Diplomarbeit am Fachbereich Informatik
der Universität Hamburg
Arbeitsbereich AGN
in Zusammenarbeit mit der
Axel Springer Verlag AG, Abt. Interactive Media

Konzeption und Ansätze der Realisierung eines digitalen Publikationssystems für personalisierte TV-Programminformationen

Diplomanden:

Nils Münch
Rutschbahn 27
20146 Hamburg
Telefon: 040 / 41 81 27
Matr.Nr.: 40 60 874

Jan Krogmann
Rissener Str. 72 a
22880 Wedel
Telefon: 04103 / 97 09 83
Matr.Nr.: 45 27 014

Betreuer an der Universität Hamburg:

Prof. Dr. Klaus Brunnstein
Vogt-Kölln-Str. 30, Haus C
22527 Hamburg
Tel.: 040 / 5494 - 2405

Dipl.-Ing. Dr. Werner Hansmann
Vogt-Kölln-Str. 30, Haus F
22527 Hamburg
Tel.: 040 / 5494 - 2544

Betreuer in der Axel Springer Verlag AG:

Dipl.-Inf. Jörn Kropfgans
Abt. Interactive Media
Axel-Springer-Platz 1
20350 Hamburg
Tel.: 040 / 347 – 27153

Hamburg, den 30. Dezember 1998

VORWORT ..... **III**

1 EINLEITUNG **1**

2 AXEL SPRINGER VERLAG AG **3**

2.1 AKTIVITÄTEN DER AXEL SPRINGER VERLAG AG IM INTERNET 3
2.2 VERWENDETE INTERNET-TECHNOLOGIEN 4

3 KONZEPT EINES PUBLIKATIONSSYSTEMS FÜR PERSONALISIERTE TV-PROGRAMMINFORMATIONEN **5**

3.1 TRADITIONELLE VERSUS ELEKTRONISCHE TV-PROGRAMMINFORMATIONEN 5
3.2 AUFBAU DES SYSTEMS 8
3.3 ONLINE-VERSION 9
3.3.1 Gast-Bereich 9
3.3.2 Abonnenten-Bereich 10
3.4 PERSONALISIERTER E-MAIL-SERVICE 12

4 VORÜBERLEGUNGEN FÜR DIE REALISIERUNG **15**

4.1 BETRIEBSUMGEBUNG UND ENTWICKLUNGSUMGEBUNG 15
4.2 MODELLIERUNGS-WERKZEUGE 16
4.3 DATENBANKEN 19

5 MODELLIERUNG **23**

5.1 EINFÜHRUNG IN OMT 23
5.1.1 Methodologie 23
5.1.2 OMT-Notation 25
5.1.3 Modellierungsprozeß 31
5.2 PROBLEMANALYSE 34
5.2.1 Objektmodellierung 34
5.2.2 Dynamische Modellierung 44
5.2.3 Funktionale Modellierung 52
5.3 SYSTEMENTWURF 56
5.3.1 Aufbrechen des Systems in Teilsysteme 56
5.3.2 Datenspeicher-Strukturen 60
5.3.3 Sicherheit 62
5.3.4 Behandlung von Grenzbedingungen 65
5.4 OBJEKTENTWURF 70
5.4.1 Objektmodell 71
5.4.2 Dynamisches Modell 87
5.4.3 Entwurf von Algorithmen 107

6 IMPLEMENTATION **109**

6.1 DATENBANKMODELL 109
6.2 DATENBANKZUGRIFF 113
6.3 DOKUMENTEN-GENERIERUNG 122

7 NACHWEIS DER FUNKTIONALITÄT **135**

7.1 VERFAHRENSWEISE 135
7.1.1 Regelbasierter Test 136
7.1.2 Nachrichtenbasierter Test 137
7.1.3 Nutzungsbasierter Test 138
7.1.4 Systemabnahme 138
7.2 TESTVORGEHENSMODELL 139

8 AUSBLICK **141**

9 ANHANG 143

9.1 ALLGEMEINES LITERATURVERZEICHNIS 143
9.2 SPEZIELLES LITERATURVERZEICHNIS 144
9.3 URL VERZEICHNIS 147
9.4 TABELLENVERZEICHNIS 148
9.5 ABBILDUNGSVERZEICHNIS 149
9.6 GLOSSAR 151
9.7 DATA-DICTIONARY 155
9.8 PEP MINI TRANSFER PROTOKOLL 197
9.9 SOFTWARE 200
9.10 BEISPIEL FÜR DEN INHALT EINES FERNSEHBEITRAGES 201
9.11 ERKLÄRUNG DER VERFASSER 202

Vorwort

Diese Diplomarbeit befaßt sich mit der Konzeption und in Ansätzen mit der Realisierung eines im World Wide Web (WWW) zu betreibenden elektronischen Publikationssystems für TV-Programminformationen.

Die Arbeit wurde als Gruppenarbeit am Fachbereich Informatik, Arbeitsbereich *Anwendungen der Informatik in Geistes- und Naturwissenschaften* (AGN), angefertigt. Die Ausarbeitung begann in der Abteilung *Electronic Publishing* der Axel Springer Verlag AG.

Grundlage für diese Arbeit bildet die im Jahr 1997 in der Abteilung *Electronic Publishing* erstellte Studienarbeit von Nils Münch [Münch97]. Die Studienarbeit umfaßt eine Bestandsanalyse der 1997 im Internet verfügbaren elektronischen TV-Programminformationssysteme und eine Grobarchitektur für die Entwicklung eines neuartigen personalisierten TV-Programminformationssystems. Die Idee dieses neuen TV-Programminformationssystems wurde dem Vorstandsvorsitzenden der Axel Springer Verlag AG, Prof. Dr. Jürgen Richter, im Rahmen einer Präsentation vorgestellt. Im Anschluß an die Präsentation wurde die Entscheidung getroffen, das System im Rahmen einer Diplomarbeit zu realisieren.

Ab dem 1. Oktober 1997 wurden seitens der Axel Springer Verlag AG zwei Arbeitsplätze sowie eine geeignete Entwicklungsumgebung für die Realisierung des Systems zur Verfügung gestellt. Am 1. Januar 1998 übernahm August A. Fischer den Vorsitz des Vorstandes der Axel Springer Verlag AG. Dieser Vorstandswechsel führte zu Umstrukturierungen innerhalb des Unternehmens, die unter anderem die Auflösung der Abteilung *Electronic Publishing* zur Folge hatten. Im Mai 1998 konnte die Bearbeitung dieser Diplomarbeit in der Abteilung *Interactive Media* wieder aufgenommen werden. Zu diesem Zeitpunkt kristallisierte sich heraus, daß der ursprüngliche Anspruch, das System zu konzeptionieren und zu realisieren, in der verbleibenden Zeit nicht mehr erfüllt werden konnte. Aus diesem Grund liegt der Schwerpunkt dieser Arbeit auf der Konzeption des Publikationssystems, die nach der OMT-Methodologie vorgenommen wurde.

Die gesamte im Rahmen dieser Diplomarbeit beschriebene Konzeption, inklusive des OMT-Modells, wurde von beiden Diplomanden gemeinsam erarbeitet. Für die schriftliche Ausarbeitung der Diplomarbeit wurde eine Aufteilung der einzelnen Kapitel vorgenommen, die im folgenden aufgelistet sind.

Folgende Kapitel der Diplomarbeit wurden von beiden Diplomanden gemeinsam geschrieben:

	Vorwort
Kapitel 1	Einleitung
Kapitel 2	Axel Springer Verlag AG
Kapitel 6.1	Datenbankmodell
Kapitel 7	Nachweis der Funktionalität
Kapitel 8	Ausblick

Von Nils Münch wurden folgende Kapitel geschrieben:

Kapitel 3	Konzept eines Publikationssystems für personalisierte TV-Programminformationen
Kapitel 5.1	Einführung in OMT
Kapitel 5.4	Objektentwurf
Kapitel 6.2	Datenbankzugriff

Von Jan Krogmann wurden folgende Kapitel geschrieben:

Kapitel 4	Vorüberlegungen für die Realisierung
Kapitel 5.2	Problemanalyse
Kapitel 5.3	Systementwurf
Kapitel 6.3	Dokumenten-Generierung

Danksagungen
Wir möchten uns an dieser Stelle bei der Axel Springer Verlag AG dafür bedanken, daß uns die Möglichkeit geboten wurde, diese Diplomarbeit im Verlag anfertigen zu dürfen. Wir danken allen Personen, die uns während der Diplomarbeit mit Rat und Tat zur Seite standen. Vor allem aber Jörn Kropfgans, Sven Hauptvogel, Anke Krahn, Krista Kniel, Lothar Mehlhorn und Miki.

Ganz besonderer Dank gilt Herrn Professor Brunnstein für seine Unterstützung und Ermutigung während der schwierigen Phase der Abteilungsauflösung. Weiterhin möchten wir ihm sehr für seine hilfreichen Anregungen und für die Betreuung dieser Arbeit danken.

1 Einleitung

Seit der Einführung des World Wide Web im Jahr 1990 wächst das Internet mit eindrucksvoller Geschwindigkeit. Zwischen Juli 1992 und Juli 1998 stieg die Anzahl der Internet-Hosts weltweit von 992.000 auf 36,7 Millionen [URL-NetWiz98].

Deutschsprachige Angebote machen, gemessen am weltweiten Gesamtangebot, einen vergleichsweise geringen Anteil aus. Dennoch betrug die Anzahl der Hosts unter der deutschen Toplevel-Domain „.de“ im Juli 1998 bereits 1,3 Millionen, gegenüber nur 46.903 im Juli 1992 [URL-DENIC98]. Das entspricht einer Steigerungsrate von 2645% in 6 Jahren.

Deutsche Unternehmen haben die wachsende Bedeutung des Internets für sich erkannt. Dies gilt in zunehmendem Maße auch für Verlage, die das Internet intensiv als neue Publikationsplattform nutzen. Dieses Engagement trägt bereits Früchte. Werden Suchmaschinen und Vermarktungsgemeinschaften ausgenommen, so finden sich unter den zehn erfolgreichsten deutschsprachigen Angeboten insgesamt sieben Zeitschriften und Zeitungen.[1] Interessant ist in diesem Zusammenhang auch, daß unter diesen zehn Angeboten drei dem Bereich Fernsehen zuzuordnen sind. Dies sind die Angebote von „TV-Spielfilm“, „SAT 1“ und „TV Movie“. „TV Spielfilm Online“ war im November 1998 mit einer Reichweite von 2.982.294 Visits und 10.760.740 PageImpressions das zweiterfolgreichste deutschsprachige Verlagsangebot.[2]

Diese Diplomarbeit beschäftigt sich damit, ein neuartiges TV-Programminformationssystem für das Internet zu entwickeln. Das zu entwickelnde System bietet, wenn es ausschließlich online genutzt wird, eine zu den bereits im Internet existierenden Systemen vergleichbare Leistung. Das heißt, es werden alle notwendigen Funktionalitäten geboten, um einem Benutzer eine komfortable Fernsehprogrammauswahl zu ermöglichen. Der Unterschied dieses Systems zu bereits im Internet existierenden TV-Programminformationssystemen liegt in der Fähigkeit, personalisierte TV-Programmzeitschriften im PDF-Format generieren zu können. Diese personalisierten TV-Programmzeitschriften werden anhand der von Benutzern eingerichteten Profile generiert und enthalten nur noch Beiträge, die den persönlichen Interessen bzw. Profilen der Benutzer entsprechen. Die so generierten Fernsehzeitschriften werden auf elektronischem Wege an die Benutzer des Systems versendet. Anschließend können die Zeitschriften von den Benutzern auf S/W- oder Farb-Druckern ausgedruckt und wie herkömmliche Programmzeitschriften vor dem Fernseher verwendet werden. Die Verwendung des PDF-Formats garantiert dabei, daß die generierte Zeitschrift in gleichbleibendem Layout reproduziert wird, unabhängig davon, welches Betriebssystem bzw. welcher Drucker beim Benutzer zum Einsatz kommt.

---

[1] Die aktuellen Reichweiten aller gemeldeten, deutschsprachigen Online-Angebote finden sich unter [VDZ98].

[2] Die Definitionen der Zählmethoden von PageImpressions und Visits finden sich im Glossar und unter [VDZ98].

In Kapitel 3 wird, aufbauend auf der Studienarbeit von Nils Münch [Münch97], das Konzept des in dieser Diplomarbeit modellierten Systems im Detail vorgestellt.

Kapitel 4 schildert die für die Realisierung des Systems notwendigen Vorüberlegungen. Diese beinhalten unter anderem Entscheidungen bezüglich der Betriebs- und Entwicklungsumgebung. Weiterhin findet eine Auswahl zwischen verschiedenen objektorientierten Werkzeugen für die Modellierung des Systems statt.

In Kapitel 5 beginnt die Beschreibung der Modellierung des Systems nach der OMT-Methodologie von J. Rumbaugh. Zu Beginn des Kapitels wird eine kurze Einführung in die OMT-Methodologie gegeben. Inhalt der verbleibenden Abschnitte des Kapitels ist ein repräsentativer Ausschnitt aus dem OMT-Modell des Systems. Das vollständige Modell befindet sich in Band 2 zu dieser Arbeit.

Kapitel 6 schildert die in Ansätzen durchgeführte Implementation des Systems. Schwerpunkt dieses Kapitels bilden die Beschreibungen der Mechanismen für den Datenbankzugriff und die Generierung von Dokumenten. In diesem Kapitel werden die für die Generierung eines Dokumentes verwendeten Vorlagen beschrieben.

Abschließend wird in Kapitel 7 ein objektorientiertes Testverfahren für den Nachweis der Funktionalität des Systems vorgestellt.

2 Axel Springer Verlag AG

Die Axel Springer Verlag AG gehört mit einem Konzernumsatz von 4.599 Mill. DM und 12.195 Mitarbeitern zu den bedeutendsten Medienunternehmen in Europa.[1] Das Kerngeschäft des Verlagshauses wird traditionell mit Zeitungen, Zeitschriften und Spezialtiteln im In- und Ausland, mit Anzeigenblättern und Buchverlagen sowie eigenen Druckereien und Vertriebsorganisationen bestritten. Darüber hinaus hat der Verlag frühzeitig die wachsende Bedeutung der elektronischen Medien erkannt und entsprechend seine Aktivitäten auch in diese Bereiche ausgedehnt. Hier bestehen heute Beteiligungen im TV-Bereich (SAT.1, HAMBURG 1) und bei Hörfunksendern (z.B. Radio Hamburg). Mit Audiotex (CompuTel Telefonservice GmbH), Teletext (z.B. SAT.1 Text) und den Online-Diensten ist die Axel Springer Verlag AG zudem auch in den neuen elektronischen und interaktiven Medien erfolgreich engagiert. Insgesamt erwirtschaftet der Bereich elektronische Medien 1997 mit ca. 160 Mitarbeitern einen Umsatz von über 100 Mill. DM.

2.1 Aktivitäten der Axel Springer Verlag AG im Internet

Die Axel Springer Verlag AG ist zum heutigen Zeitpunkt im Internet mit den Online-Ausgaben von 13 Zeitungen [URL-ZT98] und Zeitschriften [URL-ZS98] präsent. Daneben gibt es umfangreiche Presse- und Fachinformationen sowie ein Unternehmensportrait des Verlagshauses [URL-ASV98]. Im Online-Bereich konnte die Axel Springer Verlag AG die Erfolge seiner Print-Objekte auch auf das Internet übertragen: „Bild online“ war im November 1998 mit 2.113.190 Visits und 5.578.147 PageImpressions unter den fünf erfolgreichsten deutschsprachigen Verlagsangeboten. Auch weniger auflagenstarke Print-Objekte sind im Online-Bereich erfolgreich vertreten. So erreichte zum Beispiel „WELT online“ im November 1998 759.215 Visits und 2.839.152 PageImpressions und das regionale Angebot „BZ auf Draht“ erreichte 885.214 Visits und 4.277.061 PageImpressions. Damit konnten sich diese beiden Angebote unter den erfolgreichsten 20 deutschsprachigen Verlagsangeboten plazieren.

Darüber hinaus betreibt die Axel Springer Verlag AG mehrere „special interest“ Angebote im Internet, zu denen es kein Print-Pendant gibt. Dies sind zum Beispiel „Sport 1“, „fussball.de“, „ticker.de“ und „verreisen.de“. Der Ende 1996 gestartete regionale Online-Dienst GO ON wurde im April 1998 mit dem Angebot "Cityweb" der Verlagsgruppe WAZ unter dem Namen "Cityweb Network" zusammengefaßt. Ende 1998 kündigte die Axel Springer Verlag AG jedoch an, sich aus diesem Joint-venture zurückzuziehen. Zusammen mit dem Ausstieg aus der 10%igen Beteiligung an „AOL Bertelsmann Online“ zieht sich der Verlag aus dem „Internet Providing“ Geschäft zurück und setzt seinen Schwerpunkt in den Online-Aktivitäten auf die Lieferung und Vermarktung von Inhalten („Content Providing“).

Zu diesem Zweck hat die Axel Springer Verlag AG unter anderem gemeinsam mit den Unternehmen T-Online, Verlagsgruppe Georg von Holtzbrinck und Infoseek Corporation die Firma WSI Webseek Infoservice GmbH & Co. KG gegründet. Dieses Joint-venture, an

[1] Siehe dazu "Geschäftsbericht 1997" unter http://www.asv.de/htm/3_0/3_1.htm

der die vier Firmen zu gleichen Anteilen beteiligt sind, wird im Frühjahr 1999 den deutschsprachigen Navigations- und Suchdienst Infoseek (www.infoseek.de) in Betrieb nehmen. Die Inhalte werden von den beiden Verlagshäusern geliefert. Der Zugang zu dem neuen Angebot wird unter anderem auf der Portal Site von T-Online plaziert.

2.2 Verwendete Internet-Technologien

Für die Entwicklung und den Betrieb der Web-Angebote des Verlages existieren keine unternehmensweiten Richtlinien in bezug auf die zu verwendenden Internet-Technologien. Dies hat historische Gründe. Innerhalb des Verlagshauses wird traditionell Wert auf die Unabhängigkeit der einzelnen Print-Objekte gelegt. Dies betrifft vorrangig die journalistische Unabhängigkeit der Redaktionen. Aber auch wirtschaftliche, konzeptionelle und technische Entscheidungen obliegen der Geschäftsführung des jeweiligen Print-Objektes. Dies hat auch Auswirkungen auf die entsprechenden Web-Angebote der Print-Objekte. Sie werden von den Print-Objekten sowohl inhaltlich als auch in Programmierung und Betrieb eigenverantwortlich betreut. So haben zum Beispiel „BILD“, „Sport BILD“ und „Computer BILD“ jeweils eine eigene Online-Redaktion mit jeweils einem Technik-Team für Programmierung und Betrieb des Web-Angebotes. Für den Start der einzelnen Angebote war es zunächst von ausschlaggebender Bedeutung, auch die Online-Produktion in die existierende Produktionsumgebung mit einzubetten. Also hat auch für die neuen Medien diese heterogene Struktur weiter Bestand.

Web-Auftritte, die nicht an ein Print-Objekt des Hauses gebunden sind, wie zum Beispiel das Unternehmensportrait, die Presse- und Fachinformationen oder „fussball.de“, „ticker.de“ und „verreisen.de“, werden zentral von der Abteilung „interactive media“ betreut.

Die ansonsten dezentrale Organisation der einzelnen Web-Angebote führte dazu, daß jedes Print-Objekt für den Betrieb des Web-Angebotes unterschiedliche Internet-Technologien einsetzt. Aus diesem Grund reicht beispielsweise das Spektrum der genutzten Datenbank Management Systeme vom einfachen Shareware-System wie „mSQL“ oder „MySQL“ bis hin zu professionellen Systemen wie „Oracle“. Entsprechendes gilt für die verwendeten Programmiersprachen: Von Java über Perl bis C++ sind für die Programmierung von Web-Server Funktionalitäten alle Facetten der Programmiersprachen und –paradigmen vertreten. Zusätzlich zu diesen serverseitig verwendeten Technologien werden clientseitige Technologien eingesetzt, wie zum Beispiel Macromedia Shockwave, Java-Applets und JavaScript.

Einzig für den Betrieb von Internet gestützten Anwendungen in der Axel Springer Verlag AG werden in der Regel SUN-Server mit Solaris als Betriebssystem eingesetzt. Für das Hosting der eingesetzten Server sind externe Unternehmen zuständig. So steht zum Beispiel der Web-Server des Hamburger Abendblattes im Rechenzentrum der Deutschen Telekom in Kiel.

3 Konzept eines Publikationssystems für personalisierte TV-Programminformationen

In diesem Kapitel wird das Konzept des im Rahmen dieser Diplomarbeit modellierten und in Ansätzen implementierten **P**ersonalisierten **E**lektronischen TV-**P**rogrammberaters (**PEP**) vorgestellt. PEP wurde 1997 im Rahmen einer Studienarbeit mit dem Titel „Analyse bestehender digitaler TV-Programminformationssysteme in interaktiven Medien“ [Münch97] am Fachbereich Informatik der Universität Hamburg entworfen. Dabei wurden an PEP folgende Kernanforderungen definiert:

PEP ist eine Internet basierte Anwendung, die TV-Programminformationen für ein breites Online-Publikum zur Verfügung stellt. Weiterhin bietet PEP den Benutzern die Möglichkeit, ein individuelles Profil einzurichten, um eine **personalisierte** TV-Programmzeitschrift im Abonnement per E-Mail beziehen zu können.

In den folgenden Abschnitten 3.2, 3.3 und 3.4 wird der Aufbau von PEP näher beschrieben. Zuerst werden jedoch in Abschnitt 3.1 die Nutzungsgewohnheiten und Erwartungen des Lesers an eine traditionelle TV-Programmzeitschrift dargestellt werden. Darauf aufbauend werden Möglichkeiten diskutiert inwieweit der Leser bereit ist, ein elektronisches Programminformationssystem zu nutzen und was dieses leisten müßte.

