Diplomarbeit

Konzeption eines objektorientierten Applikationsservers zur Erstellung webbasierter, datenbankabhängiger Anwendungen und Strategien zur Datenmigration aus Altsystemen

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Karl Riedling
E366
Institut für Sensor und Aktuatorsysteme

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik

von

Alexander Bergolth

Mat.Nr. 9226470
Karl Domanig Gasse 4–6/5
3400 Klosterneuburg

Klosterneuburg, 30. Mai 2008

Danksagung

Ich möchte mich bei allen bedanken, die mich bei der Verfassung dieser Arbeit unterstützt haben.

Auf der fachlichen Ebene ist hier vor allem Professor Riedling hervorzuheben, der mir mit seiner freundlichen und unbürokratischen Betreuung sehr geholfen hat. Thomas Peterseil habe ich richtungsweisende Ideen bei der `TaskEngine` zu verdanken. Auch mein Vorgesetzter Willi Langenberger hat mir bei vielen unklaren Punkten als kompetenter Gesprächspartner beratend unter die Arme gegriffen.

Privat gebührt mein ganz besonderer Dank meiner Lebenspartnerin Ildiko für ihre Motivation und vor allem für die Geduld, die sie (nicht nur) während der langen Entstehungsgeschichte der Arbeit mit mir hatte. Ich danke auch meinen Eltern, die mir das Studium ermöglicht haben und die auch mit ihren kontinuierlichen Anfragen zum Fortschritt der Diplomarbeit dafür sorgten, dass ich nie auf die Fertigstellung desselben vergessen konnte. `;-)`

Zusammenfassung

Herkömmliche Web-Anwendungen passen sich in ihrem Design und Aufbau meist an das, dem World-Wide-Web zugrundeliegende, zustandslose Hypertext Transfer Protokoll an. Insbesondere bei komplexeren Aktionen, die mehrere Dialogschritte erfordern, sind dagegen die Implementierungen klassischer (nicht-webbasierter) Anwendungen häufig übersichtlicher und problemorientierter gestaltet. In der vorliegenden Arbeit wird daher ein Applikationsserver konzipiert, dessen Ziel es ist, diesen ereignisorientierten Programmierstil auch für die Entwicklung von Web-Anwendungen verfügbar zu machen.

Desweiteren wird nach Möglichkeiten gesucht, die bei einer Neuentwicklung oder Umstrukturierung einer bestehenden Applikation notwendige Übertragung der vorhandenen persistenten Daten in eine andere Datenbank und gegebenenfalls in ein anderes Datenmodell zu automatisieren.

Abstract

Traditional web applications typically adapt their design and composition to the stateless Hypertext Transfer Protocol, that forms the basis of the World Wide Web. Especially with complex tasks, that require many dialog-steps to complete, the structure of classical (non web-based) applications is often more clear and problem-oriented. This thesis designs an application server, that aims to transform this event-oriented programming style to the development of web applications.

Moreover it discusses approaches to automate the migration and, if applicable, on-the-fly restructuring of persistant application data that is needed when redesigning or refactoring an application.

Inhaltsverzeichnis

Abbildungsverzeichnis **6**

1 Einleitung **7**

2 Aufgabenstellung **9**
2.1 Entwicklung einer modularen Web-Entwicklungsumgebung . . . 9
2.1.1 Abstrahierung der Datenbankzugriffe 10
2.2 Konzepte zur Datenmigration 11

3 Grundlagen **12**
3.1 Eigenschaften und Probleme webbasierter Anwendungen 12
3.1.1 Das Transferprotokoll (HTTP) 12
3.1.2 Zustandssynchronisierung 16
3.1.3 Das Post-Redirect-Get Schema 18
3.2 Softwarearchitekturen 19
3.2.1 n-Tier Architekturen 19
3.2.2 Das Model-View-Controller Konzept (MVC) 23

4 Implementierung **27**
4.1 Auswahl einer geeigneten Plattform 27
4.1.1 Alternativen 28
4.1.2 Auswahl 33
4.2 ZOPE 33
4.2.1 Übersicht 35
4.2.2 ZPublisher - die Abarbeitung von Requests 36
4.2.3 ZODB - eine objektorientierte Datenbank 41
4.2.4 Acquisition 43
4.2.5 Page Templates - Präsentation dynamischer Inhalte . . . 43
4.2.6 Erweiterung durch Produkte 45
4.3 Das Web-Application-Framework „TaskEngine“ 46
4.3.1 Konzepte 46
4.3.2 Aufbau 52

4.3.3 Das Zusammenspiel 58
4.3.4 Das TaskEngine API 66
4.3.5 Vordefinierte Components 81
4.3.6 Beispiel-Tasks 94
4.4 Datenbankmodule 95
4.4.1 TableJoins - dynamisch generierte SQL-Abfragen 95
4.4.2 DCOracle2Row - Abfrageergebnisse als Objekte 105
4.4.3 ExtSQL 107
4.5 Datenmigration 109
4.5.1 Überblick 109
4.5.2 Datenmigrationstool 110

5 Zusammenfassung **119**

Literaturverzeichnis **121**

Abbildungsverzeichnis

4.1 Aufbau des TaskEngine Frameworks 53
4.2 Typische Task-Component-Hierarchie 56
4.3 Ablauf eines Request-Response Zyklus, Teil 1 59
4.4 Ablauf eines Request-Response Zyklus, Teil 2 60
4.5 Ausschnitt aus einem Datenmodell 97
4.6 Und-Verknüpfung über addPath() 99
4.7 Aus Abbildung 4.6 resultierende Abfrage 100
4.8 Oder-Verknüpfung über addPathsOr() 100
4.9 Aus Abbildung 4.8 resultierende Abfrage 101
4.10 Beispiel für die Verwendung der DCOracle2Row Ergebnisobjekte 107
4.11 Syntax der @migrate Datenstruktur 111
4.12 Konfigurationsbeispiel: Datenkonvertierungen 114
4.13 Konfigurationsbeispiel: Aufspaltung 115
4.14 Konfigurationsbeispiel: Aufspaltung Alternative 116
4.15 Konfigurationsbeispiel: Konsolidierung mehrerer Datenbanken . 118

1 Einleitung

Die Publikationsdatenbank der Technischen Universität Wien ist eine mittlerweile äußerst komplexe Web-Anwendung, die im Laufe ihrer Entstehungsgeschichte grundlegende Veränderungen sowohl in Hinsicht auf das Anforderungsprofil als auch auf den implementierten Funktionsumfang erfahren hat.

Nach einer ersten Implementierung auf Basis von Microsoft Access gelangte 2001 erstmals die noch heute verwendete PHP-basierte Lösung zum Produktiveinsatz. Diese erste webbasierte Version hat mit der heutigen Publikationsdatenbank jedoch nur noch das Grundgerüst gemeinsam. Immerhin wuchs der Programmcode in den letzten sieben Jahren von circa 7 500 auf derzeit etwa 60 000 Zeilen [Rie08]!

Auch die Anforderungen an das Produkt (Publikationsevaluierung, externe Installation bei den Austrian Research Centers) und der Umfang der verwalteten Daten wuchsen mit der Zeit dramatisch an, während sich die technischen Rahmenbedingungen ständig veränderten.

Aus diesen Gründen und vor allem auch weil die Zusammenführung der bisher für jede Fakultät der Universität getrennt gehaltenen Daten in eine einzige zentrale Datenbank überlegt wurde, war der langjährige und auch aktuelle Hauptentwickler der Anwendung, Professor Riedling, an Konzepten und Lösungsansätzen für eine Neuentwicklung der Applikation in Verbindung mit Migrationsstrategien der bestehenden Daten interessiert.

Diese Überlegungen waren schließlich die Motivation für die vorliegende Arbeit, deren Ziel es sein soll, einen Implementierungsvorschlag für das Grundgerüst einer Web-Anwendung zu liefern, welcher eine problemorientierte Programmierung der Geschäftslogik ermöglicht und das technische Eigenheiten und Unzulänglichkeiten einer Web-Anwendung (keine permanente Verbindung zum Client, Browser-Abhängigkeit, Layout-Details) soweit als möglich kapselt und zentral behandelt.

Auch die Thematik der Datenübernahme aus einem Altsystem und die potentiellen Probleme, die bei der Zusammenführung mehrerer Datenbanken und auch

beim Wechsel des Datenbankproduktes oder bei Änderungen im Datenmodell ergeben können, werden anhand eines Hilfsprogrammes zur Datenbankmigration und einiger Konfigurationsbeispiele kurz erörtert.

2 Aufgabenstellung

Klassische Web-Anwendungen investieren oft sehr viel Aufwand in die Entwicklung von Hilfskonstruktionen zur Behandlung webspezifischer Eigenheiten. Das Fehlen einer permanenten Verbindung zwischen der Anzeige (dem Browser) und der Programmlogik, die Zustandslosigkeit des HTTP-Protokolls und die ursprünglich (vor der Einführung von Cascading-Style-Sheets) fehlenden Möglichkeiten, Layout und Daten sauber zu trennen, sind nur einige Gründe, warum klassische GUI-Programme meist klarer strukuriert und besser auf die eigentliche Aufgabenstellung fokussiert sind.

Es sollte daher nach Möglichkeiten gesucht werden, diese Nachteile browserbasierter Anwendungen problemunabhängig zu lösen, um möglichst nahe an die Vorzüge der klassischen Anwendungsentwicklung heranzukommen und dem Anwendungsentwickler eine problemorientierte Programmierung zu ermöglichen.

2.1 Entwicklung einer modularen Web-Entwicklungsumgebung

Anwendungen mit mehrschichtigem Aufbau, die eine Trennung zwischen Layout, Applikationslogik und Datenhaltung implementieren, bieten, wie in Abschnitt 3.2.1 noch ausführlich beschrieben, zahlreiche Vorteile, die, bei steigender Komplexität der Aufgabe immer mehr ins Gewicht fallen.

Um die Entwicklung solcher Programme optimal unterstützen zu können, sollte natürlich auch das verwendete Framework Hilfestellungen zu diesen Teilgebieten liefern.

Nachdem die Anwendungsschicht im Idealfall nur auf die konkrete Aufgabenstellung ausgerichtet ist und lediglich auf die Schnittstellen zu den anderen beiden Schichten zugreifen muss, ist an dieser Stelle keine Framework-Unterstützung notwendig. Durch eine ereignisorientierte Implementierung des

Frameworks sollen in der Applikationslogik wiederverwendbare Programmmodule entstehen, die zum Beispiel über *Remote Procedure Calls* auch für Zugriffe die nicht über das World Wide Web erfolgen, verfügbar gemacht werden könnten.

Beim Layout der Anwendung soll die Möglichkeit der Ausgliederung von Darstellungseigenschaften mit Hilfe von *Cascading Style Sheets* (CSS) berücksichtigt werden. Auch das bei CSS realisierte Prinzip der Vererbung von Layout-Eigenschaften zwischen Darstellungselementen soll vom Framework unterstützt werden.

2.1.1 Abstrahierung der Datenbankzugriffe

Auf Seite der Persistenzschicht sollte die Datenbank ausreichend abgekapselt sein, damit ein späterer Wechsel des darunterliegenden Datenbankproduktes leichter möglich ist, Änderungen am Datenmodell leichter realisierbar sind und auch eine Aufgabenteilung zwischen mehreren Entwicklern möglich ist.

Zugriffe aus der Anwendungsschicht sollen auf logisch höherer Ebene stattfinden und nicht auf die konkrete Datenhaltung Rücksicht nehmen müssen. SQL-Abfragen sollen beispielsweise der Anwendungsschicht komplett verborgen bleiben. Darüberhinaus soll allerdings auch noch nach Möglichkeiten gesucht werden, SQL-Abfragen anhand einer Beschreibung der Verknüpfungsbeziehungen zwischen den Tabellen automatisch zu generieren. Dies ist bei komplexen dynamischen Suchabfragen nützlich: Je nach Formulierung der Suchanfrage und abhängig von den verwendeten Suchattributen soll von einem Modul das optimale Abfragestatement generiert werden, das nur die für die aktuelle Suche erforderlichen Tabellen verwendet.

Eine weitere wichtige Eigenschaft, die die Datenbankmodule erfüllen sollen, ist die aktive Unterstützung bei der Entwicklung einer sicheren Web-Anwendung. Laut [LTM06] ist die Verwundbarkeit durch SQL-Injections die zweithäufigste Sicherheitsschwäche bei Web-Anwendungen. Bei einem solchen Angriff wird das ursprüngliche SQL-Statement durch Benutzereingaben so verändert, dass sich die Semantik der Abfrage verändert. Es können dadurch, abhängig von den Eingabedaten, böswillige Abfragen oder sogar Modifikationen an den Daten abgesetzt werden.

Ursache dieser Verwundbarkeit ist die direkte Verwendung von Benutzereingaben in SQL-Statements bei fehlender oder unzureichender Filterung von SQL-Steuerzeichen in diesen Daten. Daher bieten sich zur Lösung dieses Problems

zwei Möglichkeiten an: Eine Option ist es, sämtliche variable Parameter, die in SQL-Statements verwendet werden, durchgehend und konsequent zu maskieren (*escapen*). Dieser Ansatz ist jedoch fehleranfällig und birgt unter anderem auch die Gefahr einer mehrfachen Maskierung. Der bessere und von praktisch allen modernen Datenbanken und Datenbank-APIs unterstützte Weg ist jedoch die vom SQL-Statement getrennte Übergabe der variablen Parameter als sogenannte Bind-Parameter. Hier werden Benutzereingaben von der Datenbank-Engine immer nur als Daten behandelt und können nie als SQL-Anweisung interpretiert werden. Dieser Ansatz hat den Vorteil einer höheren Effizienz der Abfragen (siehe Seite 98) und soll daher in diesem Projekt angestrebt werden.

2.2 Konzepte zur Datenmigration

Bei der Neuimplementierung eines bestehenden datenbankbasierten Softwareprojektes sind in den meisten Fällen auch Änderungen an der zugrundeliegenden Datenbank erforderlich. Es ist daher oft sinnvoll, die Daten aus dem Altsystem in eine neue Datenbank zu übertragen.

Damit ein solcher Wechsel transparent und nachvollziehbar erfolgen kann, ist die detaillierte Vorbereitung einer Migrationsstrategie notwendig, die alle erforderlichen Schritte bei der Übersiedlung der Daten dokumentiert. Möglicherweise dabei auftretende Probleme müssen analysiert und vorzugsweise automatisiert gelöst werden, der gesamte Migrationsvorgang sollte skriptgesteuert reproduzierbar sein.

Es soll daher beispielhaft ein parametrisierbares Konfigurationstool entwickelt werden, mit dem dieser Prozess realisiert werden könnte. Desweiteren soll untersucht werden, welche Probleme bei einer automatisierten Datenübertragung zu erwarten sind, wobei im Speziellen die Zusammenführung mehrerer Datenquellen und Änderungen im Datenmodell berücksichtigt werden sollen.

3 Grundlagen

Dieses Kapitel beschreibt einige wichtige technische Grundlagen, die für die Entwicklung webbasierter Anwendungen relevant sind. Nachdem sich Web-Anwendungen von herkömmlichen, nicht-verteilten Programmen in vielen Punkten deutlich unterscheiden, ergeben sich auch einige Probleme, die vom Entwickler oft übersehen oder nur im Anlassfall behandelt werden und nicht von vornherein in die Konzeption des Programmes miteinbezogen werden.

Der erste Abschnitt beschreibt solche Eigenschaften und Probleme, für die im Implementierungskapitel schließlich Lösungen vorgestellt werden.

Die nächsten Abschnitte stellen Design-Patterns vor, die den modularen Aufbau komplexerer Applikationen zum Ziel haben und die bei der Erstellung des praktischen Teils dieser Arbeit als Vorlage dienten.

3.1 Grundlegende Eigenschaften und Probleme webbasierter Anwendungen

3.1.1 Das Transferprotokoll (HTTP)

Web-Applikationen benutzen das *Hypertext-Transfer-Protocol* (*HTTP*) als Transport-Protokoll, welches bereits im Jahr 1991 entworfen und erstmals implementiert wurde [BL91]. Ein erster Internet-Draft wurde ein Jahr später von Tim Burners-Lee veröffentlicht [BL92]. Das ursprüngliche Einsatzgebiet dieses Protokolls war hauptsächlich der Transfer statischer Seiten. Dies ist leicht daran zu erkennen, dass schon im ersten Entwurf festgelegt wurde, dass HTTP-Requests idempotent sein sollen, also dass mehrmalige Aufrufe des selben Requests immer das selbe Ergebnis (also die selbe Seite) liefern sollen. Diese Forderung ist mit modernen, dynamischen Web-Applikationen oft nicht vereinbar. Daher wurde der ursprüngliche Vorschlag später etwas entschärft,

indem die Ergebnisse (also die HTTP-Responses) um eine Ablaufzeit ergänzt wurden.

Die meisten Protokolleigenschaften blieben allerdings erhalten, so dass dynamische webbasierte Anwendungen oft mit Problemen konfrontiert sind, die bei herkömmlichen, nicht verteilten GUI-Anwendungen oder bei klassischen Client-Server Anwendungen keine Rolle spielen. Im folgenden werden einige dieser Probleme beschrieben.

3.1.1.1 Das Request-Response Prinzip

HTTP arbeitet nach einem *Request-Response*-Prinzip. Der Ablauf der Kommunikation ist genau und strikt festgelegt. Ausgangspunkt einer Anfrage muss immer der Browser (oder allgemeiner *User-Agent*) sein, der Server beantwortet die Antworten schließlich. Damit ist auch die Richtung der Kommunikation festgelegt: Jeder Request muss immer vom User-Agent ausgehen, der Server kann nie von sich aus aktiv werden. Er reagiert nur und antwortet auf Anfragen. Damit bietet HTTP auch serverseitig keinerlei Möglichkeit, den User-Agent ereignisgesteuert zu benachrichtigen. (Etwa über Veränderungen oder den Fortschritt einer asynchron abgesetzten Aktion.) Es handelt sich um eine typische Verteilung nach dem Client-Server-Prinzip. Diese Kommunikationsart ergibt sich auch aus dem Prinzip der Zustandslosigkeit des Servers, nach dem das World-Wide-Web ursprünglich gestaltet wurde. (Siehe unten.)

Die Entwicklung moderner, hochinteraktiver Web-Applikationen, die beispielsweise asynchron auf Serverereignisse reagieren sollen, erfordern daher den Einsatz zusätzlicher Hilfsmittel wie etwa der Verwendung der *Ajax*-Technologie (*Asynchronous JavaScript and XML*). Hier kann zum Beispiel durch den Aufbau mehrerer asynchroner Kommunikationskanäle zum Server dynamisch auf Ereignisse reagiert werden, es können mit Hilfe von „Unter-Requests“ Daten für die Anzeige in der aktuellen Seite nachgefordert werden oder ähnliche Effekte erzielt werden, die man von klassischen GUIs kennt. Der Einsatz dieser Techniken setzt allerdings relativ neue Browser-Versionen voraus - eine Anforderung, die bei Anwendungen mit nicht kontrollierbarer, heterogener Client-Landschaft oft nicht leicht zu erfüllen ist.

3.1.1.2 Zustandslosigkeit

HTTP ist per Definition ein zustandsloses (*stateless*) Protokoll. Das bedeutet, dass der Server den Zustand seiner Clients nicht kennt, zumindest nicht auf Ebene des Transfer-Protokolls. Für den HTTP-Server ist ein Request nach Absenden einer Antwort abgeschlossen, es werden keinerlei Client-spezifische Informationen abgespeichert. Der Grund dafür ist, dass ein Server im Allgemeinen mit sehr vielen Clients kommuniziert, es wäre also im Allgemeinen sehr aufwendig für den Server, sich den Zustand aller Clients zu merken. Da HTTP, wie bereits erwähnt, für den Transfer statischer *Hypermedia*-Seiten entworfen wurde und darüberhinaus die Forderung nach Idempotenz besteht, darf der Zustand eines Clients auch für die Beantwortung einer Anfrage keine Rolle spielen.

Für den Programmierer einer Web-Applikation ergeben sich daraus zwei Möglichkeiten:

- Die Applikation wird serverseitig zustandslos gestaltet. Das bedeutet, dass sämtliche Informationen, die für die Bearbeitung eines Requests notwendig sind, in der Anfrage selbst enthalten sein müssen. Das ist eine Forderung des *Representational State Transfer* Architekturstils, einem in der Dissertation des HTTP-Mitbegründers Roy Fielding [Fie00] vorgestellten Modell für das Design von Web-Projekten.

  Nachdem aber praktisch jede dynamische Anwendung auf Zustandsdaten angewiesen ist, müssen diese Zustandsdaten dann clientseitig gespeichert werden. Dazu existieren mehrere Möglichkeiten:

  – Die Daten können Bestandteil des URLs sein. (Pfad oder *Query-String*)

  – Sie können in (eventuell versteckten) Formularfeldern abgelegt werden, also in der vom User-Agent angezeigten Seite gespeichert sein.

  – Der Programmierer kann sich auch einer Erweiterung des HTTP-Protokolls bedienen, die genau für das Weiterreichen von Zustandsinformationen über eine Request/Response Transaktion hinweg entwickelt wurde: *Cookies* [KM97].

  Bei der Speicherung von Zustandsdaten beim User-Agent ist zu beachten, dass beim Weiterreichen von Zustandsdaten über mehrere Requests hinweg, bei unsachgemäßer Programmierung ein Sicherheitsproblem entstehen kann. Da die Daten beim Endbenutzer gespeichert sind und über

das Netz zum Server gesendet werden, ist eine Veränderung der Daten am Client und am Übertragungsweg möglich. Eine Überprüfung der Session-Daten ist daher nicht nur nach der Eingabe durch den Benutzer, sondern auch vor der Verarbeitung durch die Anwendung notwendig [CEH+02]. Diese mehrfache Überprüfung kann bei komplexeren Problemstellungen aufwendig sein.

Zustandsloses Design und Idempotenz sind eine Voraussetzung für die Entwicklung cachebarer Seiten, eine Eigenschaft, die bei der Entwicklung stark frequentierter, aber wenig individualisierter Web-Anwendungen eine große Rolle spielt.

- Das Abspeichern von Zustandsdaten und die ihre Zuordnung zum Client wird auf höherer Ebene im Server implementiert. Das ist bei Applikationen sinnvoll, die eine größere Menge an Zustandsdaten benötigen, beziehungsweise bei denen die ständige Neugenerierung dieser Daten aus bestimmten Schlüsselinformationen zu aufwendig wäre. Auch die mehrmalige Validierung der Informationen bei Anwendungen mit mehren Dialogschritten kann, wie bereits oben erwähnt, ein Argument für diesen Ansatz sein.

  Anwendungen, die Daten am Server manipulieren, sind meist auf diese Art implementiert, da auch eine Veränderung des gespeicherten Datenbestandes serverseitig eine Veränderung des Zustandes bewirkt. Eine solche Applikation, die sich rein auf den clientseitigen Zustand verlässt, kann bei unvorsichtiger Programmierung leicht zu Problemen führen. Näheres dazu folgt im nächsten Abschnitt.

3.1.1.3 Caching

Das HTTP-Protokoll erlaubt und unterstützt[1] das Cachen von Anfragen. Diese Zwischenspeicherung von Inhalten (von HTTP-Responses) kann an mehreren Orten passieren: Im User-Agent, im sogenannten Browser-Cache, in einer Zwischenschicht bei der Übertragung, in einem Web-Proxy oder im Webserver.

Während diese Eigenschaft der Zwischenspeicherung die Skalierbarkeit von Web-Services verbessern, die Latenzzeit minimieren und die Effizienz der Datenübertragung maximieren soll, kann sie bei dynamischen Web-Anwendungen zum Problem werden. Bei Anfragen, deren Ergebnis sich über die Zeit ändern

[1] zumindest seit HTTP/1.1 explizit

kann, oder bei Requests, die den Zustand der Applikation verändern, muss sichergestellt werden, dass die Seiten tatsächlich aktuell vom Server geholt werden und daher sämtliche Caching-Mechanismen deaktiviert werden.

Auch bei Seiten, die vertrauliche Informationen beinhalten, ist eine Zwischenspeicherung der Seiten in Caches unerwünscht.

3.1.2 Zustandssynchronisierung

Da bei Web-Applikationen keine permanente Verbindung zwischen User-Agent und Applikationsserver existiert, besteht die Gefahr, dass die auf beiden Seiten gespeicherten Zustände durch bestimmte Aktionen und Mechanismen divergieren. Dies ist nur ein Problem, wenn der Zustand der Anwendung überhaupt im Server (etwa in Form einer *Session*) gespeichert wird. Eine Alternative, die sich für einfache Anwendungen, die vorwiegend lesend auf Daten zugreifen, anbietet, wurde in 3.1.1.2 vorgestellt.

Die Gefahr inkonsistenter Zustände kann sich aus mehreren Gründen ergeben:

3.1.2.1 Caching-Effekte

Wie bereits erwähnt, muss gewährleistet sein, dass vom User-Agent initiierte Zustandsveränderungen den Server auch wirklich erreichen. Theoretisch kann dies durch geeignetes Setzen von Response-Parametern (`Cache-Control`-Header in HTTP/1.1 [FGM+99] bzw. `Pragma`-Header[2] in HTTP/1.0) und durch Angabe einer Verfallszeit (`Expires`-Header ab HTTP/1.0) bei den Antworten erreicht werden. Da diese Angaben aber nicht von allen User-Agents honoriert werden, werden darüberhinaus oft noch die Request-URLs durch Anhängen eindeutiger *Query-Strings* eindeutig gemacht, die Verwendung einer gecachten Seite als Antwort ausschließen.

3.1.2.2 Browser-Navigationselemente

In klassischen Client-Server-Anwendungen bestimmt der Entwickler das Aussehen und die Funktionalität des Clients. Der Benutzer kann sich im Programm nur durch Verwendung der ihm vorgegebenen Navigationselemente bewegen.

[2] Berücksichtigung ist keine Muss-Anforderung im Standard!

Bei webbasierten Applikationen hat der Programmierer keinen Einfluss auf die verwendeten User-Agents. Entsprechend Ihres Haupteinsatzgebietes, der Hypermedia-Navigation, verfügen herkömmliche Web-Browser aber über zusätzliche, nicht von der Anwendung zur Verfügung gestellten, Navigationselemente.

- *Back-* und *Forward-* Buttons sowie die *Browser-History* ermöglichen das „Blättern“ innerhalb der Applikation. Gerade bei diesen Navigationsfunktionen verhalten sich User-Agents extrem unterschiedlich, was die Verwendung des Browser-Caches betrifft, da der Standard hier *vorschlägt*, dass bei Verwendung der History-Mechanismen die Caching-Einstellungen der entsprechenden Seiten ignoriert werden.

  Da die Verwendung der Navigationselementen in vielen GUI-ähnlichen Anwendungen ohnehin oft nicht sinnvoll ist, wird oft versucht, die Funktion dieser Steuerelemente, etwa mittels *JavaScript*, zu deaktivieren.

- Der *Reload-Button* veranlasst den Browser, die aktuell angezeigte Seite nochmals anzufordern. Wenn der letzte Request einen Zustandsübergang bewirkt hat und vor allem, wenn dadurch Daten im Server verändert wurden, kann eine mehrmalige Ausführung problematisch sein. In solchen Fällen ist eine spezielle Behandlung dieser Funktion notwendig.

- Web-Browser bieten außerdem die Möglichkeit der direkten Navigation zu bestimmten Seiten, entweder über die Verwendung von *Bookmarks*, der Browser-History oder über direkte Veränderung des URLs in der Adresszeile. Auch hier besteht natürlich das Potential, die Zustandbehandlung in der Anwendung durcheinanderzubringen. Sofern eine Applikation den direkten Einstieg an mehreren Stellen ermöglicht, muss der Entwickler also jederzeit auf Sprünge in der Anwendungslogik vorbereitet sein und entsprechende Maßnahmen treffen um die falsche Verwendung kontextabhängiger Daten zu verhindern.

3.1.2.3 Mehrere Browser-Fenster

Eine weitere Möglichkeit, die Zuordnung zwischen client- und serverseitigem Zustand zu stören, ist die Verwendung mehrerer Browser-Fenster. Gerade bei Verwendung von Cookies zur Zustandssynchronisation kann die Verwendung mehrerer Fenster zu unerwünschten Ergebnissen führen: Cookies sind einem Browser und nicht einem Browser-Fenster zugeordnet, bei der Verwendung

mehrerer Fenster können daher mehrere Client-Instanzen mit denselben Zustandsinformationen an unterschiedlichen Stellen im Programm eingreifen.

Die in diesem Abschnitt vorgestellten potentiellen Probleme bei der Zustandssynchronisierung können teilweise durch Setzen von Meta-Informationen in den Seiten und teilweise über Client-seitige Mechanismen wie beispielsweise Java-Script vermieden werden. Durch diese Techniken können die beschriebenen Effekte allerdings nie ausgeschlossen werden, schließlich handelt es sich um ein verteiltes System und die Lösungsansätze setzen außerdem voraus, dass sich der Client kooperativ verhält und die verwendeten Verfahren ordnungsgemäß unterstützt. Aufgrund der Vielfalt der am Markt befindlichen Clients und der Tatsache, dass sich der Client außerhalb des Einflussbereiches der Applikationsentwickler befindet, kann das nie garantiert werden. Es ist daher vorteilhaft, wenn die Web-Applikation einen Mechanismus vorsieht, zumindest zu erkennen, falls die client- und serverseitigen Zustände auseinanderlaufen.

3.1.3 Das Post-Redirect-Get Schema

In Zusammenhang mit den Navigationselementen des Browsers ergibt sich auch ein mögliches Problem bei der Übermittlung von Daten an den Server, das in Web-Entwicklerkreisen oft als *DoubleSubmit*-Problem bezeichnet wird:

Benutzereingaben werden meistens in Form von Formulardaten mit der *POST*-Methode des HTTP-Protokolls an den Server übermittelt. Diese Methode ist jedoch ein „normaler“ Request und nach dem in 3.1.1.1 beschriebenen Request-Response-Prinzip, muss der Server darauf mit einer Ergebnisseite antworten. In vielen Web-Applikationen handelt es sich dabei um eine normale HTML-Seite. Mit Hilfe der Browser-Navigationselemente (Reload-, Back-, Forward- Buttons, etc.) kann der Benutzer jedoch genau diese Ergebnisseite wieder anfordern, was den User-Agent dazu veranlasst, den ursprünglichen (POST-) Request zu wiederholen. Wenn dieser Request jedoch Daten im Server verändert hat, ist eine Wiederholung derselben Aktion meist nicht erwünscht.

