprise Java Beans Specification [24] beschrieben. Wie Beans grundsätzlich innerhalb einer JSP angesprochen werden, kann in der JSP Spezifikation [19] nachgelesen werden. Quellcode 5.6 zeigt eine mögliche Verwendung innerhalb des Portals.

**Quellcode 5.6 Zugriff auf Java Beans in JSP**

```
<nameoutput>
  <jsp:useBean id="u" scope="page"
               class="portal.beans.User"
               type="portal.beans.User" />
  Dein Name ist: <jsp:getProperty name="u" property="name"/>
</nameoutput>
```

### 5.3.4 Tag Libraries

Die andere Möglichkeit zur komponentenorientierten Entwicklung mit JSP sind *Tag Libraries*. Der Code wird dabei über selbst definierte *Tags* eingebunden. Jeder dieser Tags ist als eigene Java Klasse implementiert. Funktional zusammengehörige Gruppen dieser Tags werden in einer entsprechenden Tag–Library definiert.

Die im Portal zur Verfügung stehenden Tag–Libraries müssen beim Prototyp in der Konfigurationsdatei `web.xml` des Portal Servlets angegeben werden. Für die Verwendung innerhalb der Portal JSP–Seite, muss entweder die Direktive

```
<%@ taglib uri="http://www.vis.ethz.ch/~kai/da/taglib/user"
           prefix="user" %>
```

mit dem entsprechenden URI und Prefix angegeben werden, oder für JSP in XML–Syntax der Namespace angegeben werden, wie das folgende Beispiel zeigt:

```
<jsp:root
 xmlns:jsp="http://java.sun.com/jsp_1_2"
 xmlns:user="http://www.vis.ethz.ch/~kai/da/taglib/user" >
```

Für den Entwickler bestehen die Vorteile dabei primär in der einfachen Verwendbarkeit. Die Tag–Implementation hat, im Gegensatz zu den Java Beans, Zugriff auf den Kontext und die Elemente der Seite. Die Verwendung empfiehlt sich, wenn viel Informationen zwischen Code und Seite transferiert werden soll. Für eine weitergehende Beschäftigung mit dem Thema empfiehlt sich das Tutorial in [31].

## 5.4 Zusammenfassung

Der Portal Prototyp ermöglicht eine flexible Applikationsentwicklung mit den unterschiedlichsten Methoden. Was im Prototyp nicht verwirklicht wurde, sind die Trennung von Framework und Applikationen. Unter anderem mittels eines eigenen Java Classloaders kann diese aber nahezu erreicht werden. Aufgrund der zur Verfügung stehenden Flexibilität eignet sich der Prototyp momentan vorwiegend zum Testen und Entwickeln von neuen Konzepten. Sicherlich ist

die Entwicklung mit Hilfe von JSP und XML ein Schritt in die richtige Richtung. Es wird sich aber zuerst noch zeigen müssen, inwiefern sich die Verwendung von Java Beans und Tag–Libraries bewährt.

# Kapitel 6

# Folgerungen und Ausblick

Das erstellte Konzept und der Prototyp werden an dieser Stelle auf ihre Tauglichkeit für das Portal untersucht. Die Vorstellungen über was ein Portal ist und sein soll, haben sich im Laufe der Diplomarbeit geändert. Aus den neuen Ideen und den Erfahrungen mit dem Prototyp resultieren Verbesserungsvorschläge und neue Konzeptideen.

## 6.1 Portalvorstellungen

***Informations–quellen***

Das Bedürfnis des Benutzers für ein Portal entsteht dadurch, dass er die nicht lokal vorhandenen Informationen strukturieren und personalisieren will. Der Umgang mit entfernten Informationen unterscheidet sich grundsätzlich von dem mit lokaler Information. Die entfernt ablaufenden Applikationen des Portals verwenden andere Informationsquellen und andere Methoden der Interaktion mit dem Benutzer. Dies führt zu unterschiedlichen Benutzerkonten (lokal und auf dem Portal) und konzeptioneller Trennung von Information (zum Beispiel Ablage von Bookmarks an zwei verschiedenen Orten).