3.1 Traditionelle versus elektronische TV-Programminformationen

In einer traditionellen deutschen TV-Programmzeitschrift werden dem Leser wöchentlich ca. 12000 Sendungsbeiträge[1] präsentiert. Diese Informationsflut stellt nicht nur den Herausgeber der Programmzeitschrift bei der übersichtlichen Gestaltung vor große Probleme, sondern erschwert auch dem Leser das Auffinden aller seinen Interessen entsprechenden Sendungen. Die neuen Medien, speziell das Internet, bieten hier ideale Voraussetzungen, über datenbankgestützte Anwendungen die Suche nach interessanten Sendungen zu vereinfachen oder auch die Datenmenge auf ein benutzerfreundliches Maß zu reduzieren. Doch wie müßte ein Internet basiertes TV-Programminformationssystem aussehen, bzw. was muß es leisten können, um beim Verbraucher eine hohe Akzeptanz zu erreichen oder sogar die klassische Programmzeitschrift zu ersetzen?

Um dieser Frage auf den Grund zu gehen, soll zuerst betrachtet werden, welche Rolle Programmzeitschriften eigentlich spielen, wie diese genutzt werden und was der Leser von ihnen erwartet. Hierzu hat die Axel Springer Verlag AG 1994 mehr als 3.000 Bundesbürger nach ihren Fernsehgewohnheiten und ihrem Nutzungsverhalten von Programmzeitschriften befragt und die Ergebnisse in der Studie "Relations, Programmzeitschriften und Fernsehen" [ASV95] zusammengefaßt. Danach ist die mit Abstand wichtigste Programminformationsquelle die Programmzeitschrift (58% der Gesamtbefragten gaben dies an). Supplements, die größtenteils kostenlos Zeitungen und

[1] Hochrechnung: In der HÖRZU vom 22. Juni 1998 hatte das ARD-Programm 38 Sendungsbeiträge. Bei insgesamt 46 aufgeführten Sendern entspricht das 1748 Sendungen pro Tag oder 12236 pro Woche.

Zeitschriften beiliegen, Programmübersichten in der Tagespresse und Teletext werden zwar auch genutzt, haben aber keinen signifikanten Anteil (siehe Tabelle 1).

Informationsquelle	Prozent
Programmzeitschrift	58%
Supplements	12%
Hin- und Herschalten	9%
Tageszeitung	8%
Programmhinweise im Fernsehen	3%
Hinweise von Familienangehörigen	3%
Programmvorschau in anderen Zeitschriften	2%
Videotext bzw. Teletext	2%

Tabelle 1: Wichtigste Informationsquellen für das Fernsehprogramm

Der Erfolg von Programmzeitschriften ist unter anderem darin begründet, daß diese schon eine Woche vor Programmbeginn im Haushalt zur Verfügung stehen[1] und somit eine langfristige Fernsehplanung zulassen. 68% der Leser nutzen bereits in der Vorprogrammwoche den Programmteil, um rechtzeitig über das Programm der kommenden Woche informiert zu sein. Während der laufenden Programmwoche greifen dann nahezu alle Leser (95%) beinahe täglich darauf zurück (an 5,9 Tagen).

Die eigentliche Sendungsauswahl bestätigt ebenfalls, daß die Mehrheit der Fernsehzuschauer aktiv ihren Fernsehkonsum plant. So gaben 65% der Befragten an, sich ganz gezielt für eine bereits vorher ausgesuchte Sendung vor den Fernseher zu setzen. Die Mehrheit (73%) trifft sogar auch dann, wenn sie "einfach nur" fernsehen will, ganz bewußt eine Programmauswahl bevor sie das Gerät einschaltet. Nur 28% gaben an, so lange zwischen den Programmen hin- und herzuschalten, bis sie etwas Interessantes finden.

Auch in der weiteren Planung des Fernsehtages spielen Programmzeitschriften eine große Rolle. Jeder vierte der Befragten informiert sich während der laufenden Sendung oder danach in der Programmzeitschrift wie das weitere Programm aussieht. Interessant in diesem Zusammenhang ist auch, daß Programmzeitschriften in hohem Maße zu Hause gelesen werden (90%). Zum Vergleich: Aktuelle Zeitschriften erreichen hier nur 77%. Beim eigentlichen Lesen bzw. Studieren des Programms in einer Programmzeitschrift verhalten sich die Leser höchst individuell. Einzig das Verhältnis zwischen Lesern, die beim Lesen der Programminformationen mit dem TV-Listing eines Programmtages und denen, die mit den redaktionellen Tips beginnen, ist ausgewogen.

Bei den Erwartungen an eine Programmzeitschrift sind 66% der Leser der Meinung, daß außer dem Programmteil auch aktuelle Informationen und Unterhaltung, wie in einer Illustrierten, geboten werden sollte. Für 55% kommt eine Programmzeitschrift ohne einen ausführlichen Lese- und Unterhaltungsteil überhaupt nicht in Frage. Abschließend ist noch zu erwähnen, daß Programmzeitschriften viele Stammleser besitzen. Rund 80% der Programmzeitschriftenleser gaben an, sich an ihre Zeitschrift gewöhnt zu haben und daß es

[1] Bei der HÖRZU gelangen zum Beispiel 83% aller verkauften Exemplare bereits eine Woche vor Programmbeginn in die Haushalte.[ASV95].

ihnen sehr schwerfallen würde sich umzustellen. Damit ist die Leser-Blatt-Bindung deutlich höher als bei den meisten anderen Zeitschriftengattungen.

Werden die aus der Studie gewonnenen Ergebnisse nun unter dem Aspekt betrachtet, ein neues Internet basiertes TV-Programminformationssystem auf dem Markt zu positionieren, so scheint dieses auf den ersten Blick wenig erfolgversprechend. Rund 80% der Programmzeitschriftenleser ist überhaupt nicht bereit, die Informationsquelle zu wechseln. Auf das TV-Programm wird in der Programmwoche fast jeden Tag zugegriffen, und das auch zwischen den Sendungen, was natürlich voraussetzt, daß das TV-Programm schnell zu Hause bzw. vor dem Fernseher verfügbar ist. Auch in der Vorprogrammwoche wird schon ausgiebig darin gelesen. Der Leser erwartet neben den Programminformationen zudem einen Lese- und Unterhaltungsteil. Diesen intensiven und täglichen Nutzungsgewohnheiten der Leser, kann schon Aufgrund der hohen Telefongebühren und dem Aufwand des "Online Gehens" kein Internet basiertes System gerecht werden. Der vollständige Ersatz der klassischen Programmzeitschrift durch ein elektronisches Pendant, ist demnach gegenwärtig nur bei einer sehr kleinen Leserzahl vorstellbar.

Wird jedoch der Anspruch an ein Internet basiertes TV-Programminformationssystem dahingehend korrigiert, daß die elektronische Variante dem Leser nicht als Ersatz sondern als ergänzende Informationsquelle zur traditionellen Programmzeitschrift angeboten wird, so eröffnen sich ganz neue Perspektiven. Anstatt mit der traditionellen Programmzeitschrift zu konkurrieren, würden in der elektronischen Variante neue Leistungen, die die herkömmliche Programmzeitschrift so nicht bieten kann, angeboten. Zum Beispiel werden bei der Betrachtung einer Programmzeitschrift schnell zwei Nachteile deutlich. In den täglichen Tagestips wird versucht, die Interessen eines repräsentativen Publikums zu treffen; es können aber nicht die Interessen eines einzelnen Lesers berücksichtigen werden. Die TV-Programm-Listings sind so umfangreich, daß das Auffinden **aller** seinen Interessen entsprechenden Sendungen viel Zeit in Anspruch nimmt. Würde einem Leser/Benutzer in der Online-Version die Möglichkeit geboten, ein eigenes Profil mit den persönlichen Interessen einzurichten und abzuspeichern, so könnten unter Verwendung dieses Profils individuelle personalisierte TV-Programmtips für jeden Tag zusammengestellt werden. Damit der Benutzer diese persönlichen Programmtips, wie gewohnt, auch vor dem Fernseher jederzeit lesen kann, müßten sie nur noch in einem adäquaten Format zum Herunterladen und Ausdrucken vorliegen.

Eine noch weiter gedachte Idealvorstellung für dieses Szenario wäre sicherlich, daß die gesamten Programminformationen **und** die persönlichen Programmtips in einem zum Herunterladen geeigneten Format vorliegen. Aufgrund der zur Zeit noch geringen Bandbreite privater Internet-Zugänge und der hohen Druckkosten auf dem privaten Drucker, ist diese zur Zeit nicht umsetzbar. Wird aber der zum Herunterladen bereitgestellte Programmteil auf die personalisierten Programmtips reduziert, so kann der Benutzer als Ergänzung zu der wöchentlich abonnierten Programmzeitschrift seine eigene "Sonderausgabe" beziehen. Diese personalisierte Programmvorschau umfaßt nur noch wenige Seiten und könnte wöchentlich oder täglich per E-Mail an ihn verschickt werden. Die Online-Version ist damit nur als Plattform für das Einrichten der Profile zu betrachten, wenngleich auch nützliche Suchfunktionen usw. geboten werden können. Das

Kernprodukt ist aber die personalisierte Programmzeitschrift aus wenigen Seiten, die problemlos zu Hause ausdruckbar ist und dann zusammen mit der traditionellen Programmzeitschrift vor dem Fernseher genutzt werden kann.

Im folgenden Abschnitt wird nun das Konzept von PEP, ein System das genau diesen Anforderungen entspricht, vorgestellt.

3.2 Aufbau des Systems

Die Dienstleistungen des **P**ersonalisierten **E**lektronischen TV-**P**rogrammberaters (**PEP**) sowie die Teilbereiche des Systems sind in Abbildung 1 auf konzeptioneller Ebene dargestellt.

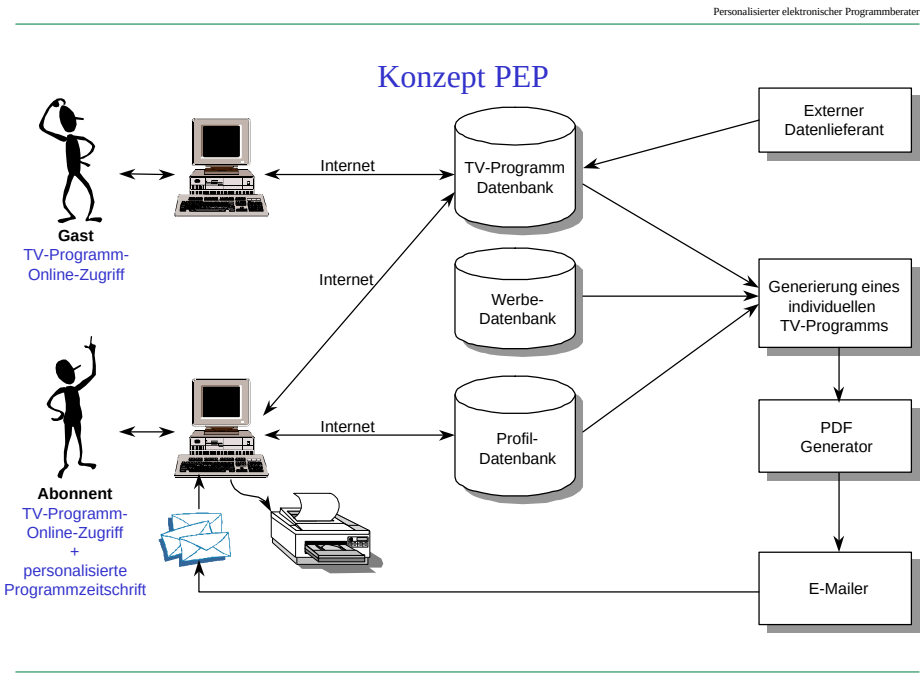


Abbildung 1: Aufbau von PEP

Deutlich ist die Aufteilung des Systems in den öffentlich zugänglichen Bereich ***Gast***- und den ***Abonnenten***-Bereich zu sehen, wobei der Dienstleistungsschwerpunkt auf dem nur für letztere zugänglichen E-Mail-Service in Form einer personalisierten TV-Programmzeitschrift liegt.

Dem ***Gast*** wird in PEP die Möglichkeit gegeben, die TV-Programmdatenbank zu benutzen und gegebenenfalls die dort enthaltenen Informationen über Selektions- oder Suchfunktionen einzugrenzen. Weitere Serviceleistungen, wie zum Beispiel das Abspeichern der individuell gewählten Einstellungen und Interessen in einem persönlichen Profil, bleiben dem Abonnenten vorbehalten. Im *Gast*-Bereich ist außerdem noch eine ausführliche Beschreibung der eigentlichen Dienstleistung von PEP, dem E-Mail-Service, und das dafür benötigte Anmelde- bzw. Abonnementformular.

Ein ***Abonnent*** betritt auf demselben Weg wie der *Gast* das System, hat aber dann die Möglichkeit über eine Login-Prozedur (Benutzername, Paßwort) in seinen persönlichen Bereich einzutreten. Dort erwartet ihn eine personalisierte Online-Version des TV-Programms, die anhand der individuellen Einstellungen seines in der **Profildatenbank** abgelegten persönlichen Profils generiert wird. Die beiden Online-Versionen *Gast* und *Abonnent* werden in den Abschnitten 3.3.1 und 3.3.2 detailliert beschrieben.

Die **TV-Programmdatenbank** beinhaltet sämtliche deutsche Programminformationen einer Woche (ca. 12.000 Beiträge), nach Sendern, Sparten, Regisseuren, Schauspielern usw. indiziert, und Bilddaten zu ausgewählten Beiträgen. Der komplette Inhalt eines Fernsehbeitrags ist in Anhang 9.10 dargestellt. Geliefert werden diese Daten von einem **externen Lieferanten** je nach gewünschter Aktualität täglich oder wöchentlich im ASCII-Format. Der Austausch der Daten in der Datenbank erfolgt immer komplett, d.h. die alten Daten werden gelöscht. Weiterhin beinhaltet PEP eine **Werbedatenbank**, in der die Werbebanner für die HTML-Seiten und für die TV-Programmzeitschrift verwaltet werden.

Den wichtigsten Punkt von PEP stellt die **Generierung der individuellen TV-Programmzeitschrift im PDF-Format** dar. In Abhängigkeit vom persönlichen Profil des Abonnenten wird von dem **PDF-Generator** eine layoutete Programmzeitschrift, z.B. im Stil der HÖRZU, erzeugt und täglich bzw. wöchentlich per E-Mail versandt. Diese kann dann vom Empfänger auf einem beliebigen Schwarzweiß- oder Farbdrucker in gleichbleibender Qualität gedruckt und anschließend wie eine herkömmliche Programmzeitschrift verwendet werden. Der personalisierte E-Mail-Service wird ausführlich in Abschnitt 3.4 beschrieben.

3.3 Online-Version

Die Online-Version besteht aus einem *Gast*- und *Abonnenten*-Bereich. Im *Gast*-Bereich werden lediglich Anzeige-, Selektions- und Suchfunktionen für das TV-Programm geboten, während im Abonnenten-Bereich die Möglichkeit besteht, die Programmdaten, unter Verwendung eines persönlichen Profils, individuell aufbereitet präsentiert zu bekommen.

3.3.1 Gast-Bereich

Ein *Gast* erhält in PEP auf **alle** TV-Programminformationen der aktuellen Woche freien Zugriff, so daß er sich vor dem Abonnieren der personalisierten TV-Programmzeitschrift, von der Leistungsfähigkeit des Systems überzeugen kann. Um dies zu erreichen, ist bei PEP auf eine übersichtliche und schnell zu erreichende TV-Programmübersicht, besonderer Wert gelegt worden. Als Darstellungsform wurde deshalb, entgegen anderen deutschen Online Programm-Informationssystemen[1] die eine Listendarstellung des TV-Programms anbieten, eine Tabellenstruktur verwendet (siehe Abbildung 2).

---

[1] Siehe dazu die Internet-Angebote deutscher TV-Programminformationssysteme [URL-TV97] und in [Münch97, Kap 4.2] "Beurteilung bestehender Programm-Informationssysteme".

Abbildung 2: PEP TV-Programmtabelle

Der in der Tabelle dargestellte zweistündige TV-Programmausschnitt aller Sender erlaubt optimale Vergleichsmöglichkeiten zwischen allen Beiträgen zu einer gewünschten Uhrzeit auf einer einzigen HTML-Seite. Für eine bessere Übersichtlichkeit sind die Tabellenfelder der einzelnen Beiträge zusätzlich mit dem über der Tabelle erläuterten Farbschlüssel nach Sparten eingefärbt. Mit den beiden Pfeilen Rechts und Links oben kann der zu betrachtende Zeitausschnitt beliebig "verschoben" werden. Für größere Sprünge können die gewünschten *Uhrzeiten* bzw. *Tage* auch direkt über die Select-Felder ausgewählt werden. Außerdem kann der Umfang der Tabelle noch über die Select-Felder *Kategorien* und *Sender* reduziert werden, indem z.B. durch Wahl der Kategorie *Spielfilm* nur Spielfilme in der Tabelle angezeigt werden. Eine Mehrfachauswahl verschiedener Kategorien oder das Einbeziehen von bevorzugten Genres wird nicht geboten. Die Möglichkeit, die TV-Programmtabelle exakt den eigenen Interessen anzupassen, wird nur den *Abonnenten* von PEP (siehe 3.3.2) geboten. Jeder Beitrag in der Tabelle ist zu einer HTML-Seite mit ausführlichen Informationen (Programmtext, Schauspieler, ShowView,...) und eventuell auch Bildmaterial „verlinkt“. Zusätzlich zu dieser Tabelle wird dem *Gast* über eine Volltextsuche die Möglichkeit geboten, direkt auf dem gesamten Datenbestand, nach z.B. Schauspielern oder bestimmten Sendungstiteln zu recherchieren.

Im *Gast*-Bereich befinden sich, zusätzlich zu dem TV-Programm, noch ausführliche Hinweise auf die Kerndienstleistung von PEP, den personalisierten E-Mail-Service und ein dazu erforderliches Anmelde- bzw. Abonnementformular. Darin wird der neue *Abonnent* aufgefordert, einen **Benutzernamen** und ein **Paßwort** seiner Wahl sowie seine **E-Mail Adresse** einzutragen. Diese Daten werden an PEP übertragen und dort, sofern der Benutzername und die E-Mail Adresse noch nicht in der Profildatenbank vergeben sind, in einem neu angelegten Profil abgespeichert. Bei erfolgreicher Anmeldung gelangt der neue *Abonnent* direkt in seinen persönlichen *Abonnenten*-Bereich.

3.3.2 Abonnenten-Bereich

Der *Abonnenten*-Bereich von PEP bietet dem Benutzer, gegenüber der für jeden zugänglichen *Gast*-Version, zwei entscheidende Vorteile. Die online in der Tabelle bereitgestellten TV-Programminformationen beinhalten in der Grundeinstellung nur noch die Fernsehbeiträge, die für den Abonnenten von Interesse sind; der Abonnent erhält täglich bzw. wöchentlich seine individuelle personalisierte TV-Programmzeitschrift.

Damit diese personalisierten TV-Programminformationen von PEP generiert werden können, muß der Abonnent zuerst seine Interessen und Fernsehgewohnheiten auf den

dafür angebotenen Profileinstellungsseiten eingeben. Hier können bevorzugte **Fernsehzeiten**, **Kategorien** und **Sparten** sowie **Sender** über Check-Boxen ausgewählt werden. Für eine möglichst präzise Beschreibung der persönlichen Interessen und als Entscheidungshilfe, ist dieser Bereich besonders umfangreich gestaltet. Zudem gibt es noch Eingabefelder, in denen der Abonnent persönliche **Schlüsselworte** in das Profil einbringen kann (z.B. Sean Connery, Synchronschwimmen,...). Neben diesen Konfigurationsmöglichkeiten, die direkt das TV-Programm betreffen, werden noch Eingabefelder und Check-Boxen angeboten um z.B. die Empfangsfrequenz der E-Mail von wöchentlich auf täglich umzustellen, die Tabellenfarben auszuschalten oder das Benutzerpaßwort zu ändern. Lediglich der Benutzername, der als Primärschlüssel für das Profil verwendet wird, kann vom Abonnenten nicht geändert werden. Der Inhalt eines Profils ist in der Tabelle 2 zu finden.

NAME	BESCHREIBUNG	BEISPIEL
Benutzername	vom Abonnenten gewählter Name	tvschlau
Paßwort	vom Abonnenten gewähltes Paßwort	**********
E-Mail-Adresse	E-Mail-Adresse des Abonnenten	mustermann@compuserve.com
Sender Nr.	vom Abonnenten gewählte Sender	1,2,4,5,....
Sparten Nr.	vom Abonnenten gewählte Sparten	2,7,9,...
Genre Nr.	vom Abonnenten gewählte Genres	5,17,18,22,...
Schlüsselworte	erweiterte Profilanpassung	Sean Connery,Synchronschwimmen,...
Uhrzeiten	individuelle Fernsehzeiten	7,8,19,20,21,22,23
Fernbedienung	eigene Fernbedienungstastenbelegung	1,n,4,3,5,6,2,n,14,....
E-Mail-Empfang	Flag für täglich T, wöchentlich W	W
VPS	VPS Zeit anzeigen (Flag Ja/Nein)	Ja
ShowView	ShowView Code anzeigen (Flag Ja/Nein)	Ja
Bundesland	für regionale Senderinformationen	Hamburg
Letzer Zugriff	Zeitstempel für den letzten Zugriff	01.01.1998, 17:30

Tabelle 2: Beispiel für den Inhalt eines Benutzerprofils

Nachdem der Abonnent sein persönliches Profil eingerichtet hat, wird als Grundeinstellung in der TV-Programmtabelle nicht mehr das ganze Programm, sondern nur das aus dem persönlichen Profil ermittelte individualisierte TV-Programm angezeigt (siehe Abbildung 3). Selbstverständlich kann der Abonnent durch Verändern der Optionen in den Select-Feldern die ausgeblendeten Beiträge oder Sender beliebig wieder aktivieren.