Ein anerkanntes Rezept, diesen Effekt zu beseitigen, ist das in einem Artikel von Michael Jouravlev [Jou04] vorgestellte *POST-REDIRECT-GET* Pattern (*PRG*). Dieser Ansatz teilt den problematischen Request indirekt in zwei Requests:

Der Server antwortet auf einen POST-Request des Client (in dem die Formulardaten übermittelt werden) nicht mit einer Ergebnisseite, sondern mit einem HTTP-Redirect. Durch diese Umleitung zeigt der User-Agent vorerst

keine neue Seite an, sondern verfolgt das Ziel der Umleitung, schickt also eine neue Anfrage. Diese wird jedoch als „gewöhnlicher“ GET-Request abgeschickt, es werden keine Daten mehr an den Server übermittelt, es wird lediglich die nächste HTML-Seite angefordert.

Diese Vorgangsweise hat den Vorteil, dass die erste Serverantwort (der HTTP-Redirect) nicht im Browser angezeigt wird und daher der zugehörige Request (der problematische POST-Request) nicht in der Browser-History gespeichert wird. Erst die Ergebnisseite und der entsprechende GET-Request werden in der History eingetragen. Diese Anfrage sollte den Server jedoch nur dazu veranlassen, den aktuellen Zustand auszugeben, es werden keine Daten mehr verändert. Das Drücken des Reload-Buttons führt also lediglich zur Anzeige des aktuellen Serverzustandes, es kommt zu keinem DoubleSubmit.

Durch die Aufteilung in zwei Teilaufgaben wird eine klare Trennung der Applikation nach dem in 3.2.2 beschriebenen Model-View-Controller-Konzept ermöglicht: Der erste Schritt (der POST-Request) bewirkt eine Veränderung der Daten des Model. Der zweite Schritt entspricht dem Aktualisieren der View.

3.2 Softwarearchitekturen

3.2.1 n-Tier Architekturen

Die meisten interaktiven Anwendungen haben grundsätzlich folgende Aufgaben zu erfüllen:

- Datenspeicherung,
- Geschäftslogik (Anwendungslogik), sowie
- Präsentation der Daten

Es bietet sich daher an, diese allgemeinen Aufgaben zur Strukturierung von Softwaresystemen heranzuziehen, eine Anwendung also in Schichten zu gliedern, die obige Aufgaben erfüllen [DH03]. Im Sinne modularer Softwarearchitektur sollten diese Schichten nur mit benachbarten Schichten kommunizieren, das heißt eine Schicht i stellt nur Dienste für die Schicht $i+1$ zur Verfügung und nutzt nur Dienste der darunterliegenden Schicht $i-1$. Die Schichten sind also lose gekoppelt, es existiert eine harte Trennung der Schichten [DH03, Mül04].

3.2.1.1 Klassische (two-tier) Systeme

In den Anfängen der verteilten Systeme folgten die meisten Anwendungen dem Client/Server Prinzip. Es handelte sich hierbei um eine zweischichtige (*two-tier*) Architektur: Der Client ist für die Präsentation der Daten, also für deren Aufbereitung und Anzeige zuständig, der Server übernimmt die Speicherung der Daten. Je nachdem, ob der größte Teil der Geschäftslogik im Client oder im Server angeordnet ist, spricht man von *Fat Clients* bzw. *Fat Servers* [Lew98].

Fat Clients Hier liegt die Applikationslogik größtenteils im Client. Das entlastet zwar einerseits den Server von möglicherweise aufwendigen Rechenoperationen und verteilt diese in Richtung User, andererseits schränkt eine solche Konstruktion die Wartbarkeit des Gesamtsystems deutlich ein: Änderungen in der Business-Logic werden erst wirksam, wenn sämtliche Clients getauscht sind. Änderungen am Interface zur Datenbank-Schicht erfordern entweder ebenfalls einen Austausch sämtlicher Clients oder das Weiterpflegen der alten Schnittstelle, um Kompatibilität zu alten Clients zu gewährleisten.

Fat Servers Nachdem der größte Teil der Applikationslogik im Server verankert ist, der Client also „schlank“ gehalten ist (*Thin-Client*) und die Daten hauptsächlich einfach darstellen muss, sind Änderungen an der Software meist leicht zu realisieren. Updates müssen nur einer Stelle (serverseitig) passieren und wirken sich sofort auf alle Clients aus. Bei rechenintensiven Aufgaben oder einer hohen Anzahl an Clients, wirkt sich die Tatsache, dass der Hauptteil der Anwendung im Server abläuft, hingegen natürlich nachteilig aus.

3.2.1.2 n-Tier Systeme

Wenn ein System anhand eingangs erwähnter Aufgabenbereiche in die drei Schichten Datenspeicherung (Persistenzschicht), Geschäftslogik (Anwendungsschicht) und Präsentation (Präsentationsschicht) gegliedert wird, spricht man von einer *dreistufigen Architektur* (*Three tier architecture*).

Presentation Tier Die Präsentationsschicht ist für die (graphische) Darstellung der Informationen und deren benutzerspezifische Aufbereitung in der Benutzeroberfläche zuständig. Eine weitere wichtige Aufgabe ist die Dialogkontrolle [DH03]: Es wird auf Benutzereingaben reagiert und gegebenenfalls unter Zuhilfenahme von Funktionen der Anwendungsschicht

Daten für die Folgeaktion angefordert. Die Ergebnisse werden wiederum aufbereitet und dargestellt. Benutzerspezifische Einstellungen, die für die Verarbeitung, Auswertung oder Anzeige der Informationen relevant sind, können hier verarbeitet werden [Hir96].

Logic Tier Die Anwendungsschicht ist für die Abbildung der Geschäftsprozesse zuständig und sollte die Daten im logisch besten Format, ohne Rücksicht auf die Art der Darstellung der Informationen für den jeweiligen Benutzer, an die Präsentationsschicht übermitteln. Hier sitzt eine i.A. auch von mehreren Anwendungen verwendete Geschäftslogik, die in einer vom Endbenutzer unabhängigen Form realisiert ist. Der Zugriff auf die gespeicherten Daten sollte, unabhängig von der aktuell verwendeten Realisierung des Datenspeichers, über eine genau definierte Schnittstelle zur Persistenzschicht erfolgen [DH03]. Anforderungen an die Anwendungsschicht sind üblicherweise außerdem Transaktionsmanagement und Datensicherung, d.h. Authentisierung und Autorisierung, also die Überprüfung der Zugangsberechtigung und das Rollenmanagement [Hir96].

Data Tier Die Persistenzschicht ist für die dauerhafte (persistente) Speicherung der von der Applikation benötigten Daten zuständig. Nur diese Schicht kennt das aktuell verwendete Datenmodell, die Zugriffe aus der Anwendungsschicht erfolgen auf abstrakterer, funktionalerer Ebene. Im Falle einer dahinterliegenden relationalen Datenbank (*RDBMS*) wird hier auch die Umwandlung der Objektdaten in ein relationales Schema durchgeführt, das sogenannte O/R-Mapping (*object to relational mapping*) [DH03].

Erweiterungen dieses Schichtenkonzeptes führen zur Einführung einer oder mehrerer zusätzlicher Schichten, also zu *n-Schichten Architekturen*. Die Aufgabe dieser zusätzlichen Schichten kann je nach Einsatzgebiet variieren. Vor allem bei webbasierten Anwendungen ist es oft sinnvoll und weit verbreitet, die Präsentationsschicht von der eigentlichen Benutzerschnittstelle zu trennen. Hier wird das User-Interface in eine Client-Schicht in den Web-Browser ausgelagert, während der restliche Teil der Präsentationsschicht im Webserver abläuft. Bei dem Client handelt es sich also um einen „Thin Client“ [Mül04].

Andere Beispiele für die Einführung zusätzlicher Schichten sind die Einführung von Multiplexern zur Lastverteilung oder zur Erreichung von Fehlertoleranz, Zugriffskontrollschichten (*access control tiers*) zwecks Vereinfachung oder logischer Ausgliederung der Sicherheitsaspekte an Spezialisten, oder die Zwischen-

schaltung von Schnittstellenschichten zur Abstraktion der Kommunikation mit Legacy-Systemen.

3.2.1.3 Vorteile mehrschichtiger Systeme

Wie bereits erwähnt ist die klare Trennung der Aufgaben einer Anwendung in einzelne Schichten ein Mittel zur Modularisierung von Softwareprojekten. Diese Aufspaltung bringt zahlreiche Vorteile, die im Folgenden erläutert werden [DH03]:

- Aufgrund der Abgrenzung der Geschäftslogik von den übrigen Schichten ergibt sich ein übersichtlicherer und besser lesbarer Programmcode. Die Funktionen der Business Logic beispielsweise befassen sich ausschließlich mit den fachlichen Teilen der Anwendung, mit der Modellierung der Geschäftsprozesse. Hier muss keine Rücksicht auf die Darstellung der Informationen oder die Speicherung der Daten genommen werden. Äquivalente Überlegungen gelten für die übrigen Schichten.
- Es ergibt sich die Möglichkeit größere oder kompliziertere Projekte gemäß der Schichtung auf spezialisierte Teams zur Implementierung zu verteilen. Diese Einteilung hat auch den Vorteil, dass die Abhängigkeiten zwischen den einzelnen Expertengruppen aufgrund der geforderten harten Trennung einfach gehalten werden. Sie erfolgt über schlanke, genau definierte Schnittstellen zwischen den Schichten.
- Es besteht die Möglichkeit, große Teile der Präsentationsschicht bzw. der Persistenzschicht, sofern sie in Form von Bibliotheken entwickelt wurden, in anderen Anwendungen wiederzuverwenden. Andererseits können, wie bereits erwähnt, auch die Funktionen der Anwendungsschicht mehreren ähnlichen Anwendungen zur Verfügung gestellt werden.
- Auch die Wiederverwendbarkeit innerhalb eines Projektes wird durch dieses Architekturkonzept gefördert und Redundanzen dadurch vermieden. Beispielsweise ist es möglich, für eine Anwendung mehrere Benutzeroberflächen anzubieten oder die Oberfläche zu ersetzen, ohne die Applikationslogik angreifen zu müssen.
- Durch die lose Kopplung zwischen den Schichten sind Fehler leichter zu lokalisieren und zu isolieren. Auch Änderungen in den einzelnen Schichten sind dadurch leichter möglich. Insgesamt wird also die Wartbarkeit der Anwendung verbessert.

- Beim Einsatz von Thin-Clients (z.B. Browsern) in einer vierschichtigen Architektur entfällt die clientseitige Installation anwendungsspezifischer Software komplett. Außerdem wird bei so konzipierten Anwendungen das Ausrollen von Änderungen nochmals erleichtert, da sich, wie bereits erwähnt, sämtliche Änderungen unmittelbar auf alle Clients auswirken und die Notwendigkeit, weiterhin ein kompatibles Interface für alte Clients anzubieten, entfällt.
- Die klar definierten Schnittstellen erleichtern auch die Aufteilung der Anwendung auf mehrere Rechner, wie es in komplexeren Szenarien aufgrund der verbesserten Skalierbarkeit oder auch wegen der Abteilungszuständigkeiten oft notwendig ist.

3.2.2 Das Model-View-Controller Konzept (MVC)

Nachdem die Präsentation der Daten eine zentrale Funktion interaktiver Anwendungen ist, kann das Softwaredesign dieser Funktionalität von entscheidender Bedeutung für die Skalierbarkeit, Wartbarkeit, zukünftige Erweiterbarkeit und Portierbarkeit einer Applikation sein.

Das Model-View-Controller Konzept (MVC) ist ein relativ altes Architekturmuster aus den 70er Jahren, dessen Ursprung eng mit der objektorientierten Programmiersprache und Entwicklungsumgebung Smalltalk-80 verknüpft ist, in deren Userinterface MVC eine wichtige Rolle spielt [KP88]

Grundgedanke ist eine Unterteilung einer interaktiven Applikation in drei Teile: Kernfunktionalität und Datenhaltung (*Model*), Darstellung (*View*) und Ablaufsteuerung (*Controller*). Anwendungsdaten werden im Modell gekapselt und sind unabhängig von der aktuell verwendeten Ausgabeform der Daten.

Der Controller kommuniziert im klassischen Konzept mit den anderen Komponenten über *Messages*, die Reaktion auf Änderungen im System (Benutzereingaben oder Änderungen der Daten) erfolgt ereignisgesteuert über Event-Handling-Mechanismen.

Model Das Model repräsentiert die Geschäftslogik einer Applikation. Hier ist die Kernfunktionalität der Anwendung angesiedelt, in diesem Teil werden auch die Daten der Anwendung gespeichert und den Views über eine funktionale Schnittstelle in Form fachlicher Objekte zur Verfügung gestellt. Bezogen auf ein mehrschichtige Architektur ist das Model nicht

der Präsentationsschicht sondern der Applikationsschicht zuzuordnen [DH03].

View Im View-Teil einer Anwendung erfolgt die Darstellung der Informationen über das User-Interface. Über eine definierte Schnittstelle bezieht diese Komponente die dazu benötigten Daten vom Model. Darüberhinaus werden dem Benutzer Objekte zur Dateneingabe bereitgestellt. Welche Art von Darstellungselementen verwendet wird, ist nur vom vorliegenden View und eventuell vom Controller abhängig, nicht aber vom Model. In einem klassischen GUI beispielsweise kommen Fenster und darin sogenannte *Widgets* wie Tabellen, Labels, Checkboxen, Eingabefelder usw. zum Einsatz. In einer Web-Oberfläche sind es HTML-Seiten mit ähnlichen Komponenten (z.B. HTML-Formularelemente) [DH03]. Darüberhinaus besteht auch die Möglichkeit der Verschachtelung, das heißt, Views können auch Unter-Views (*Subviews*) enthalten bzw. von *Superviews* verwendet werden. Die Kommunikation mit den Subviews erfolgt wieder mit Hilfe von *Messages* [KP88].

Controller Aufgabe dieses Teils ist die Dialogkontrolle. Über die Benutzeroberfläche werden Aktionen gesetzt, die im Controller Ereignisse (*Events*) auslösen. Die Aufgabe der Ablaufsteuerung ist es nun, diese Events zu verarbeiten, zu überprüfen und gegebenenfalls entsprechende Funktionen der Anwendungsschicht aufzurufen. Schließlich wird, wenn notwendig, die Anzeige einer aktualisierten Oberfläche mit Hilfe von *Messages* an den View veranlasst.

Die Überprüfung der Eingabedaten im Controller darf aber keinesfalls anwendungsspezifische Logik beinhalten. In der Dialogkontrolle werden die Eingaben daher ausschließlich auf Plausibilität geprüft, weitere, fachliche Validierungen bleiben dem Model und damit der Anwendungsschicht vorbehalten [DH03]

Nachdem Controller Ereignisse verarbeiten, die von Elementen der View-Komponenten ausgelöst werden und darüberhinaus für sämtliche Abläufe in der Benutzeroberfläche zuständig sind, sind sie oft eng mit der Darstellung verbunden. Die beiden Teile sind daher meist nicht getrennt voneinander verwendbar [DGH03]. Die Kombination von View und Controller entspricht der Präsentationsschicht in einer dreischichtigen Architektur (siehe 3.2.1.2, S. 20).

Die Beziehung dieser beiden Teile zur Model-Komponente sind, wie bereits durch die Schnittstelle zwischen *Presentation Tier* und *Logic Tier* aus Kapitel 3.2.1 nahegelegt, asymmetrisch: View und Controller müssen das Model

kennen, schließlich sind sie ja von dessen Funktionalität abhängig. Umgekehrt darf allerdings das Model keinerlei Kenntnis vom verwendeten View-Controller-Paar haben. Diese Unabhängigkeit ist wichtig, andernfalls wäre die Austauschbarkeit des User-Interfaces gefährdet.

3.2.2.1 Vorteile des Model-View-Controller Architekturprinzips

Eine wichtige Eigenschaft des Model-View-Controller Konzeptes ist die strikte Trennung von Ablaufsteuerung und Darstellung. Dadurch wird nicht nur die Modularität der Anwendung gefördert sondern auch einige weitere positive Eigenschaften:

- Die Übersichtlichkeit wird gesteigert, da der Dialogablauf an einer zentralen Stelle einsehbar wird und nicht über die Darstellungselemente verstreut ist
- Änderungen in der Präsentationsschicht werden erleichtert.
- Es besteht die Möglichkeit, mehrere (ähnliche) Views unter Beibehaltung eines einzigen Controllers anzubieten. (Beispiel: benutzerspezifische *Themes*)
- Der ereignisgesteuerte Ablauf ermöglicht eine synchronisierte Darstellung, die sich automatisch aktualisiert. Änderungen in den Anwendungsdaten oder im Zustand der Applikation können Events auslösen, die sämtliche Views auf den neuesten Stand bringen [DGH03].
- Die Austauschbarkeit von View und Controller fördern die Portierbarkeit auf andere Plattformen. Meist sind nur diese beiden Teile an eine konkrete Ablaufumgebung gebunden. Das Model und damit die Applikationslogik kann meist plattformunabhängig implementiert werden und kann daher einfach auf andere Systeme portiert werden [DGH03].

Angesichts dieser Vorteile hat das MVC-Modell weite Verbreitung gefunden und liegt vielen aktuellen GUI-Implementierungen und Frameworks zu Grunde. Bekannte Beispiele sind das Qt-Toolkit, Swing (Java), Ruby on Rails oder das wxWidgets-Toolkit. Vor allem in umfangreichen Projekten macht sich die Aufteilung der Komponenten oft bezahlt.

3.2.2.2 Probleme

Dennoch stehen den zahlreichen Vorzügen auch potentielle Probleme gegenüber [DGH03]:

- Vor allem in kleinen Projekten ist der Aufwand der Trennung der Präsentation in mehrere Teile oft übertrieben und steigert nur die Komplexität.
- Die Ereignissteuerung kann, in Verbindung mit Benutzereingaben, Ineffizienzen bei der Bearbeitung von Dialogen bedeuten. Bei schlechter Implementierung können bestimmte Benutzerinteraktionen sehr viele Ereignisse auslösen, die wiederum jeweils möglicherweise aufwendige Aktionen in der Anwendungsschicht anstoßen. Genauso können mehrere Änderungen der Anwendungsdaten auch viele Aktualisierungs-Events an die Views bewirken. Hier bietet sich eine Zusammenfassung der Benachrichtigungen an.
- Aufgrund der bereits erwähnten, oft starken Abhängigkeit zwischen View und Controller, deren Aufgabenbereiche in manchen Anwendungen nicht leicht trennbar sind, ist eine getrennte Verwendung oft nicht möglich.

3.2.2.3 Dokument-View Architektur

In manchen Anwendungen sind View und Controller so eng miteinander verknüpft, dass eine saubere Implementierung in zwei Teilen nicht mehr möglich ist. Es wird dann gelegentlich eine Aufweichung des MVC-Konzeptes verwendet, die sogenannte *Dokument-View* Architektur. Hier sind View und Controller aus MVC zu einer Komponente zusammengefasst. Der View-Teil dieser Variante umfasst die Aufgaben der Benutzerschnittstelle, also die Aufgaben der Darstellung sowie die Verarbeitung der Eingaben. Der *Document*-Teil entspricht dem Model im klassischen Konzept, erweitert um einen Event-Handling Mechanismus. Trotz der Aufgabe einiger Vorteile des MVC-Modells ist auch hier noch die Realisierung mehrerer *Views* zu einer Anwendung möglich. Eine bekannte Implementierung des Document-View-Architekturmodells ist die Microsoft Foundation Class Library (MFC).

4 Implementierung

In diesem Kapitel wird die Implementierung eines Frameworks für die Erstellung von webbasierten Applikationen beschrieben. Das entwickelte Framework eignet sich besonders zur Entwicklung von session-orientierten, komplexeren Anwendungen, die sich über mehrere Dialogschritte erstrecken und Datenbankzugriffe benötigen.

4.1 Auswahl einer geeigneten Plattform

Eine grundlegende und nicht nur für die Implementierung sondern auch für den zukünftigen Erfolg des Projektes wesentliche Entscheidung sind die Auswahl der verwendeten Programmiersprache und gegebenenfalls des eingesetzten Web-Frameworks.

Bei der Betrachtung der in Frage kommenden Programmiersprachen wurde besonders auf folgende Kriterien geachtet:

- Die Programmiersprache sollte objektorientierte Programmierung beherrschen und die Anwendung dieses Programmierparadigmas auch durch eine übersichtliche, klare, und leicht lesbare Syntax fördern.
- Es sollte sich um eine dynamische High-Level-Programmiersprache handeln, in der Rapid-Prototyping möglich ist, ohne aufwändige Details wie Speicherallozierung und manuelle Typdefinitionen berücksichtigen zu müssen.
- Die Performance sollte ausreichen um auch auf Systemen mit schwächeren Systemresourcen die erhöhten Anforderungen an eine Web-Applikation (interaktive Bedienbarkeit, größerer Pro-Request-Aufwand durch das zustandslose Prinzip des HTTP-Protokolls) erfüllen zu können.

- Es werden nur gut etablierte Programmiersprachen in Betracht gezogen. Sie sollten einer stabilen Version verfügbar sein und über eine breite Benutzerbasis verfügen damit auf umfangreiche Community-Ressourcen zurückgegriffen kann.
- Die Sprache sollte aktiv entwickelt und unterstützt werden.
- Sie sollte unter einer freien Software-Lizenz stehen, mit der sich für die damit entwickelte Software möglichst wenig Einschränkungen ergeben.

In den vielen Fällen bringt die Programmiersprache selbst kaum Unterstützung für bei der Web-Programmierung immer wiederkehrende Tätigkeiten wie Templating, Session Management, Zugriffskontrolle oder URL-Mapping mit. Es bietet sich dann die Verwendung eines Web-Application-Frameworks an.

4.1.1 Alternativen

Im folgenden Abschnitt werden einige Programmiersprachen und Web-Frameworks, die in der engeren Auswahl standen, kurz beschrieben und auf ihre Eignung untersucht.

4.1.1.1 PHP

PHP ist die im Bereich der Web-Programmierung wahrscheinlich am weitesten verbreitete Programmiersprache. Die 1994 aus einer Sammlung von CGI-Scripts entstandene Sprache (***P****ersonal* ***H****ome* ***P****age*) entwickelte sich PHP rasch zu einer eigenständigen und sehr leistungsfähigen Programmiersprache mit starkem Fokus auf Web-Entwicklung. Seit PHP 3 wird die Basisfunkionalität der objektorientierten Programmierung unterstützt, die allerdings erst nach einer kompletten Umstrukturierung in Version 5 sinnvoll einsetzbar ist. PHP verwendet *dynamische Typisierung*. Der Typ einer Variablen ist daher vom zugewiesenen Wert abhängig und muss nicht zum Zeit der Kompilierung in Bytecode feststehen. Die eingesetzte, *schwache Typisierung* bedeutet allerdings auch eine potentielle Fehlerquelle, denn die Laufzeitumgebung versucht auch bei Operationen mit unterschiedlichen Datentypen implizite Typkonvertierungen, wodurch unerwünschte Fehlbesetzungen von Variablen oft verschleiert werden.

Obwohl die weite Verbreitung und die Tatsache, dass die aktuell produktiv eingesetzte Version der Publikationsdatenbank in dieser Sprache programmiert

wurde, natürlich für die Verwendung von PHP sprachen, fiel die Wahl schließlich doch gegen eine Implementierung auf Basis von PHP. Hauptkritikpunkte waren die Vermischung von Programmcode und HTML, die inkonsistente Sprachsyntax und der unübersichtliche und schlecht lesbare Programmaufbau. Grund dafür ist, dass die Leistungsfähigkeit der Sprache über viele Jahre gewachsen ist und somit wichtige Sprachkonstrukte (vor allem die zum Thema Objektorientierung) erst nachträglich eingebaut wurden - teilweise auf Kosten der Lesbarkeit.

4.1.1.2 Java

Java wurde von Beginn an mit Objektorientierung als fixem Bestandteil der Programmiersprache entwickelt. Die Syntax der Sprache ist der von C++ ähnlich, allerdings rein auf objektorientierte Programmierung ausgelegt. Java zeichnet sich durch umfangreiche standardisierte Class-Libraries aus, die gemeinsam mit dem Java-Compiler und dem Runtime Environment die Java-Platform bilden. Dem Programmierer steht außerdem eine automatische Speicherverwaltung und Exception Handling und ausgereifte Entwicklungsumgebungen wie Eclipse oder NetBeans zur Verfügung.

Java-Programme müssen vor der Ausführung zumindest in Bytecode übersetzt werden, es existieren jedoch auch Native Code Compiler. Die Sprache verwendet *Static Typing* und *Strong Typing*. Das bedeutet, dass der Datentyp einer Variable bereits zur Kompilierung feststehen muss. Außerdem muss die Umwandlung zwischen Datentypen explizit erfolgen.

Java Applikationsumgebungen

Für die Programmiersprache Java existieren auch einige umfangreiche, leistungsfähige und hoch skalierbare Umgebungen für die Entwicklung von Web-Applikationen. Besonders hervorzuheben ist hier der freie Applikationsserver *JBoss*, der neben Modulen für Caching, Persistance und Datenbank-Zugriff (*JDBC*) auch Enterprise Features wie Clustering und Load-Balancing bietet. Über die freie Implementierung Apache Tomcat werden auch *JavaServer Pages* und *Java Servlets* unterstützt.

Aufgrund der durchgängig erzwungenen objektorientierten Struktur von Java ist der Spezifikations- und Deklarationsaufwand rund um die Erstellung eines neuen Programmmoduls beträchtlich. Auch die sehr strikte und explizite

Typisierung produziert viel Programmier-Overhead und erzwingt sehr umfangreiche Libraries, da entsprechende Methoden für viele ähnliche Datentypen zur Verfügung gestellt werden müssen. Daher und auch aufgrund der notwendigen Kompilierung des Quelltextes eignet sich die Sprache nicht für Rapid Prototyping.

4.1.1.3 Perl

Perl ist eine Programmiersprache, die ursprünglich für die Analyse und Verarbeitung von Textdateien (Konfigurationen oder Log-Files) entwickelt wurde, weshalb auch die besonders stark ausgeprägte Unterstützung von Regular Expressions von Anfang an fix in die Sprache integriert war. Erst mit Version 5 wurde objektorientierte Programmierung unterstützt, ein Umstand, der sich ähnlich wie bei PHP auch in Inkonsistenzen der Sprachsyntax bemerkbar macht. Perl verwendet *dynamische, schwache Typisierung* und leidet daher unter dem bereits bei PHP beschriebenen Problem möglicher unerwünschter automatischer Typkonvertierungen.

Web-Application-Frameworks, die auf Perl basieren verwenden meist entweder einen CGI-basierten Ansatz oder funktionieren in Verbindung mit dem Apache-Webservermodul *mod_perl*. Der CGI-Ansatz bedeutet erheblichen Overhead durch das für jeden Web-Request notwendige Starten eines neuen Prozesses, die mod_perl-Schnittstelle ist dagegen (vor allem was Version 2 betrifft) schlecht und inaktuell dokumentiert. Eine dritte mögliche Integrationslösung über *FastCGI* hat, obwohl sie seit mehr als zehn Jahren verfügbar ist, nie nennenswerte Verbreitung gefunden.

Die schlecht lesbare, inkonstante Syntax[1], viele implizite „magische“ Typkonvertierungen, das stark kontextsensitive Verhalten vieler Perl-Befehle und die unzureichend dokumentierte oder inperformante Integration in den Webserver waren entscheidende Gründe warum die Wahl nicht auf eine Perl-basierte Entwicklung fiel. Der Werbespruch „There's more than one way to do it.“ ist nach Ansicht des Autors zwar für schnelle Hacks im Bereich der Systemadministration sehr nützlich, trägt aber leider meist nicht zur besseren Verständlichkeit des Programmcodes bei.

[1] Jamie Zawinsky, der berühmte Co-Autor des Web-Browsers und Mail-Programms Mosaic Netscape antwortete 1997 auf die Frage „What's wrong with Perl?“: „It combines all the worst aspects of C and Lisp: a billion different sublanguages in one monolithic executable. It combines the power of C with the readability of PostScript.“

4.1.1.4 Python

Die objektorientierte Programmiersprache *Python* besticht vor allem durch die klare und übersichtliche Syntax und den, im Vergleich zu anderen Scriptingsprachen, schlanken Sprachkern. Python verfügt über dynamische aber starke Typisierung, der Typ einer Variable wird also erst zur Laufzeit festgelegt, was den Deklarationsaufwand erspart. Datentypen werden aber nicht implizit untereinander umgewandelt, Typfehler werden also zumindest zur Laufzeit erkannt. Bei der Behandlung von Objekten verwendet Python *Duck Typing*[2]. Das bedeutet, dass für den Interpreter nicht die Abstammung eines Objektes von einer bestimmten Basisklasse sondern das Vorhandensein der verwendeten Methoden und Attribute relevant ist [MRA05].

Die positiven Eigenschaften der Sprache werden am besten durch das „Zen of Python“ [Pet] beschrieben, die teilweise in starken Kontrast zu anderen, hier vorgestellten Ansätzen stehen:

> Beautiful is better than ugly.
> Explicit is better than implicit.
> Readability counts.
> ⋮
> There should be one — and preferably only one — obvious way to do it.

Bis vor kurzem war die Verbreitung von Python im Bereich der Web-Anwendungsentwicklung hauptsächlich auf den Einsatz rund um den Applikationsserver *Zope* beschränkt, in den letzten Jahren wurden aber auch einige andere leistungsfähige Web-Anwendungsumgebungen geschaffen.

Zope

Das *Z Object Publishing Environment* ist ein sehr leistungsfähiger Applikationsserver, der sich vor allem durch den integrierten, in Python implementierten Webserver und die eingebaute, transaktionsfähige Objektdatenbank *ZODB* auszeichnet. Auch die umfangreichen Funktionsbibliotheken und die saubere Trennung von Layout und Programmlogik über mehrere zur Verfügung stehende Templating-Mechanismen sind hervorzuheben. Dem steht allerdings eine

[2] Benannt nach einem Gedicht von James Whitcomb Riley, aus dem folgendes Zitat stammt: „When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.“

relativ flache Lernkurve gegenüber: Viel Einarbeitungsaufwand ist notwendig, um Zopes Interna für sinnvolle Projekte nutzen zu können.

Django

Django ist ein sehr junges Web-Application Framework, dessen Designgrundsatz *DRY* (***D**on't **r**epeat **y**ourself*), also die Zielsetzung einer maximalen Wiederverwendbarkeit der Komponenten, vermutlich eine gute Basis für das vorliegende Projekt gebildet hätte. Auch das eingebaute Dispatcher-System, das eine ereignisbasierte Kommunikation zwischen Komponenten ermöglicht, der integrierte *Object Relational Mapper* (*ORM*), der einen abstrakten Zugriff auf relationale Datenbanken ermöglicht, und die erweiterbare Templating-Engine wären interessante Features gewesen.