Aus technischer Sicht sind die im Portal ablaufenden Applikationen und die Lokalen ebenfalls ganz verschieden aufgebaut. Dies betrifft nicht nur Ausgabe– und Interaktionsmethoden, sondern auch Entwicklung, Betrieb und Wartung der Applikationen. Wünschenswert wäre deshalb ein Grundkonzept, welches mindestens folgende Eigenschaften enthält:

- einheitliches Modell für *Zugriff* auf lokale und entfernte Informationen
- einheitliches *Ausgabe- und Interaktionsmodell* für lokal und entfernt ablaufende Applikationen
- verteiltes *Sicherheitskonzept* bezüglich Identifikation, Authentifizierung und Zugriffsrechten

***Ubiquitous Portal***

Sind diese Punkte einmal erreicht, verschmilzt das Portalsystem mit dem lokalen System des Benutzers. Dieses dient ihm als Ausgangspunkt für die weltweit verfügbaren Informationen. Oder mit anderen Worten: *Das Portal wird zur zentralen, allen Applikationen zugrundeliegenden Basis*. Natürlich ist dies im Moment reines Wunschdenken, zeigt aber mögliche Arten von zukünftiger Applikationssoftware. Bekannte Grossfirmen investieren zurzeit Unsummen in die Entwicklung von Web–basierten Applikationen. Ein gewisses Grundbedürfnis scheint also durchaus vorhanden zu sein.

## 6.2 Prototyp und Konzept

***Probleme***

Der entwickelte Prototyp eignet sich noch nicht als allgegenwärtiges Portal, zeigt aber Möglichkeiten und Probleme, welche zu lösen sind. Auch dem Konzept fehlen noch grundlegende Eigenschaften, ohne die es sich nicht für solche Portale eignet. Im Folgenden sollen die technischen Problembereiche des Prototyps angeschaut werden:

**Applikationen.** Die Bindung zwischen der Portal–Infrastruktur (Steuerung und Basisdienstleistungen) und den ablaufenden Applikationen ist im Moment sehr eng. Die Java–Klassen der Applikationen laufen innerhalb der gleichen Virtual Machine und liefern einen DOM–Teilbaum zurück. Eine Erweiterung des Systems sollte verteilte Applikationen vorsehen, zum Beispiel Applikationen als unabhängige CORBA–Komponenten. Die Ausführung von JSP Seiten könnte durch eine eigene Komponente geschehen.

**Skalierbarkeit.** Innerhalb einer verteilten Umgebung ergeben sich neue Möglichkeiten zur Ausführung. Zusammengesetzte Portal–Anfragen könnten etwa parallel (anstatt seriell) ausgeführt werden. Auch eine Lastverteilung ist denkbar.

**Sicherheit.** Das implementierte Sicherheitskonzept ist nicht ohne weiteres verteilbar. Es darf kein direkter Zugriff des Benutzers auf die Applikationen möglich sein. Die korrekte Identifikation und Authentifizierung des Benutzers und dessen Rechte müssen jederzeit gewährleistet sein.

Bis zu einem allgegenwärtiges Portal ist noch ein weiter Weg. Es müssen dabei, wie oben bemerkt, zunächst grundsätzliche Probleme gelöst werden. Im Zentrum könnte ein einheitliches Interaktionsmodell stehen. *Wie* dieses aber genau aussieht, ist noch völlig offen. In diesem Sinne schliesst die Arbeit hier und die Lösung aller Probleme wird in die Zukunft verlegt.

# Anhang A

# Portal für ETH World

## A.1 Aufgabenstellung

### Motivation

**ETH World.** Der virtuelle Campus ETH World, so die Zielsetzung des Projektes, soll eine Platform werden, welche Information optimal strukturiert zur Verfügung stellt und die Kommunikation innerhalb der ETH (Forschung und Lehre) und ausserhalb der ETH (internationaler Austausch, Alumni) fördern. ETH World soll so integriert werden, dass die physische Universität mit der virtuellen verschmilzt, d.h. die beiden Welten sollen optimal interagieren. Innerhalb dieses grossen ETH Projektes untersucht die Datenbankgruppe des Instituts für Informationssysteme die folgenden Aspekte:

- Integration und Koordination von traditionellen Datenbeständen in die virtuelle Umgebung. Z.B. verwaltet die ETH Bibliothek ihre Bilder mit ImageFinder, ein weit verbreitetes Produkt zur Verschlagwortung zur Bildern. ImageFinder ermöglicht aber keinen direkten (gesicherten) Zugang zu den Daten über's Web und verfügt nur über rudiementäre Suchmöglichkeiten. Ein weiterer Punkt ist die Koordination: Daten müssen über mehrere Quellen hinweg konsistent gehalten werden. Z.B. bei der Raumplanung: ändert sich der Raum einer Vorlesung, so muss dies auf allen relevanten Seiten nachgeführt werden.
- Verbesserte Suchmöglichkeiten: Anhand der Bildsuche soll aufgezeigt werden, wie man in ETH World effizient und zielgerichtet multimediale Daten finden kann. Für die Konsistenzerhaltung der Suchindexe werden die Technlogien aus dem ersten Punkten eingesetzt. Dies soll zu einer schnelleren Indexierung von Daten führen (Minuten anstatt Tage) und soll Inkonsistenzen weitmöglichst verhindern.