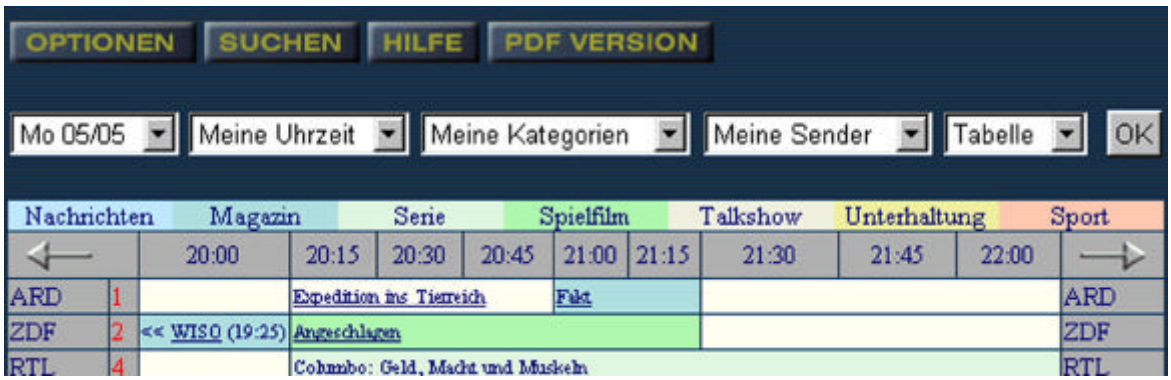


Abbildung 3: Personalisierte Online Version von PEP

Die oben links befindliche Schaltfläche ***Optionen*** führt zu den Profileinstellungen, so daß der Abonnent jederzeit Korrekturen bzw. Änderungen darin vornehmen kann. Über die Schaltfläche ***PDF-Version*** kann der Abonnent direkt, z.B. zu Testzwecken für die Optimierung des persönlichen Profils, seine individuelle TV-Programmzeitschrift im PDF-Format generieren lassen.

3.4 Personalisierter E-Mail-Service

Der Dienstleistungsschwerpunkt von PEP ist die Generierung individueller TV-Programmzeitschriften im PDF-Format mit anschließendem Versand an die *Abonnenten*. Die Voraussetzungen für den Betrieb dieser Serviceleistung wurden, bis auf die eigentliche Generierung im PDF-Format, bereits in den vorangegangenen Bereichen geschaffen. Das heißt, alle benötigten Programm-, Bild- und Werbedaten sowie die Profildaten mit allen Informationen zu den persönlichen Interessen der Abonnenten stehen für die Generierung der personalisierten Programmzeitschriften zur Verfügung. Da bis heute keine vollständige Bibliothek für das Generieren von PDF-Dokumenten direkt aus dem Programmfluß existiert, müssen hierfür alle benötigten Routinen noch entwickelt werden[1]. Dabei ist darauf zu achten, daß die generierten Dokumente über ein ausgefeiltes Layout und eine Begrenzung der Seitenzahlen auf eine für den Benutzer akzeptable Dateigröße gebracht werden. Eine wichtige Hilfe liefert hierfür auch die ab PDF-Version 1.2 eingeführte Datenkompression.

Das für die personalisierte TV-Programmzeitschrift gewählte Layout, bzw. die Anordnung der Beiträge, entspricht im wesentlichem dem Layout der Programmtips in einer traditionellen Programmzeitschrift. Die mit dem persönlichen Profil ermittelten Beiträge eines Tages werden, ohne Rücksicht auf die Senderzugehörigkeit, nur nach Anfangszeiten sortiert angeordnet (Abbildung 4). Entstehende Lücken zwischen den Sendungsbeiträgen werden mit zum Profil passender Werbung aufgefüllt.

[1] Es bestehen zwar zwei Bibliotheken zum dynamischen generieren von PDF-Dokumenten (C Bibliothek von Thomas Merz [URL-PDF-C], Perl Bibliothek [URL-PDF-Pl]), diese werden aber nur unzulänglich den hier gestellten Ansprüchen gerecht (z.B. keine Datenkompression).

Abbildung 4: Layout-Entwurf der personalisierten TV-Programmzeitschrift

Die hier abgebildete PDF-Seite hat eine Dateigröße von ca. 40 KB. Wird davon ausgegangen, daß ein überlegt konfiguriertes Profil zwischen einer halben bis einer Seite Programmtips pro Tag liefert, so ist bei einem wöchentlichen Versand mit vier bis sieben Seiten (160 bis 280 KB) zu rechen.

Für ausführlichere Informationen zur Entstehungsgeschichte von PEP und zu dem ersten konzeptionellen Entwurf wird auf die Studienarbeit [Münch97] verwiesen. Eine detaillierte Modellierung des Systems findet sich in Kapitel 5 dieser Arbeit, zuvor werden im folgenden Kapitel 4 die grundlegenden Entscheidungen für die Realisierung von PEP getroffen.

4 Vorüberlegungen für die Realisierung

Im Vorfeld der Realisierung dieser Diplomarbeit wurden seitens der Axel Springer Verlag AG verschiedene Rahmenbedingungen geschaffen. Die Kosten für die Entwicklung und den Betrieb des TV-Programminformationssystems sollten so gering wie möglich ausfallen und die Hausstandards für den Betrieb von Internet gestützten Anwendungen in der Axel Springer Verlag AG mußten eingehalten werden (vgl. Kap. 2.2). Die von der Axel Springer Verlag AG zur Verfügung gestellte Betriebs- bzw. Entwicklungsumgebung gab den Rahmen für die Entscheidungen bezüglich der zur Realisierung des TV-Programminformationssystems zu verwendenden Software.

Innerhalb dieses Kapitels werden Entscheidungen getroffen, die zumindest teilweise erst nach Beendigung bzw. während der in Kapitel 5 erläuterten Modellierung getroffen werden sollten. Dies betrifft vor allem die Auswahl der für die Realisierung zu verwendenden Programmiersprache und Datenbank-Systeme. Durch die Vorgaben, die von der Axel Springer Verlag AG getroffen wurden, blieb diesbezüglich allerdings ein extrem eingeschränkter Entscheidungsspielraum. Dementsprechend wurden einige Entscheidungen, nicht zuletzt aus finanzplanerischen Gründen vor der Modellierung getroffen.

4.1 Betriebsumgebung und Entwicklungsumgebung

Die Axel Springer Verlag AG stellte eine SUN Ultra Sparc II mit Solaris 2.6 als Entwicklungs- bzw. Betriebshardware zur Verfügung. Die Auswahl der für den Betrieb des TV-Programminformationssystems zu verwendenden Software beschränkte sich damit auf SUN/Solaris kompatible Applikationen. Ferner standen während der Entwicklungsphase zusätzlich vier PC-Systeme mit INTEL Prozessoren bereit, die jeweils zur Hälfte unter Windows95 und Linux betrieben wurden. So konnte für die Entwicklung auch Software zum Einsatz kommen, die nicht SUN/Solaris kompatibel war. Als Programmiersprachen für die Implementation wurden Java und C++ in Erwägung gezogen.

Für **Java**, als plattformunabhängige, von SUN entwickelte Programmiersprache, war die Forderung nach Solaris-Kompatibilität per se erfüllt. Für die alternativ zu Java zu verwendende Programmiersprache **C++** standen zwei mögliche Software-Pakete zur Auswahl. Der unter der GNU Public License stehende GNU C++ Compiler und der SUN C++ Compiler.[1] Ob für die Implementation des TV-Programminformationssystems Java oder C++ zum Zuge kommen sollte, wurde endgültig erst mit der Festlegung auf das zu verwendende Datenbanksystem festgelegt. Grundlegend mußte allerdings schon im Vorfeld geklärt werden, ob die Vorteile von Java oder die von C++ überwiegen würden. War also die Unabhängigkeit von der Zielplattform und die sehr strikte Umsetzung des objektorientierten Paradigmas in Java Grund genug, auf die performantere Alternative C++ zu verzichten? Da innerhalb der Axel Springer Verlag AG für internetbasierte

[1] Wie sich später herausstellte war der GNU C++ Compiler nicht kompatibel mit der verwendeten POET Datenbank (vgl. Kapitel 4.3), so daß der SUN C++ Compiler verwendet worden ist.

Anwendungen ausschließlich SUN/Solaris Plattformen eingesetzt wurden, konnte einer der entscheidenden Aspekte von Java – die Plattformunabhängigkeit – vernachlässigt werden. Grundlage für die spätere Entscheidung bildete somit die Performanz des fertigen Systems. Vor der Entscheidung für eine konkrete Programmiersprache stand jedoch die Suche nach einem geeigneten Modellierungswerkzeug, um das in Kapitel 3 beschriebene System modellieren zu können.

4.2 Modellierungs-Werkzeuge

Schon zu einem sehr frühen Zeitpunkt der Diplomarbeit wurde entschieden, daß für die Realisierung ein objektorientiertes Paradigma Verwendung finden sollte. Hauptsächlich, um die Wiederverwendbarkeit des Source-Codes zu erleichtern. Da diese Diplomarbeit eine Gemeinschaftsarbeit zweier Diplomanden ist, mußte das Gesamtsystem außerdem in zwei möglichst unabhängige Teilsysteme aufgebrochen werden. Diese Teilung bedingte unter anderem eine saubere Schnittstellen-Spezifikation zwischen den beiden Teilsystemen. Die Verwendung erprobter softwaretechnischer Methoden und Werkzeuge lag nahe, um die Erstellung eines konsistenten Systementwurfs und damit auch die Definition der benötigten Schnittstellen zwischen den einzelnen Teilbereichen, bzw. Moduln der Applikation zu realisieren.[1]

Der ursprüngliche Ansatz sah die Verwendung der **U**nified **M**odeling **L**anguage (UML) vor, die den aktuellen Standard für die objektorientierte Modellierung darstellt [Burkhardt97], zur Realisierung des Systementwurfs vor. UML kombiniert die drei anerkannten objektorientierten Methodologien von Jacobsen (Use Case) [Jacobsen92], Booch [Booch94] und Rumbaugh (**O**bject **M**odeling **T**echnique, OMT) [Rumbaugh94], wobei letztere die bisher am weitesten verbreitete objektorientierte Methodologie war.

Das Auffinden eines geeigneten Modellierungs-Werkzeuges (CASE-Tools) gestaltete sich allerdings sehr schwierig, da es sich bei UML um einen relativ jungen Standard handelt und die Auswahl dementsprechend gering ausfiel. Erschwerend kam hinzu, daß die wenigen am Markt erhältlichen CASE-Tools (Rational Rose, Paradigm Plus, Together/Professional) den für diese Diplomarbeit gesetzten Kostenrahmen gesprengt hätten. Ein Ausweichen auf eine andere Modellierungstechnik wurde somit unausweichlich. Bei der weiteren Recherche stellte sich heraus, daß speziell im kostengünstigen Shareware-Bereich die **O**bject **M**odeling **T**echnique (OMT, s. Kapitel 5.1) von Rumbaugh die größte Auswahl an entsprechenden CASE-Tools bot. Die Beurteilung der einzelnen CASE-Tools wurde dabei anhand folgender Kriterien vorgenommen.

1. Korrekte und optisch ansprechende Umsetzung der OMT-Notation
2. Exportmöglichkeit für Teilmodelle, zwecks Verwendung innerhalb des schriftlichen Teils der Diplomarbeit

[1] Um die durch eine Modellierung erhaltene Konsistenz über die Modellierungsphase hinaus zu erhalten, sollte ein Modellierungs-Werkzeug (CASE-Tool, **C**omputer **A**ided **S**oftware **E**ngineering) verwendet werden. Diese Art von „Entwicklungshilfe“ stellt unter anderem sicher, daß bei der Umsetzung des logischen Modells in Source-Code keine „Übertragungsfehler“ auftreten.

3. Source-Code-Generierung (C++, Java)
4. Bedienbarkeit
5. Preis

Die Beurteilung der Bedienbarkeit der Werkzeuge war hier ausschließlich von subjektiven Wahrnehmungen und Vorlieben geprägt, so daß keineswegs von einem objektiven „Test“ gesprochen werden kann. Da es aber vorrangig darauf ankam, mit möglichst geringem zeitlichen Aufwand ein geeignetes Werkzeug zu finden, wurde die Objektivität bezüglich der Bedienbarkeit an dieser Stelle vernachlässigt.[1]

Da die Software **Rational Rose** sowohl OMT als UML unterstützt, und um eine Referenz-Applikation für den Vergleich mit Shareware CASE-Tools zu erhalten, wird die Software Rational Rose außer Konkurrenz im folgenden Vergleich aufgenommen. **Rational Rose**, hergestellt und vertrieben von Rational Software Corporation, wäre unter Vernachlässigung des Kostenfaktors die beste Alternative gewesen. Die grafische Umsetzung der UML- bzw. OMT-Notation wurde gut gelöst. Der wesentliche Vorteil bestand jedoch in der Tatsache, daß die Implementierung des Programms von den oben genannten Vätern des UML-Standards stammt. Von einer korrekten und vollständigen Umsetzung des Standards war daher mit hoher Wahrscheinlichkeit auszugehen. Die Bedienbarkeit des Programms ließe sich, im Hinblick auf die Integration des enormen Funktionsumfanges der Software, sicher noch verbessern.

Visio, ein Produkt der Visio Corporation, deckte nur Teilaspekte der Funktionalitäten eines CASE-Tools ab. Trotzdem wurde es, wegen seiner hervorragenden grafischen Möglichkeiten, in diese Betrachtung aufgenommen. Das Werkzeug ist vornehmlich für die Erstellung von Diagrammen jeglicher Art geeignet, so daß lediglich der grafische Aspekt der Modellierung (Visual Modeling) hätte umgesetzt werden können. Über OLE (**O**bject **L**inking and **E**mbedding) ermöglicht Visio die Verwendung des erstellten Modells in anderen Applikationen. Die Umsetzung der UML- bzw. OMT-Notation wurde allerdings nicht vollständig vorgenommen und Visio bot außerdem keine Möglichkeit, aus dem erzeugten Modell Source-Code zu generieren. Diese fehlenden Eigenschaften gaben letztendlich den Ausschlag für eine Entscheidung gegen Visio.

WithClass 97 von MicroGold war das erste betrachtete CASE-Tool aus dem Shareware-Bereich. Es unterstützte, wie alle hier betrachteten Shareware-Tools, lediglich die OMT-Notation. Alle Facetten der Notation wurden abgedeckt, allerdings ließ die grafische Umsetzung der OMT-Symbolik zu wünschen übrig und die Benutzeroberfläche war wenig intuitiv implementiert. WithClass bot ferner keine Möglichkeit, die erstellten visuellen Modelle zu exportieren. Ein Mangel, der auch nicht durch die vollständige Umsetzung der OMT-Notation ausgeglichen werden konnte.

Object Domain, von Object Domain Systems, überzeugte vor allem durch eine durchdachte Benutzerführung, eine korrekte Umsetzung der OMT Methodologie und nicht zuletzt durch einen günstigen Preis. Allerdings wurde die sogenannte funktionale

[1] Die hier betrachteten Werkzeuge stellen nur eine Auswahl dar. Eine komplette Liste aller untersuchten Tools findet sich in Kapitel 9.9.

Modellierung (vgl. Kapitel 5.1) in Object Domain nicht berücksichtigt. Auch die Umsetzung eines Teiles der dynamischen Modellierung ließ zu wünschen übrig. Aus diesem Grund wurde Visio, entgegen der vorherigen Entscheidung, für Teilaspekte des „visual modeling“ verwendet. Die Generierung des Source-Codes wurde über die Skriptsprache TCL gelöst. Die mitgelieferten Skripten konnten leicht auf die eigenen Bedürfnisse angepaßt werden. Ein Test der Fachhochschule Lübeck ermittelte außerdem Object Domain als beste und günstigste Alternative der objektorientierten CASE-Tools für Windows [URL-OOC97]. Dieser Test gab letzten Endes den Anreiz für die Verwendung von Object Domain als Modellierungs-Werkzeug.

	Visio	**Object Domain**	**WithClass**	**Rational Rose**
Version	**4.5**	**1.19**	**2.5**	
Hersteller	Visio Corporation	Object Domain Systems	Microgold	Rational Software Corporation
Methodologien				
UML	ja	nein	nein	ja
Booch	nein	ja	ja	ja
OMT	ja	ja	ja	ja
Use Case	ja	nein	nein	ja
Code-Generierung				
C++	nein	ja[1]	ja	ja
Java	nein	ja[1]	ja	ja
Vollständigkeit[2]	nein	nein	ja	ja
Export	OLE	Bitmap	nein	OLE
Bedienbarkeit	++	+	-	+
Preis pro Lizenz	DM 349,-	DM 168,30	DM 331,50	DM 9974,-

[1] Über Scriptsprache TCL programmierbar [2] Vollständige Umsetzung der OMT-Notation

Tabelle 3: Vergleich objektorientierter CASE-Tools

Allen hier betrachteten CASE-Tools aus dem Shareware-Bereich haftete der Nachteil an, daß das Hauptaugenmerk auf die visuelle Modellierung gelegt wurde. Eine durchgehende Umsetzung der gesamten Methodologie (vgl. Kapitel 5.1), von der Analyse, über Objektmodellierung bis zur funktionalen Modellierung, war in keinem der Werkzeuge außer Rational Rose realisiert. Die Einhaltung der logischen und semantischen Konsistenz des Systementwurfs lag somit vollständig in der Verantwortung der Entwickler. Auf Hilfestellungen, die ein Werkzeug wie Rational Rose geboten hätte, mußte weitgehend verzichtet werden. Dies betraf unter anderem eine möglichst fehlerfreie Übertragung der Analyseergebnisse in das OMT-Modell, aber auch die Verknüpfung der einzelnen Teilmodelle, die während der Modellierungsphasen entstehen. Einer der gravierendsten Nachteile, der sich allerdings erst zu einem späteren Zeitpunkt herausstellte, war die fehlende Anbindung der Shareware-Tools an das verwendete Datenbank-System. Für Rational Rose war beispielsweise ein Add-On erhältlich, mit dessen Hilfe direkt aus dem OMT-Modell die Struktur einer POET-Datenbank generiert werden konnte. Diese oder ähnliche Funktionalitäten fehlten den anderen CASE-Tools. Trotzdem konnte mit Object Domain die in Kapitel 5 erläuterte Modellierung realisiert werden.

4.3 Datenbanken

Die Verwendung eines objektorientierten Paradigmas bei der Realisierung des TV-Programminformationssystems einzusetzen, beeinflußte auch die Auswahl der Datenbank. Der wohl entscheidende Vorteil einer objektorientierten Datenbank ist der fehlende Paradigmenwechsel – objektorientiert ↔ relational – zwischen Programmiersprache und Datenbank. Diese Paradigmengleichheit versprach erstens eine erhebliche Zeitersparnis bei der Implementation, zweitens würde auf diese Weise eine weitere Fehlerquelle eliminiert, nämlich die Übertragung des persistenten Anteils des Objektmodells in ein relationales Datenbankmodell. Die Verwendung einer „ausgereiften" relationalen Datenbank fand im Hause allerdings sehr viel größeren Anklang, als die Verwendung einer innerhalb der Axel Springer Verlag AG weitgehend neuen Technologie, wie eine objektorientierte Datenbank. Speziell mit dem System von Oracle wurden in internetbasierten Anwendungen bereits einige Erfahrungen gesammelt, so daß diesem System prinzipiell der Vorrang gegeben wurde. Oracle unterbreitete mehrere Vorschläge für den Erwerb verschiedener Oracle Softwarepakete, die die Realisierung des Systems vorantreiben sollten. Das endgültige, nach einem Zeitraum von etwa acht Wochen, unterbreitete Angebot von Oracle wies einen Preis von ca. DM 750.000,- aus.[1]

Parallel zu den Verhandlungen mit Oracle wurde Kontakt zu Herstellern objektorientierter Datenbanken aufgenommen. Die Hersteller **Object Design** und **POET** (**P**ersistent **O**bjects and **E**nhanced Database **T**echnology) unterbreiteten jeweils Angebote für ihre Datenbank-Systeme **Object Store** bzw. **POET**. Im Verlauf der folgenden Verhandlungen mit Vertretern dieser beiden Firmen kristallisierte sich eine Entscheidung zugunsten einer der beiden objektorientierten Datenbanksystem heraus. Dies ist nicht zuletzt auf die mangelnde Flexibilität des Consultings der Firma Oracle zurückzuführen. Auch der Sachverhalt, daß beide Angebote für die objektorientierten Systeme ein deutlich niedrigeres Preisniveau als das Angebot von Oracle hatten, lenkte die Entscheidung in diese Richtung.

Insgesamt vier Kriterien sollten eine Entscheidung zwischen den beiden Kandidaten Object Store und POET ermöglichen.

1. Unterstütze Programmiersprachen
2. Integration der Programmiersprachen in die Datenbank API
3. Qualität des Consulting
4. Preis

Über die Qualität des Consulting (Punkt 3) sollten Rückschlüsse auf die Unterstützung des Herstellers während der Entwicklung gezogen werden. Diese Unterstützung nahm einen hohen Stellenwert bei der Entscheidung ein, da innerhalb der Axel Springer Verlag AG keinerlei Erfahrungen mit objektorientierten Datenbanken gesammelt worden sind. Es war daher mit einigen programmiertechnischen Schwierigkeiten während der Entwicklung zu

[1] Dieser Preis beinhaltete allerdings, aufgrund eines Mißverständnisses seitens Oracle, ein Angebot für die Implementierung des gesamten Systems PEP. Die Einzelpreise für die benötigten Datenbank-Komponenten waren diesem Angebot nicht zu entnehmen.

rechnen, die durch eine qualitativ hochwertige Unterstützung seitens des Datenbankherstellers möglichst schnell und reibungslos beseitigt werden sollten.