Leider wurde die erste frei verfügbare Version erst im Laufe der Implementierungsphase dieses Projektes veröffentlicht, wodurch eine Entwicklung auf der Basis von Django nicht mehr in Betracht gezogen werden konnte.

4.1.1.5 Ruby

Die vollständig objektorientierte Sprache Ruby wurde erst vor wenigen Jahren außerhalb ihres Entstehungslandes Japan bekannt. Ruby wird als äußerst komplexe Sprache, die beispielsweise ähnlich wie Perl eine sehr große Anzahl an Funktionen in die Sprache direkt integriert. Auch die variantenreiche Syntax ist Perl-ähnlich, während die modernen objektorientierten Sprachkonzepte eher an Python erinnern. So verwendet Ruby beispielsweise dynamisches Duck-Typing, was für die Code-Reusability sehr förderlich ist.

Ruby on Rails

Im Umfeld der Web-Programmierung ist Ruby vor allem durch das Application-Framework Ruby on Rails stark vertreten. Durch die Designgrundsätze *„Convention over Configuration“* und *„Don't repeat Yourself“* (*DRY*) eignet sich Ruby on Rails gut für Rapid Application Development und begünstigt die modulare Softwareentwicklung.

Die fehlenden Sprachkenntnisse des Autors und der Umstand, dass es sich bei Ruby um eine relativ junge Sprache handelt, deren Entwickler- und Nutzer-Community (zumindest bis vor kurzem) hauptsächlich auf den Raum Japan

konzentriert war, waren schließlich die ausschlaggebenden Argumente gegen den Einsatz von Ruby, obwohl die Konzepte der Sprache sehr vielversprechend klingen.

4.1.2 Auswahl

Nach Abwägung oben angeführter Vor- und Nachteile der einzelnen Programmiersprachen und Web-Application-Frameworks, fiel die Wahl auf eine Lösung auf Basis von Zope beziehungsweise Python.

Python empfahl sich vor allem wegen des einfachen, logischen Sprachaufbaus, der dennoch, vor allem durch die ausgereiften dynamischen Fähigkeiten der Sprache, auch komplexe Problemlösungen ermöglicht.

Der Hauptnachteil der hohen Einarbeitungszeit in diesen komplexen Anwendungsserver wird bei der Entwicklung des vorliegenden Projektes durch die vorhandene Vorerfahrung des Autors mit dieser Umgebung abgeschwächt. Für potentielle Entwickler, die das Ergebnis dieser Arbeit nutzen sollen, wird diese Hürde durch die Kapselung Zope-spezifischer Eigenheiten im Application-Framework vermindert. Dennoch steht im Bedarfsfall aber die volle Funktionalität und Leistungsfähigkeit von Zope zur Verfügung. Durch die klare Objektstruktur von Zope können einzelne Bibliotheken problemlos aus dem entwickelten Framework heraus dem Anwender zur Verfügung gestellt werden.

Ein weiteres wichtiges Argument für den Einsatz von Zope war das bereits vorhandene Know-How in der ADV-Abteilung der Technischen Universität Wien, da Zope die Basis für das webbasierte Informationssystem *Tuwis++* der TU-Wien darstellt. Die ADV-Abteilung des Zentralen Informatikdienstes war zum Zeitpunkt der Projektvergabe ein möglicher Kandidat für die Übernahme und Weiterentwicklung der Publikationsdatenbank.

4.2 ZOPE

Die Umsetzung des Projektes erfolgte also auf Basis des Applikationsservers *Zope* (*Z Object Publishing Environment*). Die Wahl fiel unter anderem auf dieses Produkt, da Zope viele in den vorigen Kapiteln vorgestellten Konzepte zur systematischen Entwicklung von modularen, erweiterbaren, wartbaren und portablen Web-Applikationen unterstützt oder zumindest fördert.

Einige positive Eigenschaften von Zope sind:

- Wie der Name schon sagt, handelt es sich um eine objektorientierte Umgebung. Zope ist nicht nur selbst objektorientiert programmiert, auch die damit entwickelten Komponenten und Anwendungen sind in einer Objekthierarchie organisiert.
- Es handelt sich um ein aktives Open-Source Projekt, das ständig erweitert und verbessert wird. Eine große Community sorgt für Feedback bei Fehlern und Unterstützung bei Problemen während der Anwendungsentwicklung.
- Durch seinen modularen Aufbau ist Zope leicht erweiterbar. Zusätzliche Funktionalität kann in vielen Fällen auch aus einem großen Pool bereits vorhandener Erweiterungen (in der Zope Diktion: *Produkte*) ergänzt werden.
- Mit Zope entwickelte Anwendungen sind größtenteils plattformunabhängig. (Zope selbst läuft sowohl unter Windows als auch auf den meisten Unix-Varianten.)
- Zope unterstützt die Trennung von Geschäftslogik und Präsentation durch Template-Sprachen wie *Zope Page Templates*.
- Der Applikationsserver unterstützt mehrere Internet-Protokolle, wie HTTP, XML-RPC, FTP oder WebDAV. Der Zugriff auf angebotene Objekte wird protokollunabhängig ermöglicht. Dadurch können Teile der Geschäftslogik beispielsweise gleichzeitig auch als Remote-Procedures über XML-RPC angeboten werden. Dies fördert die Wiederverwendbarkeit des Programmcodes.
- Applikationen mit Datenbankzugriffen profitieren von einer in Zope integrierten Transaktionssteuerung. Tritt während der Behandlung eines Requests ein Fehler auf, werden sämtliche offenen Datenbanktransaktionen rückgängig gemacht.
- Durch ein flexibles Lizenzmodell unterstützt Zope sowohl die kommerzielle als auch die freie Nutzung der Software. Die Zope Lizenz ist auch kompatibel mit vielen anderen Open-Source Lizenzen.

4.2.1 Übersicht

An der Behandlung eines Requests sind in einer Zope-Anwendung unter anderem mehrere zentrale Komponenten beteiligt:

- ZServer
- ZPublisher
- Transaction Manager
- ZODB

Zope behandelt Web-Requests rein objektorientiert. Der angefragte URL repräsentiert ein Objekt in einer hierarchischen Objektdatenbank, der *ZODB*.

Der typische Ablauf eines Requests sieht bei einer Zope-Anwendung folgendermaßen aus:

Über eines der unterstützten Protokolle (HTTP, XML-RPC, WebDAV, FTP) erreicht die Anfrage den *ZServer*. Hier wird der Request in eine protokollunabhängige Form zur weiteren Bearbeitung umgewandelt.

Auf der Basis des angefragten URLs lokalisiert ein *Object Request Broker*, der *ZPublisher*, ein Objekt, das für die Behandlung zuständig ist:

Objekte sind in Zope in einer hierarchischen Objektdatenbank, der *ZODB* organisiert. Objekte können andere Objekte enthalten (z.B. *Folder*-Objekte). Der Request-URL entspricht in ZOPE einem Pfad zu einem Objekt in dieser Hierarchie, die Aufgabe des ZPublishers ist es, das jeweils zuständige Objekt nach bestimmten Regeln (siehe *Aquisition* auf Seite 43) zu finden, dessen Berechtigungen zu überprüfen und und schließlich gegebenenfalls eine Methode des Objekts aufzurufen.

An der Abarbeitung der Anfrage sind im Allgemeinen noch weitere Objekte der ZODB beteiligt, beispielsweise Code-Objekte, die Funktionen der Geschäftslogik beinhalten (z.B. *Python-Scripts*), Datenbank-Konnektoren oder *Z-SQL*-Methoden, die die Verbindung zu relationalen Datenbanken herstellen oder *Page-Templates* zur Präsentation der Daten.

Wird die Abarbeitung eines Requests erfolgreich abgeschlossen, so ist es die Aufgabe des TransactionManagers, am Ende des Publishing-Vorganges alle transaktionsorientierten Operationen durchzuführen (Commit). Tritt hingegen während der Bearbeitung ein Fehler auf, werden alle Transaktionen rückgängig gemacht (Rollback).

4.2.2 ZPublisher - die Abarbeitung von Requests

Der ZPublisher begleitet einen Request über die gesamte Abarbeitungsphase hinweg, bis die endgültige Antwort vorliegt.

Aufgaben des ZPublishers sind unter anderem [LPMS]:

4.2.2.1 Bereitstellen der zentralen Objekte `REQUEST` und `RESPONSE`

Das `REQUEST`-Objekt kapselt alle Informationen über die aktuelle Anfrage. Das Objekt kann direkt als assoziatives Array[3] verwendet werden, für exakteren Zugriff auf die gespeicherten Informationen stehen allerdings Kategorien in Form von Attributen zur Verfügung[4]:

- `.environ`: Environment-Variablen nach der *CGI*-Spezifikation
- `.form`: Formulardaten (aus dem Query-String beziehungsweise aus dem Request-Body bei POST-Requests)
- `.cookies`: HTTP-Cookies
- `.other`: weitere Informationen, darunter:
  - `PUBLISHED`, `PARENTS`: Referenzen auf das angeforderte Objekt beziehungsweise auf dessen übergeordnete Objekte in der Objekthierarchie.
  - `REQUEST`, `RESPONSE`: Das Objekt selbst, beziehungsweise das zugehörige Antwort-Objekt
  - `URL`, `URL`*n*, `BASE`*n*, `BASEPATH`*n*: diverse Informationen über den angeforderten URL

Diese Attribute können wiederum als Mapping Objekte verwendet werden.

Andere Attribute des `REQUEST`-Objektes erlauben den Zugriff auf möglicherweise serverseitig gespeicherte Session-Daten (`SESSION`) oder geben Auskunft über den verbundenen Client `getClientAddr()` und über (HTTP-) Request-Header `get_header()`.

[3] genauer: als Python-Mapping-Objekt
[4] Einige dieser Informationen sind nur bei HTTP-Requests verfügbar.

Das `RESPONSE`-Objekt wird zur Formulierung der Antwort herangezogen. Wichtige Aufgaben dieses Objekts sind daher unter anderem[5]:

- Setzen des Status-Codes: `setStatus()`
- HTTP-Umleitungen: `redirect()`
- Manipulation von Response-Header-Zeilen: `addHeader()`, `setHeader()`
- Manipulation von Cookies: `setCookie()`, `addCookie()`, `appendCookie()`, `expireCookie()`
- Manipulation des Inhaltes der Antwort: `setBody()`
- Das Streamen von Daten: Über die `write()`-Methode ist es auch möglich, Daten „scheibchenweise“ an den Client zu liefern. Diese Methode eignet sich beispielsweise für den Transfer großer Datenmengen.

4.2.2.2 Dekodieren und Aufbereiten von Request-Argumenten (Marshalling)

Üblicherweise enthält ein Zope-Request Nutzdaten, die, je nach verwendetem Protokoll, auf unterschiedliche Art an den Server übermittelt werden können. Der Zope-Publisher versucht, diese Daten in eine einheitliche Form zu bringen, damit die aufgerufene Applikation möglichst nicht auf das aktuell verwendete Transportprotokoll und dessen Eigenheiten Rücksicht nehmen muss. Diese Aufbereitung wird als *Marshalling* bezeichnet [MPH].

Am Ende des Marshalling-Prozesses ist das `REQUEST`-Objekt, wie oben beschrieben, mit Daten befüllt. Zusätzlich stellt der Publisher die Daten aber auch noch in anderer Form zur Verfügung:

Bei PythonScript-Methoden wird die Parameterliste jenes Objektes, das schließlich vom Publisher ausgeführt wird, automatisch mit den Request-Argumenten befüllt. Erwartet ein Python-Script-Objekt also beispielsweise den Parameter `name`, so enthält dieser Parameter den Inhalt des gleichnamigen Formularfeldes, wenn das Objekt über das Web aufgerufen wurde, egal, ob die Daten mit GET oder POST übermittelt wurden.

Darüberhinaus unterstützt der Zope-Publisher auch Methoden zur automatischen Konvertierung der Argumente. Über Namenssuffixes können Argumente

[5] Einige der Methoden sind nur verfügbar, wenn HTTP als Transportprotokoll verwendet wird.

automatisch in spezielle Datentypen umgewandelt werden. Werden Formularfelder zum Beispiel mit `numbers:list:int` bezeichnet, wird der Parameter `numbers` in eine Liste von Integer-Werten konvertiert. Es ist auch möglich, über einen Suffix (`:method`) den Pfad zum aufzurufenden Objekt zu verändern oder Argumente in strukturierter Form zusammenzufassen (`:record`). Details zu den verfügbaren Konvertierungen finden sich in [MPH].

4.2.2.3 Traversal

Eine zentrale Aufgabe des Publishers ist das Lokalisieren des aufzurufenden Objektes anhand des URLs beziehungsweise des Objektpfades. Dazu wird der Pfad in Komponenten aufgeteilt. Anschließend wird, ausgehend vom Root-Objekt, die Objekthierarchie der ZODB durchschritten, bis die letzte Komponente des URLs erreicht ist oder ein Fehler auftritt. Dieser Vorgang wird als *Traversal* bezeichnet.

Die Tatsache, dass Objekte in der ZODB hierarchisch strukturiert sind, dass also Objekte im Allgemeinen andere Objekte enthalten, wird von Zope für eine spezielle Form der Vererbung genutzt, eine Technik namens *Acquisition* (beschrieben im nächsten Abschnitt). Zum Auffinden von Objekten wird nicht nur der „physische“ Objektpfad berücksichtigt, es kommt auch Acquisition zum Einsatz.

Über die Definition spezieller Hooks[6] ist es möglich, das Verhalten des Objekt-Lookups zu verändern.

Ob das Objekt, das am Ende des Objektpfades steht, schließlich direkt aufgerufen wird oder ob der Publisher ein „Ersatzobjekt“ präsentiert, wird durch folgende Regeln bestimmt[7]:

- Besitzt das Objekt eine Methode namens `__browser_default__`, so wird diese aufgerufen, um ein „Ersatzobjekt“ und einen Ersatzpfad zu diesem Objekt zu bestimmen.
- Existiert eine Methode namens `__index_html__` und handelt es sich um einen HTTP-Request der Methoden GET oder POST, wird diese Methode verwendet. (Dieses Verhalten ist vergleichbar mit der Default-Seite,

[6] Über die Methode `__before_publishing_traverse__()`.

[7] Die Reihenfolge dieser Regeln wurde gegenüber [MPH] abgeändert und entspricht der tatsächlich implementierten Logik

die bei den meisten Webservern bei Anforderung eines Verzeichnisses geliefert wird.)

- In allen anderen Fällen wird das eigentliche Objekt verwendet.

4.2.2.4 Veranlassen der Authentisierung

Um den Zugriff auf auf geschützte Objekte regeln und überwachen zu können, muss Zope den Benutzer authentisieren. Vor dem Publizieren eines Objektes wird diese Aufgabe an einen sogenannten *User Folder* delegiert[8]. Seine Aufgabe ist sowohl die Authentisierung des Benutzers, als auch seine `Autorisierung`.

Objektseitig wird die Autorisierung durch Berechtigungen (*Permissions*) festgelegt, der Zugriff auf Attribute eines Objektes wird durch die ihnen jeweils zugeordneten Permissions bestimmt.

Die Verknüpfung zwischen Benutzern und Berechtigungen erfolgt nicht direkt, sondern über eine zusätzliche Abstraktionsschicht, die *Rollen*. Einem Benutzer sind eine oder mehrere Rollen zugeordnet, wobei jede Rolle eine Menge von Permissions repräsentiert. Rollen können einem Benutzer global, im User Folder, zugewiesen werden, sie können aber auch lokal, als Objektattribute, nur für einen Teil der Objekthierarchie, definiert werden [LPMS].

User Folder sind als Objekte in der ZODB abgelegt. Eine oft genutzte Variante für die flexible Behandlung verschiedenster Authentisierungsarten ist das *PluggableAuthService*, ein User Folder, mit dem verschiedenste Benutzerdatenbanken (z.B. LDAP, SQL, Kerberos, Radius, etc) kombiniert werden können.

Nicht authentisierte Benutzer werden auf den User `Anonymous` abgebildet, dem standardmäßig die gleichnamige Rolle `Anonymous` zugeordnet ist. Der aktuell authentisierte Benutzer ist über das Request-Attribut `AUTHENTICATED_USER` ermittelbar.

4.2.2.5 Aufruf des gewünschten Objektes

Ist die Lokalisierung erfolgreich abgeschlossen und wurden alle Sicherheitsvoraussetzungen erfüllt, muss das angeforderte Objekt schließlich vom Publisher aufgerufen werden.

[8] Über dessen Methode `validate()`

Handelt es sich bei dem aus den Regeln des Traversal resultierenden Ziel um ein aufrufbares Objekt, so wird es mit den benötigten Argumenten aus dem Request versehen und ausgeführt. Das Ergebnis wird schließlich bei der Formulierung der Antwort in einen String umgewandelt.

Wurde ein Objekt gefunden, das nicht ausführbar ist, wird das Objekt selbst in einen String gewandelt und als Antwort zurückgeliefert. (Dies ist zum Beispiel bei allen Objekten der Fall, die statische Seiten repräsentieren.)

4.2.2.6 Exception-Handling, Transaktionen

Wie bereits erwähnt, wird die Abarbeitung eines Requests durch eine Transaktionsumgebung geschützt. Tritt also während der Ausführung ein Fehler auf (was sich durch Auftreten einer Exception bemerkbar macht), sorgt der Publisher dafür, dass alle transaktionsfähigen Operationen rückgängig gemacht werden. War die Ausführung hingegen erfolgreich, erledigt der Publisher über ein Commit des Transaction Managers die dauerhafte Durchführung der Aktionen.

Beispiele für transaktionsfähige Operationen sind Schreibzugriffe auf relationale Datenbanken oder auch Schreibzugriffe auf die ZODB, es ist aber auch möglich eigene Produkte (Erweiterungen) beim Transaktionsmanager zu registrieren, die Operationen nach dem ACID-Prinzip (***A****tomicity*, ***C****onsistency*, ***I****solation*, ***D****urability*) ausführen müssen oder bestimmte Aktionen nur durchführen sollen, wenn die Abarbeitung des Requests fehlerfrei abgeschlossen wurde. Das Beschreiben externer Dateien oder das Versenden von Druckaufträgen oder E-Mails wären etwa Anwendungsfälle.

Das Auftreten einer Exception namens `ConflictError` wird vom Publisher speziell behandelt. Diese Exception wird abgefangen und bewirkt, dass der aktuelle Request abgebrochen und neu gestartet wird. (Die aktuelle Transaktion wird dabei rückgängig gemacht.) Dieser Vorgang wird beispielsweise nach einem Zugriffskonflikt durch mehrere parallele Schreiboperationen auf die ZODB angestoßen. Andere Anwendungsfälle wären generelle Recovery-Aktionen, wie beispielsweise der Wiederaufbau einer getrennten Datenbankverbindung.

4.2.3 ZODB - eine objektorientierte Datenbank

Die ***Z**ope **O**bject **D**ata**B**ase* ist ein hierarchischer Speicherplatz für Objekte. Die meisten von Zope verwendeten Objekte sind in der ZODB dauerhaft abgelegt.

In einem typischen Zope-Server sind statische Inhalte gemeinsam mit dynamischen Page Templates, Scripts und Hilfsobjekten, die die Funktionalität von Zope erweitern, als Objekte in einer Folder-(Objekt-)struktur organisiert und können, zumindest bei Zope Version 2, über ein Management Interface (*ZMI*) graphisch verwaltet werden.

4.2.3.1 Persistenz

In der ZODB gespeicherte Objekte werden in einem konsistenten Zustand auf einem nicht-flüchtigen Speichermedium[9] gesichert und überleben daher mit ihren Attributen und Unterobjekten auch einen Neustart des Servers.

Eine Voraussetzung für diese dauerhafte Speicherung eines Objektes ist seine Serialisierbarkeit. Der Zustand des Objektes muss als Datenstrom darstellbar sein, eine Bedingung, die hauptsächlich bei offenen Ressourcen, die vom Betriebssystem zur Verfügung gestellt werden, nicht erfüllbar ist. Beispiele sind offene Filehandles, Netzwerkverbindungen oder ähnliches. Oft können aber in solchen Fällen über eine Hilfsroutine, die bei der Serialisierung aufgerufen wird, Informationen abgespeichert werden, die beim neuerlichen Instanzieren des Objektes ausreichen, um ein Äquivalent des ursprünglichen Zustandes wiederherstellen zu können.

Nähere Informationen zu diesem Thema finden sich in [Lut06] und [MRA05] unter dem Stichwort `pickle`, dem Modul, das für die Serialisierungsaufgaben der ZODB verantwortlich ist.

4.2.3.2 Transparenz

Um Änderungen an einem persistenten Objekt in der ZODB zu speichern, ist kein expliziter Befehl notwendig. Ein abschließendes `commit()` der aktuellen Transaktion sichert automatisch alle Objekte, die seit Beginn der Transaktion

[9] mit Ausnahme des TempStorage Datenbankmoduls, welches die Daten im RAM speichert

verändert wurden. Ändert sich ein Attribut, so wird das von der Persistence-Maschinerie automatisch erkannt, lediglich die Veränderung eines Elements von *Mutable-Python-Objekts* (zum Beispiel Python-Listen oder Dictionaries) können nicht automatisch erkannt werden. Es existieren aber äquivalente Zope-Klassen, die dieses Problem nicht haben, beziehungsweise hilft als Workaround auch eine Neuzuweisung des Objektes oder das Setzen des speziellen Attributs `_p_changed`. Attribute, die mit „`_v_`“ beginnen sind für temporäre Verwendung bestimmt und werden nicht persistent gespeichert[MPH, Fulb].

Objekte der ZODB werden für einen effizienteren Zugriff auch transparent gecached, müssen also nicht zu Beginn jeder Transaktion neu aus der serialisierten Repräsentation erstellt werden.

4.2.3.3 Versionierung und Undo-Fähigkeit

Dadurch, dass die meisten ZODB-Datenbank-Backends bei der Serialisierung alte Versionen eines Objektes nicht überschreiben und jedes Objekt mit einem Versionszähler versehen, kann in diesen Implementierungen auch auf ältere Versionen eines Objektes zurückgegriffen werden. Auf diese Art wird ein *Undo*-Mechanismus für Objekte implementiert.

4.2.3.4 Clusterfähigkeit (ZEO)

Die Speicherung der Zope-Objekte kann auch auf einen entfernten Server ausgelagert werden. Mit Hilfe der ***Z**ope **E**nterprise **O**bjects* (*ZEO*) kann der Speicherort der Objekte (ZEO-Server) von den Ausführungsorten der Objekte (ZEO-Clients) getrennt werden. Auf diese Art kann Clustering erreicht werden. Objekte werden zentral gespeichert, die Programmlogik wird aber zur Lastverteilung auf mehreren Rechnern ausgeführt.

Da der ZEO-Server, zum Beispiel zu Debugging-Zwecken, auch unabhängig von Zope aus Python-Programmen heraus verwendet werden kann, wird die Verwendung von ZEO als ZODB-Backend generell für die Zope-Softwareentwicklung empfohlen, auch wenn Clustering nicht benötigt wird[LPMS].

4.2.4 Acquisition

Da die in der ZODB abgelegten Objekte in einer Baumstruktur gespeichert sind, können diese hierarchischen Beziehungen zwischen Objekten auch für eine Variante der Vererbung verwendet werden, die in ZOPE als *Acquisition* bezeichnet wird.

Bei aktivierter Acquisition erbt ein Objekt die Eigenschaften seiner Elternobjekte in der Baumstruktur. Wird also ein Attribut nicht direkt am Objekt selbst gefunden, so wird auch in der Hierarchie nach oben, an den Elternobjekten gesucht.

Ein Objekt kann also mittels Acquisition über verschiedene Zugriffspfade gefunden werden. Die Lokalisierung erfolgt, indem der Suchpfad elementweise von links nach rechts behandelt wird. Dabei wird der Pfad, über den der Zugriff erfolgt, *Context* genannt, der Pfad, in dem das Objekt tatsächlich enthalten ist, wird als *Container* bezeichnet. Ist ein Attribut sowohl im Context, als auch im Container vorhanden, wird der Container bevorzugt. Eine mathematische Beschreibung dieses komplexen Verfahrens findet sich in einer Präsentation seines Erfinders unter [Fula].

Acquisition kann entweder implizit oder explizit aktiviert werden. Ersteres bedeutet, dass bei Zugriff auf Attribute eines Objektes automatisch und transparent die Regeln der Acquisition zur Anwendung kommen, bei zweiterem wird die hierarchische Suche nur bei Verwendung der speziellen Suchmethode `ac_acquire()` veranlasst.

Acquisition wird von Zope an vielen Stellen verwendet. Der Publisher lokalisiert Objekte basierend auf dem Request-URL beispielsweise mit Hilfe impliziter Acquisition. Auch Zugriffsberechtigungen auf Zope-Objekte werden standardmäßig über diese Technik „vererbt“.

4.2.5 Page Templates - Präsentation dynamischer Inhalte

Eine Forderung vieler Software-Architekturmuster wie Model-View-Controller (siehe 3.2.2) oder *Presentation-Abstraction-Control* ist die scharfe Trennung zwischen Logik und Präsentation der Daten. Um dieser Forderung gerecht werden zu können, bietet Zope eine eigene Template-Sprache namens *Zope Page Templates* an.

Page Templates sollen die Zusammenarbeit zwischen Designern und Programmierern erleichtern und erfüllen daher folgende Eigenschaften:

- Templates enthalten keine Geschäftslogik sondern lediglich einfache Logikanweisungen, die zur formatierten Ausgabe von Daten notwendig sind
- Page Templates können mit HTML-Editoren nach dem WYSIWYG-Prinzip angesehen und sogar editiert werden, ohne dass ihre dynamischen Steueranweisungen verloren gehen oder beschädigt werden. (Hier wird die Empfehlung des Standards ausgenutzt, dass unbekannte HTML-Attribute von Browsern ignoriert und auch von Editoren nicht verändert werden sollen [KM95].)
- Sowohl die gerenderte Seite, als auch die Vorlage sind valides HTML.

Über eine XHTML-basierte Syntax definieren Page Templates direkt das Aussehen einer Web-Seite. Dynamische Inhalte werden ausschließlich in Form spezieller XML-Attribute, die in „normalen“ XHTML-Elementen enthalten sind, eingefügt.

Die Syntax dieser Attribute ist durch die *Template Attribute Language (TAL)* definiert. Diese Sprache bietet einfache Befehle (repräsentiert durch Attribute), um die Vorlage dynamisch mit Inhalt zu befüllen:

- HTML-Elemente können ersetzt (`tal:replace`), befüllt (`tal:content`) oder gelöscht (`tal:omit-tag`) werden.
- Attribute können ersetzt oder gelöscht werden. (`tal:attributes`)
- Globale oder lokale Variablen können definiert oder verändert werden. (`tal:define`)
- Elemente und ihr Inhalt können bedingt weggelassen werden. (`tal:condition`)
- Einfache Schleifen (`tal:repeat`) können implementiert werden.

Obige TAL-Kommandos verwenden als Argumente Ausdrücke (*Expressions*), die durch die ***TAL E**xpression **S**yntax (TALES)* beschrieben werden und die TAL-Kommandos mit dynamischen Daten versorgen.

TALES bietet Möglichkeiten

- über *path-expressions* andere Objekte aufzurufen und deren Rückgabewert zu verwenden

- Objektpfade auf Existenz zu überprüfen (`exists:`*`path`*)
- Objektpfade zurückzuliefern, ohne sie auszuführen (`nocall:`*`path`*)
- Ausdrücke zu negieren (`not:`*`expression`*)
- Texte nach Variablensubstitution zurückzuliefern (`string:`*`text`*)
- Python-Ausdrücke auszuführen (`python:`*`expression`*)

Zur Förderung der Modularität können Seitenteile, die in mehreren Teilen einer Applikation (oder applikationsübergreifend) benötigt werden, an zentraler Stelle mit Hilfe der ***M****acro* ***E****xpansion* ***T****ag* ***A****ttribute* ***L****anguage (METAL)* beschrieben werden (`metal:define-macro`). Auf diese Definitionen kann dann mittels `metal:use-macro` zugegriffen werden. Da solche Textbausteine oft variable Elemente beinhalten, können Teile der Makros (*slots*) austauschbar gestaltet werden. Die werden in Makros mittels `metal:define-slot` definiert und bei deren Aufruf durch `metal:fill-slot` mit dem gewünschten Inhalt befüllt.

4.2.6 Erweiterung durch Produkte

Funktionalität, die in einer Basis-Installation vermisst wird, kann über sogenannte Produkte nachgerüstet werden. Produkte sind Add-On-Pakete, die den Funktionsumfang von Zope erweitern oder das bestehende Verhalten in gewisser Weise verändern.

Meist deckt ein Produkt eine logisch abgeschlossene Problematik ab. Ziel ist es, die Funktionalität anderen Entwicklern zur Verfügung zu stellen beziehungsweise Softwareteile in leicht wartbarer Form modular zu organisieren. Produkte werden in abgepackter Form der Community angeboten und im Zuge der Installation am Zope-Server im Dateisystem entpackt. So kann etwa bei einem Versionsupgrade des Zope-Servers das Produkt meist in unveränderter Form übernommen werden.

Kann auf die Integration in die Zope-Managementoberfläche verzichtet werden, besteht ein Produkt praktisch nur aus den Klassen, die anderen Zope-Produkten zur Verfügung gestellt werden sollen.

Oft bieten Produkte jedoch neue Objekttypen an, die über das Zope Management Interface (*ZMI*) instanziert und in der Objekthierarchie der ZODB

abgelegt werden können. Dafür werden dann zusätzlich zum eigentlichen Programmcode noch Funktionen benötigt, die die graphische Verwaltung der Objektinstanzen über das ZMI erlauben.

Die Zope Developers Guide [MPH] empfiehlt bereits für Zope 2.x, die Produktentwicklung in *Interface* und *Implementierung* zu trennen. Dabei beschreibt das Interface die Schnittstelle zur „Außenwelt“, also das *API*. Es definiert welche Methoden für die öffentliche Verwendung bestimmt sind, beschreibt deren Verhalten in Form einer Dokumentation und beinhaltet möglicherweise Testroutinen in Form von Unit-Tests, die eine korrekte Implementierung der Schnittstelle sicherstellen sollen. Dabei enthält das Interface selbst keinen Programmcode, es beschreibt lediglich die Funktionalität.

Die eigentliche programmtechnische Umsetzung erfolgt schließlich in der Implementierung. Zu jedem Interface existieren eine oder mehrere Implementierungen. Die Implementierung bestimmt, *wie* die vom Interface beschriebenen Aufgaben umgesetzt werden.