**Web-Portale.** Ein typisches Web-Portal stellt Benutzer spezifisch die wichtigste Information und die zentralen Suchmöglichkeiten einer Web-Site zur Verfügung. Im Rahmen von ETH World spielt das Portal eine noch wichtigere Rolle: es soll den Kontakt und die Kommunikation der sich momentan in

ETH World befindenden Personen ermöglichen. Während die Gestaltung und die Kommunikationsprimitiven durch die Gewinner des ausgeschriebenen Projektwettbewerbs zu ETH World definiert werden, bleiben die technischen Aspekte und die Suchprimitiven weitgehend unserer Vorstudie und der Realisierung von ETH World vorbehalten.

### Aufgabe

Im Rahmen dieser Diplomarbeit soll ein Portal für ETH World entwickelt und die technischen Probleme weitgehend gelöst werden. Im Detail sind dies:

- Dynamische Darstellung: Web-Seiten mit wechselndem Inhalt werden nicht statisch sondern dynamisch aufgebaut. Es soll ein Konzept entworfen werden, dass die Darstellung klar von den Daten trennt. Ferner soll auch die Logik und die Benutzer spezifischen Aspekte entkoppelt sein. Dadurch ist die Wartung der Seiten einfacher und sowohl die Präsentation als auch die Daten lassen sich einfach austauschen.
- Benutzerführung: Informationsexploration ist ein komplexer Prozess: verschiedenste Informationsbedürfnisse müssen befriedigt werden, wobei Rücksicht auf die Fähigkeiten und Möglichkeiten des Bedieners genommen werden soll.
- Benutzerkontrolle und Benutzerprofil: Information soll abhängig vom aktuellen Benutzer dargestellt und zur Verfügung gestellt werden: Authorisierung, Authentifizierung und die Speicherung der Benutzerdaten sind dabei ein Thema.

Die Datenbankgruppe hat in den vergangenen Jahren ein Bildsuchsytem entwickelt, welches für ETH World um die Aspekte Textsuche, kontextgeleitete Suche, Relevanz Feedback, etc. erweitert wird. Ziel der Diplomarbeit ist die exemplarische Verwirklichung der entwickelten Konzepte im Rahmen dieses Bildsuchsystems. Die so gewonnen Erkenntnisse können später in ETH World integriert werden.

## A.2 Dank

An dieser Stelle möchte ich mich ganz herzlich bei Roger Weber für die ausgezeichnete Betreuung und bei Emanuel Schefer für das Korrekturlesen der Dokumentation bedanken.

# Anhang B

# Diagramme

1. UML Klassendiagramm
   Portal Application Server
2. UML Sequenzdiagramm
   portal.framework.Dispatcher.doGet
3. UML Sequenzdiagramm
   portal.application.JSPWrapperApplication.handleRequest

# Anhang C

# Beispiele

**Quellcode C.1 Hello World Portalapplikation (HelloWorld.java)**

```
package portal.demo;

import org.w3c.dom.DocumentFragment;
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Text;
import portal.framework.PortalRequest;
import portal.framework.PortalResponse;
import portal.application.Application;

public class HelloWorld extends Application {

    public DocumentFragment
           handleRequest(PortalRequest req, PortalResponse resp) {

        Document doc = resp.getDocument();
        DocumentFragment docPart = doc.createDocumentFragment();
        Element docRoot = resp.createElement("hellotext");
        Text t = doc.createTextNode("Hello World!");
        docRoot.appendChild(t);
        docPart.appendChild(docRoot);

        return docPart;
    }
}
```

**Quellcode C.2 HelloWorld XSLT Stylesheet (HelloWorld.xsl)**

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">

<xsl:template match="/">
<html><head><title>Hello Sample</title></head>
<body>
    <xsl:apply-templates select="*" />
</body></html>
</xsl:template>

<xsl:template match="hellotext">
    <h1><xsl:value-of select="." /></h1>
</xsl:template>