Als zu unterstützende Programmiersprachen wurden ausschließlich Java und C++ vorausgesetzt. Beide Kandidaten boten sowohl Java- als auch C++-Unterstützung und die Integration in die jeweilige API verfolgte zwar unterschiedliche, aber ähnlich unkomplizierte Ansätze. Die Auswahlkriterien wurden somit auf die Punkt 3 und 4 reduziert. Das Consulting der Firma POET erschien um einiges flexibler als das der Firma Object Design. Das mag zum Teil daran gelegen haben, daß es sich bei POET im Vergleich zu Object Design um eine relativ kleine Firma handelt, die an der Axel Springer Verlag AG als Referenzkunden sehr interessiert war. Dieser Sachverhalt war aber zugleich auch ein weiteres Entscheidungskriterium für POET, da damit gerechnet werden konnte, daß POET an einer erfolgreichen Zusammenarbeit mit der Axel Springer Verlag AG gelegen war und sich die hervorragende Zusammenarbeit mit dem Consulting auf die Unterstützung während der Entwicklung übertragen würde. Die Tatasache, daß der Großteil des technischen Personals von POET in Hamburg ansässig war, versprach eine schnelle Abwicklung von Support-Anfragen und damit eine schnelle Lösung von programmiertechnischen Schwierigkeiten. Da das POET Datenbanksystem im direkten Preisvergleich zusätzlich als eine um den Faktor zehn günstigere Alternative gegenüber Object Store hervortrat, wurde dem POET System der Vorrang gegeben.[1]

Im Verlauf der Verhandlungen mit den beiden Herstellern wurden auch technische Einzelheiten der beiden Datenbanksysteme erörtert. Obwohl sie letztendlich keinen Einfluß auf die Entscheidung hatten, sollen sie an dieser Stelle erwähnt werden, um einen objektiveren Maßstab für die Unterschiede zwischen POET und Object Design zu gewinnen.

Der bedeutendste Unterschied zwischen den beiden betrachteten Datenbanken ist die Methode, mit der persistente Objekte in der Datenbank gespeichert werden. Bei Object Store handelt es um einen sogenannten **Page-Server** (s. Abbildung 5). Der Datenbank-server speichert alle Objekte innerhalb eines Abbildes einer Seite des Arbeitsspeichers (page). Auf diese Weise wird es dem Server ermöglicht, bei einem Lesevorgang aus der Datenbank eine komplette Speicherseite in den Arbeitsspeicher des Rechners zu laden. Da verschiedene Instanzen einer Klasse häufig innerhalb einer einzelnen solchen Speicherseite zu finden sind, werden mehrere schreibende Speicherzugriffe eingespart, wenn auf Instanzen der bereits geladenen Seite zugegriffen wird. Ein clientseitiges Cache-Management verwaltet die geladenen Speicherseiten und regelt den Zugriff auf die einzelnen Objekte. Der Nachteil dieser Technik besteht in der womöglich unnötigen Übertragung einer kompletten Speicherseite an den Client, immer dann wenn vom Client nur einzelne, nicht zusammenhängende Objekte benötigt werden, die nicht innerhalb einer Speicherseite plaziert sind. Im schlimmsten Fall überträgt der Server für jedes angeforderte Objekt eine komplette Speicherseite.

---

[1] Der Preis des POET ODBMS lag bei ca. DM 25.000,- (Windows- und Solaris-Version). Object Design verlangte für das Object Store ODBMS ca. DM 300.000,- (Solaris-Version).

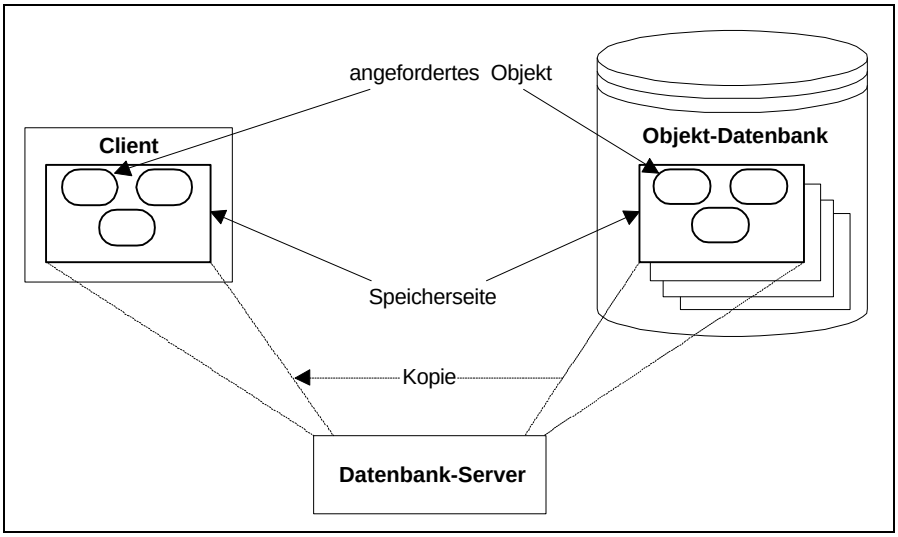


Abbildung 5: Page-Server

Diesen Nachteil der unnötigen Übertragung von Daten über eine Client-Server Verbindung umgeht POET indem es nur die Daten an den Client übermittelt, die tatsächlich benötigt werden. POET ist ein sogenannter **Object-Server.** Das heißt, der POET Datenbankserver überträgt alle angeforderten Objekte einzeln an den Client (s. Abbildung 6).

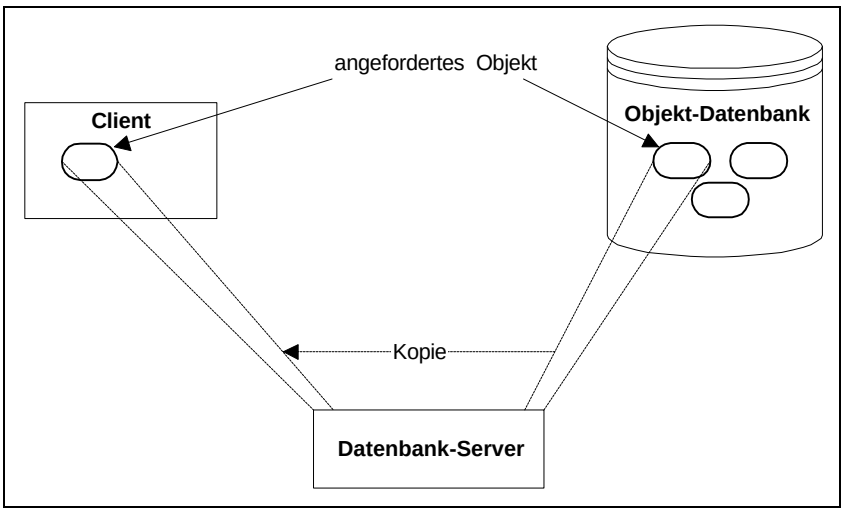


Abbildung 6: Object-Server

Die Entscheidung, ob für ein gegebenes Szenario ein Page-Server oder ein Object-Server die bessere Wahl ist, hängt vom jeweiligen applikationsspezifischen Engpaß („bottleneck“) ab. Wird die Geschwindigkeit einer Applikation in höherem Maße von Speicherzugriffen beeinflußt, ist ein Page-Server die performantere Alternative. Ist die Menge der über eine Client-Server Verbindung (socket) übertragenen Daten der entscheidende Faktor für die Performanz der Applikation, sollte ein Object-Server Verwendung finden.

Beide Systeme bieten, unabhängig von der zugrundeliegenden Technologie, die Möglichkeit, Objekte in ihrer, innerhalb der Applikation genutzten Ausprägung („object as

is“), direkt in einer Datenbank zu speichern. Die so erreichte Persistenz eines Objektes ist in beiden Systemen unabhängig von der verwendeten Programmiersprache. Ein in C++ deklariertes und innerhalb einer C++-Applikation gespeichertes Objekt kann ohne Probleme auch in einer Java-Applikation aus der Datenbank gelesen und sofort verwendet werden.

Unter den in diesem Kapitel erarbeiteten Voraussetzungen konnte mit der Modellierung des Publikationssystems begonnen werden. Das folgende Kapitel schildert den Modellierungsprozeß und die Ergebnisse, die während der Modellierung erarbeitet worden sind.

5 Modellierung

In diesem Kapitel wird die Modellierung des in Kapitel 3 auf konzeptioneller Ebene beschriebenen Systems PEP vorgestellt. Die Modellierung wurde auf Basis der OMT-Methodologie vorgenommen und bildet den Schwerpunkt dieser Arbeit. Zunächst wird in Kapitel 5.1 eine Einführung in die OMT-Methodologie gegeben. Anschließend wird die Modellierung von PEP beschrieben. Die Beschreibung wird in die drei Phasen der OMT-Methodologie Problemanalyse, Systementwurf und Objektentwurf gegliedert (Kapitel 5.2, 5.3 und 5.4). In diesem drei Abschnitten werden jeweils wesentliche Teilbereiche der einzelnen Modellierungsphasen vorgestellt. Eine vollständige Beschreibung aller Teilbereiche hätte den Rahmen dieser Diplomarbeit gesprengt. Das komplette OMT-Modell des Systems PEP findet sich im Ergänzungsband zu dieser Diplomarbeit.

5.1 Einführung in OMT

Die "Object Modeling Technique" (OMT) wurde Ende der 80er Jahre von J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy und W. Lorensen bei der Firma *General Electric* entwickelt und 1991 in ihrem Buch "Object-Oriented Modeling and Design" [Rumbaugh91] veröffentlicht. Die darin vorgestellte OMT-Methodologie unterstützt den gesamten Lebenszyklus der objektorientierten Softwareentwicklung von der **Analyse** über den **Entwurf** bis hin zur **Implementierung**. Im folgenden soll eine kurze Einführung in die OMT-Methodologie, die darin verwendete Modellierungsnotation und den eigentlichen Modellierungsprozeß gegeben werden. Eine vollständige Beschreibung findet sich in [Rumbaugh94].

5.1.1 Methodologie

Die Kernidee der OMT-Methodologie besteht darin, für ein zu entwickelndes System zuerst ein abstraktes, programmiersprachenunabhängiges Modell zu modellieren (Analyse) und diesem erst anschließend Implementierungsdetails hinzuzufügen (Entwurf). Das Analysemodell stellt dabei in einer kompakten, präzisen Abstraktion dar, was das System leistet, nicht aber, wie dieses geschieht. Erst im Entwurf wird dieses Modell durch Details, wie z.B. Datenstrukturen und Algorithmen, erweitert. Der Vorteil dieser Verfahrensweise ist, daß dieses Modell quasi eine Kommunikationsschnittstelle zwischen Anwendern und Entwicklern darstellt, die beide Seiten verstehen und kritisieren können. Konzeptionsfehler können so frühzeitig erkannt und behoben werden. Nachdem dies sichergestellt ist, bietet das Analysemodell durch hinzufügen von Implementierungsdetails die Grundlage für den eigentlichen Entwurf des Systems. Folgende vier Phasen sind in der OMT-Methodologie für die Softwareentwicklung vorgesehen:

Analyse:
In der Analyse wird, ausgehend von einer Problembeschreibung, ein Modell der

"realen Welt" mit deren wichtigsten Eigenschaften erstellt. Dieses Analysemodell[1] ist eine kompakte, präzise Abstraktion dessen, was das zu entwickelnde System leisten muß. Die Objekte im Modell entstammen der Anwendungsdomäne, Implementierungskonzepte sollen darin nicht enthalten sein.

Systementwurf:
Im Systementwurf werden vor Beginn des detaillierten Objektentwurfs grundlegende Entscheidungen über die Gesamtarchitektur des Systems getroffen. Anhand dieser Architektur und der Analysestruktur wird das System in Teilsysteme unterteilt. Es werden Nebenläufigkeiten identifiziert, Teilsysteme auf Prozessoren und Tasks verteilt, Hard- und Software bzw. benötigte Ressourcen ausgewählt und die Strategie der Datenverwaltung geklärt. Zudem werden noch Randbedingungen wie z.B. Systemfehler, Systemterminierung und -initialisierung sowie Anwortzeiten behandelt.

Objektentwurf:
Der Objektentwurf setzt auf die Analyse und den Systementwurf auf. Dabei wird das Analysemodell durch spezifische Implementierungsdetails, wie z.B. Entwerfen von Datenstrukturen und Algorithmen oder Hinzufügen von zusätzlichen internen Klassen, Attributen und Assoziationen, erweitert. Ziel ist es, das Modell so weit zu verfeinern und zu erweitern, bis es detailliert genug für die Implementierung ist.

Implementierung:
Übersetzung der entwickelten Objektklassen und Relationen in eine bestimmte Programmiersprache, Datenbank oder Hardware.

Die OMT-Methodologie verwendet zur Beschreibung der einzelnen Entwicklungsphasen **jeweils drei** Modelle, die im wesentlichen aus erweiterten Entity-Relationship-Diagrammen [Chen76], Zustandsdiagrammen [Harel87] und Datenflußdiagrammen [Yourdon89], [DeMarco79] und [Gane78] bestehen. Diese drei Modelle (nach OMT: Objektmodell, dynamisches Modell und funktionales Modell) werden erstmals in der Analysephase entwickelt und anschließend in den Entwurfsphasen bis zur Implementierungsfähigkeit verfeinert. Jedes dieser drei Modelle fängt die Aspekte des Systems unter einem anderen Blickwinkel ein, für eine vollständige Beschreibung sind alle Modelle erforderlich.

Objektmodell
Das Objektmodell beschreibt die statische Struktur der Objekte in einem System, die Relationen zwischen den Objekten und die Attribute sowie Operationen, die jedes Objekt charakterisieren. Unter einem Objekt ist dabei ein Konzept, eine Abstraktion oder eine Sache mit präzisen Abgrenzungen und einer klaren Bedeutung für eine Anwendung zu verstehen. Das Objektmodell wird grafisch durch Objektdiagramme (Klassendiagramme) repräsentiert (siehe Abbildung 7). In

[1] Genaugenommen handelt es sich bei dem Analysemodell um drei Modelle (Objektmodell, dynamisches Modell, funktionales Modell), die jeweils andere Sichten auf das System repräsentieren (siehe folgende Absätze).

der OMT-Methodologie fällt dem Objektmodell die größte Bedeutung zu, es liefert den Bezugsrahmen für das dynamische und funktionale Modell.

Dynamisches Modell
Das dynamische Modell beschreibt jene Aspekte des Systems, die mit Zeit bzw. Interaktionsreihenfolgen verbunden sind. Hier werden die Ereignisse zwischen den einzelnen Objekten betrachtet und in Ereignisfolgen alle erreichbaren Zustände identifiziert und auf Konsistenz verifiziert. Das dynamische Modell definiert die Reihenfolge der Operationen des Systems. Grafisch beschrieben wird es durch Zustandsdiagramme (siehe Abbildung 8), Ereignispfad- und Ereignisflußdiagramme sowie Szenarien.

Funktionales Modell
Das funktionale Modell beschreibt die Datenwert-Transformationen innerhalb eines Systems. Es beschreibt die funktionalen Beziehungen der von einem System berechneten Werte sowie Einschränkungen und funktionale Abhängigkeiten derselben. Es zeigt jedoch nicht, wann oder wie diese berechnet werden. Grafisch wird das funktionale Modell mit Datenflußdiagrammen beschrieben (siehe Abbildung 9).

Das Objektmodell spezifiziert also den Bezug bzw. **auf wen oder was** etwas geschieht, das dynamische Modell spezifiziert **wann** etwas geschieht und das funktionale Modell spezifiziert **was** passiert. Die drei Modelle unterteilen ein System dabei in orthogonale Sichten, die weitgehend für sich betrachtet und verstanden werden können, obwohl sie nicht vollständig voneinander unabhängig sind. So beschreibt das Objektmodell, welche Objekte, Attribute und Verknüpfungen in einem System existieren und liefert damit den Bezugsrahmen für die Operationen der beiden anderen Modelle. Es verdeutlicht für das dynamische Modell, welche Objekte ihren Zustand ändern und auf welchen Operationen ausgeführt werden. Für das funktionale Modell zeigt es die Struktur der Handlungsobjekte, Datenspeicher und -flüsse. Darauf aufbauend spezifiziert das dynamische Modell für alle Objekte aus dem Objektmodell die erreichbaren Zustände und durchgeführten Operationen bei Eintreffen eines Ereignisses. Weiterhin definiert das dynamische Modell die Reihenfolge, in der die Prozesse des funktionalen Modells – diese entsprechen den Operationen auf den Objekten des Objektmodells – durchgeführt werden. Das funktionale Modell beschreibt dann abschließend die Operationen auf den Klassen und die Argumente jeder Operation, sowie die Einschränkungen im Objektmodell und der Aktionen im dynamischen Modell. Diese drei Modelle werden in jeder OMT-Phase weiter verfeinert und dann für die Implementierung vereint.

5.1.2 OMT-Notation

Die Objekt Modeling Technique verwendet für das Objektmodell, das dynamische und funktionale Modell jeweils eine eigene Notation, die in jeder Entwicklungsphase der Methodologie beibehalten wird.

Objektmodell

Die Notation für das Objektmodell ist im wesentlichen auf das Entity-Relationship-(ER)-Modell von [Chen76] zurückzuführen. Im folgenden werden die in der Modellierung von PEP verwendeten Notationen der Objektdiagramme (grafische Repräsentation des Objektmodells) vorgestellt (siehe Abbildung 7) und kurz erläutert. Eine vollständige Beschreibung der Konzepte und Auflistung der Notation für das Objektmodell findet sich in [Rumbaugh94, Kap. 3,4 und S.548-549].

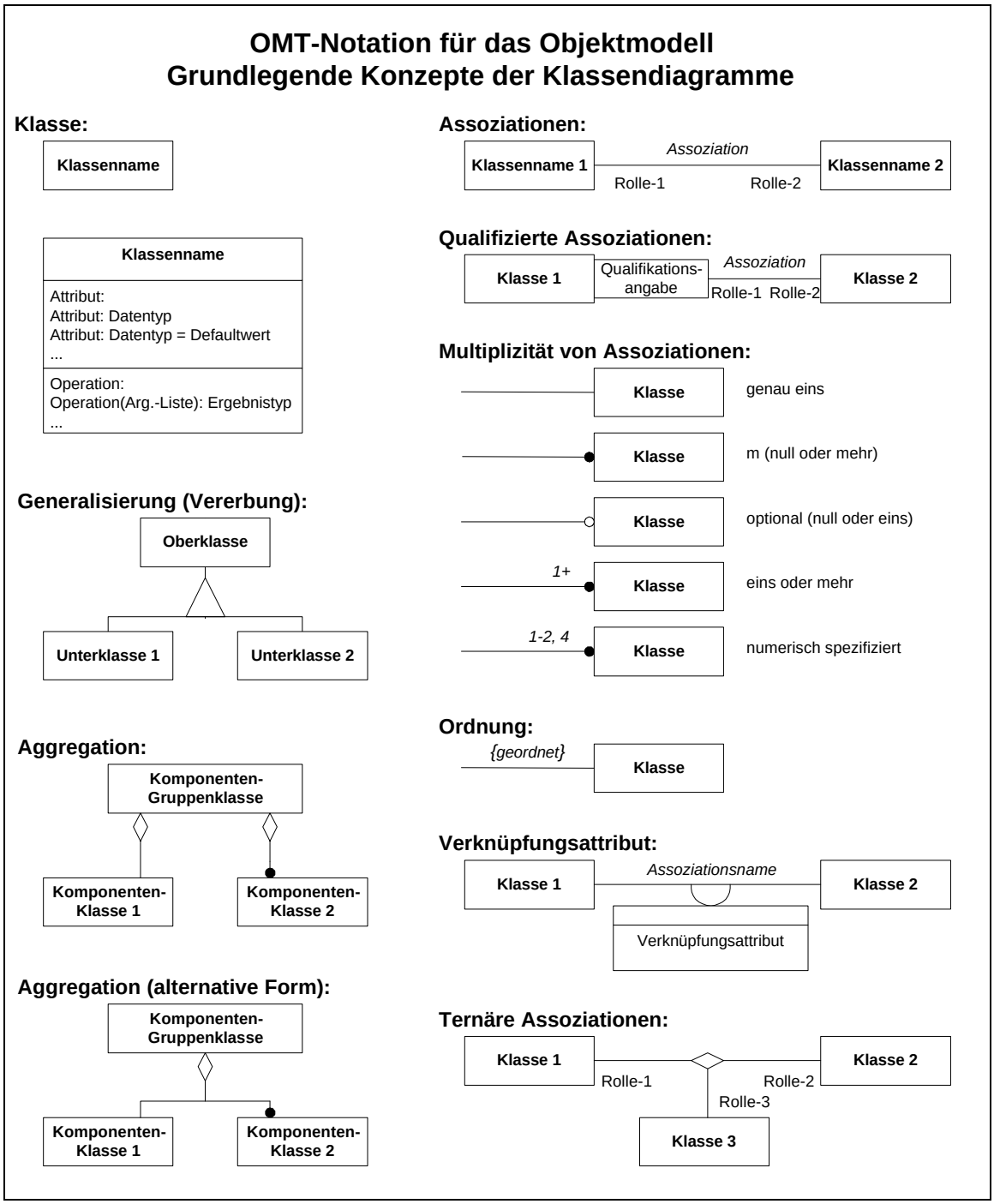


Abbildung 7: Ausschnitt der OMT-Notation für das Objektmodell [Rumbaugh94]

Objektdiagramm	Objektdiagramme stellen eine formale grafische Notation für die Modellierung von Objekten, Klassen und ihren Relationen zueinander bereit. Es gibt zwei Arten von Objektdiagrammen: *Klassendiagramme* und *Instanzdiagramme*. *Klassendiagramme* beschreiben Objektklassen, *Instanzdiagramme* Objektinstanzen.
Objektklasse	Bzw. **Klasse**, beschreibt eine Gruppe von Objekten mit gleichen Eigenschaften (→*Attributen*), gleichem Verhalten (→*Operationen*) und gleichen Relationen (→*Verknüpfung*, →*Assoziation*) zu anderen Objekten (z.B. Sender).
Objektinstanz	Bzw. **Objekt**, beschreibt ein Objekt (eine Instanz) einer Klasse (z.B. ARD).
Attribut	Datenwert, den die Objekte einer Klasse besitzen.
Operation	Eine Funktion oder Transformation, die von Objekten einer Klasse verwendet werden kann.
Assoziation	Eine Assoziation beschreibt eine Gruppe von →*Verknüpfungen* mit gleicher Struktur und gleicher Bedeutung (vgl. Unterschied zwischen einer →*Objektinstanz* und einer →*Objektklasse*)
Verknüpfung	Eine Verknüpfung ist eine physikalische oder konzeptionelle Verbindung zwischen zwei (oder mehreren) →*Objektinstanzen*
Rolle	Ein Rollenname identifiziert eindeutig das Ende einer Assoziation.
Qualifikationsangabe	Ist ein Attribut einer →*Assoziation*, das die effektive Multiplizität einer Assoziation herabsetzt. Qualifiziert werden können 1:m und m:m Assoziationen.
Multiplizität einer Assoziation	Spezifiziert wieviele Instanzen einer Klasse mit wievielen Instanzen einer anderen Klasse assoziiert sein können.
Verknüpfungsattribut	Ist eine Eigenschaft einer →*Assoziation*.
Generalisierung	Ist die Beziehung einer Oberklasse und einer oder mehreren Unterklassen, die Verfeinerungen der Oberklasse darstellen. Die in Unterklassen identischen Attribute und Operationen werden diesen entnommen und in einer gemeinsamen Oberklasse vereint. Der gegensätzliche Vorgang zur Generalisierung ist die **Vererbung**, dort werden von einer Oberklasse ausgehend verfeinerte Unterklassen definiert.
Aggregation	Ist die "Teil-Ganzes" oder "ist-Teil-von" Relation, in

	der Klassen als Komponentenklasse einer Komponentengruppenklasse repräsentiert werden. Als wichtigstes Merkmal der Aggregation ist die Transitivität zu erwähnen. Wenn A ein Teil von B ist und B ein Teil von C, dann ist auch A ein Teil von C, wobei A, B und C Objektklassen darstellen.
Ordnung	Bietet die Möglichkeit Objekte auf der "m"-Seite einer Assoziation explizit zu ordnen.