In Zope 3 sind Interfaces bereits zentraler Bestandteil der Komponenten-Infrastruktur [Wei06].

4.3 Das Web-Application-Framework „TaskEngine“

4.3.1 Konzepte

Klassische Web-Applikationen folgen mit ihrem Programmkonzept meist dem chronologischem Ablauf eines HTTP-Requests. Sie passen sich damit also dem zustandslosen Request-Response-Prinzip (vgl. Abschnitt 3.1.1.1) des Transportprotokolls an:

- Ein Request wird analysiert, die Argumente und übermittelten Daten werden aufbereitet.
- Eine der Anfrage entsprechende Aktion wird lokalisiert und Zugriffsberechtigungen werden überprüft.
- Die Aktion wird auf Basis der bereitgestellten Daten ausgeführt und eine Antwort wird vorbereitet.

- Die Antwort wird an den User-Agent übermittelt.

Modularität wird zumeist durch Unterteilung der Anwendung in Subroutinen und Funktionen erreicht, man spricht von prozeduraler Programmierung. Diese Call-and-Return Architektur ([DGH03]) hat allerdings zum Beispiel aufgrund der vom Programmierer fix vorgegebenen Reihenfolge des Programmflusses gerade bei interaktiven Anwendungen einige Schwächen.

In dieser Arbeit wurde daher versucht, eine Technik, die von der Programmierung klassischer graphischer Benutzeroberflächen bekannt ist, auf Web-Anwendungen umzusetzen: die ereignisorientierte Programmierung.

Da sich die ereignisorientierte Programmierung sehr gut mit den Konzepten der Objektorientierung verbinden lässt, wurde danach gestrebt, diese Prinzipien sowohl bei der Modellierung der Kontrollelemente, also auch bei der Ereignissteuerung und, soweit möglich, auch bei der Applikationslogik konsequent umzusetzen.

4.3.1.1 Ereignisorientierte Programmierung

Im Gegensatz zu einer linearen Programmstruktur, die einem fix vorgegebenen Programmablauf folgt, bietet eine ereignisgesteuerte Architektur die Möglichkeit, asynchron auf sowohl auf äußere Einflüsse als auch auf Veränderungen des inneren Zustandes zu reagieren.

Auslöser für Programmaktionen sind Ereignisse (*Events*). Ereignisse können von außen in das System einfließen, sie verbinden aber auch lose gekoppelte Programmteile.

Beispiele für Ereignisse in interaktiver Software sind:

- Benutzeraktionen
  - Änderung des Textes in Formularfeld
  - Auswahl eines Menüpunktes
  - Drücken eines Buttons
- Systemereignisse
  - Komponente (z.B. Formularelement) wird neu erstellt und initialisiert

  - Komponente soll neu gezeichnet werden
  - Überprüfung auf Gültigkeit der Eingabewerte
  - Komponente wird entfernt
  - Ablauf eines gesetzten Timers
- von Komponenten ausgelöste Ereignisse
  - selbst definierte Benachrichtigungen
  - Verkettung von Event-Handlern (Folgeereignisse)

Auftretende Events werden meist an zentraler Stelle von einem *Dispatcher* verarbeitet und analysiert. Bei ihm registrieren sich Komponenten, die sich für ein Ereignis (oder für eine Kategorie von Ereignissen) interessieren, mit einer Ereignisbehandlungsroutine, dem *Event-Handler*. Die Aufgabe des Dispatchers ist es, beim Eintreffen von Ereignissen, die entsprechenden Komponenten zu benachrichtigen, indem die hinterlegten Event-Handler aufgerufen werden.

In vielen Fällen ist es notwendig (beziehungsweise sinnvoll), Ereignisse sequentiell, in der Reihe des Eintreffens abzuarbeiten. Beim Auftreten neuer Ereignisse sollen also gerade ausgeführte Event-Handler nicht unterbrochen werden. Dieses Verhalten kann durch Implementierung einer *Event-Queue* erreicht werden. Neue Events werden hinten in die Queue eingereiht und vom Dispatcher der Reihe nach von vorne entnommen und verarbeitet.

Bei der Entwicklung interaktiver Software sind die ereignisorientierten Konzepte hauptsächlich aus der Programmierung graphischer User-Interfaces bekannt. Die meisten etablierten GUI-Bibliotheken basieren auf diesem Programmierparadigma.

Eine wichtige Konsequenz dieses Ansatzes ist, dass dabei das Framework den grundlegenden Ablauf vorgibt und die Applikation sich nur für bestimmte Ereignisse registriert. Die Anwendung reagiert also nur auf Benachrichtigungen. Das Konzept wird daher in der Literatur oft auch als *Inversion of Control* bzw. *Hollywood-Prinzip* („Don't call us, we call you!“) bezeichnet [Fow05].

Bei der Entwicklung webbasierter Anwendungen ist die Umsetzung dieser Grundsätze derzeit noch nicht weit verbreitet. Ein Grund dafür ist wahrscheinlich die komplexe Umsetzung des zustandslosen Request-Response Prinzips der HTTP-Infrastruktur in ein zustandsbehaftetes, session-orientiertes Komponenten-Konzept.

Bekannte Web-Application-Frameworks, die ereignisorientierte Programmierung ermöglichen sind Asp.net oder Ruby on Rails.

4.3.1.2 Standardisierte Kommunikation: Messages

Damit der Dispatcher allgemeine Ereignisse verarbeiten und zwischenspeichern kann, ohne den eigentlichen Inhalt oder die Bedeutung des Ereignisses kennen zu müssen, ist es notwendig, dass die Ereignisse in einer normierten Form verwaltet werden können.

Da einerseits der Dispatcher die Aufgabe hat, Komponenten über auftretende Ereignisse zu informieren und andererseits auch Komponenten die Möglichkeit haben sollten, untereinander zu kommunizieren, wird eine Infrastruktur für den Austausch von Informationen benötigt. Wie in der Interprozesskommunikation üblich wurde auch in diesem Projekt das Konzept von Nachrichten (*Messages*) gewählt um die zu transferierenden Daten zu kapseln.

Messages sind Kommunikationsobjekte, die folgende Eigenschaften aufweisen:

- Sie beinhalten definierte Adressierungsinformationen (Absender- und Empfängeradresse).
- Sie enthalten die Nutzdaten der Kommunikation in gekapselter Form.

Darüberhinaus enthalten die Komponenten, die Messages versenden und empfangen, Methoden die eine einheitliche Behandlung dieser Aufgaben ermöglichen.

Ein Vorteil dieser Kommunikationsform ist die lose Kopplung, die Messages zwischen Sender und Empfänger der Informationen bewirken. Das ermöglicht den Einsatz mehrerer Kommunikationsformen:

asynchrone Kommunikation Der Sender (*Producer*) der Nachricht muss nicht auf die Abarbeitung im Empfänger warten. Nachrichten können beispielsweise vom Dispatcher zwischengespeichert und zeitversetzt an den oder die *Consumer* zugestellt werden. Es wird keine unmittelbare Antwort an den Absender der Message zurückgeschickt. Dadurch wird auch eine Zustellung an mehrere Empfänger ermöglicht. Auch eine „unzuverlässige“ Zustellung von Nachrichten vergleichbar mit *Datagrams* in der Netzwerktechnik wäre denkbar.

Nach Qusay [Qus04] wird bei asynchroner Kommunikation zwischen verschiedenen Methoden der Zustellung unterschieden:

- Bei der Nachrichtenzustellung nach dem *Point-to-Point* Modell werden Nachrichten vom Producer über eine Queue an einen Consumer zugestellt.
- Beim *Publish/Subscribe* Modell werden die Nachrichten nicht direkt an einen Consumer adressiert, sondern sind an bestimmte „Themenbereiche“ (*Channels*) innerhalb der Messaging-Engine gerichtet (*publish*). Ein Consumer, der sich für einen bestimmten Channel interessiert, kann sich dafür registrieren (*subscribe*) und erhält damit alle Nachrichten zu diesem Thema. Nachdem üblicherweise auch mehrere Subscriber pro Channel erlaubt sind, sind auch „one to many“ und „many to many“ Beziehungen möglich.

synchrone Kommunikation Wie bei klassischen RPC-Aufrufen wartet der Sender der Nachricht auf eine Antwort vom Empfänger.

Die Komponentenarchitektur dieses Projekts bildet Messages standardmäßig auf Methoden der Komponenten ab, eine eintreffende Nachricht bewirkt also den Aufruf der entsprechenden Methode dieser Komponente.

4.3.1.3 Notifikationen

Obwohl das Messaging im vorliegenden Projekt nach dem Point-to-Point Prinzip gestaltet wurde, besteht für Komponenten, wie auf Seite 48 beschrieben, dennoch die Möglichkeit, über das Eintreten bestimmter Ereignisse informiert zu werden.

Beispielsweise folgt der Request/Response Ablauf einer Web-Anwendung einem immer wiederkehrenden Schema, in dem bestimmte Stellen für bestimmte Komponenten interessant sein können. So werden etwa Komponenten benachrichtigt, wenn POST-Daten eines abgeschickten Formulars für sie vorhanden sind. Ähnlich wird auch der Windowmanager nach Abarbeitung aller Requestdaten von der Ablaufsteuerung benachrichtigt, dass nun die Fenster zu zeichnen wären. Komponenten, die sich für solche Benachrichtigungen interessieren, implementieren spezielle Methoden, sogenannte *Hooks*, die aufgerufen werden, wenn das entsprechende Ereignis eintritt.

Zusätzlich zu diesen fix vorgesehenen Benachrichtigungen, haben Komponenten auch noch die Möglichkeit, sich für benutzerdefinierte Ereignisse zu registrieren. Die Ereignisse und die resultierenden Messages werden von der Klasse `TaskEngine` zentral verwaltet und gesteuert.

4.3.1.4 Kapselung der Applikationsdaten

Um die Zustandslosigkeit des Hypertext-Transfer Protokolls zu umgehen, nutzen viele Web-Applikationen serverseitige Datenspeicher, in denen sessionspezifische Anwendungsdaten abgelegt werden können um bei folgenden Requests derselben Session wieder zur Verfügung zu stehen.

Bei komplexeren Anwendungen, die aus vielen Modulen und Komponenten bestehen, entsteht hier allerdings das Problem, dass die Verwaltung (die Organisation, Gültigkeitsüberprüfung und gegebenenfalls eine Invalidierung) der Sessiondaten, in der Verantwortung des Entwicklers liegt.

Das komponentenorientierte Design des TaskEngine-Projektes, unterstützt hingegen eine strukturierte Speicherung der Applikationsdaten. Via HTTP übermittelte Daten werden automatisch der zugehörigen Objekt zugeteilt. Darüberhinaus besitzt jede Komponente (genauer gesagt jede Instanz einer Komponente) ihren eigenen Datencontainer um Daten über mehrere Requests hinweg abzulegen. Die Lebensdauer dieses Containers ist automatisch auf die der Komponenteninstanz beschränkt, die Notwendigkeit einer manuellen Invalidierung kann daher in den meisten Fällen entfallen.

Werden also beispielsweise Formularelemente als Komponenten realisiert, dann werden auch die zugehörigen Formulardaten automatisch in den entsprechenden Objekten gespeichert. Auch bei mehreren Ausprägungen desselben Formularelementes stellt das kein Problem dar, da die Datencontainer ja instanzspezifisch gespeichert sind.

4.3.1.5 Persistence

Ein großer Unterschied zwischen Web-Applikationen und GUI-Anwendungen besteht darin, dass der Anwendungsprozess und damit der aktuelle Ausführungskontext des Programmes zwischen zwei Requests im ersteren Fall nicht erhalten bleibt. Bei einer klassischen Web-Anwendung ist es beispielsweise nicht

möglich, wie z.B. in der GUI-Programmierung üblich, Closures oder Objekte mit der Verwaltung von Kontrollelementen zu beauftragen oder gar eigene Threads zu starten, um asynchrone Aufgaben durchzuführen.

Das TaskEngine-Framework versucht, diese Einschränkung mit Hilfe sogenannter Tasks und Components zu umgehen, die prozess- beziehungsweise threadähnliche Eigenschaften besitzen. Zwischen zwei Requests bleiben nicht nur, wie im vorigen Abschnitt beschrieben, die Applikationsdaten erhalten, auch die Component- und Task-Objekte selbst werden serialisiert und bei der nächsten Browseranfrage gemeinsam mit Ihrem gesamten Kontext wieder „aufgetaut“. Die Notwendigkeit, den Programmzustand händisch zu speichern und ihn zum Beispiel gemeinsam mit den Anwendungsdaten in Session-Daten unterzubringen, entfällt damit großteils.

4.3.1.6 Multiple Inheritance, Mixins

Nach dem Vorbild der Architektur von Zope 2 wurden auch bei der Erstellung des vorliegenden Web-Frameworks die Prinzipien der Mehrfachvererbung (*Multiple Inheritance*), im speziellen die der sogenannten *Mixin-Classes*, intensiv eingesetzt.

Mixins sind eine Anwendung der Mehrfachvererbung. Mixin Classes werden verwendet, um Klassen modular um optionale Funktionalität zu erweitern. Dabei werden Mixin-Classes nie direkt instanziert, die sind nur dazu bestimmt, dass Subklassen (unter anderem) von ihnen erben [GHJV95].

Da Mixins nur eine sehr spezielle Form der Mehrfachvererbung darstellen und darauf abzielen, Funktionalität zu ergänzen und nicht zu verändern, werden Probleme, wie die Gefahr der Mehrdeutigkeit bei mehreren indirekten Vererbungen derselben Basisklasse (das sogenannte *Diamond-Problem*) im Allgemeinen vermieden.

4.3.2 Aufbau

Das TaskEngine-Framework besteht aus mehreren Bausteinen die, wie in Abbildung 4.1 gezeigt, miteinander zusammenhängen und interagieren. Eine Ablaufsteuerung übermittelt Ereignisse an Tasks, in denen die Geschäftslogik implementiert ist. Tasks sind im allgemeinen zur Erhöhung der Modularität in unabhängige, wiederverwendbare Components unterteilt. Die Kommunikation

zwischen Tasks beziehungsweise Components erfolgt über Messages, wobei die Möglichkeiten zum Nachrichtenaustausch von einer allen gemeinsamen TaskEngine bereitgestellt werden. Sie verwaltet auch die Tasks.

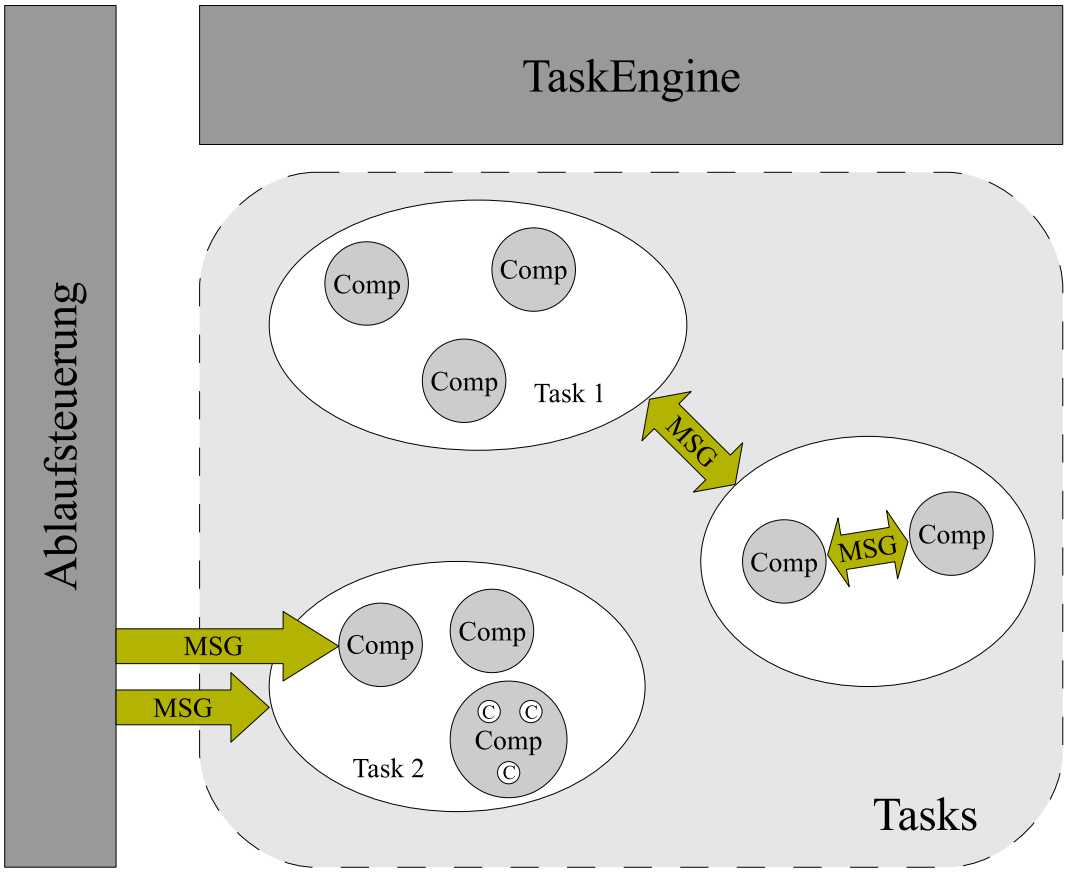


Abbildung 4.1: Aufbau des TaskEngine Frameworks

4.3.2.1 Tasks

Die eigentliche Applikationslogik, beziehungsweise bei komplexeren Anwendungen Teilaufgaben der Applikation werden als Task realisiert. Ein Task entspricht etwa einem Prozess in einem Multitasking Betriebssystem. So können beispielsweise auch die Tasks des TaskEngine-Frameworks neue Tasks generieren und werden benachrichtigt, wenn diese beenden. Desweiteren verfügen Tasks auch über eine *Environment*-Datenstruktur, die an Sub-Tasks weitergegeben wird.

Tasks erhalten beim Start eine eindeutige TaskID und sind über diese auch adressierbar.

Tasks werden als Folder-Objekte in der Zope-Objektdatenbank (ZODB) implementiert. Ihre Methoden (hauptsächlich Message-Handler) werden einfach als Python-Script-Objekte in diesem Folder angelegt.

4.3.2.2 Components

Tasks bestehen im Allgemeinen aus mehreren Komponenten. Diese weitere Unterteilung wurde zur Erhöhung der Reusability eingeführt. Da das primäre Einsatzgebiet dieses Frameworks die Erstellung interaktiver Applikationen ist, werden in dieser Arbeit beispielsweise Bauelemente des Benutzerinterfaces als Components modelliert. Diese ab Seite 81 beschriebenen Elemente sind eine Mischung aus sogenannten GUI-Widgets, die bei der Programmierung klassischer graphischer Benutzeroberflächen verwendet werden ([SA88]), und Formularelementen, die die Kontrollelemente bei Web-Anwendungen darstellen.

Komponenten können nach dem Vorbild der *Object Composition* ([GHJV95]) wiederum andere Komponenten verwenden, wodurch beispielsweise komplexere User-Interface-Bausteine realisiert werden können.

Ein typisches Beispiel einer Task- und Komponentenhierarchie ist in Abbildung 4.2 auf Seite 56 dargestellt.

Die Implementierung der Komponenteninfrastruktur erlaubt es, Komponenten entweder in Form von Modulen und Klassen im Filesystem des Zope-Servers zu installieren, oder in Form von Folder-Objekten in der ZODB abzulegen. Ersteres ist für die Erstellung von Basiskomponenten gedacht, die beispielsweise gemeinsam mit der TaskEngine-Installation ausgeliefert werden oder als Plugins in Form von Erweiterungspaketen angeboten werden können. Der zweite Ansatz erlaubt die „Through the Web“ Entwicklung spezialisierter Komponenten, die ihrerseits auf der Basis aufsetzen können.

4.3.2.3 TaskEngine

Die Aufgaben der `TaskEngine`-Klasse liegen in der Verwaltung von Tasks und der Bereitstellung einer Kommunikationsinfrastruktur:

Aufgaben des Task-Management umfassen das Öffnen und Beenden (Schließen) von Tasks. Auch die innerhalb einer Session eindeutige Durchnummerierung der Tasks und die Verwaltung dieser TaskIDs wird von der TaskEngine erledigt.

Zur Kommunikation zwischen Tasks und auch zur Benachrichtigung über Ereignisse der Ablaufsteuerung oder der TaskEngine selbst, wird eine Messaging-Schnittstelle angeboten. Über sie können Nachrichten sofort oder asynchron (gequeued) übermittelt werden.

Über Notifikationen wird Tasks auch die Möglichkeit eingeräumt, sich für die Beobachtung von Ereignissen zu registrieren. Bei Auftreten des entsprechenden Ereignisses wird schließlich eine Message an den entsprechenden Task übermittelt.

Messages sind auch die Möglichkeit, einem Task die Kontrolle über den weiteren Programmablauf zu übertragen. Nachdem die TaskEngine keine Betriebsmittelverwaltung durchführt und auch keine Möglichkeit vorgesehen ist, einem Task die Kontrolle über die weitere Ausführung zu entziehen, müssen die Tasks diese Aufgabe selbst übernehmen. Sie können die Kontrolle entweder die durch Beendigung des Message-Handlers an die TaskEngine zurückgeben oder sie durch Senden einer synchronen Message temporär einem anderen Task übertragen. In einem Mehrprozessbetriebssystem würde diese Art der Prozessverwaltung dem *kooperativem Multitasking* entsprechen.

4.3.2.4 Ablaufsteuerung

Die Ablaufsteuerung verknüpft eine Web-Anfrage mit der ihm zugehörigen TaskEngine und bewirkt über die Ereignissteuerung und das Messaging-System der TaskEngine die eigentliche Abarbeitung eines Web-Requests, nämlich:

- Die Zuordnung eines Web-Requests zu einer Session.
- Die (Re-) Aktivierung des TaskEngine-Objekts. Initialisierung einer neuen TaskEngine beziehungsweise „Auftauen“ einer bestehenden TaskEngine-Objektstruktur.
- Unterteilung des Request-Response Ablaufs in mehrere Stufen und Benachrichtigung der beteiligten Komponenten über Messages:
  - Authentisierung des Requests
  - Validierung der RequestID

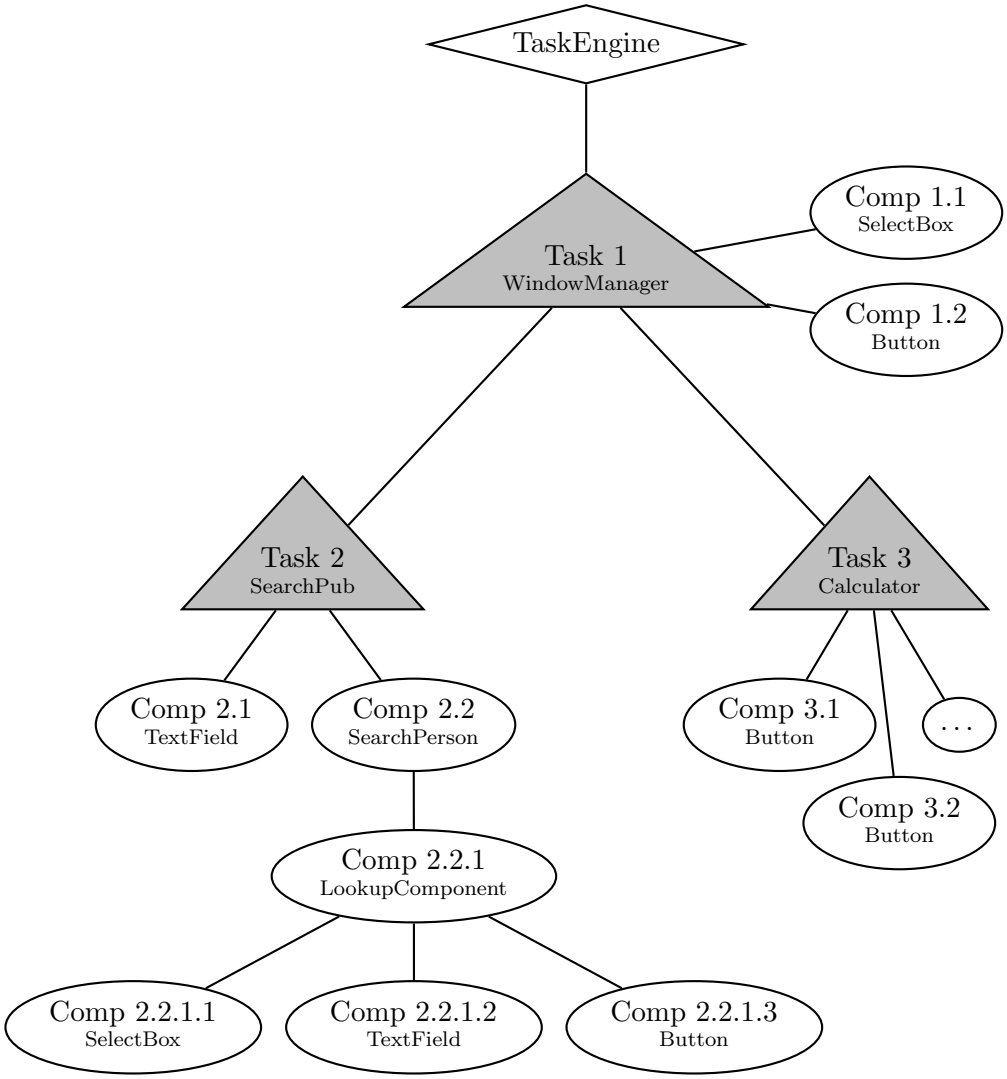


Abbildung 4.2: Typische Task-Component-Hierarchie

  - Übermittlung von Formulardaten beziehungsweise von Nachrichten, die über die GET-Methode an Komponenten gesandt werden.
  - Post-Redirect-Get Behandlung (siehe Abschnitt 3.1.3)
  - Versenden der Aufforderung zum Erzeugen einer Response
- Das „Einfrieren“ der TaskEngine
- Die Ausgabe der Antwort

Die Implementierung dieser Funktionalität erfolgt zur Zeit über ein Python-Script-Objekt namens *index_html* das im Zope Objektfilesystem im Hauptverzeichnis des Projektes abgelegt ist.

4.3.2.5 Der WindowManager

Der erste Task einer Session (er erhält von der TaskEngine die TaskID Nummer 1) ist immer der *WindowManager*. Seine Aufgaben bestehen in der Vorgabe eines Basis-Layouts sowie der Verwaltung der übrigen Tasks. Der WindowManager ist vergleichbar mit der Kombination aus WindowManager und Desktop-Manager eines klassischen UNIX X-Windows Systems.

Wichtige Aufgaben des WindowManagers sind:

- Das Zeichnen eines Grundgerüstes der Seite. (Z.B. der Rahmen, diverse Style-Anweisungen und das Navigationsmenü)
- Die Entscheidung, welche der aktiven Tasks auf der Ausgabeseite erscheinen sollen und gegebenenfalls die Aufforderung dieser Tasks, Ihre Ausgabe zu zeichnen. (Über das Senden einer `paint`-Message.)
- Das Starten neuer Tasks. (Beispielsweise neuer Programmmodule.)
- Eventuell das Bereitstellen von Möglichkeiten, zwischen Tasks zu wechseln.
- Das Setzen einer adäquaten Aktion beim Beenden eines Tasks. (Z.B. die Anzeige des zuletzt aktiven Tasks oder eines Startmenüs.)

Üblicherweise enthalten Web-Seiten, die mit Hilfe der TaskEngine erstellt werden, nur ein Web-Formular, dessen Grundgerüst (die eröffnende `<FORM ...>` Anweisung beziehungsweise der schließende `</FORM>`-Befehl) vom WindowManager ausgegeben wird. Dadurch wird erreicht, dass alle weiteren Tasks innerhalb dieses Bereiches gezeichnet werden und sichergestellt, dass sämtliche Benutzerinteraktionen auf der Seite beim nächsten Request übermittelt werden, egal welche Tasks betroffen sind. Da die Namen der Formularelemente über Methoden der Komponenten vergeben werden, ist auch sichergestellt, dass die abgesandten Daten bei der Analyse des Requests, automatisch an die richtigen Komponenten übermittelt werden können.

4.3.3 Das Zusammenspiel

4.3.3.1 Der Request-Response-Zyklus

Dieser Abschnitt beschreibt die schrittweise Abarbeitung eines typischen (Web-) Requests bis hin zur Ausgabe einer Seite. Zunächst wird dieser Vorgang aus der Sicht der Ablaufsteuerung erklärt, der zweite Teil befasst sich mit den Ereignissen, die im Zuge eines solchen Verlaufs generiert und an die entsprechenden Tasks und Komponenten übermittelt werden.

Die Ablaufsteuerung

Der Kontrollfluss über einen Request- Response Zyklus wird von der Ablaufsteuerung moderiert und folgt dem in Abbildung 4.3 und 4.4 dargestellten Schema. Die im Flussdiagramm nur stichwortartig behandelten Stufen des Programmablaufes werden nachfolgend beschrieben:

Session In dieser Phase wird überprüft, ob der aktuell behandelte Request bestehenden Session gehört?

- Falls Ja, wird das bestehende TaskEngine-Objekt „aufgetaut“.
- Falls Nein, wird ein neues Task-Engine Objekt instanziert und der *WindowManager* wird als erster Task gestartet.