</xsl:stylesheet>
```

# Literaturverzeichnis


[1] Verein Polyguide an der ETH Zürich. *my.polyguide Portal*. http://my.polyguide.ch.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. http://www.w3.org/Protocols/rfc2068/rfc2068, January 1997. RFC 2068.

[3] D. Kristol and L. Montulli. *HTTP State Management Mechanism*. http://www.ietf.org/rfc/rfc2109.txt.

[4] W3C World Wide Web Consortium. *Extensible Markup Language (XML)*. http://www.w3.org/TR/2000/REC-xml-20001006. W3C Recommendation, October 6th 2000.

[5] W3C World Wide Web Consortium. *HyperText Markup Language*. http://www.w3.org/MarkUp/. W3C Recommendation, October 6th 2000.

[6] W3C World Wide Web Consortium. *Extensible HyperText Markup Language*. http://www.w3.org/MarkUp/. W3C Recommendation, January 26th, 2000.

[7] W3C World Wide Web Consortium. *Cascading Style Sheets (CSS) Specification, Level 2*. http://www.w3.org/TR/REC-CSS2/. W3C Recommendation, May 12th, 1998.

[8] Semiotek Inc. *WebMacro Java Servlet and HTML Template Script Language*. http://www.webmacro.org.

[9] W3C World Wide Web Consortium. *Extensible Stylesheet Language (XSL) Version 1.0*. http://www.w3.org/TR/xsl/. W3C Candidate Recommendation, November 21th, 2000.

[10] W3C World Wide Web Consortium. *XSL Transformations (XSLT) Version 1.0*. http://www.w3.org/TR/xslt.html. W3C Recommendation, November 16th, 1999.

[11] W3C World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*. http://www.w3.org/TR/xpath. W3C Recommendation, November 16th, 1999.

[12] W3C World Wide Web Consortium. *Document Object Model (DOM) Level 2 Core Specification Version 1.0*. http://www.w3.org/TR/DOM-Level-2-Core/. W3C Recommendation, November 13th, 2000.

[13] XML-DEV mailing list. *Simple API for XML parsing (SAX2)*. http://www.megginson.com/SAX/sax.html.

[14] The Apache XML Project. *XSL Formatting Objects for PDF*. http://xml.apache.org/fop/index.html.

[15] Macromedia Inc. *Dreamweaver 4 Web site development tool*. http://www.macromedia.com/software/dreamweaver.

[16] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Client/Server Survival Guide*. John Wiley & Sons, Inc., third edition edition, 1999.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc., California, 18th printing edition, 1995. ISBN: 0-201-63361-2.

[18] The Jakarta Project. *Tomcat 4, Reference Implementation of Servlet 2.3 and Java Server Pages 1.2*. http://jakarta.apache.org/tomcat/index.html.

[19] Sun Microsystems. *Java Servlet API 2.3 and Java Server Pages 1.2*. http://java.sun.com/aboutJava/communityprocess/first/jsr053/, October 2000. Proposed Final Draft.

[20] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and S. Carter. *HTTP Extensions for Distributed Authoring – WEBDAV*. http://www.ietf.org/rfc/rfc2518.txt, February 1999.

[21] The Apache XML Project. *Xerces Java XML parser 1.3.0*. http://xml.apache.org/xerces-j/index.html.

[22] The Apache XML Project. *Xalan Java XSLT processor*. http://xml.apache.org/xalan-j/index.html.

[23] Sun Microsystems Rajiv Mordani et al. *Java API for XML processing (JAXP)*. http://java.sun.com/aboutJava/communityprocess/jsr/jsr_063_jaxp11.html, January 2001.

[24] *Enterprise Java Beans Specification 1.01*. http://java.sun.com/products/javabeans/docs/spec.html.

[25] The Jakarta Project and Jonathan Locke. *Java Regular Expression package*. http://jakarta.apache.org/regexp/index.html.

[26] Borland Inc. *JBuilder 4*. http://www.borland.com/jbuilder. Java Visual Development Tool.

[27] Togethersoft. *Together 4.2*. http://www.togethersoft.com/together. Integrated Object–oriented development tool.

[28] JDOM Project. *Java Document Object Model (JDOM)*. http://www.jdom.org/.

[29] Microsoft Inc. *Microsoft Active Server Pages*. `http://www.microsoft.com/windows2000/en/advanced/iis/default.asp?url=/W%INDOWS2000/en/advanced/iis/htm/asp/aspguide.htm`.

[30] Sun Microsystems Inc. *Developing XML Solutions with Java Server Pages Technology*. `http://java.sun.com/products/jsp/html/JSPXML.html`, November 2000.

[31] Simon Brown. *Encapsulate reusable functionality in JSP tags*. `http://www.javaworld.com/javaworld/jw-08-2000/jw-0811-jsptags.html`.

[32] ETH Zürich. *ETH World Project Description*. `http://www.ethworld.ch`.