Für die Modellierung von PEP wurden im Objektmodell nur Klassendiagramme und keine Instanzdiagramme erstellt. Dies ist darin begründet, daß Instanzdiagramme nur jeweils eine konkrete Ausprägung eines Klassendiagramms darstellen und somit vorrangig zu Testzwecken und als Gesprächsgrundlage erstellt werden. Für das Objektmodell ist neben der grafischen Notation auch eine schriftliche Dokumentation zu erstellen. Dieses sogenannte **Data Dictionary** liefert für alle Elemente des Objektmodells Definitionen bzw. Beschreibungen und dient somit als Repositorium für die relevanten Abstraktionen des Systems.

Dynamisches Modell

Im dynamischen Modell werden für die Modellierung eines Systems Szenarien, Ereignispfad-, Ereignisfluß- und Zustandsdiagramme erstellt. Die Zustandsdiagramme (siehe Abbildung 8) bilden dabei den Schwerpunkt der dynamischen Modellierung. Szenarien bzw. die Erweiterung in Form von Ereignispfaddiagrammen dienen lediglich dazu, vor der konkreten Modellierung der Zustandsdiagramme grundlegende Ereignisfolgen zu lokalisieren und aufzuzeichnen. Ebenso verhält es sich mit den Ereignisflußdiagrammen. Ziel dieser drei vorbereitenden Modellierungen ist es, sicherzustellen, das in den Zustandsdiagrammen keine wichtigen Schritte übersehen werden und daß der allgemeine Interaktionsfluß reibungslos und richtig abläuft. Die Notation für die Zustandsdiagramme des dynamischen Modells entspricht im wesentlichen der Notation der Statecharts nach [Harel87].

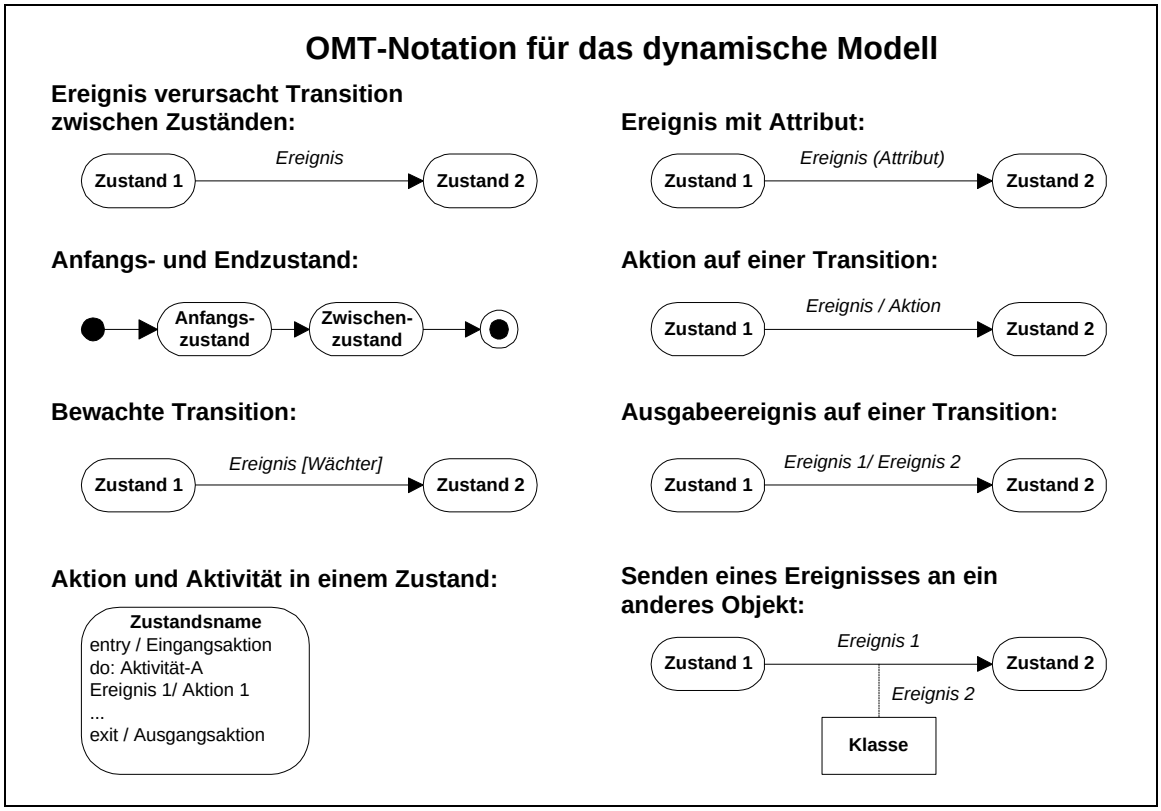


Abbildung 8: Ausschnitt der OMT-Notation für das dynamische Modell [Rumbaugh94]

Zustand — Ist eine Abstraktion der Attributwerte und Verknüpfungen eines Objektes und entspricht der Zeitspanne zwischen zwei von einem Objekt empfangenen Ereignissen.

Aktivität — Ist eine Operation, die eine Zeitdauer besitzt. Die Aktivität wird mit "*do:A*" in der Zustandsbox eingetragen und besagt, daß die Aktion *A* bei Eintritt in den Zustand beginnt und bei Verlassen des Zustandes endet.

Aktion — Ist eine auf einen Moment beschränkte Operation mit einer, gemessen an der zeitlichen Rasterung des Zustandsdiagramms, unbedeutenden Zeitdauer.

Ereignis — Ein Ereignis stellt einen „externen Reiz“ von einem Objekt auf ein anderes Objekte dar. Das Empfängerobjekt erfährt dabei als Reaktion eine Zustandsveränderung oder sendet seinerseits ein Ereignis an ein Objekt.

Transition — Ist eine Zustandsveränderung, die durch ein Ereignis verursacht wird. Der Name des auslösenden Ereignisses ist anzugeben, optional können noch Aktionen, Attribute und Wächter (Bedingungen) angegebenen werden, um die Transition näher zu spezifizieren.

Das dynamische Modell enthält für jede Objektklasse des Objektmodells, die ein dynamisches bzw. zeitlich veränderliches Verhalten hat, ein Zustandsdiagramm. Der hier

beschriebene Ausschnitt der dynamischen Notation entspricht dem in der Modellierung von PEP verwendeten. Eine vollständige Beschreibung der Konzepte und Auflistung der Notation für das dynamische Modell findet sich in [Rumbaugh94, Kap. 5 und S.550].

Funktionales Modell

Für die Beschreibung des funktionalen Modells werden in OMT Datenflußdiagramme verwendet.

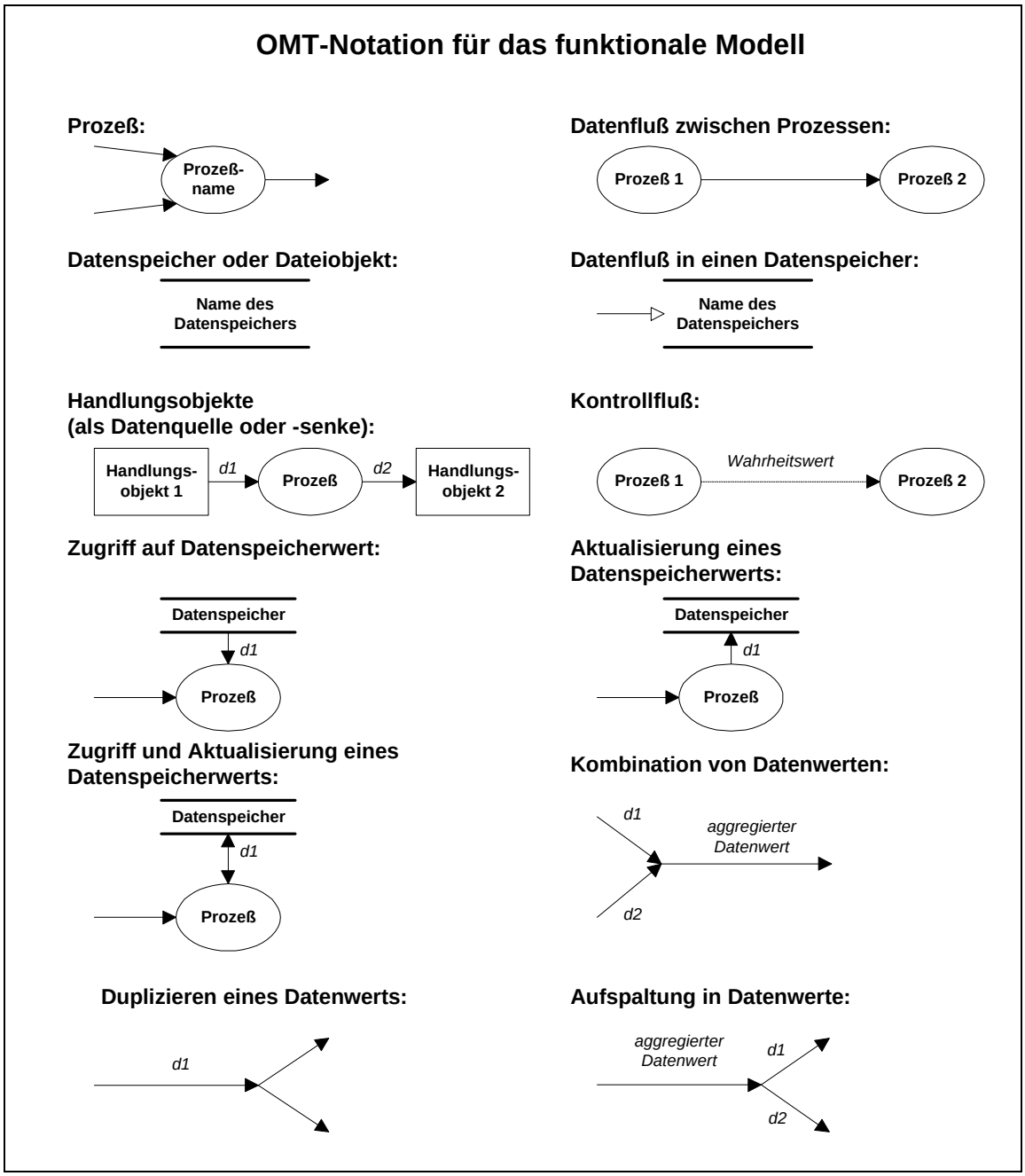


Abbildung 9: OMT-Notation für das funktionale Modell [Rumbaugh94]

Prozeß Ein Prozeß transformiert Datenwerte und stellt so z.B. einfache Funktionen dar. Prozesse entsprechen den Aktivitäten oder

Aktionen der Klassen in den Zustandsdiagrammen oder den Operationen des Objektmodells.

Datenfluß Der Datenfluß (dargestellt durch einen Pfeil) verbindet die Ausgabe eines Objektes oder Prozesses mit der Eingabe eines anderen Objektes oder Prozesses. Der Datenfluß repräsentiert einen Daten- bzw. Zwischenwert einer Berechnung, der durch den Datenfluß nicht geändert werden kann. Der Pfeil wird mit einer Beschreibung der Daten beschriftet. Datenflüsse entsprechen den Objekten oder Attributwerten des Objektmodells.

Handlungsobjekt Ist ein aktives Objekt des Datenflußgraphen, das den Datenfluß aktiviert, indem es Werte/Daten erzeugt oder verbraucht. Handlungsobjekte repräsentieren explizite Objekte des Objektmodells.

Datenspeicherobjekt Datenspeicher sind passive Objekte im Datenflußdiagramm, die Daten speichern und auf denen auf Daten zugegriffen werden kann. Jeder Fluß in einen Datenspeicher ist eine Aktualisierungsoperation, jeder Fluß aus einem Datenspeicher ist eine Anfrageoperation.

Eine vollständige Beschreibung der Konzepte und Auflistung der Notation für das funktionale Modell findet sich in [Rumbaugh94, Kap. 6 und S.551].

5.1.3 Modellierungsprozeß

Grundsätzlich wird, nach der OMT-Methodologie, die objektorientierte Entwicklung in die vier Phasen Analyse, Systementwurf, Objektentwurf und Implementation aufgeteilt. In der Analysephase wird ausgehend von einer Problembeschreibung ein Objektmodell, ein dynamisches und dann ein funktionales Modell entwickelt. Das Ziel ist dabei, ein Problem und den Anwendungsbereich vollständig zu spezifizieren, ohne dabei schon eine bestimmte Implementierungsrichtung bzw. Implementierungsdetails festzulegen. Folgende Schritte sind nach OMT für die einzelnen Modellierungen in der Analyse zu beachten.

Objektmodell

- Finden der Objekte, Klassen, Attribute und ihren Beziehungen zueinander
- Erstellen eines Data Dictionary für die ermittelten Klassen, Attribute und Assoziationen
- Organisieren und Vereinfachen der Klassen durch Vererbung
- Zugriffspfade testen
- Gruppierung von Klassen zu Moduln

Dynamisches Modell

- Typische Interaktionssequenzen in Szenarien verarbeiten
- Identifizieren der Ereignisse zwischen den Objekten und Erstellung eines Ereignispfaddiagramms für jedes Szenario
- Ereignisflußdiagramme für Gruppen von Klassen (z.B. Moduln) erstellen

- Erstellung der Zustandsdiagramme für jede Klasse mit wichtigem dynamischen Verhalten

Funktionales Modell

- Die Ein- und Ausgabewerte des Systems identifizieren
- Datenflußdiagramme zur Beschreibung der funktionalen Abhängigkeiten entwickeln
- Funktionen deklarativ beschreiben
- Identifizieren von Einschränkungen zwischen Objekten und Optimierungskriterien spezifizieren

Die Entwicklung dieser drei Modelle ist in der Analysephase der OMT-Methodologie keinesfalls als rein linear zu betrachten. Vielmehr handelt es sich um einen iterativen Prozeß, in dem die Modelle mit den Erkenntnissen aus den jeweils anderen Modellen schrittweise verfeinert und gegebenenfalls verändert werden (siehe Abbildung 10). Als Ergebnis liefert die Analyse Objektdiagramme mit dem zugehörigen Data Dictionary, Zustands-, Ereignispfad- und Ereignisflußdiagramme sowie Datenflußdiagramme. Der nächste Entwicklungsschritt ist nach der OMT-Methodologie der Entwurf, der in den System- und Objektentwurf unterteilt ist. Der Übergang von der Analyse zu den Entwurfsphasen ist dabei fließend. Detaillierte Bereiche des Analysemodells können durchaus schon Überlegungen des Entwurfs beinhalten, genauso beeinflussen noch Entwurfskriterien die Struktur des Analysemodells.

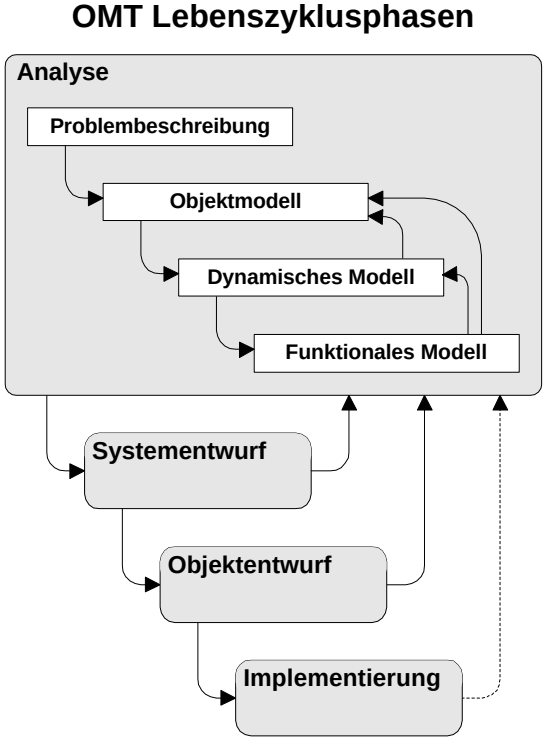


Abbildung 10: OMT Lebenszyklusphasen

Im Systementwurf, der ersten Phase des Entwurfs, werden die grundlegenden Entscheidungen für die zukünftige Architektur des Zielsystems getroffen. Die OMT-Methodologie sieht für den Systementwurf folgende Schritte vor:

Systementwurf

- Aufbrechen des Gesamtsystems in Teilsysteme
- Auffinden von Nebenläufigkeiten
- Zuweisung der Teilsysteme an Prozessoren und Tasks
- Wahl einer Strategie zur Verwaltung von Daten (Datenstrukturen, Dateien, Datenbanken)
- Identifizieren benötigter globaler Ressourcen und Organisation der Zugriffsmechanismen für diese Ressourcen
- Wahl eines Ansatzes zur Implementierung der Steuerungssoftware (Prozedurgesteuertes-, Ereignisgesteuertes-, Nebenläufiges-System)
- Behandlung von Grenzbedingungen (Initialisierung, Terminierung, Absturz)
- Kompromißprioritäten über die Ziele des Systemdesigns ermitteln.

In der zweiten Entwurfsphase, dem Objektentwurf, werden auf Grundlage der im Systementwurf getroffenen Entscheidungen die drei Modelle weiter verfeinert und komplettiert. Ziel ist es, das Analysemodell in einen implementierungsfähigen Entwurf zu überführen. Der wichtigste Bezugsrahmen ist dabei das Objektmodell. In diesem werden die in Operationen umgewandelten Ereignisse des dynamischen Modells und Prozesse des funktionalen Modells den Klassen zugeordnet. Zusätzlich wird das Objektmodell z.B. durch eine Restrukturierung der Klassen oder durch Hinzufügen neuer interner Klassen, Attribute, Assoziationen und Operationen weiter optimiert. Folgende Schritte sind in der OMT-Methodologie für den Objektentwurf vorgesehen.

Objektentwurf

- Definition von Operationen für die Ereignisse des dynamischen Modells und Prozesse des funktionalen Modells
- Entwurf von Algorithmen für alle Operationen des funktionalen Modells
- Optimierung von Zugriffspfaden auf Daten
- Implementation der Zustandsdiagramme unter Berücksichtigung der im Systementwurf gewählten Kontrollmechanismen für die Steuersoftware
- Anpassen der Klassenstruktur, mit dem Ziel, die Vererbung zu erhöhen
- Implementierung der Assoziationen
- Festlegung der exakten Repräsentationen von Objekten und Attributen
- Ordnen der Klassen und Assoziationen zu Moduln

Nach Abschluß des Entwurfs, stehen dem Softwareentwickler für die eigentliche Implementierung des Systems drei detaillierte Modelle und deren Dokumentationen zur Verfügung. Der Entwickler muß diese "nur" noch in Code übersetzen, weil die

schwierigsten Entscheidungen bereits beim Entwurf getroffen wurden [Rumbaugh94, S.321].

Im Rahmen dieser Diplomarbeit wurde festgestellt, daß der Übergang vom Entwurf zur Implementierung nicht so reibungslos zu realisieren ist, wie bei [Rumbaugh94] beschrieben. Dies ist unter anderem darin begründet, daß für den Entwickler/Modellierer schwer zu erkennen ist, wann der Entwurf ausreichend detailliert für die Implementierung ist. Wird die Entwurfsphase zu früh beendet, fehlen eventuell entscheidende Details für die Implementierung. Wird der Entwurf bis zum letzten Detail verfeinert, so geht eventuell wertvolle Implementierungszeit verloren, ohne einen entscheidenden Vorteil für die Implementierung gewonnen zu haben. Die Schwierigkeit besteht also darin, genau zu erkennen, ab wann Modellierungsvorgänge ein Stadium erreichen, ab dem weitere Verfeinerungen besser während der Implementierung zu leisten sind.

Im folgenden wird, ausgehend von der Problemanalyse 5.2, über den Systementwurf 5.3 und Objektentwurf 5.4 die Modellierung von PEP näher erörtert.

5.2 Problemanalyse

Ziel der Problemanalyse ist, die Entwicklung eines groben Objektmodells um ein präzises, verständliches und korrektes Modell der „realen Welt“ zu entwickeln. Als Problembeschreibung für die Gewinnung der einzelnen Klassen, Assoziationen und Attribute des zu modellierenden Systems wurde [Münch97, Kapitel 5] verwendet, das in Kapitel 3 dieser Diplomarbeit zusammengefaßt wurde. Die Festlegung der statischen Strukturen innerhalb des Systems ist der erste Modellierungsschritt während der Analysephase. Das nun folgende Kapitel setzt sich mit der Objektmodellierung auseinander, in deren Verlauf diese Strukturen erarbeitet werden.