Überprüfung Request-ID Hier wird die Nummer des aktuellen Requests (die *RequestId*) mit dem erwarteten Wert verglichen. Falls die ID nicht gültig ist, hat der Benutzer hat einen der Navigations Buttons im Browser

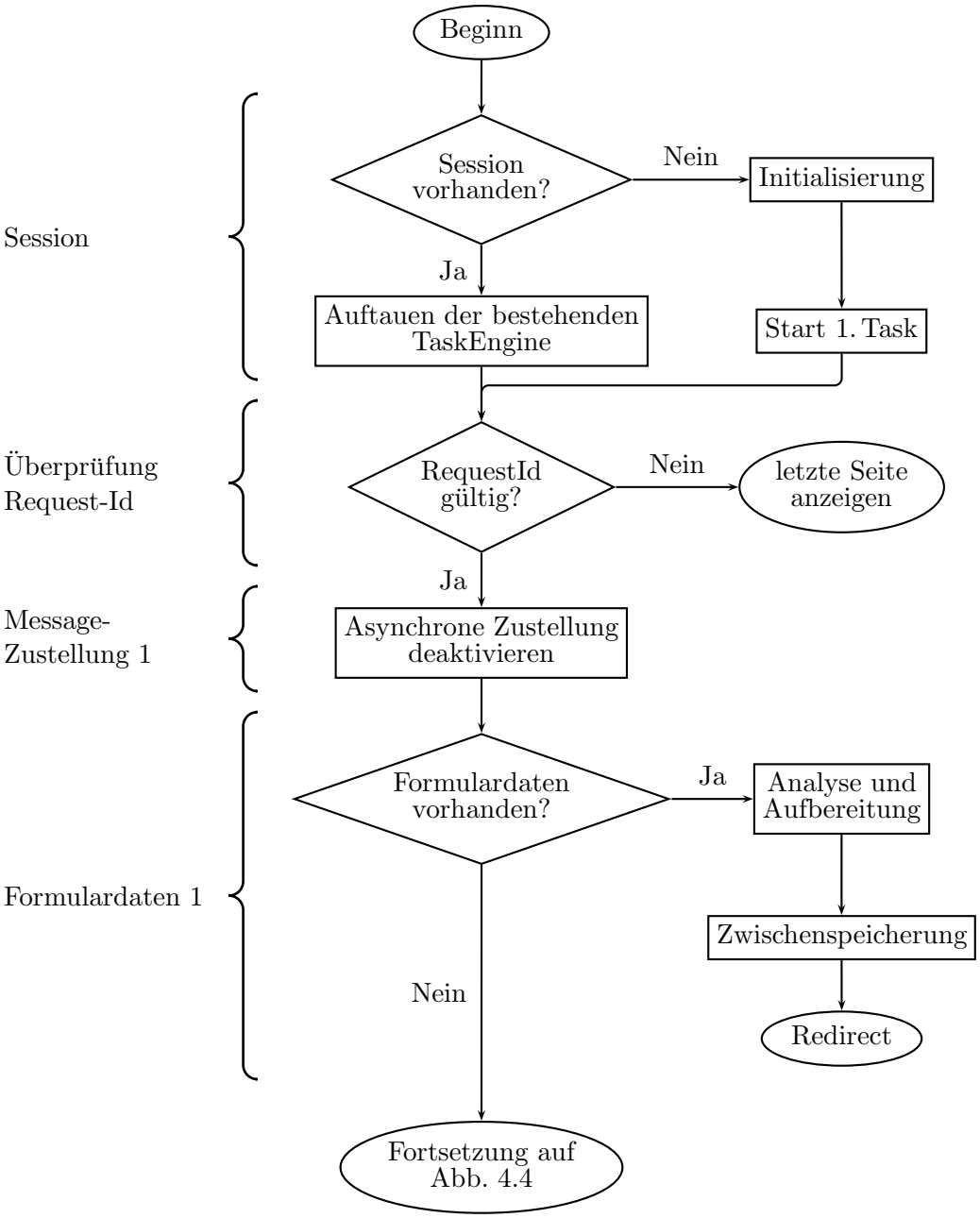


Abbildung 4.3: Ablauf eines Request-Response Zyklus, Teil 1

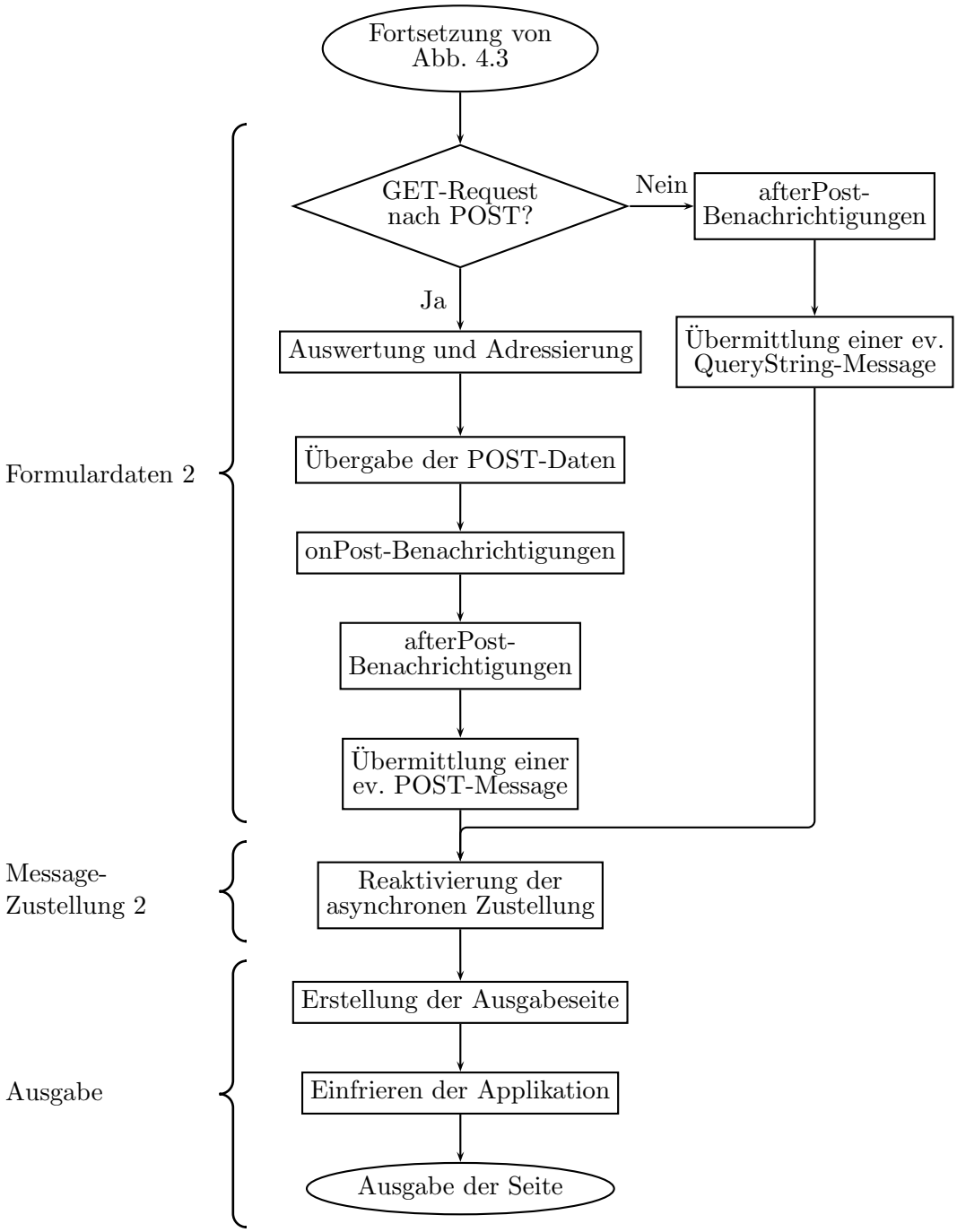


Abbildung 4.4: Ablauf eines Request-Response Zyklus, Teil 2

betätigt oder den URL manipuliert. In diesem Fall wird die zuletzt angezeigte Seite wieder ausgegeben. Der interne Status der Applikation bleibt ansonsten unverändert.

Message-Zustellung 1 und 2 Um einen konsistenten Zustand der Applikation gewährleisten zu können, ist es notwendig, vor der Auswertung der Formulardaten und der Übermittlung der Daten an die Komponenten, die asynchrone Nachrichtenzustellung temporär zu deaktivieren. Nur so wird sichergestellt, dass zum Zeitpunkt des Eintreffens der entsprechenden Benachrichtigungen (der `onPost`-Messages) bereits alle Daten an die Komponenten übermittelt wurden. Während die Zustellung stillgelegt ist, werden über die Methode `postMessage` versandte Nachrichten gequeued. Bei der Reaktivierung wird diese Warteschlange schließlich abgearbeitet.

Formulardaten 1 An dieser Stelle werden Formulardaten, die über die Request-Methode POST übermittelt werden, entgegengenommen. Im Zuge der Analyse und Aufbereitung der Formulardaten werden die als Text-String in die Formulardaten kodierten Adressierungsinformationen und Nutzdaten extrahiert und in deserialisierter Form in der von Zope verwalteten Session gespeichert. Als nächster Schritt des *POST-REDIRECT-GET* Schemas (vergleiche Abschnitt 3.1.3) wird der aktuelle Request durch das Versenden einer *Redirect*-Antwort beendet. (Der Browser wird dadurch veranlasst, einen neuen GET-Request zu übermitteln, wodurch ein neuer Ablauf bei „Beginn“ gestartet wird.)

Formulardaten 2 Im zweiten Teil der POST-REDIRECT-GET-Behandlung werden die vom vorigen Request zwischengespeicherten Daten ausgewertet und an die entsprechenden Komponenten direkt übergeben. Anschließend werden `onPost`-Benachrichtigungen für die betreffenden Components vorbereitet. (Die Nachrichten kommen in die Warteschlange, da die Zustellung ja ausgesetzt wurde.) Damit auch jede Komponenten berücksichtigt werden, die keine Post-Daten empfangen haben, wird schließlich auch allen Komponenten, die sich für dieses Ereignis registriert haben, über eine `afterPost`-Message mitgeteilt, dass eine Verteilung der Daten erfolgt ist.

Zusätzlich zur Möglichkeit der Datenübermittlung besteht auch die Option, eine Message über die POST-Daten des Web-Requests zu übermitteln. Falls eine solche Message existiert wird sie an dieser Stelle in die

Warteschlange für asynchrone Nachrichten eingereiht. Die derzeitige Implementierung der Ablaufsteuerung sieht die Übermittlung von nur einer Message pro Post-Request vor.

Wird der „Nein"-Zweig eingeschlagen, handelt es sich um einen GET-Request, der nicht Teil des POST-REDIRECT-GET Ablaufes ist. Um eine Symmetrie zum Ablauf bei POST-Requests zu erreichen, wird auch hier für Komponenten, die sich dafür angemeldet haben, eine `afterPost`-Message vorbereitet. Auch eine etwaige, im Query-String des GET-Requests vorhandene Nachricht wird in die Queue eingeordnet.

Ausgabe In der Ausgabephase wird zunächst die Generierung der Ergebnisseite durch Übermittlung einer `paint`-Message an den Window-Manager (Task 1) angestoßen. Dieser veranlasst schließlich selbstständig (wiederum durch Senden von `paint`-Messages) das Zeichnen der anzuzeigenden Tasks. Das Resultat dieser Aktionen wird in einem internen Puffer der TaskEngine gespeichert.

Je nach verwendeter Persistence-Technik wird im folgenden Schritt die gesamte TaskEngine-Objekthierarchie möglicherweise serialisiert und bei den Zope-Sessiondaten gespeichert. In der bevorzugten Implementierungsvariante ist diese Objekthierarchie allerdings in einem von allen Threads des Zope-Systems gemeinsam genutzten Speichersegment abgelegt. In dieser Ausführung ist also kein „Einfrieren" und „Auftauen" erforderlich.

Waren alle vorhergehenden Schritte erfolgreich, wird die aktuelle Anfrage schließlich durch die Ausgabe der vorbereiteten Seite abgeschlossen.

Die Kommunikation mit den Tasks und Komponenten

Um die Applikationslogik implementieren zu können, besteht für Tasks und Komponenten die Möglichkeit, sich an bestimmten Punkten der Request-Bearbeitung „einzuhängen". Hierzu werden, wie aus dem Flussdiagramm der Ablaufsteuerung ersichtlich ist, an diesen relevanten Stellen Messages generiert, die (je nach Event) entweder an bestimmte Komponenten direkt versandt werden, oder an alle verschickt werden, die sich für ein bestimmtes Ereignis interessieren.

Abbildung 4.1 zeigt eine typische Kommunikation zwischen der Ablaufsteuerung und Komponenten, sowie schließlich auch zwischen den Komponenten.

Ablaufstufe	**Message**	**Absender**	**Empfänger**
Formulardaten 2	`onPost`	Ablaufsteuerung (via TaskEngine)	Comp 1.1
			Comp 2.1
			Comp 2.2.1.1
			Comp 2.2.1.2
	`afterPost`	TaskEngine	Comp 1.1
			Comp 2.1.1.1
Message-Zustellung 2	`msgSearch`	Comp 2.1	Comp 2
Ausgabe	`paint`	Ablaufsteuerung	Task 1
	`paint`	Task 1	Task 2

Tabelle 4.1: Chronologischer Ablauf der Nachrichtenzustellung (Beispiel)

POST/GET Request Behandlung

Um die saubere Behandlung der Daten gewährleisten zu können, sollten Programme, die Formularfelder beinhalten, immer die HTTP-Methode POST für sämtliche Aktionen verwenden. Nur dadurch kann sichergestellt werden, dass auch bei der gleichzeitigen Darstellung mehrerer Components oder sogar mehrerer Tasks sämtliche Benutzereingaben an den Server übermittelt werden.

Da die TaskEngine die automatische Behandlung von POST-Requests nach dem *PRG-Schema* (siehe Abschnitt 3.1.3) in ihrer Ablaufsteuerung vorsieht, ist von Seiten des Benutzers kein weiter Aufwand mehr erforderlich. Eine mit diesem Framework erstellte Applikation unterstützt also automatisch das *PRG-Pattern*.

Um eine direkte Verlinkung zu bestimmten Zuständen der Applikation zu ermöglichen (Bookmarks), besteht auch die Möglichkeit, Messages als HTTP-GET-Request, also als Teil des URLs zu übermitteln. Hier ist aber zu beachten, dass etwaige Benutzereingaben, die in Formularfeldern der laufenden Applikation vorgenommen wurden, durch Anklicken dieses URLs verworfen werden. Das GUI-ähnliche Verhalten der Anwendung, das durch dieses Framework angestrebt wird, geht also dadurch verloren.

4.3.3.2 Zugriffssicherung

Die Authentisierung und Rollenzuteilung wird in der aktuellen Implementierung von Zope erledigt. Durch entsprechende Definition der Zope-Security-Attribute des Projektfolders kann eine generelle Authentisierung erzwungen werden. Durch selektive Berechtigungsvergabe auf die entsprechenden Task-Verzeichnisse kann der Zugriff auch individuell für einzelne Tasks beschränkt werden.

Darüberhinaus besteht innerhalb der Anwendung (in der Ablaufsteuerung und natürlich auch in den benutzerdefinierten Tasks) die Möglichkeit, die Authentisierungsdaten zu verwenden, um eine eigene Autorisierungslogik zu entwickeln. Hierzu muss über die Methode `getSecurityManager()` des Zope-Moduls `AccessControl` ein Security-Manager Objekt geholt werden, über das schließlich der aktuell angemeldete Benutzer mit `getUser()` ermittelt werden kann. Der resultierende Objekt, das von der Klasse `BasicUser` abstammt, enthält außer dem Usernamen (`getUserName()`) auch noch Informationen über die dem User zugeteilten Rollen (`getRoles()` und `getRolesInContext(`*`object`*`)`).

Weitere Details zu den erwähnten Klassen sind in [LPMS] dokumentiert.

4.3.3.3 Persistence

Da der Zope-Server die Anfragen in mehreren Threads abarbeitet, ist es notwendig, den Zustand der Anwendung nach jedem behandelten Request threadunabhängig, in einem gemeinsam genutzten Bereich sichern zu können.

Zu diesem Zweck, wurden in der Ablaufsteuerung zwei alternative Mechanismen implementiert, die grundsätzlich unterschiedliche Ansätze verfolgen:

Die erste Variante sichert die gesamte TaskEngine-Objekthierarchie in einem globalen Speicherbereich. Der Zugriff auf diesen Bereich ist mit Hilfe des Moduls `SharedResource` des Zope Entwicklers Dieter Maurer realisiert[10], das auch Locking-Mechanismen zur Verfügung stellt, um den konkurrierenden Zugriff auf die Daten aus mehreren Threads kontrollieren zu können.

Die zweite Version nutzt die Persistence-Maschinerie der ZODB und das Zope Session Management zur Speicherung des Applikationszustandes. Hier wird nach jedem Request die gesamte Objekthierarchie serialisiert und in den

[10] `http://www.dieter.handshake.de/pyprojects/zope/SharedResource.html`

Session-Daten abgespeichert. Beim nächsten Request wird die TaskEngine wieder aus der Session geholt und damit von Zope automatisch wieder „aufgetaut“.

Der Vorteil der ersten Methode ist die höhere Geschwindigkeit, da Serialisierungsaufwand nach jedem Request und die anschließende Deserialisierung bei der nächsten Anfrage entfällt. Dafür belegen die gesicherten Anwendungsdaten virtuellen Arbeitsspeicher des Servers, ein Nachteil, der allerdings erst bei sehr vielen Sessions ins Gewicht fallen wird. In der derzeitigen Version der „`SharedResource`“-Implementierung überstehen die Sitzungsdaten auch einen Serverstart nicht, ein Problem, dass allerdings durch eine erzwungene Serialisierung bei einem Server-Shutdown gelöst werden könnte. Da die Daten im Speicher des Servers abgelegt sind, ist auch ein Clustering mehrerer Zope-Server nicht möglich. (Zumindest muss garantiert werden, dass alle Requests einer Session immer zum selben Zope-Server geleitet werden.) Bei der zweiten Implementierung ist nachteilig anzumerken, dass nicht alle Objekte automatisch serialisierbar sind. In einigen Fällen sind manuelle Eingriffe in Form von Helper-Methoden notwendig, wie die Methoden `__getstate__()` und `__setstate__()` in einigen TaskEngine-Klassen zeigen.

In beiden Fällen werden die gesicherten Objekte über den Zope-Session-Mechanismus nach konfigurierbarer Inaktivitätszeit gelöscht, um den belegten Speicher wieder freizumachen.

4.3.3.4 Das Starten eines neuen Tasks

Ein Task kann mit Hilfe der `openTask()` Methode einen neuen Untertask starten. Dieser Subtask erhält von der TaskEngine eine neue TaskEngine zugewiesen die auch dem öffnenden Task als Resultat das `openTask()`-Aufrufs zurückgeliefert wird.

Damit der neu erzeugte Task Initialisierungsaufgaben vornehmen kann, wird ihm von der TaskEngine unmittelbar (synchron, mit Hilfe der Methode `sendMessage()`) eine `start`-Message übermittelt. Im `start`-Message-Handler können beispielsweise Datenstrukturen angelegt oder Komponenten beziehungsweise weitere Tasks erzeugt werden.

4.3.3.5 Das Einbinden von Components in einen Task

Komponenten werden über die Methode `addComponent` aus Tasks oder anderen Methoden heraus generiert. Die erhalten eine hierarchisch aufgebaute, eindeutige Komponentenadresse, über die sie für die Übermittlung von Messages oder Formulardaten direkt adressierbar sind.

Im Unterschied zu Tasks sind Komponenten mit ihrem Erzeuger eng gekoppelt. Die Methode `addComponent()` liefert direkt das neue `Component`-Objekt zurück, über das ein direkter Zugriff auf die Methoden und Attribute der Komponente möglich ist.

4.3.3.6 Das Beenden eines Tasks

Das TaskEngine-Framework sieht zwei Möglichkeiten für das Beenden eines Tasks vor: Er kann sich entweder selbst beenden (`shutdown()` oder `closeTask()`), oder von einem anderen Task unter Angabe der TaskId beendet werden (`closeTask(taskId)`).

Das Terminieren eines Tasks bewirkt in jedem Fall das rekursive Schließen aller Subtasks. Dieser Vorgang wird von der TaskEngine ausgeführt. Zuletzt wird schließlich der Parent-Task durch Übermittlung einer `sigchld`-Message (mit entsprechende TaskId) darüber informiert, dass einer seiner Child-Tasks soeben beendet hat.

Durch diese Logik sichergestellt, dass zu jedem Task (mit Ausnahme von Task 1, dem WindowManager) immer ein Parent-Task existiert. Andererseits bewirkt das Schließen eines Tasks auch immer, dass gleichzeitig sämtliche (meist für Hilfsaufgaben erzeugte) Sub-Tasks aufgeräumt werden.

4.3.4 Das TaskEngine API

4.3.4.1 class TaskEngine

Öffentliche Methoden

`getTaskList()`

Liefert die TaskIDs der aktiven Tasks in Form einer Liste zurück.

`getTask(`*`taskID`*`[, `*`default`*`])`

Liefert das Task-Object des Tasks mit der angegebenen *taskID*. Falls der der Task nicht existiert, wird *default* zurückgeliefert beziehungsweise eine `KeyError`-Exception ausgelöst, falls kein Wert für *default* angegeben wurde.

`postMessage(Message(...))`
`postMessage(`*`fromaddr`*`, `*`toaddr`*`, `*`message`*`, `*`data`*`=())`
`postMessage()`

Versendet eine Message asynchron. Die Message wird in eine Queue eingereiht beziehungsweise zugestellt, falls gerade keine andere Message bearbeitet wird.

Als Argument kann, wie in der ersten Variante gezeigt, eine `Message`-Instanz (erste Form) übergeben werden. Alternativ könne auch dieselben Argumente, die zur Instanzierung einer Message verwendet werden, angegeben werden. Die dritte Form erlaubt die Wiederaufnahme der Message-Zustellung nachdem das asynchrone Versenden mittels `queueMessages(1)` deaktiviert wurde.

`sendMessage(Message(...))`
`sendMessage(`*`fromaddr`*`, `*`toaddr`*`, `*`message`*`, `*`data`*`=())`

Versendet eine Message synchron. Die Message wird sofort zugestellt. Die Ausführung der Programmlogik setzt mit dem Message-Handler der versendeten Message fort. Rückgabewert von `sendMessage()` ist der Rückgabewert des Message-Handlers.

Als Argument kann entweder eine `Message`-Instanz (erste Form) übergeben werden. Alternativ können auch dieselben Argumente, die zur Instanzierung einer Message verwendet werden, angegeben werden.

`queueMessages(`*`switch`*`)`

Aktiviert (*switch*=`False`) oder deaktiviert (*switch*=`True`) die asynchrone Nachrichtenzustellung. Bei deaktivierter Zustellung werden mittels `postMessage()` gesendete Nachrichten gequeued.

`notifyComponents(`*`eventname`*`)`

Benachrichtigt die Components, die sich zuvor über die Methode `addNotifyEvent()` für das Ereignis *eventname* registriert haben.

Siehe auch: `addNotifyEvent()`, `removeNotifyEvent()`

`getRequest()`

Liefert das aktuelle ZOPE-`REQUEST`-Objekt zurück. (Siehe [LPMS].)

`getResponse()`

Liefert das aktuelle ZOPE-`RESPONSE`-Objekt zurück. (Siehe [LPMS].)

`getContext()`

Liefert das aktuelle ZOPE-`context`-Objekt zurück. (Siehe [LPMS].)

`getTasksFolder()`

Liefert das Tasks-Folder-Objekt des aktuellen Projektes zurück. Die ist üblicherweise der Subfolder namens `Tasks` des Projektverzeichnisses in der ZODB.

`getCompsFolder()`

Liefert das Components-Folder-Objekt des aktuellen Projektes zurück. Die ist Üblicherweise der Subfolder namens `Components` des Projektverzeichnisses in der ZODB.

`getOutputBuffer()`

Liefert den aktuellen Ausgabepuffer (in Form eines `unicode`-Objektes) zurück.

`getRequestId()`

Liefert die aktuelle RequestId zurück.

`getBaseUrl()`

Liefert den Basis-URL des Projektes, inklusiver der nächsten gültigen Request-Id zurück.

Interne Methoden

Diese Methoden werden zwischen den Basis-Klassen der TaskEngine verwendet und sollten von benutzerdefinierten Tasks und Komponenten nicht aufgerufen werden. Private Methoden, die nur innerhalb einer Klasse verwendet werden, werden hier nicht angeführt.

`__init__(`*environment*`={})`

Initialisierung einer neuen Taskengine. Als optionales Argument kann ein Environment in Form eines Dictionarys angegeben werden. Eine Kopie dieses Environments wird allen neu gestarteten Tasks beim Start übergeben.

`isNew()`

Liefert `True`, wenn die TaskEngine soeben instanziert wurde und noch nicht initialisiert ist.

`openTask(`*pTaskId*`,`*taskName*`,`*startData*`=None)`

Diese Methode wird von der Task-Klasse beziehungsweise von der Ablaufsteuerung verwendet, um einen neuen Task zu generieren.

pTaskId ist die Parent Task-Id, *taskName* gibt den Namen (als String) des zu startenden Tasks an. Bei Tasks, die im Zope-Object-Filesystem implementiert sind, entspricht der *taskName* dem gleichnamigen Container-Objekt im `Tasks`-Folder des Projektes.

Die optionalen Daten *startData* werden der `start`-Message als Argument mitgegeben. (Siehe 4.3.3.4.)

Benutzerdefinierte Tasks sollten die `openTask()`-Methode der `Task`-Klasse verwenden.

`closeTask(`*`taskId`*`)`

Beendet den Task mit der ID *taskId*.

Benutzerdefinierte Tasks sollten die `closeTask()`-Methode der `Task`-Klasse verwenden.

`getComponent(compaddr,` *`default`*`)`

Liefert das Component-Object der Komponente mit der angegebenen Adresse *compaddr*.

Falls die gesuchte Komponente nicht existiert, wird *default* zurückgeliefert beziehungsweise eine `KeyError`-Exception ausgelöst, falls kein Wert für *default* angegeben wurde.

`appendBuffer(`*`ustring`*`)`

Hängt den `unicode`-String *ustring* an den aktuellen Ausgabepuffer an.

Benutzerdefinierte Tasks sollten die `paint()`-Methode der `Task`-Klasse verwenden.

`setContext(`*`context`*`)`

Setzt das aktuelle ZOPE-`context`-Objekt zur weiteren Verwendung des Objekts innerhalb des TaskEngine-Frameworks mittels `getContext()`.

`setRequestId(`*`reqid`*`)`

Setzt die aktuelle RequestId zur weiteren Verwendung über die Methode `getRequestId()`.

`setEnvironment(`*`environment`*`):`

Setzt das Environment der TaskEngine. Siehe auch `__init__()`.

`getRequestData(`*`addr`*`=(0, ))`

Stellt eine temporäre Datenstruktur in Form eines Dictionaries bereit, das von Task oder der Komponenten mit der Adresse *addr* zur Speicherung von Per-Request-Daten verwendet wird. Falls bereits vorhanden, wird die bestehende Datenstruktur zurückgeliefert.

`mkMessage(`*`fromaddr`*`, `*`toaddr`*`, `*`message`*`, `*`data`*`=())`

Factory-Methode der `Message`-Klasse. Erzeugt eine neue `Message`-Instanz. Siehe 4.3.4.6.

`decodeMsgStr(`*`toaddr`*`, `*`msgstr`*`, `*`fromaddr`*`=[-1])`

Wird von der Ablaufsteuerung verwendet, um eine serialisierte Message aus den POST-Daten des Requests in ein `Message`-Objekt umzuwandeln.

`decodeAddrStr(`*`addrstr`*`)`

Wird von der Ablaufsteuerung verwendet, um eine serialisierte Component-Adresse aus den POST-Daten des Requests zurück zu wandeln.

`encodeDataForm(`*`toaddr`*`, `*`data`*`)`

Serialisiert die Adresse *todata* und den Datenstring *data* für die Verwendung als Name eines HTML-Formularfeldes.

Benutzerdefinierte Components und Tasks sollten die `encodeDataForm()`-Methode der `Component`-Klasse verwenden.

`encodeMsgForm(`*`toaddr`*`, `*`msg`*`, `*`msgdata`*`=None)`

Serialisiert eine Message für die Verwendung als Name eines HTML-Formularfeldes.

Benutzerdefinierte Components und Tasks sollten die `encodeMsgForm()`-Methode der `Component`-Klasse verwenden.

`encodeMsgUrl(`*`toaddr`*`, `*`msg`*`, `*`msgdata`*`=None)`

Serialisiert eine Message für die Verwendung als Link in einer HTML-Seite. Liefert einen vollständigen URL auf das Projekt zurück.

Benutzerdefinierte Components und Tasks sollten die `encodeMsgURL()`-Methode der `Component`-Klasse verwenden.

`addNotifyEvent(addr, eventname, msgname, data=())`

Registriert die Komponente mit der Adresse *addr* für den Event *eventname*. Beim Eintreffen des Events (siehe `notifyComponents()`) wird die Nachricht *msgname* mit den optionalen Daten `data` an die Komponente *addr* geschickt.

Benutzerdefinierte Components und Tasks sollten die `addNotifyEvent()`-Methode der `Component`-Klasse verwenden.

`removeNotifyEvent(addr, eventname)`

Deaktiviert weitere Benachrichtigungen für den Event *eventname* für die Komponente mit der Adresse *addr*.

Benutzerdefinierte Components und Tasks sollten die `removeNotifyEvent()`-Methode der `Component`-Klasse verwenden.

`log(msg, level='debug', tag='TaskEngine')`

Schreibt die Log-Meldung *msg* mit dem Log-Level *level* in das Log-File der TaskEngine. Über *tag* kann der Name der loggenden Komponente zur Kennzeichnung angegeben werden.

Siehe auch: `log()`-Methode der `Component`-Klasse.

`cleanup()`

Löscht Per-Request Daten und bereitet die TaskEngine auf das „Einfrieren“ am Ende eines Requests vor.

4.3.4.2 Tasks

Tasks werden über Folder-Objekte im Unterverzeichnis `Tasks` des Projektes implementiert. Der Folder trägt den Namen des Tasks und beinhaltet Message-Handler für die Messages, die der Task behandeln soll.

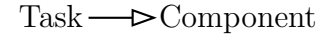
Message-Handler

msghandler(*self*, *message*)

Innerhalb der Message-Handler, die als Zope-`Script`-Objekte in Python implementiert sind, steht das API des TaskEngine-Frameworks zur Verfügung. Das erste Argument, das an einen Message Handler Übergeben wird, ist immer das aktuelle Task-Objekt. Es sind daher sämtliche Methoden der Task-Klasse (siehe 4.3.4.3) direkt verwendbar. Da über die `Task`-Methode `getTaskEngine()` auch das aktuelle `TaskEngine`-Objekt verfügbar ist, steht auch dessen API (siehe 4.3.4.1) zur Verfügung. Als zweites Argument folgt das Message-Objekt (siehe 4.3.4.6), das zusätzlich zur versandten Nachricht unter anderem auch optionale Daten enthalten kann.

Desweiteren ist in Message-Handlern natürlich die Verwendung sämtlicher Python-Funktionen möglich, die in Zope-`Script`-Objekten erlaubt sind. (Siehe [LPMS].)

4.3.4.3 class Task



Die Klasse `Task` erbt von `Component`. Es stehen daher sämtliche Methoden der `Component`-Klasse zur Verfügung (siehe 4.3.4.5 ab Seite 75). Im folgenden Abschnitt werden ausschließlich die `Task`-spezifischen Methoden beschrieben.

Öffentliche Methoden

openTask(*taskName*, *startData*=None)

Startet einen neuen Task mit dem Namen *taskName* als Subtask des aktuellen Tasks.

taskName gibt den Namen (als String) des zu startenden Tasks an. Bei Tasks, die im Zope-Object-Filesystem implementiert sind, entspricht der *taskName* dem gleichnamigen Container-Objekt im `Tasks`-Ordner des Projektes.

Die optionalen Daten *startData* werden der `start`-Message als Argument mitgegeben. (Siehe 4.3.3.4.)

`closeTask(`*`taskid`*`=None)`

Beendet den Task mit der ID *taskId*. Falls keine *taskId* angegeben wird, wird der aktuelle Task beendet.

Siehe auch 4.3.3.6.

`shutdown()`

Entspricht `closeTask(getTaskId())`, beendet also diesen Task.

`paint(`*`ustring`*`)`

Schreibt den `unicode`-String *ustring* in den Ausgabepuffer. Diese Methode wird üblicherweise vom `paint`-Messagehandler eines Tasks aufgerufen, um die Ausgabe des aktuellen Tasks zu veranlassen.

`getBaseUrl()`

Entspricht der Methode `getBaseUrl()` der Klasse `TaskEngine`.

`getTaskId()`

Liefert die ID des aktuellen Tasks zurück.

`getPTaskId()`

Liefert die ID des Parent-Tasks zurück, beziehungsweise den Wert 0, wenn der aktuelle Task keinen Parent-Task besitzt.

`getTaskFolder()`

Liefert das Folder-Objekt der ZODB zurück, das den aktuellen Task implementiert.

4.3.4.4 Components

Components, die „Bausteine“ eines Tasks, können entweder als benutzerdefinierte Components in `Folder`-Objekten der ZODB implementiert sein, oder in Form von vordefinierten Components vorliegen, die Bestandteil der TaskEngine-Paketes sind und als Python-Module im Filesystem des ZOPE-Servers liegen.

Benutzerdefinierte Components werden, analog zu Tasks, über Folder-Objekte im Unterverzeichnis `Components` des Projektes realisiert. Der Folder trägt den Namen des Components und beinhaltet Message-Handler für die Messages, die die Komponente behandeln soll.

Components stammen in jeden Fall von der Klasse `Component` ab und besitzen somit die in 4.3.4.5 beschriebenen Methoden.

`msghandler(self, message)`

Der Aufbau der Message-Handler für Components entspricht genau jenem für Tasks. (Vergleiche 4.3.4.2). Der einzige Unterschied besteht im ersten Argument, das bei Tasks das aktuelle Component-Objekt darstellt. Der zugehörige Task und dessen Methoden sind über die Methode `getTask()` verfügbar.

4.3.4.5 class Component

Öffentliche Methoden

`getTask()`

Liefert das zugehörige `Task`-Objekt zurück.

`getTaskEngine()`

Liefert das `TaskEngine`-Objekt zurück.

`getParent()`

Liefert das `Component`-Objekt der übergeordneten Komponente beziehungsweise des übergeordneten Tasks.

Anstelle der obigen Methoden können auch die gleichwertigen Property-Attribute `task`, `taskengine` und `parent` verwendet werden. Die zusätzliche Implementierung als Methoden ist notwendig, da Zope aus Sicherheitsgründen das direkte Manipulieren von Attributen eines Objektes aus Python-`Script`-Objekten heraus verbietet.

`addComponent(`*`classname`*`='ZODBComponent', `*`compname`*`=None, **kwargs)`

Erzeugt ein neues `Component`-Objekt. *classname* ist der Name der Komponenten-Klasse. Bei mitgelieferten vordefinierten Components ist dies der Name der entsprechenden Klasse.

Beispiel:

```
textfield = self.addComponent(
  'TextField',
  text='bitte Suchbegriff eingeben',
  focus=1, selectOnFocus=1,
  callbacks=(('change', self.getAddress(),
              'msgSearch', 'post'),),
  )
```

Bei benutzerdefinierten Components, die über Folder-Objekte in der ZODB realisiert werden (im Verzeichnis `Components` des Projektes), ist hier ZODBComponent anzugeben (der Default-Wert). Bei ZODB-basierten Components ist es zusätzlich noch notwendig, über den Parameter *compname* den Namen des Folder-Objektes anzugeben, das die Implementierung der Komponente beinhaltet.

`closeComponent(`*`compid`*`)`

Beendet die Komponente mit der ID *compid* (einschließlich aller Subcomponents).

`shutdown()`

Beendet diese Component, einschließlich aller Subcomponents.

`iterComponents()`

Liefert einen Iterator zurück, der über alle Subcomponent-Objekte iteriert.

`iterCompItems()`

Liefert einen Iterator zurück, der *(compid, compobj)*-Tupel aller Subcomponents liefert.

`getComponentsSorted()`

Liefert eine nach der Component-ID sortierte Liste aller Subcomponent-Objekte zurück.

`getComponentIds()`

Liefert eine Liste aller Subcomponent-IDs zurück.

`getSubComponent(`*`compid, default`*`)`

Liefert das Component-Objekt mit der angegebenen Component-Id *compid* zurück.

Falls die Komponente nicht existiert, wird *default* zurückgeliefert beziehungsweise eine `LookupError`-Exception ausgelöst, falls kein Wert für *default* angegeben wurde.

`getAddress()`

Liefert die hierarchische Component-Adresse der aktuellen Komponente zurück.

`postMessage(Message(...))`
`postMessage(`*`toaddr, message, data`*`=())`

Versendet eine Message asynchron. Die Message wird in eine Queue eingereiht beziehungsweise zugestellt, falls gerade keine andere Message bearbeitet wird.

Als Argument kann entweder eine `Message`-Instanz (erste Form) übergeben werden. Alternativ können auch, mit Ausnahme der Absenderadresse, dieselben Argumente, die zur Instanzierung einer Message verwendet werden, angegeben werden. Als Absenderadresse wird immer die Adresse der aktuellen Komponente verwendet.

`sendMessage(Message(...))`
`sendMessage(`*`toaddr, message, data`*`=())`

Versendet eine Message synchron. Die Message wird sofort zugestellt. Die Ausführung der Programmlogik setzt mit dem Message-Handler der versendeten Message fort. Rückgabewert von `sendMessage()` ist der Rückgabewert des Message-Handlers.

Als Argument kann entweder eine `Message`-Instanz (erste Form) übergeben werden. Alternativ können auch, mit Ausnahme der Absenderadresse, dieselben Argumente, die zur Instanzierung einer Message verwendet werden, angegeben werden. Als Absenderadresse wird immer die Adresse der aktuellen Komponente verwendet.

`mkMessage(`*`toaddr`*`, `*`message`*`, `*`data`*`=())`

Factory-Methode der `Message`-Klasse. Erzeugt eine neue `Message`-Instanz. Absenderadresse der Message ist immer die Adresse der aktuellen Komponente. Siehe 4.3.4.6.

`addNotifyEvent(`*`eventname`*`, `*`msgname`*`, `*`data`*`=())`

Registriert die Komponente für den Event *eventname*. Beim Eintreffen des Events (siehe `notifyComponents()` der Klasse `TaskEngine` auf Seite 68) wird die Nachricht *msgname* mit den optionalen Daten *data* an die aktuelle Komponente geschickt.

`removeNotifyEvent(`*`eventname`*`)`

Deaktiviert weitere Benachrichtigungen für den Event *eventname*.

`getRequestData()`

Stellt eine temporäre Datenstruktur in Form eines Dictionaries bereit, das zur Speicherung von Per-Request-Daten verwendet wird. Falls bereits vorhanden, wird die bestehende Datenstruktur zurückgeliefert.

`encodeDataForm(`*`data`*`, `*`toAddr`*`=None)`

Serialisiert die Adresse *todata* und den Datenstring *data* für die Verwendung als Name eines HTML-Formularfeldes. Falls *toAddr* gleich `None` ist, wird die Adresse der aktuellen Komponente verwendet.

`encodeMsgForm(`*`msg`*`, `*`msgdata`*`=None, `*`toAddr`*`=None)`

Serialisiert eine Message für die Verwendung als Name eines HTML-Formularfeldes. Falls *toAddr* gleich `None` ist, wird die Adresse der aktuellen Komponente verwendet.

`encodeMsgUrl(`*`msg`*`, `*`msgdata`*`=None, `*`toaddr`*`=None)`

Serialisiert eine Message für die Verwendung als Link in einer HTML-Seite. Liefert einen vollständigen URL auf das Projekt zurück. Falls *toAddr* gleich `None` ist, wird die Adresse der aktuellen Komponente verwendet.

`log(msg, level='debug')`

Schreibt die Log-Meldung *msg* mit dem Log-Level *level* in das Log-File der TaskEngine. Zur Kennzeichnung werden der Log-Meldung der Name und die Adresse der aktuellen Komponente vorangestellt.

Interne Methoden

`getComponent(addr=None, default)`

Liefert das Component-Objekt zurück, das durch die hierarchische Component-Adresse *addr* relativ in Bezug auf den aktuellen Task referenziert wird.

Falls die Komponente nicht existiert, wird *default* zurückgeliefert beziehungsweise eine `LookupError`-Exception ausgelöst, falls kein Wert für *default* angegeben wurde.

`getMessage(message)`

Wird bei der Message-Zustellung aufgerufen, um die Message *message* an die Komponente zuzustellen, beziehungsweise an deren Subcomponents weiterzuleiten, falls es sich noch nicht um das entgültige Ziel der Message handelt.

Liefert das Ergebnis des Message-Handlers zurück, das bei der synchronen Nachrichtenzustelung dann schließlich auch der Rückgabewert von `sendMessage()` ist.

4.3.4.6 class Message

`__init__(fromaddr, toaddr, message, data=())`

Erzeugt eine neue Message-Instanz. *fromaddr* und *toaddr* sind Absender- und Empfängeradresse in Form hierarchischer Component-Adressen, wie sie zum Beispiel von der Methode `getAddress()` der Klasse `Component` geliefert werden. *message* ist der Name der Message (als String), *data* enthält optionale Message-Daten und sollte ein Python-`list`- oder `tuple`-Objekt sein.

Neue Message-Objekte können auch über die Factory-Methoden `mkMessage()` der Klassen `TaskEngine` und `Task` erzeugt werden.

`getToAddr()`

Liefert die Zieladresse der Message zurück.

`setToAddr(`*`addr`*`)`

Setzt die Zieladresse der Message.

`getFromAddr()`

Liefert die Absenderadresse der Message zurück.

`setFromAddr(`*`value`*`)`

Setzt die Absenderdresse der Message.

`getMessage()`

Liefert den Namen der Message zurück.

`setMessage()`

Setzt den Namen der Message.

`getData()`

Liefert die optionalen Message-Daten zurück.

`setData(`*`data`*`=())`

Setzt die Message-Daten. *data* sollte ein Python-`list`- oder `tuple`-Objekt sein.

Anstelle der obigen Methoden können auch die gleichwertigen Property-Attribute `toaddr`, `fromaddr`, `message` und `data` lesend und schreibend verwendet werden. Die zusätzliche Implementierung als Methoden ist notwendig, da Zope aus Sicherheitsgründen das direkte Manipulieren von Attributen eines Objektes aus Python-`Script`-Objekten heraus verbietet.

`copy()`

Liefert eine Kopie des aktuellen Message-Objektes zurück.

Interne Methoden

`asLogString()`
`__str__()`

Liefert eine String-Repräsentation der Message zur kompakten Ausgabe in Log-Files aus.

4.3.5 Vordefinierte Components

Das TaskEngine-Framework enthält eine Reihe vordefinierter Komponenten, die größtenteils HTML-Formularelemente abbilden und zur Erstellung komplexerer Komponenten verwendet werden können. Diese vordefinierten Components beziehen viele Eigenschaften von Basisklassen die im folgenden Abschnitt zusammengefasst sind.

4.3.5.1 Basisklassen

Die nachfolgenden Klassen können nicht direkt verwendet werden sondern werden von anderen Klassen als Basisklassen oder Mix-Ins verwendet.

class InputComponent

Diese Klasse wird als Basisklasse für sämtliche Komponenten verwendet, die direkt als HTML-Elemente darstellbar sind. Sie erbt wiederum von den Klassen `ClassComponent`, `CallbackComponent`, und `AttributeElement`.

`__init__(`*focus*`=0)`

Das Argument *focus*, das maximal bei einem einzigen Element einer Ausgabeseite gesetzt werden sollte, legt fest, dass dieses Element direkt nach dem Laden der Seite den Eingabefokus erhalten sollte.

`focus()`
`focus(`*value*`=0)`

Liefert zurück (erste Form), beziehungsweise legt fest (zweite Form), ob das aktuelle Element direkt nach dem Laden der Seite den Eingabefokus erhalten sollte. Nur ein einziges Element einer Ausgabeseite sollte den Fokus für sich beanspruchen.

`__unicode__()`

Liefert die aktuelle Komponente in für die Ausgabe passender Form als `unicode`-String zurück.

`paint()`

Veranlasst die Ausgabe der aktuellen Komponente. (Schreibt die `unicode`-String-Repräsentation der aktuellen Komponente in den Ausgabepuffer.)

class CallbackComponent

Diese Mix-In-Klasse wird von Komponenten verwendet, die das Versenden von („Callback“-) Messages beim Eintreten bestimmter Ereignisse ermöglichen sollen.

Beispielsweise bietet die `TextField`-Komponent an, sich für das Ereignis `change` zu registrieren. Bei Änderung des Inhaltes des Feldes wird dann automatisch die entsprechende Message versandt.

`__init__(`*`callbacks`*`=())`

Bei der Initialisierung kann eine Liste von Callbacks angegeben werden, deren Format den Optionen der Methode `setCallback()` entspricht. Die einzelnen Callbacks können dabei entweder eine Liste der Positions-Argumente oder ein Dictionary mit Keyword-Argumenten von `setCallback()` sein.

`setCallback(`*`event`*`, `*`toaddr`*`, `*`message`*`, `*`type`*`='post', `*`args`*`=())`

Registriert die CallBack-Message *message* für den Event *event*. *toaddr* gibt die Adresse an, an die die Message beim Eintreten des Events versandt werden soll. *message* ist der Name der zu sendenden Message, *type* spezifiziert die Art der Message-Zustellung und kann die Werte `send` oder `post` besitzen. (Siehe `postMessage()` und `sendMessage()` der Klasse `Component`, 4.3.4.5.) *args* ist eine Liste zusätzlicher Argumente, die als zusätzliche Daten in der Message versandt werden.

`processCallback(`*`event, *args`*`)`

Versendet die Callback-Message, die für den Event *event* registriert ist. Als erstes Argument der Message-Daten wird implizit das aktuelle Component-Objekt eingefügt. Falls als Art der Message-Zustellung `send` gewählt wurde, wird die Nachricht synchron zugestellt und `processCallback()` liefert den Rückgabewert des Message-Handlers zurück. Falls für *event* kein Callback registriert ist, werden keine Aktionen durchgeführt.

class AttributeElement

Diese Klasse ist ein Mix-In für Elemente, die HTML-Attribute enthalten können. Wichtige Attribute sind beispielsweise:

`id`	Kennzeichnet ein Element eindeutig.
`class`	Gibt die Stylesheet-Klassen des Elementes an. Mehrere Klassen werden durch Leerzeichen getrennt. Zur Manipulation dieses Attributes siehe auch die Methoden `classAttr()` und `addClassAttr()`.
`accesskey`	Definiert einen Hotkey für das zugehörige Element.
`readonly`	Das Element wird im Browser als schreibgeschützt dargestellt. Der Wert des Attributes kann `0` oder `1` sein.
`disabled`	Das Element wird im Browser inaktiv dargestellt. Der Wert des Attributes kann `0` oder `1` sein.

Attribute, die boolsche Werte enthalten, sollten über die Methode `updateAttr()` beziehungsweise über die dem Attribut zugehörigen Methoden (zum Beispiel `readonly()` oder `disabled()`) gesetzt werden, damit etwa XHTML-kompatible Ausgabe ermöglicht wird.

Attribute, deren Namen mit Unterstrich beginnen, werden zwar von den Methoden der Klasse verwaltet, scheinen allerdings nicht in der Text-Repräsentation des Objektes (als bei der Ausgabe) auf. Dies kann verwendet werden, um benutzerdefinierte Daten beim Element abzuspeichern.

`__init__(`*`attrdict={}, ...`*`)`

Bei der Initialisierung werden ein oder mehrere Dictionaries mit den initialen Attributen und deren Werten übergeben. Gleichnamige Attribute in späteren Dictionaries überschreiben eventuell zuvor

gesetzte Werte. Dadurch ist es möglich, in einem ersten Dictionary Default-Attribute vorzugeben.

`updateAttr(`*`attrs, **kwattrs`*`)`

Entspricht der Python Methode `update()`, aktualisiert also die Attribute anhand der Liste an Attribut-Wert-Tupeln *attrs*, oder anhand der Keyword-Argumente *kwattrs*. Diese Methode sorgt auch dafür, dass boolsche Attribute korrekt behandelt werden.

`getAttribute(`*`attr, default`*`)`

Liefert den Wert des Attributes *attr* zurück. Falls das Attribut *attr* nicht existiert, wird *default* zurückgeliefert beziehungsweise eine `KeyError`-Exception ausgelöst, falls kein Wert für *default* angegeben wurde.

`attribute(`*`attr`*`)`
`attribute(`*`attr, value`*`)`

Liefert den Wert des Attributes *attr* zurück (erste Form) oder setzt diesen auf *value* (zweite Form). Falls das Attribut *attr* nicht existiert, wird None zurückgeliefert.

`classAttr()`
`classAttr(`*`class, ...`*`)`

Liefert den Wert des Attributes `class` zurück oder setzt dieses auf die Klasse(n), die als Argumente übergeben werden.

`addClassAttr(`*`class, ...`*`)`

Ergänzt die im `class`-Attribut gespeicherte Liste der Klassen um die als Argument übergebenen Klasse(n).

`readonly()`
`readonly(`*`value`*`)`

Liefert den Wert des `readonly`-Attributes zurück (erste Form) oder setzt diesen auf den Wert *value* (zweite Form) (`0` oder `1`).

`disabled()`
`disabled(`*`value`*`)`

Liefert den Wert des `disabled`-Attributes zurück (erste Form) oder setzt diesen auf den Wert *value* (zweite Form) (`0` oder `1`).

`delAttr(`*`attr`*`)`

Entfernt das Attribut *attr*, falls vorhanden.

`addAttrIfMissing(`*`attr`*`, `*`value`*`=None)`

Setzt das Attribut *attr* auf den Wert *value*, falls das Attribut nicht bereits vorhanden ist.

`addBoolAttr(`*`battr`*`, ...)`

Definiert die als Argumente angegebenen Attributsnamen als boolsche Attribute. Bereinigt auch die Werte eventuell bereits vorhandener Attribute mit diesen Namen.

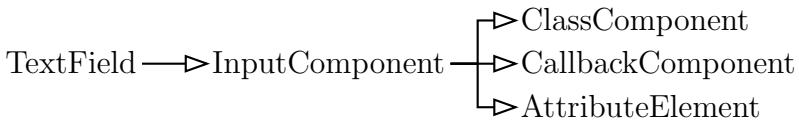
`attr2txt(`*`attrs`*`=None)`

Wandelt die durch eine Liste von Attributnamen (*attrs*) angegebenen Attribute in einen String um, der direkt im HTML-Element verwendet werden kann.

class ClassComponent

`ClassComponent` ist eine Mix-In Klasse für Modul-basierte Components, die im Filesystem und nicht in der ZODB implementiert sind. Sie ermöglicht die Verwendung von Message-Handlern als einfache Methoden der entsprechenden Component-Klasse, die sich durch ein vorgestelltes `msg` auszeichnen. (So definiert beispielsweise die Methode `msgpaint` den Message-Handler für die `paint`-Message.)1

4.3.5.2 TextField



Die Klasse `TextField` erzeugt HTML-Formularelemente der Art `INPUT TYPE="text"` oder `INPUT TYPE="password"`.

```
__init__(type='text'|'password', text='',
         size=None, selectOnFocus=0,
         focus=0,      # InputComponent
         callbacks=(), # CallbackComponent
         attributes={} # AttributeElement
        )
```

Erzeugt eine neue `TextField`-Instanz. Das Argument *type* gibt an, welchen Typ das INPUT Feld haben soll. Das Feld kann optional mit *text* vorausgefüllt werden. *size* gibt die Größe des Formularfeldes an, mit *selectOnFocus* kann der Text automatisch markiert werden, sobald das Feld edn Eingabefokus erhält. (Siehe gleichnamige Methode.)

Die Parameter `focus`, `callbacks` und `attributes` sind bei den entsprechenden Basisklassen beschrieben.

Der Konstruktor dieser Klasse wird typischerweise über die Methode `addComponent()` einer Komponente aufgerufen. (Siehe Seite 75.)

`getText()`

Liefert den aktuellen Text des Feldes zurück.

`setText(text)`

Setzt den Text des aktuellen Feldes auf den `unicode`-String *text*.

Anstelle der obigen Methoden kann auch das gleichwertige Property-Attribut `text` verwendet werden.

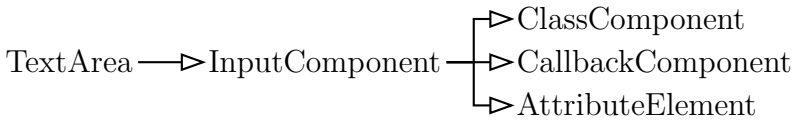
`selectOnFocus(value=None)`

Bewirkt, dass der Inhalt des Textfeldes markiert wird, sobald das Feld den Eingabefokus bekommt. Ist der Wert *value* auf 2 gesetzt, wird der Text jedes Mal markiert, wenn das Feld den Eingabefokus bekommt. Ist $value = 1$, wird der Inhalt des Feldes nur beim ersten Mal markiert. Bei anderen Werten für *value* wird das automatische Markieren ausgeschaltet.

Callback-Events

`change` Dieses Ereignis wird ausgelöst, wenn sich der Inhalt des Eingabefeldes verändert.

4.3.5.3 TextArea



Diese Klasse erzeugt ein gleichnamiges HTML-Formularfeld.

```
__init__(text='', cols=40, rows='40',
         selectOnFocus=0,
         focus=0,      # InputComponent
         callbacks=(), # CallbackComponent
         attributes={} # AttributeElement
        )
```

Erzeugt eine neue `TextArea`-Instanz. Das Feld kann optional mit *text* vorausgefüllt werden. über *cols* und *rows* kann die Größe des Testbereiches über die Anzahl an Spalten beziehungsweise Zeilen festgelegt werden. Mit *selectOnFocus* kann der Text automatisch markiert werden, sobald das Feld edn Eingabefokus erhält. (Siehe gleichnamige Methode.)

Die Parameter `focus`, `callbacks` und `attributes` sind bei den entsprechenden Basisklassen beschrieben.

Der Konstruktor dieser Klasse wird typischerweise über die Methode `addComponent()` einer Komponente aufgerufen. (Siehe Seite 75.)

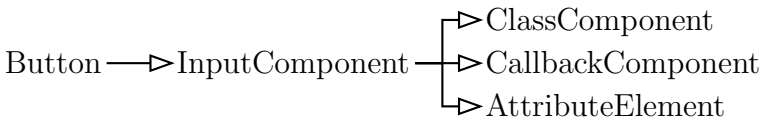
`selectOnFocus(`*value*`=None)`

Bewirkt, dass der Inhalt des Textfeldes markiert wird, sobald das Feld den Eingabefokus bekommt. Ist der Wert *value* auf 2 gesetzt, wird der Text jedes Mal markiert, wenn das Feld den Eingabefokus bekommt. Ist $value = 1$, wird der Inhalt des Feldes nur beim ersten Mal markiert. Bei anderen Werten für *value* wird das automatische Markieren ausgeschaltet.

Callback-Events

`change` Dieses Ereignis wird ausgelöst, wenn sich der Inhalt des Eingabefeldes verändert.

4.3.5.4 Button



Diese Klasse generiert `<INPUT TYPE="submit" ...>` Formularelemente.

Es sind auch Elemente vom Typ `INPUT TYPE="button"` möglich, die Reaktion auf einen Button-Klick muss in diesem Fall allerdings mit JavaScript ausprogrammiert werden. (Natürlich funktioniert wird auch das unten beschriebene Callback-Event `click` dann nicht ausgeführt.)

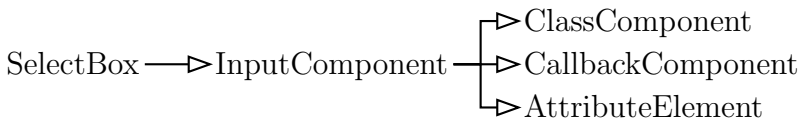
```
__init__(focus=0,       # InputComponent
         callbacks=(), # CallbackComponent
         attributes={} # AttributeElement
        )
```

Erzeugt einen neuen Button. Sämtliche Argumente werden von den Basisklassen geerbt sind daher in den entsprechenden Abschnitten beschrieben.

Callback-Events

`click` Dieses Ereignis wird ausgelöst, wenn der Button gedrückt wurde.

4.3.5.5 SelectBox



`SelectBox` erzeugt HTML-Auswahllisten vom Typ `<SELECT ...>`.

```
__init__(multiple=0, size=1, options=(),
         focus=0,      # InputComponent
         callbacks=(), # CallbackComponent
         attributes={} # AttributeElement
        )
```

Erzeugt eine Klappliste. Über den Parameter *multiple* kann angegeben werden, ob eine Mehrfachauswahl zulässig ist. *size* gibt die Anzahl der Listeneinträge an, die gleichzeitig angezeigt werden. Standardwert dafür ist `1` beziehungsweise bei `3`, falls *multiple* = 1. Über `options` wird der anfängliche Inhalt der Auswahlliste festgelegt, die Einträge werden im selben Format akzeptiert, wie in `setOptions()` beschrieben.

Die Parameter `focus`, `callbacks` und `attributes` sind bei den entsprechenden Basisklassen beschrieben.

Der Konstruktor dieser Klasse wird typischerweise über die Methode `addComponent()` einer Komponente aufgerufen. (Siehe Seite 75.)

`setOptions(`*options*`)`

Setzt den Inhalt der Auswahlliste auf *options*. *options* muss eine Liste von Elementen sein, wobei jedes Element ein Tupel vom Format `(`*text*`, { `*attribute*`=`*value*`, ... })` ist. Die optionalen Attribute sind HTML-Attribute, die den einzelnen `<OPTION ...>`-Elementen der Auswahlliste zugeordnet werden. (Siehe auch Klasse `AttributeElement` ab Seite 83.) Ein wichtiges Attribut ist `selected`, mit dem angegeben werden kann, ob das entsprechende Element vorselektiert ist. Dieses Attribut wird von der Klasse `SelectBox` auch speziell ausgewertet.

`getOptions()`

Liefert die aktuellen Elemente der Auswahlliste zurück. Das Ergebnis ist vom Typ `OptionElementList`. (Siehe unten.)

`getSelectedOptions()`

Liefert die ausgewählten Optionen als Liste zurück. Die Listenelemente stammen von `AttributeElement` ab, ihre Attribute sind daher über die Methoden `getAttribute()` beziehungsweise

`attribute()` verfügbar. Diese Methode entspricht der gleichnamigen Methode der Klasse `OptionElementList`.

Callback-Events

`change` Dieses Ereignis wird ausgelöst, wenn sich die Auswahl ändert. Die Daten der übermittelten Message enthalten nach dem Component-Objekt auch noch die Listenindizes der ausgewählten Optionen in Form eines Python-`set`. Das Ereignis wird nur ausgelöst, wenn die Auswahl vom Benutzer verändert wurde, bei Änderungen durch Methoden der Klasse `OptionElememtList`, wie `checked(`*`index, value`*`)` wird kein `change`-Event generiert.

class OptionElementList

Die von der `SelectBox` verwendete Elementliste ist eine Klasse, die vom Python `list`-Typ abstammt, jedoch einige zusätzliche Methoden anbietet. Im Folgenden sind nur die Methoden beschrieben, die die grundsätzliche Funktionalität der `list`-Klasse erweitern.

`__init__(`*`options`*`=(), `*`itemclass`*`=OptionElement)`

Initialisiert eine neue Liste. Wird üblicherweise nicht direkt, sondern über die Methode `setOptions()` oder den Konstruktor der Klasse `SelectBox` aufgerufen.

`getSelected()`

Liefert die Indzies der ausgewählten Listenelemente als Liste zurück.

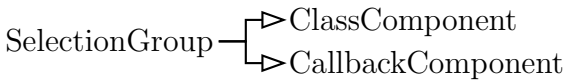
`getSelectedOptions()`

Liefert die ausgewählten Optionen als Liste zurück. Die Listenelemente stammen von `AttributeElement` ab, ihre Attribute sind daher über die Methoden `getAttribute()` beziehungsweise `attribute()` verfügbar.

`selected(`*`index`*`)`
`selected(`*`index`*`, `*`value`*`)`

Die erste Form liefert zurück, ob das Element mit dem Index *index* ausgewählt ist. Mit der zweiten Form kann das Element *index* selektiert ($value = 1$) beziehungsweise deselektiert ($value = 0$) werden.

4.3.5.6 SelectionGroup



Diese Klasse erzeugt eine Gruppe von `<INPUT TYPE="checkbox" ...>` oder `<INPUT TYPE="radio" ...>` Elementen.

```
__init__(itemtype, multiple=0, items=None, defaultattribs={},
         callbacks=(), # CallbackComponent
         )
```

Erzeugt eine neue SelectionGroup. *itemtype* kann die Werte `'checkbox'` oder `'radio'` besitzen und definiert den Typ der zu erzeugenden HTML-Formularelemente. Wird *itemtype* nicht angegeben, ist der Standardwert `'checkbox'`. Über den Parameter *multiple* kann angegeben werden, ob eine Mehrfachauswahl zulässig ist. Über *items* werden die Elemente der Gruppe im selben Format festgelegt, wie sie an `setItems()` übergeben werden. *defaultattribs* kann Standardattribute enthalten, die jedes neu angelegte Element enthalten soll.

Der Parameter `callbacks` ist bei der Beschreibung der Basisklasse `CallbackComponent` beschrieben.

Der Konstruktor dieser Klasse wird typischerweise über die Methode `addComponent()` einer Komponente aufgerufen. (Siehe Seite 75.)

`newItem(`*`attribs`*`=)`

Liefert ein neues Element zurück. Das Ergebnis ist vom Typ `SelectionElement` und muss erst mit `addItem()` in die `SelectionGroup` integriert werden.

attribs sind HTML-Attribute, die dem Element zugeordnet werden. (Siehe auch Klasse `AttributeElement` ab Seite 83.) Ein wichtiges und spezielles Attribut ist `checked`, mit dem angegeben werden kann, ob das entsprechende Element ausgewählt ist. Wie in der Dokumentation vom `AttributeElement` beschrieben, können Attributnamen, die mit einem Unterstrich beginnen, zum Abspeichern benutzerdefinierter Daten im Element benutzt werden.