5.2.1 Objektmodellierung

Das „Grundgerüst“ einer objektorientierten Applikation wird von Klassen, deren Attributen und den Beziehungen (Assoziationen) der Klassen untereinander gebildet. Zur Gewinnung dieser Grundstruktur wurden zunächst die für das System relevanten Klassen aus der Problembeschreibung „herausgeschält“. OMT schlägt hier das „Herausschreiben“ aller Nomen aus der Problembeschreibung vor. Auf diese Weise wird, je nach Umfang der Problembeschreibung, eine mehr oder weniger umfangreiche Liste von „potentiellen Klassen“ erarbeitet. Anschließend wird die so gewonnene Liste auf die relevanten Begriffe reduziert. Die Analyse der Problembeschreibung lieferte anfangs 210 Nomen, die für den ersten groben Objektentwurf auf 23 relevante Begriffe reduziert wurden. Beispielsweise wurden die beiden in der Problembeschreibung auftretenden Begriffe „HTML-Dokument“ und „PDF-Dokument“ zu dem Begriff „Dokument“ zusammengefaßt. Dies liegt darin begründet, daß schon zu Beginn der Modellierung das Ziel abgesteckt wurde, ein System zu realisieren, das möglichst unabhängig vom zu generierenden Dokumentenformat sein

soll. Die während dieses ersten Schrittes gefundenen Klassen wurden in einem Data-Dictionary festgehalten (Tabelle 4).[1]

Aktion	Abstrakte Superklasse von →*Aktion_BE* und →*Aktion_FE*.
Aktion_BE	Eine in sich geschlossene Anforderung von Back-End-Operationen auf →*Profil*, →*Betreiber*, →*Mantel*, →*Beitrag* und →*Werbung*. *Aktion_BE* füllt Instanzen der oben genannten Klassen mit Inhalten aus der Datenbank und stellt sie →*Aktion_FE* für die weitere Verwendung zur Verfügung. Beauftragt den →*Datenbank_Server* mit der Speicherung von eventuellen Änderungen der Objekte durch *Aktion_FE*. Dient der Kommunikation des Gesamtsystems mit dem *Datenbank_Server*.
Aktion_FE	Eine in sich geschlossene Anforderung von Front-End-Operationen auf →*Profil*, →*Betreiber*, →*Mantel*, →*Beitrag* und →*Werbung*. Liest und ändert Inhalte von Instanzen der oben aufgeführten Klassen, die von →*Aktion_BE* zur Verfügung gestellt werden zur Generierung von →*Dokument*en.
Beitrag	Abgeschlossene Informationsblöcke, die die Gesamtheit der Daten eines einzelnen *Beitrag*es ausmachen (zum Beispiel: ausstrahlender Sender, Darsteller, Bilder usw.). *Beiträge* werden von einem externen →*Lieferant*en geliefert und von →*Import* in eine Datenbank importiert.
Benutzer	Abstraktion einer natürlichen Person. Ein *Benutzer* hat ein →*Profil*, das seine persönlichen Interessen beschreibt. Die Einrichtung dieses *Profils* kann kostenpflichtig gestaltet werden. Dazu muß der *Benutzer* sein *Profil* mittels einer →*Zahlung* freischalten. Der *Benutzer* interagiert mit dem System über →*Client*s.
Betreiber	Betreibt ein →*Publishing_System*. →*Benutzer* sind Kunden eines *Betreibers*, der deren jeweilige →*Profil*e verwaltet bzw. führt. Das System PEP kann von mehreren Betreibern parallel genutzt werden. Jeder Betreiber verfügt über einen individuell gestalteten Web-Auftritt. Die betreiberspezifischen Informationen (zum Beispiel für die Gestaltung der dynamisch generierten HTML-Seiten) sind im Betreiber-Objekt gespeichert. Die TV-Programmdaten und Funktionalitäten von PEP werden von allen Betreibern gemeinsam genutzt.

[1] Die Berücksichtigung aller während der Modellierung erarbeiteten Diagramme und Tabellen würde den Rahmen dieser Arbeit bei weitem sprengen. Aus diesem Grund wurden alle Diagramme, die als Basis für die Implementierung dienen sollen, im Ergänzungsband „Konzeption und Ansätze der Realisierung eines digitalen Publikationssystems für personalisierte TV-Programminformationen – Band 2: OMT-Modelle“ zusammengefaßt. Dieser Ergänzungsband ist Bestandteil dieser Diplomarbeit.

Browser	Client-Software zur Kommunikation eines →*Benutzer*s mit dem System.
Client	Superklasse von →*Browser* und →*E-Mail_Client*.
Daten	Superklasse von →*Beitrag*, →*Profil*, →*Werbung* und →*Betreiber*. Instanzen der Klasse *Daten* und deren Kinderklassen sind Bestandteil einer Datenbank.
Datenbank_Server	Datenbank-Abstraktions-Schicht für die Kommunikation zwischen Datenbank und →*Aktion_BE*.
Dokument	Abgeschlossenes Dokument, das an einen →*Server* übermittelt wird. Ein *Dokument* wird in verschiedenen Formaten zur Verfügung gestellt.
Eingabe	„Zwischenklasse“ zwischen →*Server* und →*Dokument*. Erhält und analysiert alle von einem Server übermittelten Daten, um eine den Anfragedaten entsprechende Generierung eines *Dokument*s anzustoßen. Klasse, die alle vom Benutzer an das System übermittelte Dateneingaben [Stary96] verarbeitet.
E-Mail_Client	Client-Software zur Kommunikation eines →*Benutzer*s mit dem System.
Import	Eine in sich geschlossene Anforderung von Operationen, die →*Beiträge* verschiedener →*Lieferanten* einliest und anschließend in eine Datenbank importiert.
Lieferant	Anbieter von TV-Programminformationen.
Mail_Server	Serversoftware zur Übermittlung eines →*Dokuments* an einen →*Benutzer*. Dazu erhält der *Mail_Server Dokumente*, die vom →*Publishing_System* geliefert wurden. Diese *Dokumente* werden vom *Mail_Server* innerhalb einer E-Mail an einen →*E-Mail_Client* versendet.
Profil	Individuelle Einstellungen eines →*Benutzer*s bei einem →*Betreiber*.
Publishing_System	Eine in sich geschlossene Anforderung von Operationen und Klassen, die Daten unterschiedlichsten Typs verarbeitet, verwaltet und grafisch aufbereitet. Das *Publishing_System* steuert die Generierung von →*Dokumenten* auf Basis eines →*TV-Programms* und →*Profilen*.
Server	Superklasse von →*Web_Server* und →*Mail_Server*.
TV-Programm	Die Gesamtheit aller →*Beiträge*, die Bestandteil eines TV-Programms sind.
Web_Server	Serversoftware zur Übermittlung eines →*Dokumentes* an einen →*Benutzer*. Dazu fordert der *Web_Server* Dokumente an, die vom

	→*Publishing_System* geliefert werden. Diese Dokumente werden vom Web_Server an einen *Browser* gesendet.
Werbung	Textuelle und grafische Informationen eines Werbekunden bei einem →*Betreiber*, die in Abhängigkeit eines →*Profil*s in ein →*Dokument* integriert wird.
Zahlung	Eine in sich geschlossene Anforderung von Operationen, die die Funktionalitäten für den Zahlungsvorgang eines →*Benutzers*, für die Freischaltung eines die *Zahlung* betreffenden →*Profil*s, bereitstellen.

Tabelle 4: Data-Dictionary (Analyse)

Das Einfügen der beiden Klassen *Aktion_BE* und *Aktion_FE*, greift der Analyse strenggenommen etwas vor. An dieser Stelle wird bereits ein „Implementierungsdetail" eingeflochten, das eigentlich Bestandteil des Objektentwurfs sein sollte. Diese, in Tabelle 4 festgehaltene, Aufteilung von Aktionen in Front-End- und Back-End-Aktionen findet ihren Ursprung allerdings in der Problembeschreibung in [Münch97, Kapitel 5.5]. Die dort erörterte Grundstruktur des Systems sah eine Trennung zwischen datenbank- und generierungsspezifischen Funktionalitäten vor. Sie war integraler Bestandteil der Grundkonzeption und wurde aus diesem Grund in das Objektmodell übernommen. Die Klassen *Aktion_BE* und *Aktion_FE* trennen das Gesamtsystem genau an der Grenze zwischen Generierung und datenbankspezifischen Funktionalitäten.

Um die gefundenen Klassen in eine Beziehung zueinander zu setzen, schlägt die OMT-Methodologie als nächsten Modellierungsschritt die Identifikation der Assoziationen zwischen den Klassen vor. Zu diesem Zweck wurden alle Verben und Verbalphrasen aus der Problembeschreibung extrahiert. Nach Streichung der irrelevanten Begriffe, blieben insgesamt 28 Assoziationen zwischen den gewonnen Klassen erhalten (Tabelle 5).

Aktion_BE **betrifft** *Dokument*
Aktion_BE **kommuniziert mit** *Datenbank_Server*
Aktion_FE **betrifft** *Dokument*
Benutzer **führt** *Zahlung* **aus**
Benutzer **hat** *Profil*
Benutzer **interagiert mit** *Browser*
Benutzer **interagiert mit** *Client*[1]
Benutzer **interagiert mit** *E-Mail_Client*
Betreiber **betreibt** *Publishing_System*

[1] Zusammenfassung der Assoziationen „Benutzer interagiert mit E-Mail_Client" und „Benutzer interagiert mit Browser".

Betreiber **stellt** *Mantel*
Betreiber **hat** *Werbung*
Betreiber **führt** *Profile*
Browser **kommuniziert mit** *Web_Server*
Client **kommuniziert mit** *Server*[1]
Datenbank_Server **greift auf** *Daten* **zu**
Dokument **liefert Daten an** *Server*
Eingabe **initialisiert** *Dokument*
Eingabe **kommuniziert mit** *Server*
Import **importiert** *Beiträge*
Lieferant **liefert** *TV-Programm*
Mail_Server (Server) **erhält** *Dokument*
*Mail_*Server **versendet E-Mails an** E-*Mail_Client*
Publishing_System **benutzt** *TV-Programm*
Publishing_System **liefert** *Dokument*
Publishing_System **wertet** *Profil* **aus**
Web_Server (Server) **fordert** *Dokument* **an**
Zahlung **betrifft** *Profil*
Zahlung **ist für** *Dokument*

Tabelle 5: Assoziationen (Analyse)

Die erarbeiteten Klassen und Assoziationen wurden vorerst in zwei Klassendiagrammen verarbeitet (Abbildung 11 und Abbildung 12). Das erste Klassendiagramm („Objekt Analyse“) stellt das System für einen potentiellen Kunden grob dar. Es wurde ein Blickwinkel gewählt, der alle systemspezifischen Details ausblendet und dem „Kunden“ ein Modell des Systems zeigt, wie es sich „von außen“ betrachtet darstellt. Innerhalb dieses Diagramms wurde außerdem auf die Verwendung aggregierender Assoziationen (Aggregationen) verzichtet, da es bei diesem sehr abstrakten Blickwinkel ausschließlich um das Vorhandensein der Assoziationen zwischen den Klassen ging und nicht so sehr um die Art der Assoziationen.

[1] Zusammenfassung der Assoziationen „Browser kommuniziert mit Web_Server“ und „Mail_Server versendet E-Mails an E-Mail_Client“.

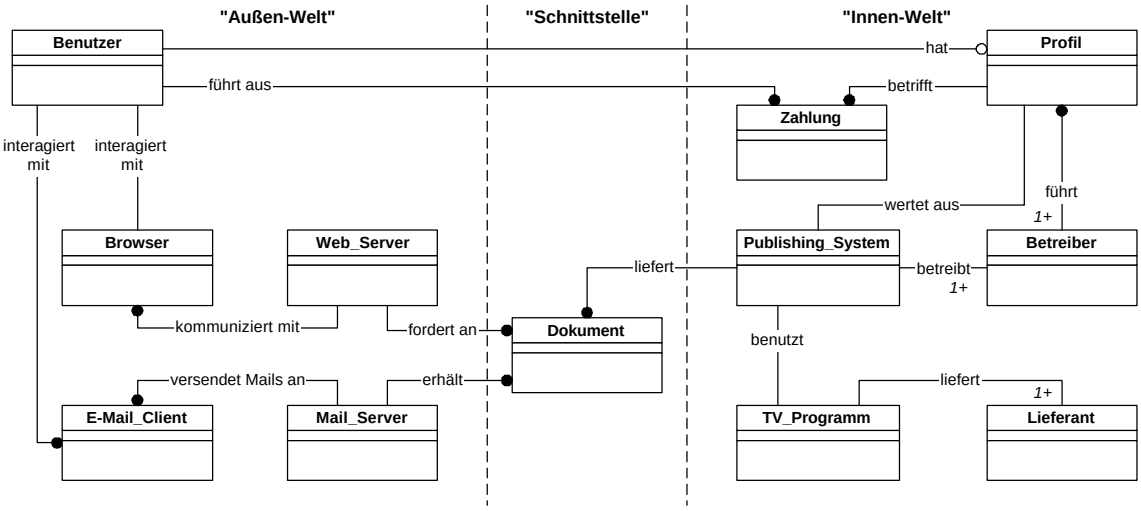


Abbildung 11: Klassendiagramm Objekt Analyse

In diesem ersten Modellierungsschritt wurden Objekte der „realen“ Welt mit „Systemobjekten“ in eine Beziehung gebracht. Dies ist in Abbildung 11 gut anhand der Dreiteilung zu erkennen, die auf der linken Seite alle Klassen aufführt, die als „außerhalb“ des zu implementierenden Systems zu betrachten sind. Das sind zum einen bereits vorhandene Softwarekomponenten, wie *Web_Server* und *Browser*, zum anderen die Abstraktion einer natürlichen Person (*Benutzer*). Diese Klassen sind im weiteren Verlauf der Modellierung nicht weiter verfeinert worden. Auf der rechten Seite des Diagramms befinden sich die „Systemkomponenten“, die in den folgenden Entwicklungsphasen näher betrachtet und verfeinert worden sind.

Als „Schnittstelle“ zwischen „Außen-Welt“ und „Innen-Welt“ dient hier die Klasse *Dokument*, die als zu beiden „Welten“ gehörig klassifiziert worden ist. Für die Klassen *Web_Server*, *Mail_Server*, *Browser* und *E-Mail_Client* gibt es bestehende Dokumentenformate (HTML, PDF etc.). Die Klasse *Publishing_System* muß diese Formate produzieren können, um die angestrebte Kommunikation mit einem Benutzer erreichen zu können. Über die Art und Weise der Produktion dieser *Dokumente* sagt das Diagramm nichts aus.[1] Es ist lediglich ersichtlich, daß die Klasse *Publishing_System* das von einem *Lieferanten* gelieferte *TV-Programm* benutzt, um ein solches *Dokument* zu erstellen. Das System soll es einem Benutzer ermöglichen, durch die Festlegung von persönlichen Interessen eine Auswahl über den Ausschnitt des anzuzeigenden TV-Programms zu realisieren. Diese Einstellungen werden in der Klasse *Profil* gespeichert. *Publishing_System* wertet die Informationen der Klasse *Profil* aus, um einen Extrakt aus dem *TV-Programm* zu erhalten, der den Interessen des Benutzers entspricht.

Neben der „Schnittstelle“ *Dokument* sind noch zwei direkte Assoziationen zwischen „Außen-Welt„ und „Innen-Welt“ zu erkennen (*Benutzer – Zahlung*, *Benutzer – Profil*). Diese Assoziationen haben allerdings einen rein logischen Charakter. Sie dienen zur Veranschaulichung des Sachverhalts, daß ein Benutzer innerhalb des Systems ein *Profil*

[1] Die Produktion der einzelnen Dokumente ist Bestandteil der folgenden Entwicklungsphasen.

hat und dieses mittels einer *Zahlung* (*Zahlung betrifft Profil*) aktiviert. Die eigentliche Kommunikation zwischen *Benutzer* und *Publishing_System* läuft jedoch **immer**, auch bei einem Zahlungsvorgang, über die Klasse *Dokument* ab. Für diese Fälle müssen ebenfalls *Dokumente* generiert werden, um zum Beispiel den Zahlungsvorgang durch einen Benutzer ausführen lassen zu können.

Die Vernachlässigung der Systemspezifika innerhalb der Abbildung 11 bietet die Möglichkeit, die Arbeitsweise und die Integration des Systems in bestehende Strukturen (zum Beispiel Web) zu veranschaulichen, ohne auf technische Details eingehen zu müssen. Für die detaillierte Modellierung wurde aber lediglich der rechte Teil des Diagramms berücksichtigt.

Im Klassendiagramm in Abbildung 12 wurden die Klassen aus Abbildung 11 teilweise zu Klassen zusammengefaßt bzw. in mehrere Klassen aufgebrochen. So wurde beispielsweise die Klasse *Publishing_System* in die einzelnen Klassen *Aktion*, *Aktion_BE*, *Datenbank_Server*, usw. verfeinert. Die beiden Klassen *Browser* und *E-Mail_Client* finden sich dagegen in der Klasse *Client* vereint.

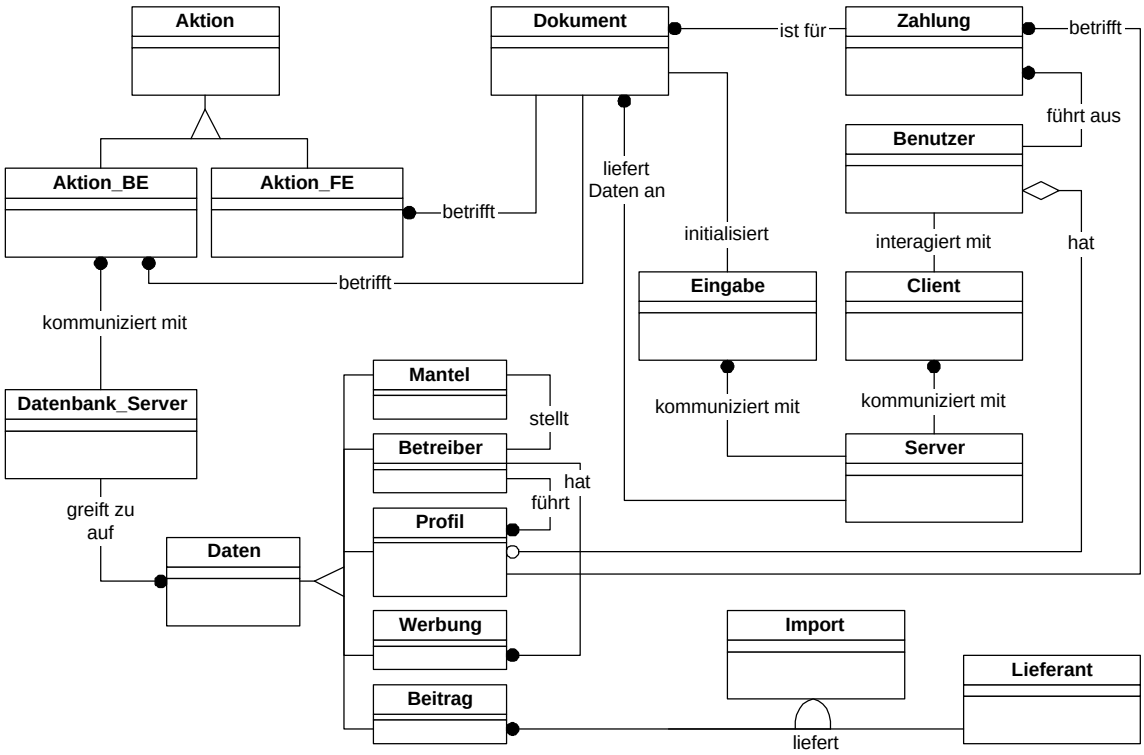


Abbildung 12: Klassendiagramm Übersicht (Analyse)

Auch bei dieser zweiten „Ansicht“ des Systems handelt es sich um ein sehr grobes Modell, das der weiteren Verfeinerung der jeweiligen Bestandteile des kompletten Systems dient. Der Blickwinkel, der in Abbildung 12 gewählt wurde, bietet einen Blick auf das System „von innen“. Eigenschaften der „äußeren“ Objekte bzw. Klassen werden hier vernachlässigt. Für dieses frühe, abstrakte Stadium der Modellierung war es beispielsweise unerheblich, um welchen Client (Browser oder E-Mail_Client) es sich bei der Kommunikation mit dem Server handelt. Wichtig war lediglich das Vorhandensein einer

wie auch immer gearteten „Kommunikations-Assoziation“ zwischen den Klassen *Eingabe*, *Dokument* und der Klasse *Server*. Weiterhin wurden, entsprechend den hinzugefügten bzw. zusammengefaßten Klassen, Assoziationen vereint oder hinzugefügt. Durch das Einfügen der Klasse *Eingabe* zwischen *Dokument* und *Server*, wurden die in Abbildung 11 vernachlässigten Kommunikationskanäle (*Server→Dokument* (input), *Dokument→Server* (output)) berücksichtigt. Die Klasse *Eingabe* ist hier nun für den Kommunikationskanal *Server→Dokument* zuständig, empfängt, analysiert und strukturiert alle vom *Benutzer* über den Weg *Client-Server* an das System übermittelten Daten (input). Nach erfolgter Analyse initialisiert *Eingabe* die Generierung eines *Dokuments* und übergibt die strukturierten Anfragedaten des *Servers*. Die Klasse *Dokument* übernimmt, nach erfolgter Generierung, die Übermittlung der Dokumenteninhalte an den *Server* (output). Unabhängig davon, ob es sich bei dem zu generierenden *Dokument* um die reine Darstellung von Inhalten oder um die Änderung von Inhalten (zum Beispiel Einstellungen eines Benutzers in seinem Profil) handelt, ist das *Dokument* für die folgenden Verarbeitungsschritte verantwortlich.

Die zur Generierung eines *Dokuments* benötigten Inhalte werden von *Aktion_BE* über den *Datenbank_Server* aus einer Datenbank gelesen. *Aktion_FE* liest auf Anforderung die Inhalte aus einem Objekt und stellt sie *Dokument* zur Verfügung. Stehen Änderungen an einem Objekt an, werden diese wiederum über *Aktion_FE* durchgeführt. *Dokument* signalisiert *Aktion_BE* dann die Durchführung einer Änderung, so daß *Aktion_BE* die Speicherung des geänderten Objektes veranlassen kann.

Die Klasse *TV-Programm* aus Abbildung 11 wurde in einzelne Beiträge aufgespalten. *Beitrag* steht nun nicht mehr ausschließlich für „TV-Sendungen“, sondern kann auch für Beiträge anderer Bereiche wie zum Beispiel Hörfunk oder Tageszeitung verwendet werden. Zusätzlich wurden die Klassen *Werbung* und *Mantel* eingeführt, die Werbeinformationen und einen redaktionellen Mantel für jeden Betreiber beinhalten.

Die Identifizierung der Klassenattribute war der nächste durchzuführende Modellierungsschritt. Die OMT-Methodologie schlägt dazu das „Herausschreiben“ aller Adjektive aus der Problembeschreibung vor. An dieser Stelle soll die Klasse *Beitrag* hinsichtlich ihrer Attribute exemplarisch betrachtet werden (siehe Abbildung 13). Da *Beiträge* der Dreh- und Angelpunkt des gesamten Systems sind, und einem Beitrag sehr viele Attribute unterschiedlichster Art zugeordnet worden sind, deckt dieses Beispiel einen großen Teil der Notation ab.

Die Analyse der Problembeschreibung ergab für *Beitrag* 29 Attribute, die sich grob in zwei Arten von Attributen einteilen ließen. Der größte Teil der Attribute bestand aus einfachen Texten, Zahlen und Datumsangaben (zum Beispiel Startzeit, ShowView etc.). Die übrigen Attribute konnten nur durch neue Klassen beschrieben werden, die dem *Beitrag* per Aggregation hinzugefügt wurden. So beinhaltet ein *Beitrag* zum Beispiel ein oder mehrere Bilder, die durch die Klasse *PEPBild* repräsentiert werden. Da PEP ein internet-basiertes System ist, genügte die schlichte Berücksichtigung der Bilddaten nicht. Es mußten noch andere Informationen gespeichert werden, die einem Bild zugeordnet werden können. Dazu gehören beispielsweise ein vom Browser anzuzeigender alternativer Text (ALT) und, da der HTML-Standard keine Deklaration von „Inline“-Grafiken zuläßt,

eine Pfadangabe, wo das Bild innerhalb einer Verzeichnisstruktur auf dem Server abgelegt ist. Gleichzeitig erschien es sinnvoll, die Informationen über die Abmaße des Bildes (Breite, Höhe in Bildpunkten) innerhalb der neuen Klasse zu hinterlegen, um eine sich wiederholende Neuberechnung bei jedem Zugriff auf das Objekt zu umgehen. An diesem Punkt war deutlich, daß die Klasse *PEPBild* auch innerhalb einer *Werbung* Verwendung finden konnte. Das Hinzufügen einer Zieladresse (URL) und eines Zielfensters (target frame) zur Klasse *PEPBild* sollte diese Verwendung ermöglichen. Ferner wurde das Attribut „bild_unterschrift“ ergänzt, um eine Verwendung der Klasse für andere Medien (Print) zu ermöglichen, die eine solche Bildunterschrift erfordern.

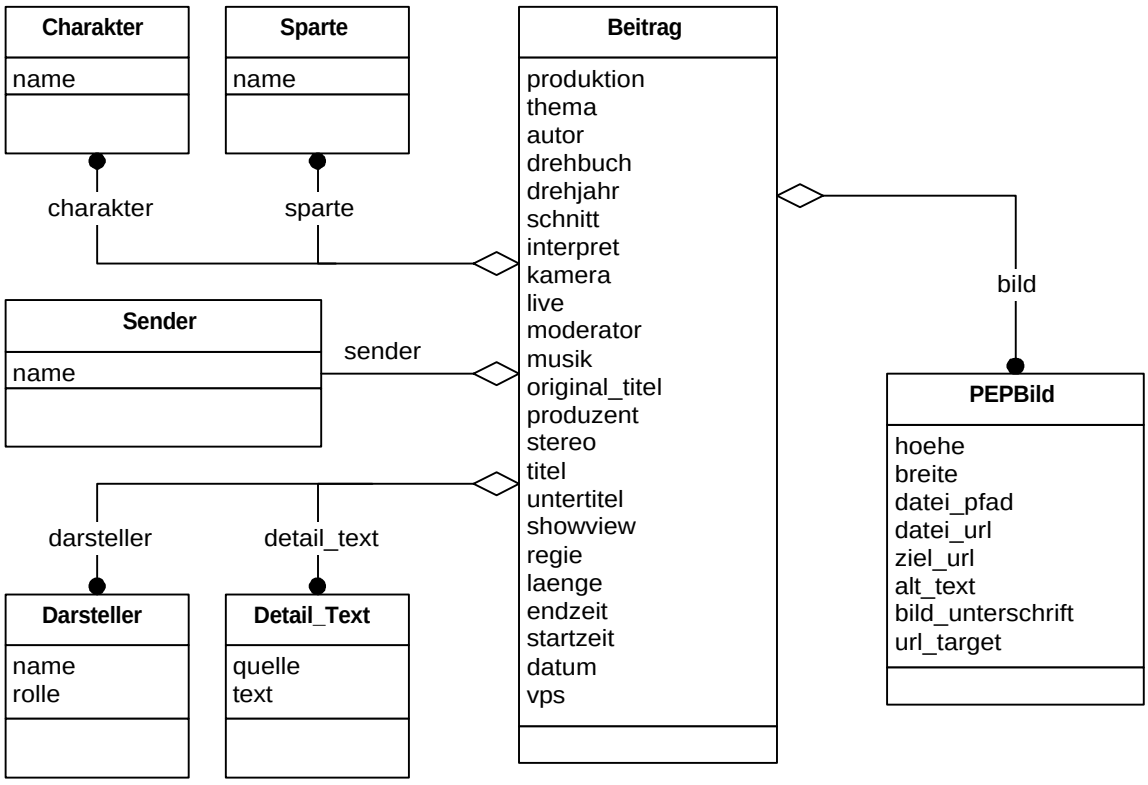


Abbildung 13: Klassendiagramm Beitrag (Analyse)

Die aggregierten Klassen *Charakter* und *Sparte* dienen der Kategorisierung eines *Beitrags*. So gibt es zum Beispiel eine *Sparte* „Spielfilm“ und ein *Charakter* „Komödie“, so daß über diese beiden Attribute ein Beitrag als „komödiantischer Spielfilm“, bzw. als „Komödie mit Spielfilmlänge“ kategorisiert werden könnte. Da die Angabe mehrerer *Sparten* und *Charaktere* möglich ist, kann ein *Beitrag* sehr genaue Kategorisierungen beinhalten. Wird zusätzlich noch der *Charakter* „Western“ eingefügt, würde der *Beitrag* umgangssprachlich als „komödiantischer Western mit Spielfilmlänge“ kategorisiert.[1] Jeder *Beitrag* hat zusätzlich genau einen *Sender*, mehrere beschreibende *Detail_Texte* und mehrere *Darsteller* als Attribute.

[1] Diese Kategorisierung von TV-Beiträgen wird von verschiedenen TV-Programm-Anbietern angeboten. Der Anbieter „PPS“ beispielsweise liefert insgesamt 52 Sparten und 146 Charaktere zur Kategorisierung der einzelnen TV-Beiträge.

Während der Analyse ist zu prüfen, ob die Klassendiagramme mit ihren Assoziationen und Attributen adäquate Zugriffspfade zur Verfügung stellen, um die gewünschten Informationen zu erhalten bzw. um die geforderten Funktionalitäten zu erreichen.

Die OMT-Methodologie schlägt als nächsten Schritt die Gruppierung der Klassen zu Moduln vor. Abbildung 14 zeigt die Verteilung der gefundenen Klassen auf die Module **Aktionen**, **Front-End**, **Back-End** und **Import**.

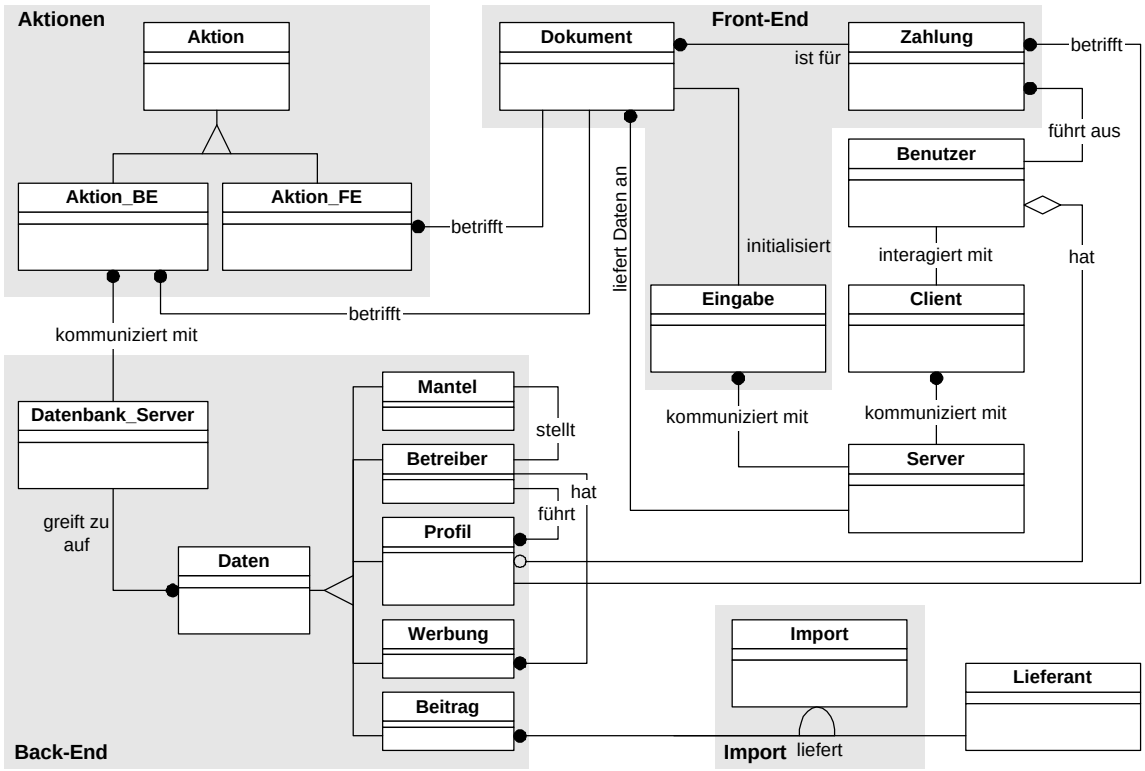


Abbildung 14: Module (Analyse)

In Abbildung 14 wurden die Einzelheiten der einzelnen Klassen aus Platzgründen nicht berücksichtigt. So fehlen zum Beispiel die Attribute der weiter oben betrachteten Klasse *Beitrag*. Alle durch eine Aggregation mit einer Klasse verknüpften Attribut-Klassen sind als Bestandteil des Moduls der Klasse zu betrachten, von der die Aggregation ausgeht. Die Klasse *Darsteller* ist somit dem Modul **Back-End** zugeordnet, da die Aggregation von der Klasse Beitrag ausgeht (vgl. Abbildung 13).

Die Klassen *Benutzer*, *Client*, *Server* und *Lieferant* wurden bei der Modularisierung nicht berücksichtigt, da sie keine integralen Bestandteile des Systems sein werden. Lediglich die Assoziationen von Klassen eines Moduls zu diesen „externen“ Klassen müssen im Systementwurf beachtet werden.

Innerhalb der Objektmodellierung wurde die statische Struktur und die Beziehungen zwischen den verschiedenen Klassen des Systems modelliert. Die folgende dynamische Modellierung übernimmt die Erkenntnisse aus der Objektmodellierung, um auf dieser Grundlage das dynamische Verhalten des Systems zu modellieren.

5.2.2 Dynamische Modellierung

Ziel der dynamischen Modellierung ist die Verdeutlichung der dynamischen Aspekte des Systems. Während der dynamischen Modellierung werden Ereignisse zwischen den in der Objektmodellierung definierten Klassen identifiziert und in eine zeitliche Reihenfolge gebracht. Im folgenden werden zu diesem Zweck zunächst sogenannte Szenarien, bzw. Ereignispfaddiagramme entwickelt, die als Grundlage für die darauf aufbauenden Ereignisfluß- und Zustandsdiagramme dienen (s. Kapitel 5.1).[1]

Szenarien und Ereignispfaddiagramme beschreiben Ereignisfolgen, die von bestimmten Ereignissen ausgelöst werden. Es ist sinnvoll, zu Beginn der dynamischen Modellierung diese auslösenden Ereignisse zu identifizieren, um Ausgangspunkte für die entsprechenden Ereignispfade zu erhalten. Da die Aufgabe von PEP ausschließlich die Generierung von HTML- und PDF-Dokumenten ist, können als „Auslöser“ die beiden Ereignisse „Server fordert HTML-Dokument an“ und „Server fordert PDF-Dokument an“ postuliert werden. Innerhalb der Problembeschreibung wurden lediglich diese beiden Dokumentenformate betrachtet.[2] Zu Beginn der Modellierung wurde jedoch das Ziel dahingehend erweitert, das System möglichst unabhängig vom Dokument-Format zu konzeptionieren (s. Kapitel 5.2.1). Um vom Format des zu generierenden Dokuments zu abstrahieren, werden diese beiden Ereignisse auf das Ereignis „Server fordert Dokument an“ reduziert.

Damit festgestellt werden kann, welche Klasse das von *Server* ausgelöste Ereignis empfängt, ist ein Blick auf das Objektmodell geboten. Abbildung 15 zeigt den für die Beantwortung dieser Frage relevanten Ausschnitt aus dem Objektmodell.

---

[1] Wie in Kapitel 5.2.1 für das Objektmodell erwähnt, wurden auch alle Diagramme des dynamischen Modells, die als Basis für die Implementierung dienen sollen, im Ergänzungsband zusammengefaßt.

[2] Diese Einschränkung auf die Formate HTML und PDF wurde eingeführt, um Ereignispfade für konkrete Ereignisse entwickeln zu können.

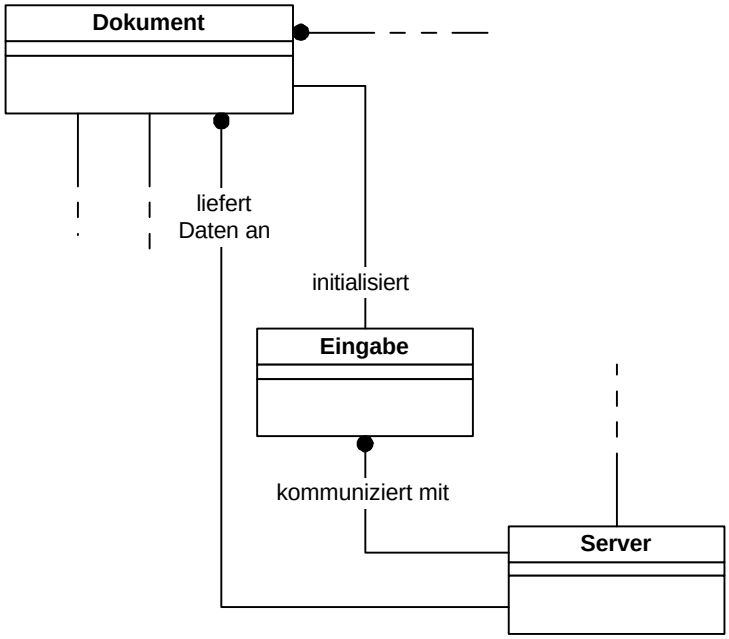


Abbildung 15: Ausriß des Klassendiagramms aus Abbildung 12

Die Klassen *Eingabe* und *Dokument* sind die einzigen Klassen des Systems, die über eine Assoziation mit der Klasse *Server* verbunden sind. Ferner ist über die Namen der betreffenden Assoziationen und durch die Beschreibung der beiden Klassen im Data-Dictionary zu erkennen, daß die Klasse Dokument lediglich Daten an den Server liefert. *Dokument* hat also nur eine unidirektionale Verbindung zum *Server*. Lediglich die Klasse *Eingabe* ist während der Objektmodellierung so definiert worden, daß sie Daten und damit auch Ereignisse vom *Server* erhalten kann. Das Ereignis, das von der Klasse *Server* ausgelöst wird, betrifft somit die Klasse *Eingabe*. Das Ereignis „Server fordert Dokument an“ kann damit auf folgendes Ereignis erweitert werden:

„*Server* **fordert Dokument** von *Eingabe* **an**“

Als folgender Schritt gilt es nun die Ereignisfolge zu beschreiben, die das oben definierte Ereignis innerhalb des Systems auslöst. Das in Tabelle 6 dargestellte Szenario, beschreibt genau diese Ereignisfolge des von der Klasse *Server* initiierten Ereignisses „Server fordert Dokument von Eingabe an“.

Server **fordert Dokument** von *Eingabe* **an**
Eingabe **wertet die Anfragedaten** des *Servers* **aus**
Eingabe fordert die **Generierung eines Dokumentes** von *Dokument* an
Dokument fordert *Aktion_BE* auf den **Betreiber** zu **holen**
Aktion_BE **fordert Betreiber** vom *Datenbank_Server* **an**
Datenbank_Server liefert **Betreiber** an *Aktion_BE*
Aktion_BE meldet an *Dokument* daß der **Betreiber geholt** wurde
Dokument fordert *Aktion_BE* auf das **Profil** zu **holen**
Aktion_BE **fordert Profil** vom *Datenbank_Server* **an**
Datenbank_Server liefert **Profil** an *Aktion_BE*
Aktion_BE meldet an *Dokument* daß das **Profil geholt** wurde

Dokument fordert *Aktion_BE* auf **Beiträge** zu **holen**
Aktion_BE **fordert Beiträge** vom *Datenbank_Server* **an**
Datenbank_Server liefert **Beiträge** an *Aktion_BE*
Aktion_BE meldet an *Dokument* daß die **Beiträge geholt** wurde
Dokument fordert *Aktion_BE* auf **Werbung** zu **holen**
Aktion_BE **fordert Werbung** vom *Datenbank_Server* **an**
Datenbank_Server liefert **Werbung** an *Aktion_BE*
Aktion_BE meldet an *Dokument* daß die **Werbung geholt** wurde
Dokument fordert *Aktion_BE* auf **Mantel** zu **holen**
Aktion_BE **fordert Mantel** vom *Datenbank_Server* **an**
Datenbank_Server liefert **Mantel** an *Aktion_BE*
Aktion_BE meldet an *Dokument* daß der **Mantel geholt** wurde
Dokument **erzeugt Dokument**
Dokument meldet an *Eingabe*, daß das **Dokument erzeugt** wurde
Dokument fordert den *Server* auf, daß **Dokument** zu **senden**

Tabelle 6: Szenario „Server fordert Dokument von Eingabe an“[1]

Die oben beschriebene Ereignisfolge wird in Abbildung 16 innerhalb eines Ereignispfaddiagramms grafisch dargestellt. Dabei wurden einige Ereignisse umbenannt, um die Lesbarkeit des Diagramms zu verbessern (zum Beispiel „fordert Dokument an“ in „gib Dokument“).

[1] Innerhalb des Szenarios werden Klassen kursiv und Ereignisse fett dargestellt.

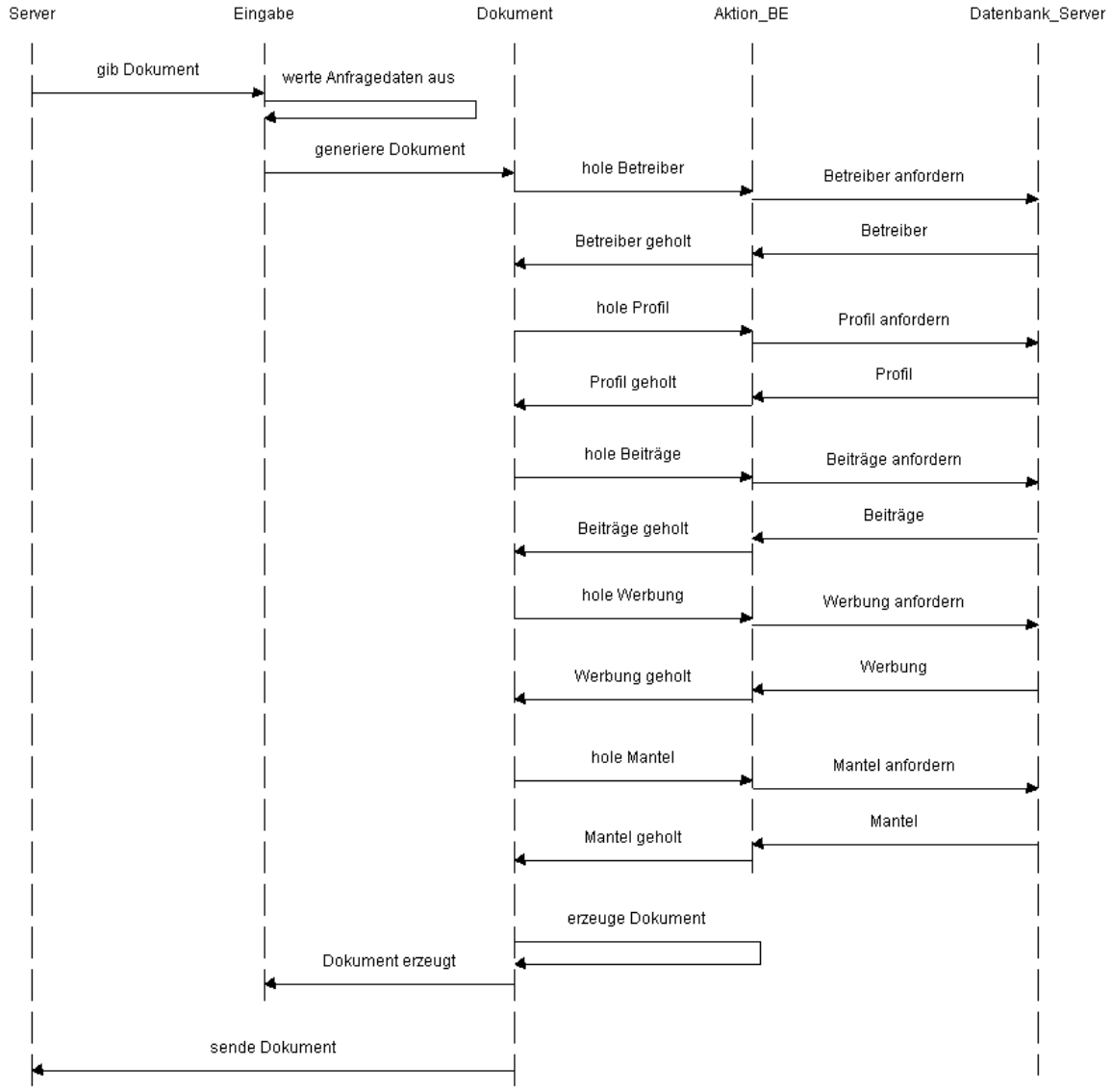


Abbildung 16: Ereignispfaddiagramm Dokument generieren (Analyse)

Das Ereignis „gib Dokument“ löst zunächst die Auswertung der vom *Server* übermittelten Anfragedaten durch *Eingabe* aus. Es folgt die Aufforderung von *Eingabe* an *Dokument*, ein Dokument zu generieren. Zu diesem Zweck fordert *Dokument* bei *Aktion_BE* nacheinander zunächst alle benötigten Daten an (Betreiber, Profil, Beiträge, Werbung und Mantel), um diese in das zu erzeugende Dokument einzubauen. Sobald die Erzeugung des Dokuments abgeschlossen ist, wird an *Eingabe* gemeldet, daß das Dokument erzeugt wurde. Abschließend wird der *Server* aufgefordert, das erzeugte Dokument an den entsprechenden Client zu senden.