`addItem(`*`SelectionElement`*`)`
`addItem(`*`attribs`*`=)`

Fügt der `SelectionGroup` ein neues Element hinzu. Als Argument kann entweder ein Objekt vom Typ `SelectionElement` oder ein Dictionary von Attributen des Elements übergeben werden. (Siehe `newItem()` für Details.)

`setItems(`*`items`*`=())`

Setzt die Elemente der `SelectionGroup` auf *items*. *items* ist eine Liste aus Objekten vom Typ `SelectionElement` oder eine Liste von Attribut-Dictionaries. (Sie an `newItem()` für Details.)

`getItems()`

Liefert die Liste der `SelectionGroup`-Elemente zurück. Die einzelnen Elemente sind Objekte vom Typ `SelectionElement`.

`checked()`
`checked(`*`index`*`)`
`checked(`*`index, value`*`)`

Die erste Form liefert eine Liste der Indizes der Elemente zurück, die ausgewählt sind. Die zweite Form liefert zurück, ob das Element mit dem Index *index* ausgewählt ist. Mit der dritten Form kann das Element *index* selektiert ($value = 1$) beziehungsweise deselektiert ($value = 0$) werden.

`checkeditems()`

Liefert die Liste der ausgewählten Elemente zurück. (Die Elemente sind vom Typ `SelectionElement`.)

`readonly()`
`readonly(`*`value`*`)`

Die erste Form liefert zurück, ob der Status der aktuellen Gruppe auf Nur-Lesen gesetzt ist.

Die zweite Form setzt alle Elemente der Gruppe auf Read-Only ($value = 1$) oder entfernt das `readonly`-Attribut von den Elementen der Gruppe ($value = 0$).

`disabled()`
`disabled(`*`value`*`)`

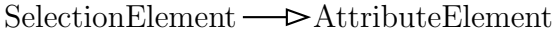
Die erste Form liefert zurück, ob die aktuelle Gruppe deaktiviert ist. Die meisten Browser zeigen die Elemente dann inaktiv (grau) an.

Die zweite Form setzt alle Elemente der Gruppe auf deaktiviert ($value = 1$) oder entfernt das `disabled`-Attribut von den Elementen der Gruppe ($value = 0$).

Callback-Events

`change` Dieses Ereignis wird ausgelöst, wenn sich die Auswahl ändert. Die Daten der übermittelten Message enthalten nach dem Component-Objekt auch noch die Listenindizes der ausgewählten Optionen in Form eines Python-`set`. Das Ereignis wird nur ausgelöst, wenn die Auswahl vom Benutzer verändert wurde, bei Änderungen durch Methoden wie `checked(`*`index, value`*`)` wird kein `change`-Event generiert.

class SelectionElement



Ein `SelectionElement` repräsentiert ein einzelnes Element vom Typ `<INPUT TYPE="checkbox" ...>` oder `<INPUT TYPE="radio" ...>`. Der Konstruktor dieser Klasse wird nicht direkt aufgerufen, Elemente werden über die Methoden `newItem()` oder `addItem()` eines `SelectionGroup`-Objektes erzeugt.

`checked()`
`checked(`*value*`)`

Die erste Form liefert zurück, ob das aktuelle Element ausgewählt ist. Mit der zweiten Form kann das Element selektiert ($value = 1$) beziehungsweise deselektiert ($value = 0$) werden.

`__unicode__()`

Liefert das aktuelle Element in für die Ausgabe passender Form als `unicode`-String zurück.

`paint()`

Veranlasst die Ausgabe des aktuellen Elements. (Schreibt die `unicode`-String-Repräsentation des Elements in den Ausgabepuffer.)

4.3.6 Beispiel-Tasks

Um die Möglichkeiten des TaskEngine-Frameworks demonstrieren zu können und um die Implementierung der Komponenten und Tasks testen zu können, wurden einige Beispiel-Tasks und -Komponenten entwickelt, die auch als Vorlagen und Startpunkte für Eigenentwicklungen dienen sollen.

4.3.6.1 WindowManager

Dieser Task ist die eine Beispielimplementierung eines Window-Manager Tasks, wie er in Abschnitt 4.3.2.5 beschrieben ist.

Diese Implementierung erlaubt das Starten und Beenden von Tasks und liefert das Grundgerüst für die Programmierung eine Web-Anwendung mit dem TaskEngine-Framework. Mehrere Tasks (und damit natürlich auch mehrere Instanzen desselben Tasks) können gleichzeitig ausgeführt werden, es wird allerdings immer nur ein Task im Vollbildmodus angezeigt. Dieser anzuzeigende

„Vordergrundtask“ kann mittels Klappliste aus den aktiven Tasks ausgewählt werden.

4.3.6.2 TestTask

Dieser Task dient dazu, die Funktion des WindowManagers und der vordefinierten Components zu demonstrieren und zu testen. Es können beliebig viele Instanzen der angebotenen Components generiert und schließlich selektiv deaktiviert, schreibgeschützt oder wieder entfernt werden. Desweiteren wird auch der Notifikationsmechanismus der Components getestet, ausgelöste Ereignisse werden angezeigt.

4.3.6.3 Calculator

Bei diesem Task handelt es sich um die Realisierung eines einfachen Taschenrechners.

4.4 Datenbankmodule

Da die Aufgabenstellung eng mit komplexen Datenbankabfragen verknüpft ist, war es auch ein Ziel, den Zugriff auf die Datenbank möglichst allgemein und implementierungsunabhängig zu gestalten. Ein weiteres Motiv für die Einführung von Abstraktionsschichten zum Datenbankzugriff war außerdem die übersichtlichere Gestaltung der Programmlogik, die sich, von der Abfragesprache SQL abstrahiert, auf die konkrete Aufgabenstellung konzentrieren sollte, ohne auf die syntaktische Formulierung der Abfragestatements Rücksicht nehmen zu müssen.

4.4.1 TableJoins - dynamisch generierte SQL-Abfragen

Dieses Modul wurde zur automatischen Formulierung komplexer SQL-Abfragen aufgrund logischer Abfragekritierien entwickelt.

Ausgehende Problemstellung war die Schwierigkeit, für wechselnde Suchbedingungen dynamische SQL-Abfragen zu generieren, vor allem, wenn die gesuchten Tabellenspalten aus mehreren Tabellen kombiniert werden können, die

direkt oder auch indirekt (über Zwischentabellen) durch Fremdschlüsselbeziehungen miteinander verknüpft sind. Je mehr Tabellen an der Suche beteiligt sind, desto aufwendiger wird der Suchvorgang, daher sollten nur jene Tabellen in eine Suchabfrage miteinbezogen werden, die auch wirklich für die aktuelle Fragestellung benötigt werden.

Strategie des Moduls ist es, ausgehend von einer Basistabelle, die die benötigten Ergebnisspalten beinhaltet, für jedes aktive Suchkriterium über die benötigten Tabellenverknüpfungen (*Table-Joins*) einen Weg in der Datenbankstruktur aufzuspannen, der in Form einer Baumstruktur im `TableJoins`-Objekt abgelegt wird. Am Ende dieses Weges wird schließlich die Filterregel eingetragen.

Auf diese Art werden sämtliche Suchbedingungen abgearbeitet, wobei bereits bestehende Wege genutzt werden können. Dabei ist zu beachten, dass die genaue Verknüpfungsbeziehung für die Knoten und Verbindungen der aufgespannten Baumstruktur entscheidend ist. Wenn also zwischen zwei Tabellen unterschiedliche Verknüpfungsbeziehungen benötigt werden (wenn zum Beispiel der Table-Join über andere Spalten erfolgt), so werden dafür zwei verschiedene Wege mit zwei verschiedenen Zielknoten im Baum eingetragen.

So so ist beispielsweise die Tabelle `PUBLIKATION` aus Abbildung 4.5 über zwei direkte Wege mit `PERSON` verknüpft: über die Fremdschlüsselfelder `created_at` und `changed_at`. Werden beide Kriterien für eine Suche verwendet, entsteht ein Baum mit zwei Knoten für die Tabelle `PERSON`, also einen eigenen für jede Fremdschlüsselbeziehung.

Eine weitere Grundidee bei der Entwicklung des TableJoins-Moduls war auch die konsequente Verwendung von *Bind-Parametern* in SQL-Statements. Abfragen, die mit Hilfe dieser Technik formuliert sind, enthalten Platzhalter für Variablen, die erst vor dem Absetzen der eigentlichen Suche mit konkreten Werten befüllt werden.

Diese Technik ist vor allem für Web-Anwendungen interessant, da die Verwendung von Platzhaltern in SQL-Statements verhindert, dass über Benutzereingaben, die als variable Teile in SQL-Statements eingefügt werden und unzureichend gequotet wurden, Datenbankbefehle eingeschleust werden. Diese Art von Sicherheitslücke, die als *SQL-Injection* bezeichnet wird, kann durch durchgängige Verwendung von Bind-Parametern gänzlich verhindert werden. Je nach verwendeter Datenbank werden nämlich die einzusetzenden Parameter entweder direkt als Parameter an die Datenbank übergeben (wodurch sie gar nicht als Teil einer SQL-Anweisung interpretiert werden) oder sie werden vom

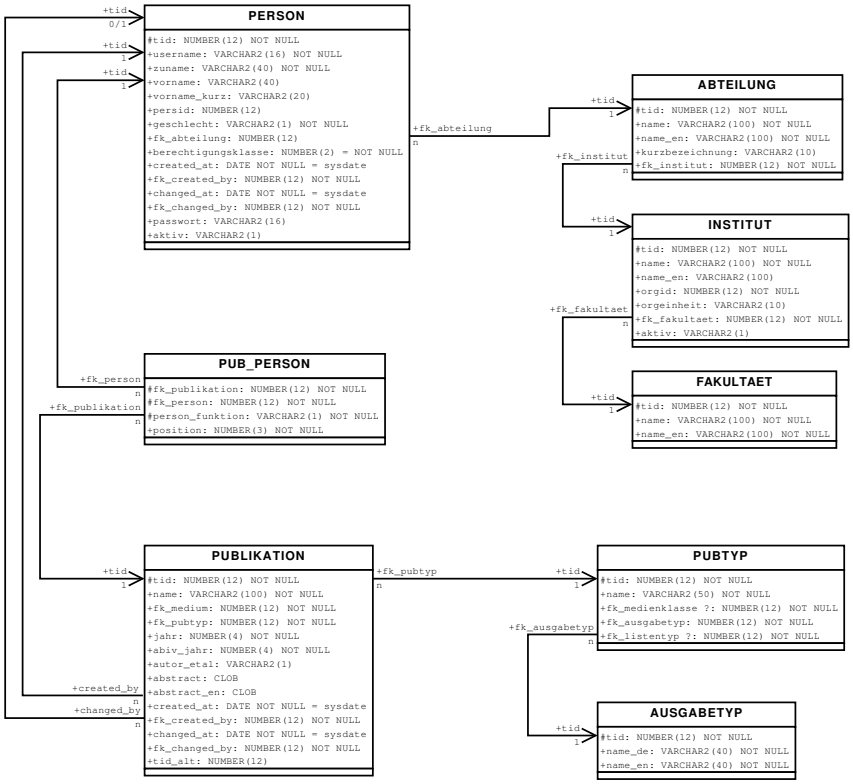


Abbildung 4.5: Ausschnitt aus einem Datenmodell

Datenbank-API auf optimale Art „escaped“. In beiden Fällen entfällt für den Programmierer die fehleranfällige Arbeit, die Eingabedaten zu überprüfen und gegebenenfalls selbst zu entschärfen.

Ein weiterer Vorteil ist, dass die SQL-Anweisung schon mit Platzhaltern, also ohne konkrete, suchspezifische Werte, compiliert und optimiert wird.[11] Dies wirkt sich vor allem, bei vielen gleichartigen Suchabfragen oder Datenmanipulationsanweisungen aus, die zum Beispiel in einer Schleife, ausgeführt werden. Das Statement wird nur einmal compiliert, und durchläuft nur einmal die Abfrageoptimierung, was signifikante Geschwindigkeitsgewinne bedeuten kann. Auch bei vielen gleichartigen Suchabfragen, die über unterschiedliche Datenbankverbindungen abgesetzt werden, ist oft ein Performance-Vorteil festzustellen, da die meisten Datenbanken über einen sogenannten *Query-Cache* verfügen, in dem SQL-Befehle gemeinsam mit ihren optimierten und vorkompilierten Versionen für den schnelleren Zugriff vorbereitet sind. In diesem Zusammenhang wirkt sich auch positiv aus, dass eine sogenannte *Cache-Pollution*, also eine Überfüllung des Cache-Speichers mit vielen individuellen Suchabfragen, die konkrete Abfragewerte enthalten und damit nur selten in der selben Form wieder benötigt werden, vermieden wird.

Bind-Parameter sind bereits bei den Datenbank-APIs der meisten Programmiersprachen vorgesehen und werden auch bereits von einer großen Anzahl an Datenbanken nativ unterstützt. Die vorliegende Implementierung wurde für die Verwendung mit Oracle in Python entwickelt, es existieren jedoch auch für Programmiersprachen wie PHP oder Perl und Datenbanken wie MySQL oder PostgreSQL Datenbank-Schnittstellen, die die selben oder einer sehr ähnlichen Syntax anzusprechen sind.[12]

4.4.1.1 Funktionsweise

Daten zu einer Abfrage, das heißt Informationen über Tabellen, die Beziehungen zwischen ihnen sowie über Abfragefilter werden in einem Objekt vom Typ `TableJoins` abgespeichert. Dieses Objekt beinhaltet anfänglich eine leere Liste an Tabellenbäumen beinhaltet. Mit jedem Suchkriterium das dem Objekt mit `addPath()` hinzugefügt wird, werden die Baumstrukturen erweitert. Eine solche Verknüpfung der Suchergebnisse entspricht einer Und-Verknüpfung der

[11] Dies setzt voraus, dass die darunterliegende Datenbank Bind-Parameter unterstützt.

[12] Für die Programmiersprache PHP stehen etwa die Datenbankabstraktionsschichten *PHP Data Objects (PDO)* oder *PEAR-DB*, beziehungsweise *PEAR-MDB2* zur Verfügung. Die PHP-Erweiterung *mysqli* ist sogar Bestandteil der meisten PHP-Pakete ab Version 5.

Suchkriterien. Abbildung 4.6 demonstriert die Verwendung dieser Methode, Abbildung 4.7 illustriert das Ergebnis.

Beispiel:

```
tj = TableJoins()

tj.addPath(('publikation', ),
           selectcols=('$(table).tid as tid', ))

tj.addPath(('publikation', ),
           ('$(table).name LIKE ?', 'Publikationstitel%'))
tj.addPath(('publikation', ('tid', 'fk_publikation'),
           'pub_person'),
           ("$(table).person_funktion = 'A'", ))
tj.addPath(('publikation', ('tid', 'fk_publikation'),
            'pub_person', ('fk_person', 'tid'),
            'person'),
           ('$(table).nachname LIKE ?', 'Berg%'))
tj.addPath(('publikation', ('tid', 'fk_publikation'),
            'pub_person', ('fk_person', 'tid'),
            'person', ('fk_abteilung', 'tid'),
            'abteilung'),
           ('$(table).bezeichnung = ?', 'Institut X'))
```

Abbildung 4.6: Und-Verknüpfung über `addPath()`

Darüberhinaus können auch alternative Suchpfade über die Methode `addPathsOr()` der aktuellen Abfrage hinzugefügt werden. Das ist zum Beispiel sinnvoll, wenn ein Suchfeld in der Eingabemaske nicht direkt einem Datenbankfeld entspricht, und die resultierende Abfrage daher mehrere Möglichkeiten berücksichtigen soll. Abbildung 4.8 zeigt die Erstellung eine solchen Abfrage, die nach Datensätzen sucht, die von einer bestimmten Person geändert oder erzeugt wurden. Die vom Modul daraus erzeugte Abfrage ist in Abbildung 4.9 dargestellt.

4.4.1.2 class TableJoins

`__init__()`

Instanziert ein neues TableJoins Objekt, das eine SQL-Abfrage repräsentiert.

Beispiel:

```
# tj.trees:
publikation: [('$(table).name LIKE ?', 'Publikationstitel%')]
    pub_person (tid->fk_publikation): [("$(table).person_funktion = 'A'",)]
        person (fk_person->tid): [('$(table).nachname LIKE ?', 'Berg%')]
            abteilung (fk_abteilung->tid): [('$(table).bezeichnung = ?',
                                             'Institut X')]

# tj.statement():
SELECT publikation.tid as tid
FROM publikation, pub_person, person, abteilung
WHERE ( publikation.name LIKE :1 )
 AND publikation.tid = pub_person.fk_publikation
 AND ( pub_person.person_funktion = 'A' )
 AND pub_person.fk_person = person.tid
 AND ( person.nachname LIKE :2 )
 AND person.fk_abteilung = abteilung.tid
 AND ( abteilung.bezeichnung = :3 )

# ts.params:
['Publikationstitel%', 'Berg%', 'Institut X']
```

Abbildung 4.7: Aus Abbildung 4.6 resultierende Abfrage

Beispiel:

```
tj = TableJoins()

tj.addPath(('publikation', ),
           selectcols=('$(table).tid as tid', ))

tj.addPathsOr((
  (('publikation', ('changed_by', 'tid'), 'person'), # path1
   ('$(table).name LIKE ?', 'Berg%') # filter
  ),
  (('publikation', ('created_by', 'tid'),  'person'), # path2
   ('$(table).name LIKE ?', 'Berg%') # filter
  ),
))
```

Abbildung 4.8: Oder-Verknüpfung über `addPathsOr()`

Beispiel:

```
# tj.trees:
publikation
    person (changed_by->tid): [('$(table).name LIKE ?', 'Berg%')]
OR
publikation
    person (created_by->tid): [('$(table).name LIKE ?', 'Berg%')]

# tj.statement():
SELECT publikation.tid as tid
FROM publikation, person
WHERE publikation.changed_by = person.tid
 AND ( person.name LIKE :1 )
UNION
SELECT publikation.tid as tid
FROM publikation, person
WHERE publikation.created_by = person.tid
 AND ( person.name LIKE :2 )

# ts.params:
['Berg%', 'Berg%']
```

Abbildung 4.9: Aus Abbildung 4.8 resultierende Abfrage

`addPath(`*path*`, `*filter*`=None, `*selectcols*`=None)`

Fügt der Abfrage ein zusätzliches Filterkriterium hinzu. *path* ist der Pfad im Datenmodell, über den die Tabellen, ausgehend von einer zentralen „Ergebnistabelle“, miteinander verknüpft werden sollen. Dabei muss *path* eine Liste sein, die alternierend Tabellennamen und Verknüpfungsbeziehungen zwischen den Tabellen, die verknüpft werden, enthält. Die Verknüpfungsbeziehungen haben dabei die Form von Tupeln der jeweils miteinandere Verknüpften Spaltennamen:

```
path = (table1, (col1x, col2x),
        table2, (col2y, col3x), ...,
        tableN)
```

filter gibt die Filterbedingung an, die auf die letzte Tabelle des Pfades (in obigen Beispiel *tableN*) angewandt werden soll. *filter* muss eine Liste sein, die aus der SQL-Bedingung (in derselben Syntax,

wie sie in `WHERE`-Klauseln der Abfrage vorkommen kann) und eventuellen Werten für Bind-Parameter[13] besteht:

```
filter = (condition, placeholder1, ...)
```

Das Argument *selectcols* ermöglicht die Angabe von Spalten, die in der Abfrage selektiert werden sollen. Es handelt sich um eine Liste von Spaltenbezeichnungen (möglicherweise mit SQL-Funktionen verknüpft) in der selben Syntax, wie sie im `SELECT`-Teil eines SQL-Statements vorkommen können:

```
selectcols = (column, ...)
```

Falls eine Tabelle in der Abfrage mehrfach vorkommt, muss das Modul sogenannte Tabellen-Aliases vergeben, um eine bestimmte Instanz der Tabelle eindeutig referenzieren zu können. Da diese Alias-Werte automatisch vergeben werden, steht der symbolische Platzhalter `$(table)` in den Parametern *filter* und *selectcols* zur Verfügung, der vom `TableJoins`-Modul automatisch durch den entsprechenden Tabellennamen oder -Alias ersetzt wird.

Die Abbildungen 4.6 und 4.8 enthalten Beispiele für die Verwendung von `addPath()`.

`addPathsOr(`*paths_and_filters*`)`

Diese Methode fügt der Abfrage alternative Suchpfade hinzu. *paths_and_filters* ist eine Liste von Pfad-Filter-Tupeln, die die diskunktiv (mittels logischer ODER-Verknüpfung) kombiniert werden sollen:

```
paths_and_filters = ((path, filter), ...)
```

path und *filter* müssen im selben Format angegeben werden, wie bereits unter `addPath()` beschrieben wurde.

Abbildung 4.8 zeigt die Verwendung dieser Methode.

[13] falls die Bedingung Platzhalter enthält

`statement()`

Liefert ein `TableStatement`-Objekt zurück, das eine fertige Abfrage repräsentiert. Über die `__str__`-Methode dieses Objekt wird das SQL-Statement ausgegeben, das Attribut `params` enthält eine Liste der Bind-Parameter, die bei der Ausführung des Statements angegeben werden müssen, um eventuell vorhandene Platzhalter aufzufüllen.

4.4.1.3 class Placeholders

Die Klasse `TableJoins` verwendet diese Klasse intern, um Platzhalter für Bind-Variablen, die in den Suchbedingungen datenbankunabhängig mit `?` markiert werden, bei der Ausgabe des SQL-Statements in das von Oracle verwendete Format (`:1`, `:2`, usw.) zu übersetzen und die zugehörigen Parameterwerte zu sammeln.

Für einfachere Anwendungsfälle, die den Einsatz der `TableTree` Klasse nicht erfordern, kann dieses Modul auch direkt verwendet werden:[14]

`__init__()`

Initialisiert ein neues `Placeholders`-Objekt.

`inFilter(`*`filter`*`)`

Ersetzt alle als `?` markierte Platzhalter durch die von Oracle verwendeten, durchnummerierten Platzhalter und speichert die zugehörigen Parameterwerte im `Placeholders`-Objekt. Diese Parameterwerte können über das Attribut `values` abgerufen werden.

Beispiel:

```
filter = filterOr("vorname||' '||zuname LIKE ?",
                  ['%Alexander%', '%Bergolth%'])
filter2 = ['Ort = ?', 'Klosterneuburg']

pholders = Placeholders()
filterstr = pholders.inFilter(filter)
filterstr2 = pholders.inFilter(filter2)
sql = """
  SELECT tid FROM person
  WHERE
```

[14] Für viele Datenbank-APIs, die eine positionsunabhängige Syntax wie `?` direkt als Platzhalter akzeptieren, wird dieses Modul jedoch nicht benötigt.

```
    %s
    AND %s""" % ( filterstr, filterstr2 )

# ... sql:
  SELECT tid FROM person
  WHERE
  vorname||' '||zuname LIKE :1 OR vorname||' '||zuname LIKE :2
  AND Ort = :3

# ... pholders.values:
['Alexander', 'Bergolth', 'Klosterneuburg']
```

4.4.1.4 Module TableJoins

Die folgenden Hilfsfunktionen sind keine Methoden der Klasse `TableJoins` sondern Funktionen des Moduls:

`filterOr(`*conditions*`, `*params*`)`

Erzeugt einen Suchfilter, in dem die Suchparameter (die Suchbegriffe) *params* logisch ODER-verknüpft werden. Diese Parameter werden mit allen möglichen Kombinationen der Suchbedingungen *conditions* ODER-verknüpft. (Es handelt sich um Kombinationen mit Zurücklegen.) *conditions* sind Suchterme, wie sie im `WHERE`-Abschnitt eines SQL-Statements vorkommen können.

Beispiel:

```
filter = filterOr(('vorname LIKE ?', 'zuname LIKE ?'),
                  ['Alexander', 'Bergolth'])
# ... ergibt:
('vorname LIKE ? OR vorname LIKE ? OR
 zuname LIKE ? OR zuname LIKE ?',
 ['Alexander', 'Bergolth', 'Alexander', 'Bergolth'])
```

`filterAnd(`*conditions*`, `*params*`)`

Erzeugt einen Suchfilter, in dem die Suchparameter (die Suchbegriffe) *params* logisch UND-verknüpft werden. Diese Parameter werden mit allen möglichen Kombinationen der Suchbedingungen *conditions* ODER-verknüpft. (Es handelt sich um Variationen mit Zurücklegen.) *conditions* sind Suchterme, wie sie im `WHERE`-Abschnitt eines SQL-Statements vorkommen können.

Vorsicht: Das Ergebnis hat $conditions^{params}$ Hauptterme, und $params * conditions^{params}$ Suchbedingungen. Es sollte daher auf eine entsprechende Beschränkung der Elemente geachtet werden.

Beispiel:

```
filter = filterAnd(('vorname LIKE ?', 'zuname LIKE ?'),
                   ['Alexander', 'Bergolth'])
# ... ergibt:
('(vorname LIKE ? AND vorname LIKE ?) \
OR (vorname LIKE ? AND zuname LIKE ?) \
OR (zuname LIKE ? AND vorname LIKE ?) \
OR (zuname LIKE ? AND zuname LIKE ?)', \
['Alexander', 'Bergolth', 'Alexander', 'Bergolth', \
 'Alexander', 'Bergolth', 'Alexander', 'Bergolth'])
```

4.4.2 DCOracle2Row - Abfrageergebnisse als Objekte

Praktisch alle klassischen Datenbank-APIs liefern Suchergebnisse zeilenweise als Arrays zurück, deren Elemente den gewählten Ergebnisspalten entsprechen.

Diese Eigenschaft bringt einige Nachteile mit sich und erschwert es, eine problemorientierte, von der konkreten Datenrepräsentation abstrahierte und damit leicht verständliche Applikationslogik zu entwerfen:

- Das verarbeitende Programm muss das zugrundeliegende SQL-Statement kennen oder zumindest die genaue Struktur der zurückgelieferten Suchergebnisse kennen, da die Position eines Elementes von der Reihenfolge der Spalten im SELECT-Statement abhängig ist.
- Das entstehende Anwendung wird zwangsläufig unübersichtlich: Entweder ist zusätzlicher Programmcode notwendig, um das Array in symbolische Variablen zu „entpacken“ oder die Elemente werden direkt weiterverwendet, was zu schwer verständlichen Konstrukten, wie `if (row[7] == 'N') ...` im Programm führt.
- Das Ergebnis ist fehleranfällig und schwer wartbar, da sich bei Änderungen der SQL-Abfrage (wenn Ergebnisspalten eingefügt oder verändert werden) die Durchnummerierung der Arrayelemente ändert.
- Der oft verwendete Ansatz, die Ergebnisdaten in ein assoziatives Array zu verpacken, das nach Spaltennamen indiziert und von der Anordnung der Spalten unabhängig ist, ist ineffizient, wenn größere Datenmengen

gelesen werden, da die Spaltenbezeichnungen und Hash-Tabellen bei jeder Ergebniszeile extra gespeichert werden. Außerdem gehen bei dieser Darstellungsform die Informationen über die Reihenfolge der Spalten verloren.

Der hier vorgestellte Ansatz verwendet hingegen Metaklassen, um dynamisch Ergebnisklassen zu generieren, die auf die jeweilige Suchabfrage und die verwendeten Spaltennamen und Datentypen zugeschnitten sind. Die an die Applikation zurückgelieferten Ergebniszeilen sind schließlich Instanzen dieser dynamisch erzeugten Klassen.

Diese Umsetzung hat gegenüber herkömmlichen Implementierungen einige Vorteile:

- Effizienz: Spaltenbeschreibungen werden über Python-Slots implementiert und werden daher nur Pro-Klasse und nicht Pro-Instanz gespeichert.
- Übersichtlichkeit: Ergebniszeilen sind Objekte, deren Spalten symbolisch (über den Namen) angesprochen werden können. Spaltennamen werden dabei als Attribute der Ergebnisobjekte abgebildet (zum Beispiel `row.fields.vorname`. Darüberhinaus sind die Objekte aber auch als assoziative Arrays (Python Dictionary-Objekte) verwendbar, was einen Zugriff der Form `row['vorname']` ermöglicht.
- Abwärtskompatibilität: Zeilenobjekte besitzen auch die Eigenschaften eines Arrays (eines Python-`tuple`-Objektes), auf Elemente kann daher auch über den numerischen Index (beispielsweise `row[7]`) zugegriffen werden.

Die Implementierung der `DCOracle2Row` Klasse bedient sich dazu eines bestehenden Python-Moduls namens `db_row`[15]. Dieses Modul übernimmt anhand einer Spaltenbeschreibung die Erzeugung einer dynamischen Klasse, die schließlich in der Lage ist, Datenbankzeilen in Listenform, wie sie von herkömmlichen Datenbank-APIs geliefert werden, aufzunehmen und daraus Objekte zu generieren, die die beschriebenen Eigenschaften erfüllen.

Das Modul `DCOracle2Row` ist ein 1:1-Ersatz für das Python Datenbank-API *DCOracle2*. Über die Methoden des *Subclassing* und teilweise auch des *Monkey-Patching*[16], wird ein objektorientierter Layer über das DCOracle2-API gelegt, der die Funktionalität des `db_row`-Moduls transparent einbindet. Das

[15] `http://opensource.theopalgroup.com/`
[16] `http://en.wikipedia.org/wiki/Monkey_patch`

Ergebnis ist zu DCOracle2 kompatibel, die von den Methoden `fetchone()`, `fetchmany()` und `fetchall()` der `cursor`-Klasse zurückgelieferten Zeilen sind allerdings Objekte, die zusätzlich zu den Eigenschaften eines Python `tuples` auch symbolischen Zugriff auf die Elemente ermöglicht.

Da das API exakt dem des `DCOracle2`-Moduls entspricht, werden hier nur Beispiele für die Zugriffsmöglichkeiten auf die Abfrageergebnisse angeführt.

Beispiel:

```
>>> sql = "SELECT vorname, zuname, username FROM person WHERE zuname = :1"
>>> cursor.execute(sql, 'Riedling');
>>> row= cursor.fetchone()
>>> row
('Karl', 'Riedling', 'user-1000007')
>>> row[0]
'Karl'
>>> row['vorname']
'Karl'
>>> row['Zuname']
'Riedling'
>>> row.fields.username
'user-1000007'
>>> row.fields.Username
'user-1000007'
>>> row.keys()
('VORNAME', 'ZUNAME', 'USERNAME')
```

Abbildung 4.10: Beispiel für die Verwendung der `DCOracle2Row` Ergebnisobjekte

4.4.3 ExtSQL

ZOPE verwendet sogenannte *Database Connection*-Objekte, um Verbindungen zu relationalen Datenbanken zu definieren und anderen Objekten in der ZODB dem Zugriff auf diese Datenbanken zu ermöglichen.

Der vorgesehene Weg des Zugriffs führt allerdings über spezielle ZSQL-Methoden, die ebenfalls in der ZODB definiert werden müssen, was für Externe Methoden und Produkte, die im Filesystem des Servers liegen, unpraktikabel ist.

Die Klasse `ExtSQL` wurde daher entwickelt um auch Programmteilen, die nicht in der ZODB entwickelt wurden, zu ermöglichen, die in der ZODB definierten *Database Connection*-Objekte nutzen zu können.

Eine zusätzliche Erweiterung dieses Moduls übernimmt den automatischen Wiederaufbau der Verbindung, falls die Konnektivität zum Datenbankserver unterbrochen wurde. Dies ist bei Database Connection Objekten oft notwendig, da der Zope-Server die Verbindungen dauerhaft offen hält und es daher mit der Zeit netzwerkbedingt oder wegen datenbankseitiger Wartungs- oder Backup-Operationen zu Unterbrechungen der Verbindung kommt.

Da theoretisch die Trennung der Verbindung mitten in einer Datenbanktransaktion passieren könnte reicht es nicht aus, nach einem automatischen Wiederaufbau der Verbindung die zum Zeitpunkt des Fehlers ausgeführte Datenbankoperation zu wiederholen, schließlich könnten in derselben Transaktion ja bereits andere Befehle mit der alten Datenbankverbindung ausgeführt worden sein, die durch die Invalidierung dieser Verbindung ungültig würden.

Daher nutzt das Modul, nachdem die Verbindung zur Datenbank neu aufgebaut wurde, eine Exception, die in Zope speziell behandelt wird und eigentlich zur Auflösung von Konflikten bei konkurrierenden Zugriffen auf die ZODB gedacht war. Wie in Abschnitt 4.2.2.6 beschrieben bewirkt diese Exception, dass der aktuelle Request unterbrochen und komplett neu abgearbeitet wird. Sämtliche bei Zope registrierten Transaktionen (also auch die der relationalen Datenbankverbindungen) werden mittels *Rollback* rückgängig gemacht.

Die Klasse `ExtSQL` ist für das Subclassing aus anwendungsspezifischen Datenbank-Klassen heraus vorgesehen[17] und definiert nur einen Konstruktor und die Methode `cursor()`.

Achtung: Objekte, die von dieser Klasse abstammen, müssen für jeden Request neu instanziert werden.

`__init__(`*`context`*`, `*`dbcname`*`='db_connection')`

> *context* ist ein ZOPE-Context-Objekt (vergleiche Abschnitt 4.2.4), von dem aus die *Database Connection* gesucht werden soll. *dbcname* ist der Name (in Zope mit *Id* bezeichnet) des *Database Connection* Objektes.

[17] Die Klasse `PubSQL` ist ein Anwendungsbeispiel dafür.

`cursor()`

Liefert analog zur gleichnamigen Methode des Python-DB-API ein `cursor`-Objekt. Die genaue Beschaffenheit dieses Objektes ist von der verwendeten Datenbank und dem zugehörigen Datenbankadapter abhängig. Die meisten Implementierungen sind allerdings zur *Python Database API Specification v2.0* [Lem] kompatibel.

4.5 Datenmigration

In diesem Abschnitt werden Probleme und deren Lösungsmöglichkeiten bei der Übersiedelung bestehender Daten behandelt. Dabei werden sowohl die Datenübernahme in ein anderes Datenbankprodukt, als auch diverse Änderungen der Datenstruktur besprochen.

4.5.1 Überblick

Eine relationale Datenbank stellt ein Kernstück vieler komplexer Softwareprodukte dar. Aufgrund dieser zentralen Bedeutung sind umfangreichere Änderungen an dieser Stelle, oft trotz zwischengeschalteter Abstraktionsschichten, meist nur sehr aufwändig zu realisieren [Fow99].

Dennoch sind Datenmigrationen in manchen Fällen notwendig und sinnvoll. Beispiele dafür sind:

- Ein notwendiger Produktwechsel auf ein Datenbankprodukt mit grundlegend unterschiedlichen Eigenschaften.
- Änderungen am Datenbankschema, um die Wartbarkeit oder die Performance zu verbessern.
- Änderungen oder Anpassungen in der Anwendungslogik, die Umstellungen im Datenmodell erfordern, um weiterhin Datenkonsistenz gewährleisten zu können.

4.5.2 Datenmigrationstool

Obwohl gelegentlich auch eine manuelle beziehungsweise eine nur halbautomatische Transformation der Daten möglich wäre, empfiehlt es sich dennoch in den meisten Fällen, die Umwandlung vollautomatisch durch ein Migrationsprogramm durchzuführen. Obwohl dies möglicherweise einen auf den ersten Blick höheren Entwicklungsaufwand bedeutet, entstehen dadurch auch einige wichtige Vorteile:

- Die Übertragung der Daten ist meist kein einmaliger Vorgang.
- Die neue Entwicklung kann im Zielsystem mit Echtdaten getestet und evaluiert werden.
- Eine automatische Datenmigration ermöglicht oft einen vorübergehenden Parallelbetrieb. In einer Übergangsphase werden dann Daten in regelmäßigen Zeitintervallen ausgetauscht, wobei Schreibzugriffe nur in einer der Datenbanken erfolgen dürfen, da sonst eine intelligente Synchronisation erforderlich ist.
- Eine erfolgreiche programmgestützte Datenkonvertierung zeigt, dass eine Abbildung basierend auf einem exakten, fixen Regelwerk möglich ist und ohne subjektive Entscheidungen auskommt.
- Ein Programm liefert jederzeit reproduzierbare Datentransfers.
- Die Konfiguration des Migrationstools ist gleichzeitig eine Dokumentation der Änderungen des Datenmodells.

Es wurde daher ein Migrationsskript entwickelt, das in der Lage ist, konfigurierbare Datenbankabfragen in mehreren Quelldatenbanken abzusetzen, die Ergebnisse zu filtern und schließlich in eine Zieldatenbank zu schreiben.

Da das Migrationstool in der interpretierten Programmiersprache-Perl entwickelt wurde, kann die Konfiguration ebenfalls in Perl-Syntax erfolgen, was die die Flexibilität des Tools stark erhöht und den Programmieraufwand für das Basisprogramm auf wenige Zeilen Code reduziert hat.

Das Konfigurationsskript muss zwei Variablen definieren: Das Array `@dst_connect` beschreibt den Verbindungsaufbau zur Zieldatenbank und besitzt dasselbe Format, wie die Argumente der Klassenmethode `DBI::connect` des Perl-Datenbank-Interfaces `DBI`.

Zentrales Konfigurationselement ist jedoch das Array `@migrate`. Hier wird definiert, wie die Verbindungen zu den Quell-Datenbanken hergestellt werden und welche Migrationsschritte mit Hilfe dieser Datenbanken ausgeführt werden sollen. Abbildung 4.11 zeigt die Syntax dieser Datenstruktur.

```
@migrate = [
  [
    \@src_connect1,
    [
       \%migrate_step11,
       \%migrate_step12, ...
    ]
  ], ...
  [
    \@src_connect2,
    [
       \%migrate_step21,
       \%migrate_step22, ...
    ]
  ], ...
];

@src_connect1 = ( $data_source, $username, $passwd, \%attr );

%migrate_step11 = (
  name => 'step1',
  active => 1,
  src_sql => $src_sql_str11,
  prefilter_sql => [ $prefilter_sql_str11 => \@prefilter_sql_params11 ],
  filter => sub {
    # filterprog
    my ($src_row, $prefilter_row, $rownum) = @_;
    ...
    return (@dst_row);
  },
  postfilter_sql => [ $postfilter_sql_str11 => \@postfilter_sql_params11 ],
  dst_sql => [
    $dst_sql_str111 => \@dst_sql_params111,
    $dst_sql_str112 => \@dst_sql_params112, ...
  ]
);
```

Abbildung 4.11: Syntax der `@migrate` Datenstruktur

Der Ablauf des Migrationsskripts ist eng mit der Struktur der `@migrate` Datenstruktur verknüpft und entspricht folgendem Schema:

1. Verbinde mit der Ziel-Datenbank.
2. Für alle SRC-Datenbanken:
   a) Verbinde mit der Quell-Datenbank.
   b) Für alle Migrationsschritte (`%migrate_stepXX`):
      i. Führe das `src_sql`-Statement auf der Quell-Datenbank aus. Hier werden die zu migrierenden Datensätze geholt.
      ii. Für jede Ergebnis-Zeile aus src_sql:
         A. Führe das `dst_sel1`-Statement auf der Zieldatenbank aus. (optional)
            Als Parameter können optional Spalten des Quell-Datensatzes verwendet werden. `@prefilter_sql_params` enthalten dann die entsprechenden Spaltenindizes.
         B. Führe den `filter`-Programmcode aus. (optional)
            Der Filter erhält den Datensatz (`$src_row`), die erste Ergebniszeile des `dst_sel1`-Statements (`prefilter_row`) und die Zeilennummer der aktuellen `src_sql`-Abfrage als Argumente. Der Rückgabewert der Filterroutine ist der veränderte Datensatz (`@dst_row`).
         C. Führe das `destsel2`-Statement auf der Zieldatenbank aus. (optional)
            Als Parameter können optional Spalten des aktuellen (möglicherweise bereits durch `filter` veränderten) Datensatzes oder die Ergebnisspalten der `prefilter_sql`-Abfrage verwendet werden. `@postfilter_sql_params` enthalten dann entweder nur die entsprechenden Spaltenindizes (für die Spalten des Datensatzes) oder Spaltennummern der Form `pre_0`, `pre_1`, etc.
         D. Führe die `dest_sql`-Statements auf der Zieldatenbank aus. Als Parameter können Spalten des aktuellen (möglicherweise bereits durch `filter` veränderten) Datensatzes oder die Ergebnisspalten der `prefilter_sql`- beziehungsweise der `postfilter_sql`-Abfrage verwendet werden.

`@dst_sql_params` enthalten dann entweder nur die entsprechenden Spaltenindizes (für die Spalten des Datensatzes) oder Spaltennummern der Form `pre_0`, `pre_1`, beziehungsweise `post_0`, `post_1`, etc.

4.5.2.1 Datenkonvertierung

In diesem Abschnitt werden beispielhaft einige Beispiele für verschiedene Konvertierungsszenarien angeführt. Sämtliche Konfigurationsbeispiele enthalten zur Veranschaulichung stark vereinfachte Datenstrukturen.

Konvertierung zwischen verschiedenen Datentypen oder Datenrepräsentationen

Die einfachsten Konvertierungen betreffen eine eindeutige Abbildung eines Spaltenwertes der Quelldatenbank auf einen neuen Wert in der Zieldatenbank.

Diese Maßnahme kann beispielsweise notwendig sein, um auf Zielspalten mit unterschiedlichen Datentypen zu migrieren, um die Formatierung der Daten zu verändern (z.B. diverse Textkonvertierungen) oder die Kodierung von Flags zu verändern.

Der optimale Ort um solche Umwandlungen durchzuführen ist im Programmcode des `filter`-Blocks, ein Beispiel für eine solche Umwandlung findet sich in Abbildung 4.12 in der Veränderung der Spalte `$s[6]`.

Falls Fremdschlüsselabhängigkeiten zu dem veränderten Wert bestehen, müssen die entsprechenden Felder natürlich ebenfalls dieselbe Transformation erfahren.

Vorbereitung auf eine Konvertierung in zweite Normalform, Aufspaltung von Tabellen, Einführung neuer Schlüssel

Als Vorbereitung auf eine Normalisierung einer Tabelle, ist es möglich, die Quelltabelle in einem Migrationsschritt auf zwei Zieltabellen aufzuspalten und dabei die referentielle Integrität zu erhalten.

Beispiel:

```
# migrate_step
{
  'src_sql' =>
    "SELECT PIndex, Titel, Medium, Type, Publikationsjahr,
            UsedInABIV, AutorEtAl,
            DATE_FORMAT(Eintragung, '%Y%m%d %T'), OrigEigentuemer,
            DATE_FORMAT(Aenderung, '%Y%m%d %T'), geaendert
       FROM Publikation",

  'filter' => sub {
    my ($s, $pre, $rowcount) = @_;
    @s= @$s;

    $s[0] += $count * $ID_STEP;        # pindex -> tid_alt
    $s[2] += $count * $ID_STEP;        # medium -> medium.tid_alt
    $s[6] = $s[6] eq 'JA' ? 'j' : 'n'; # autor_etal
    $s[8] += $count * $ID_STEP;        # fk_created
    $s[10] += $count * $ID_STEP;       # fk_changed_by

    # gefilterten Datensatz zurückliefern
    @s;
  },

  # neue tid fuer publikation und medium.tid aus medium.tid_alt
  # und ev. medium.pub_tid_alt holen
  # MAX wird verwendet, damit das Subselect auf jeden Fall eine
  # Zeile liefert
  'postfilter_sql' => [
    "SELECT publikation_seq.nextval, mtid FROM
       ( SELECT MAX(tid) AS mtid FROM medium
         WHERE tid_alt = ? AND (pub_tid_alt IS NULL OR pub_tid_alt
         = ?)
       )" => [ 2, 0 ] ],

  'dst_sql' => [
    "INSERT INTO publikation (tid, name, fk_medium, fk_pubtyp, jahr,
       abiv_jahr, autor_etal,
       created_at, fk_created_by, changed_at, fk_changed_by, tid_alt)
       VALUES (?, ?, ?, ?, ?, ?, ?,
         TO_DATE(?, 'YYYYMMDD HH24:MI:SS'), ?,
         TO_DATE(?, 'YYYYMMDD HH24:MI:SS'), ?, ?)" =>
    [ 'post_0', 1, 'post_1', 3..10, 0 ]
  ]
}
```

Abbildung 4.12: Konfigurationsbeispiel: Datenkonvertierungen

In Abbildung 4.13 wird gezeigt, wie Informationen zum Verlag aus der Tabelle `Publikation` geholt und in eine neue Tabelle `Verlag` migriert werden können. Eine alternative Implementierung ist in Abbildung 4.14 dargestellt.

Beispiel:

```
{
  # ORDER BY wird verwendet, um eine definierte Reihenfolge zu
  # gewaehrleisten (bei mehreren Gross-/Kleinschreibungsvarianten
  # wird nur der erste Verlag eingetragen)
  'src_sql' =>
    "SELECT PIndex, Titel, Verlag, VerlagOrt
       FROM Publikation
       ORDER BY PIndex",

  'prefilter_sql' => [
    "SELECT tid from Verlag
       WHERE LOWER(name) = LOWER(?) AND LOWER(ort) = LOWER(?)" =>
    [ 2, 3 ] ],

  # Sequence Nummer wird nur geholt, wenn prefilter keine
  # existierende Verlags-tid liefert
  'postfilter_sql' => [
    "SELECT verlag_seq.nextval FROM dual WHERE ? IS NOT NULL" =>
    [ 'pre_0' ] ],

  'dst_sql' => [
    # INSERT wird nur ausgefuehrt, wenn in prefilter kein Verlag
    # gefunden wurde
    "INSERT INTO Verlag (tid, name, ort)
       SELECT ?, ?, ? FROM DUAL WHERE ? IS NOT NULL" =>
    [ 'post_0', 2, 3, 'pre_0' ],

    "INSERT INTO Publikation (tid, titel, fk_verlag)
       VALUES (publikation_seq.nextval, ?, NVL(?, ?))" =>
    [ 1, pre_0, post_0 ],
  ]
}
```

Abbildung 4.13: Konfigurationsbeispiel: Aufspaltung

Dieser Schritt ist allerdings nur eine Vorbereitung auf eine Normalisierung, weitere Nachbesserungen, beispielsweise um unterschiedliche Schreibweisen desselben Verlages (beziehungsweise des Verlags-Ortes) zusammenzufassen, müssen in der Regel manuell durchgeführt werden. In vielen Fällen sind hier auch computerunterstützte Gruppierungen möglich, aufgrund der starken Kontext-

Beispiel:

```
{
    'src_sql' =>
      "SELECT DISTINCT Verlag, VerlagOrt
         FROM Publikation",

    # WHERE NOT EXISTS ist notwendig, wenn Daten aus mehreren
    # Quell-Tabellen geholt werden
    'dst_sql' => [
INSERT INTO Verlag (tid, name, ort)
SELECT verlag_seq.nextval, ?, ?
WHERE NOT EXISTS (
  SELECT tid FROM verlag WHERE name = ? AND ort = ?
)
      [ 1, 2, 1, 2 ] ]
},
{
    'src_sql' =>
     "SELECT PIndex, Titel, Verlag, VerlagOrt
         FROM Publikation
       ORDER BY PIndex",

    'dst_sql' => [
      "INSERT INTO publikation (tid, titel, fk_verlag)
       SELECT publikation_seq.nextval, ?, verlag.tid
         FROM verlag
         WHERE name = ? AND ort = ?" =>
      [ 1, 2, 3 ] ]
}
```

Abbildung 4.14: Konfigurationsbeispiel: Aufspaltung Alternative

abhängigkeit und der Notwendigkeit Heuristiken einzusetzen, sind jedoch auf den individuellen Fall abgestimmte Lösungen empfehlenswert.

Kategorisierung (Supertyp-/Subtyp-Beziehungen)

Analog zu vorigem Beispiel ist natürlich genauso eine Kategorisierung, also eine Aufspaltung einer Tabelle in eine Generalisierung und mehrere Spezialisierungen möglich. Hier ist zu beachten, dass die Spezialisierungstabellen denselben Primärschlüssel wie die Generalisierungen haben.

Zusammenführung mehrerer Tabellen / Datenbanken, Modifikation von Schlüsselfeldern

Bei der Konsolidierung einer Tabelle aus mehrerer Datenbanken in eine einzige Zieldatenbank entstehen mehrere potentielle Probleme:

- Derselbe logische Datensatz existiert möglicherweise in mehreren Quelldatenbanken und wird somit auch in die Zieldatenbank in mehrfacher Ausführung migriert.
- Eventuelle `UNIQUE`-Constraints werden daher in der Zieldatenbank möglicherweise verletzt.
- Die Nummernbereiche von Surrogarschlüsselfeldern, also von automatisch generierten Schlüsselwerten (z.B. `AUTO_INCREMENT` in MySQL oder `SEQUENCE` in Oracle), kollidieren bei der Zusammenführung aus mehreren Datenbanken meistens. Es ist daher notwendig, dies Schlüsselwerte in disjunkte Bereiche zu transformieren, damit Eindeutigkeit gewahrt bleibt. Diese Änderung muss gegebenenfalls bei sämtlichen Fremdschlüsselfeldern, die auf die modifizierten Attribute verweisen, berücksichtigt werden.

Surrogatschlüsselfelder meist durchnummerierte Sequenznummern sind, ist eine Abbildung der Form $ID_{neu} = count * DB\_STEP + ID_{alt}$ oft ausreichend. *count* ist dabei ein Zähler, der für jede Quelldatenbank hochgezählt wird, DB_STEP ist größer als der höchste ID-Wert, der in den Quell-Datenbanken vorkommt.

Beispiel:

```
@dbs = (
  [ 'db1', 'bergolth', '' ],
  [ 'db2', 'bergolth', '' ]
);
my $ID_STEP = 1000000;
my $i = 1;
for my $src_connect (@dbs) {
  # closures muessen $count als Zaehler verwenden!
  my $count = $i;

  @tables = (
    \%migrate_step11, ...
  );

  push(@migrate, [ $src_connect, \@tables ]);
  $i++;
}
```

Abbildung 4.15: Konfigurationsbeispiel: Konsolidierung mehrerer Datenbanken

Abbildung 4.15 zeigt das Grundgerüst einer solchen Migration, ein Beispiel für die eigentliche Transformation eines Schlüsselattributs findet sich im `filter`-Script von Abbildung 4.12.

5 Zusammenfassung

Mit dem Projekt TaskEngine wurde ein, für die Web-Anwendungsentwicklung eher unkonventioneller Ansatz verfolgt: Im Gegensatz zu den meisten existierenden Frameworks wurde hier versucht, den Aufbau und den Ablauf der Programme vom zustandslosen Request-Response-Prinzip des Hypertext Transfer Protocol loszulösen und eine Arbeitsweise nach den Methoden und Möglichkeiten der klassischen GUI-Programmierung anzubieten.

Funktionierende erste Testanwendungen zeigen die prinzipielle Realisierbarkeit dieses Konzeptes, auch die Zielsetzung einer hohen Modularität und Wiederverwendbarkeit der mit Hilfe der TaskEngine entwickelten Komponenten wurde durch das ereignisorientierte Design des Applikationsservers nach Ansicht des Autors erreicht.

Das entwickelte API deckt bereits viele, im Alltagsgeschäft der Web-Anwendungsentwicklung benötigten Funktionen ab und verbirgt damit in den meisten Fällen die relativ komplexen, für den Neuling oft schwer erlernbaren und unübersichtlichen Interna von Zope vor dem Entwickler.

Durch die Gliederung der Applikation in *Tasks*, die, ähnlich wie im Prozessmodell der Betriebssysteme Unix oder Windows, voneinander unabhängig ablaufen, ist eine übersichtliche Gliederung in Teilaufgaben leicht realisierbar. Trotz dieser Trennung ist eine Kommunikation über Nachrichten jederzeit möglich. Auf diese Art können Unteraufgaben (wie beispielsweise das Erfassen einer in der Datenbank noch nicht vorhandenen Person) als Sub-Tasks in den Arbeitsablauf eingeschoben werden.

Die Umsetzung dieses Prozessmodells auf eine Web-Anwendung wurde durch eine automatische Persistenzschicht ermöglicht, die nicht nur, wie bei Web-Applikationen oft üblich, die Daten der Anwendung speichert, sondern die gesamte Objektstruktur der laufenden Tasks transparent über einen Web-Request hinaus sichert.

Dennoch unterscheidet sich das Funktionsprinzip einer dynamischen Web-Anwendung mit der eigenständigen Anzeigeeinheit in Form eines mit dem

Server nur lose gekoppelten Web-Browsers, grundlegend von der Arbeitsweise einer GUI-Anwendung, sodass teilweise Kompromisse eingegangen werden mussten, um eine analoge Behandlung zu ermöglichen. So sollten beispielsweise sämtliche Bedienelemente, die den Zustand des Browsers verändern, ohne dabei mit dem Server Kontakt aufzunehmen, durch entsprechendes Setzen von HTTP-Headern oder JavaScript-Anweisungen deaktiviert werden. Auch der für das World-Wide-Web übliche direkte Einstieg in beliebige Stellen des Programms (beispielsweise via Bookmarks) erfordert derzeit eine Ausnahmebehandlung in der Ablaufsteuerung.

Im Bereich der Datenbank-Einbindung wurden die wichtigsten Grundlagen geschaffen: Mit Hilfe des Moduls `TableJoins` können Abfragen auf sichere Art, von Benutzereingaben abhängig, dynamisch generiert werden, ohne die Applikation dabei der Gefahr von SQL-Injections auszusetzen. Abfrageergebnisse können durch `DCOracle2Row` über ihre Spaltennamen symbolisch als Objekte oder als assoziative Arrays angesprochen werden. Durch den Einsatz von Metaklassen werden die dafür notwendigen Zusatzinformationen pro Abfrage und nicht, wie bei Lösungen, die assoziative Arrays zur Datenhaltung verwenden, pro Zeile gespeichert. Dieser Ansatz verbessert die Speichereffizienz bei großen Abfragen deutlich.

Zur Datenmigration aus einem Altsystem wurde schließlich noch ein Hilfsprogramm entwickelt, das eine Datenübertragung in die neue Datenbank basierend auf Konfigurationsangaben und ohne die Notwendigkeit einer Benutzerinteraktion ermöglicht und zusätzlich, bei gleichzeitigen Änderungen des Datenmodells auch noch vorbereitende Schritte für eine Konvertierung der Daten erledigen kann. In diesen Fällen wird allerdings oft, abhängig vom Individualfall, eine spezielle Nachbehandlung, basierend auf Heuristiken beziehungsweise menschlichen Entscheidungen, notwendig sein. In einigen Beispielen wurden hierzu mögliche Konvertierungsszenarien diskutiert und Lösungsvorschläge anhand von Konfigurationsmustern vorgestellt.

Literaturverzeichnis

[BL91] Berners-Lee, Tim: *The HTTP Protocol As Implemented In W3*. `http://www.w3.org/Protocols/HTTP/AsImplemented.html`. Version: 1991, Abruf: 18. Jan 2007

[BL92] Berners-Lee, Tim: *HTTP: A protocol for networked information*. `http://www.w3.org/Protocols/HTTP/HTTP2.html`. Version: 1992, Abruf: 18. Jan 2007

[CEH+02] Curphey, Mark ; Endler, David ; Hau, William ; Taylor, Steve ; Smith, Tim ; Russell, Alex ; McKenna, Gene ; Parke, Richard ; McLaughlin, Kevin u. a.: *A Guide to Building Secure Web Applications*. `http://www.cgisecurity.com/owasp/html/index.html`. Version: 2002, Abruf: 19. Jan 2007. – Version 1.1 Final

[DGH03] Dustdar, Schahram ; Gall, Harald ; Hauswirth, Manfred: *Software-Architekturen für Verteilte Systeme*. Springer, 2003. – 264 S.

[DH03] Dunkel, Jürgen ; Holitschke, Andreas: *Softwarearchitektur für die Praxis*. Springer, 2003. – 418 S.

[FGM+99] Fielding, R. ; Gettys, J. ; Mogul, J. ; Frystyk, H. ; Masinter, L. ; Leach, P. ; Berners-Lee, T.: Hypertext Transfer Protocol – HTTP/1.1 / Internet Engineering Task Force. Version: 1999. `http://rfc.net/rfc2109.html`, Abruf: 24. Jan 2007. 1999 (RFC2616). – Request for Comments

[Fie00] Fielding, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, Diss., 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`, Abruf: 19. Jan 2007

[Fow99] Fowler, Martin: *Refactoring: Improving the design of existing code*. Addison-Wesley, 1999. – 464 S.

[Fow05] Fowler, Martin: *Inversion of Control*. `http://martinfowler.com/bliki/InversionOfControl.html`. Version: 2005, Abruf: 25. Jul 2007

[Fula] Fulton, Jim: *Acquisition Algebra*. `http://zope.org/Members/jim/Info/IPC8/AcquisitionAlgebra/index.html`, Abruf: 27. Apr 2007

[Fulb] Fulton, Jim: *Introduction to the Zope Object Database*. `http://www.python.org/workshops/2000-01/proceedings/papers/fulton/fulton-zodb3.pdf`, Abruf: 20. Apr 2007. – Version 2.4

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. – 416 S.

[Hir96] Hirschfeld, Robert: *Three-Tier Distribution Architecture*. `http://citeseer.ist.psu.edu/173051.html`. Version: 1996, Abruf: 1. Dez 2006

[Jou04] Jouravlev, Michael: *Redirect after Post*. `http://www.theserverside.com/tt/articles/article.tss?l=RedirectAfterPost`. Version: 2004, Abruf: 24. Jan 2007

[KM95] Kristol, D. ; Montulli, L.: Hypertext Markup Language - 2.0 / Internet Engineering Task Force. Version: 1995. `http://rfc.net/rfc1866.html`, Abruf: 27. Apr 2007. 1995 (RFC1866). – Request for Comments

[KM97] Kristol, D. ; Montulli, L.: HTTP State Management Mechanism / Internet Engineering Task Force. Version: 1997. `http://rfc.net/rfc2109.html`, Abruf: 19. Jan 2007. 1997 (RFC2109). – Request for Comments

[KP88] Krasner, G. ; Pope, S.: A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. In: *Journal of Object Oriented Programming* 1 (1988), Nr. 3, 26–49. `http://citeseer.ist.psu.edu/krasner88description.html`, Abruf: 2. Dez 2006

[Lem] Lemburg, Marc-André: Python Database API Specification / Python Software Foundation. `http://www.python.org/dev/peps/`

pep-0249, Abruf: 20. Dez 2007 (PEP249). – Python Enhancement Proposals. – Version 2.0

[Lew98] Lewandowski, Scott M.: Frameworks for Component-Based Client/Server Computing. In: *ACM Computing Surveys* 30 (1998), Nr. 1, 3-27. http://citeseer.ist.psu.edu/lewandowski98frameworks.html

[LPMS] Latteier, Amos ; Pelletier, Michel ; McDonough, Chris ; Sabaini, Peter: *The Zope Book*. http://www.zope.org/Documentation/Books/ZopeBook/2_6Edition/ZopeBook-2_6.pdf, Abruf: 9. Mar 2007. – Version 2.6

[LTM06] Lerdorf, Rasmus ; Tatroe, Kevin ; MacIntyre, Peter: *Programming PHP*. 2. Auflage. O'Reilly, 2006. – 540 S.

[Lut06] Lutz, Mark: *Programming Python*. 3. Auflage. O'Reilly, 2006. – 1596 S.

[MPH] McDonough, Chris ; Pelletier, Michel ; Hathaway, Shane: *The Zope Developer's Guide*. http://www.zope.org/Documentation/Books/ZDG/current/DevGuide-2_4.pdf, Abruf: 29. Mar 2007. – Version 2.4

[MRA05] Martelli, Alex ; Ravenscroft, Anna M. ; Ascher, David: *Python Cookbook*. 2. Auflage. O'Reilly, 2005. – 844 S.

[Mül04] Müller, Joachim: *Workflow-based Integration: Grundlagen, Technologien, Management*. Springer, 2004. – 238 S.

[Pet] Peters, Tim: The Zen of Python / Python Software Foundation. http://www.python.org/dev/peps/pep-0020, Abruf: 6. May 2008 (PEP20). – Python Enhancement Proposals

[Qus04] Qusay, Mahmoud H. (Hrsg.): *Middleware for Communications*. Wiley, 2004. – 522 S.

[Rie08] Riedling, Karl: *Benutzerhandbuch zur Publikationsdatenbank*. http://publik.tuwien.ac.at/info/manual/Web-Publikationsdatenbank.pdf. Version: 2008, Abruf: 8. Apr 2008. – Version 2.40

[SA88] Swick, Ralph R. ; Ackerman, Mark S.: The X Toolkit: More Bricks for Building User-Interfaces or Widgets for Hire. In: *USENIX Winter*, 1988, S. 221–228

[Wei06] Weitershausen, Philipp von: *Web Component Development with Zope 3*. Second Edition. Springer, 2006. – 467 S.