Die Berücksichtigung der Assoziationen zwischen den einzelnen Klassen, die innerhalb der Objektmodellierung definiert worden sind, hat einen entscheidenden Einfluß auf die dynamische Modellierung. Die Assoziationen beeinflussen unter anderem die Reihenfolge, in der die einzelnen Bestandteile des zu erzeugenden Dokuments angefordert werden müssen. Da *Profile* beispielsweise jeweils einem bestimmten *Betreiber* zugeordnet sind (s. Abbildung 12, Seite 40), muß zuerst der entsprechende *Betreiber* angefordert werden.

Erst im Kontext des *Betreibers* kann das *Profil* angefordert werden. Die Auswahl der zu verarbeitenden *Beiträge* wiederum geschieht einerseits anhand der vom *Server* während der Anfrage übermittelten Daten. Andererseits werden auch die persönlichen Einstellungen eines *Benutzers*, die innerhalb eines *Profils* festgehalten sind, in den Auswahlprozeß einbezogen. Die Anforderung der *Beiträge* darf deshalb erst vorgenommen werden, nachdem das *Profil* geliefert worden ist. Für die Auswahl der Werbung gelten die gleichen Bedingungen wie für der Auswahl der *Beiträge*. Für die Anforderung des *Mantels* gilt es, wie beim *Profil*, lediglich die Zuordnung zu einem *Betreiber* zu berücksichtigen. Der *Mantel* könnte somit zu einem beliebigen Zeitpunkt nach Erhalt des *Betreibers* angefordert werden.

Wenn man Abbildung 16 mit dem zugrundeliegenden Szenario vergleicht, so wird deutlich, daß der Informationsgehalt beider Varianten identisch ist. Sowohl im Ereignispfaddiagramm, als auch im Szenario, wird das Auftreten von Ereignissen zwischen Objekten in eine zeitliche Reihenfolge gebracht. Das Ereignispfaddiagramm bietet allerdings eine übersichtlichere Darstellung der Ereignisfolge und der betreffenden Objekte. Aus diesem Grund wurden sowohl während der Analyse, als auch im späteren Objektentwurf, ausschließlich Ereignispfaddiagramme verwendet. Auf die Ausarbeitung entsprechender Szenarien wurde für PEP deshalb vollständig verzichtet.

In Abbildung 16 ist das Ereignis „erzeuge Dokument“ zu erkennen. Die Ereignisfolge, die dieses Ereignis auslöst, wurde im entsprechenden Diagramm vernachlässigt. Das in Abbildung 17 dargestellte Ereignispfaddiagramm verfeinert dieses Ereignis aus Abbildung 16.

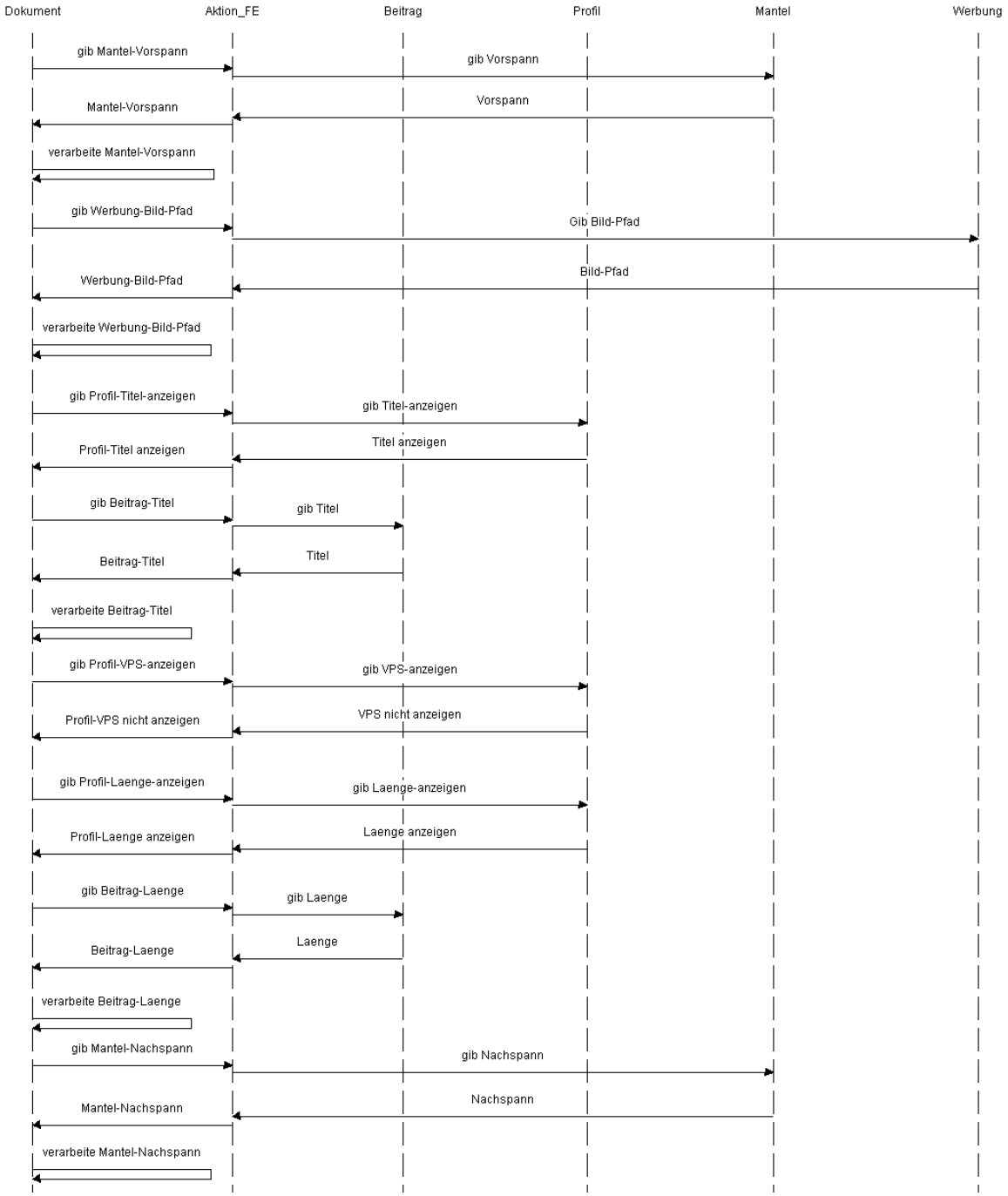
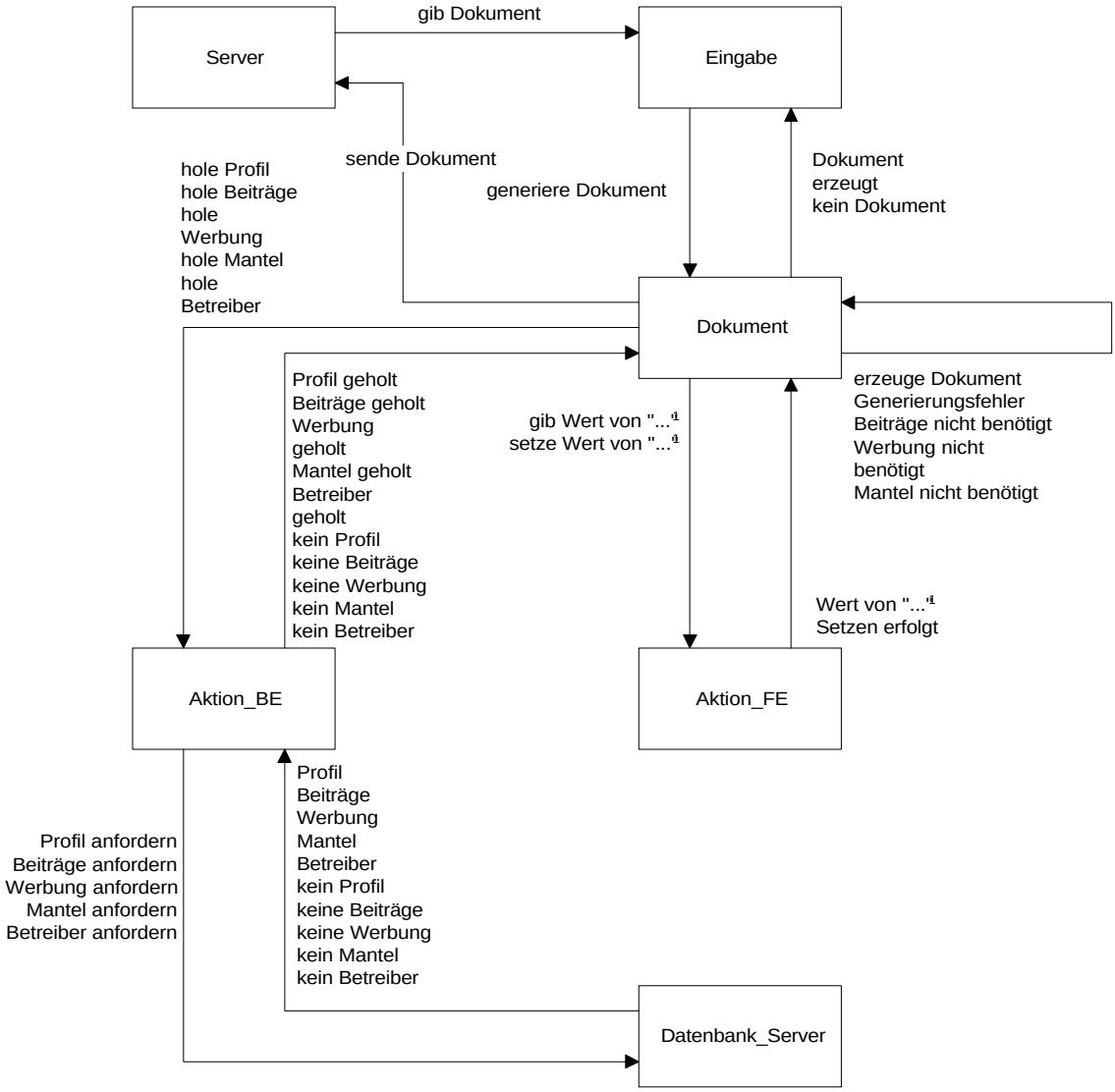


Abbildung 17: Ereignispfaddiagramm Dokument erzeugen (Analyse)[1]

Dokument verarbeitet zunächst den Vorspann des redaktionellen *Mantels* und das Bild der *Werbung*. Abbildung 17 verdeutlicht aber vor allem die Arbeitsweise der Klasse *Dokument* in bezug auf die Auswahl der einzelnen anzuzeigenden Bestandteile eines *Beitrags*. Bevor der Wert eines Beitrag-Attributs über *Aktion_FE* angefordert wird, wird zunächst überprüft, ob innerhalb des *Profils* festgelegt wurde, daß die Anzeige dieses Beitrag-

[1] Abbildung 17 zeigt die Erzeugung eines Dokumentes anhand des konkreten Beispiels der „Einzelansicht“. Die Einzelansicht umfaßt Informationen über einen bestimmten Beitrag.

Attributs vom Benutzer gewünscht wird. Ist dies der Fall (zum Beispiel für „Titel“), wird das Beitrag-Attribut angefordert und anschließend von *Dokument* verarbeitet. Wird die Anzeige des Attributs vom Benutzer nicht gewünscht (zum Beispiel „VPS“), überspringt *Dokument* die Anforderung und Verarbeitung dieses Attributs. Abschließend wird der Abspann des redaktionellen Mantels angefordert und verarbeitet. Im Verlauf der dynamischen Modellierung wurden mehrere Ereignispfaddiagramme für PEP entwickelt, die in Ereignisflußdiagrammen zusammengefaßt worden sind. Das übergeordnete Ereignisflußdiagramm für Abbildung 16 ist in Abbildung 18 dargestellt.



[1] "..." steht für alle Werte der Attribute der Klassen Beitrag, Werbung, Profil, Mantel und Betreiber (s. entsprechende Klassendiagramme)

Abbildung 18: Ereignisflußdiagramm Server – Eingabe – Dokument – Aktion_BE – Aktion_FE – Datenbank_Server

Innerhalb eines Ereignisflußdiagramms wird die zeitliche Komponente, die innerhalb der Ereignispfaddiagramme festgelegt wurde, außer acht gelassen. Dieses Diagramm dient lediglich der Auflistung aller möglichen Ereignisse zwischen den im Diagramm

aufgeführten Klassen. Der Ereignispfad aus Abbildung 16 kann innerhalb des entsprechenden Ereignisflußdiagramms nachvollzogen werden.

Ereignispfad- und Ereignisflußdiagramme dienen als Grundlage für die Erstellung von Zustandsdiagrammen. Jedes Zustandsdiagramm repräsentiert dabei die Zustände und Zustandsübergänge einer bestimmten Klasse bei Eintreten eines bestimmten Ereignisses.

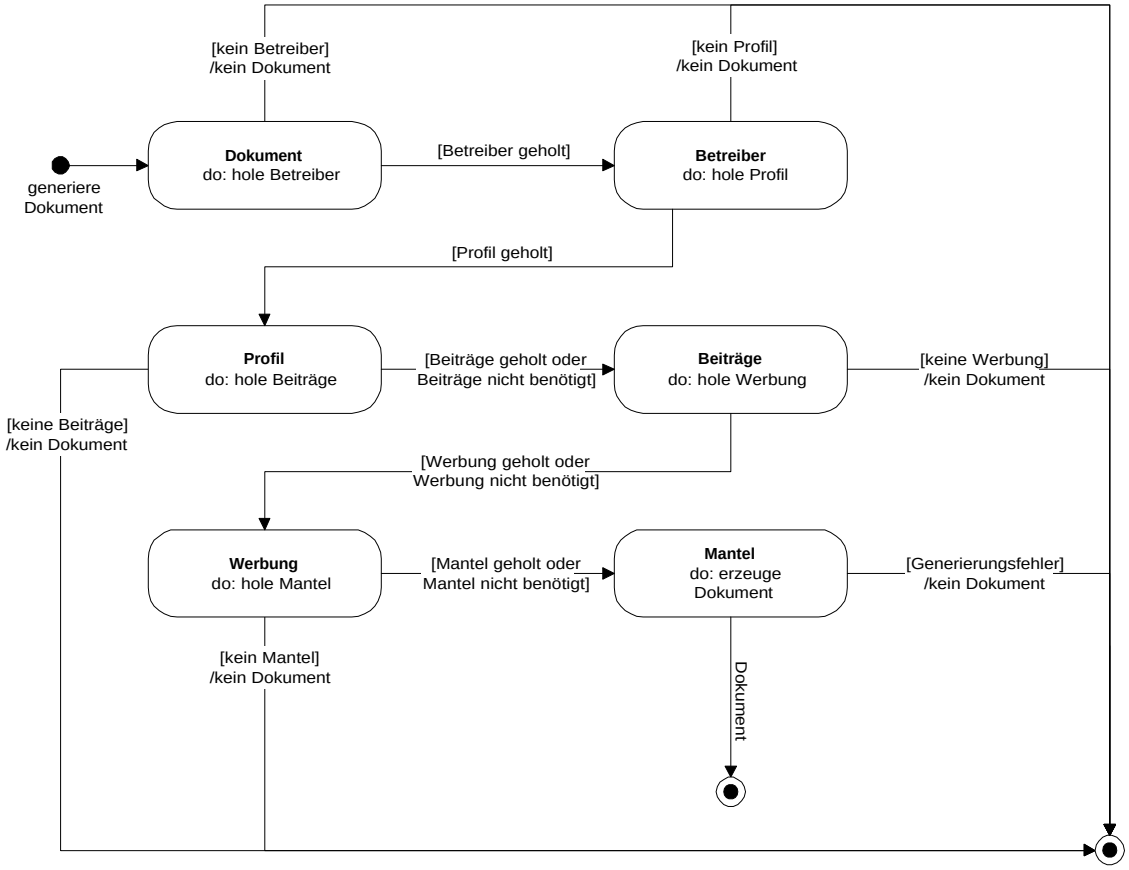


Abbildung 19: Zustandsdiagramm Dokument – Dokument generieren

Abbildung 19 zeigt das Zustandsdiagramm für das Ereignis „generiere Dokument“ (vgl. Abbildung 18). Dieses Ereignis wird von der Klasse *Eingabe* ausgelöst und von der Klasse *Dokument* empfangen. Abbildung 19 ist also aus Sicht der Klasse *Dokument* zu betrachten. Die Klasse *Dokument* erreicht zuerst den Zustand „Dokument“, der als auszuführende Aktion „hole Betreiber“ beinhaltet. Nacheinander „holt“ Dokument die einzelnen Bestandteile, die für die Generierung benötigt werden. Dabei sind für jeden Zustand des Diagramms jeweils zwei mögliche Zustandsübergänge verzeichnet, die den Erfolg bzw. Mißerfolg der jeweils anstehenden Aktion markieren. Zusätzlich ist bei drei Übergängen das Ereignis „[…] nicht benötigt“ eingefügt worden. Diese Ereignisse treten jeweils dann ein, wenn ein Bestandteil nicht für die Generierung benötigt wird. Bei einem Übergang in einen Endzustand wird von *Dokument* wiederum ein Ereignis initiiert (zum Beispiel „kein Dokument“), das von einer anderen Klasse als *Dokument* empfangen und verarbeitet wird. Bemerkenswert an Abbildung 19 ist die Tatsache, daß alle Ereignisse, die einen aufgetretenen Fehler melden, in einen einzigen Endzustand münden. Die OMT-Methodologie schlägt vor, alle Ereignisse zu gruppieren, die die gleiche Wirkung

auf den Kontrollfluß haben. Da es für die Bearbeitung des aufgetretenen Fehlers durch die Klasse *Eingabe* zu diesem Zeitpunkt der Modellierung unerheblich ist, welcher Fehler aufgetreten ist, wird hier immer der identische Endzustand erreicht.

Die Gesamtheit aller Ereignispfad-, Ereignisfluß- und Zustandsdiagramme bildet das dynamische Modell. In diesem Kapitel wurde nur ein kleiner Ausschnitt aus dem während der Problemanalyse entwickelten dynamischen Modell vorgestellt. Die für die Implementierung entscheidenden Details wurden erst während des Entwurfs modelliert und sind Bestandteil des Kapitels 5.4.

Durch die Objektmodellierung in Kapitel 5.2.1 wurde die statische Struktur des Systems modelliert. Das in diesem Kapitel entwickelte dynamische Modell gibt Auskunft über die dynamischen Aspekte der in der Objektmodellierung gefundenen Strukturen. Es wurden unter anderem Kontrollstrukturen und die resultierenden Kontrollflüsse erarbeitet und durch Zustandsdiagramme visualisiert. Das nun folgende Kapitel soll die entstehenden Datenflüsse und funktionalen Abhängigkeiten verdeutlichen, die durch die dynamischen Abläufe innerhalb des Systems entstehen.

5.2.3 Funktionale Modellierung

Nachdem im dynamischen Modell die im System vorhandenen Ereignisfolgen mit ihren Zuständen spezifiziert wurden, folgt jetzt im dritten und letzten Modell die Beschreibung der Wertetransformationen innerhalb des Systems. Ziel ist es, die Berechnungen innerhalb eines Systems mit ihren Eingabewerten und den daraus resultierenden Ausgabewerten zu lokalisieren und zu beschreiben. Weiterhin soll spezifiziert werden, inwieweit funktionale Abhängigkeiten zwischen den im System berechneten Werten bestehen und welche Werte bestimmten Einschränkungen unterliegen.

Für die Modellierung werden Datenflußdiagramme verwendet (vgl. Abschnitt 5.1.2). Diese beschreiben den Fluß von Werten zwischen Prozessen, Datenspeichern und Objekten. Für die funktionale Modellierung sind dabei die Prozesse und die Transformation der Werte durch diese Prozesse von wesentlichem Interesse. Die Datenflußdiagramme liefern keine Kontrollinformationen, wie z.B. die Reihenfolge in der Prozesse ausgeführt werden oder die Entscheidung zwischen alternativen Datenpfaden. Diese Informationen gehören in das dynamische Modell (vgl. Abschnitt 5.2.2).

Während der Entwicklung von PEP wurde schnell deutlich, daß dieses dritte Modell für die Modellierung des Systems die geringste Aussagekraft bzw. Bedeutung besitzt. Das liegt daran, daß die in PEP zu verarbeitenden Datenwerte im wesentlichen nur zwischen den Prozessen ausgetauscht (weitergereicht) werden. Berechnungen bzw. Transformationen von Datenwerten in andere Datenwerte, die den Kern des funktionalen Modells ausmachen, finden selten statt. Folgende zwei Beispiele sollen dies verdeutlichen.

Beispiel 1: Flugsimulator – Kräfte berechnen
Abbildung 20 zeigt einen abstrakten Prozeß, der die auf ein fliegendes Flugzeug einwirkenden Drehmomente und Kräfte verarbeitet, um damit z.B. die tatsächliche Beschleunigung des Flugzeugs zu berechnen[1].

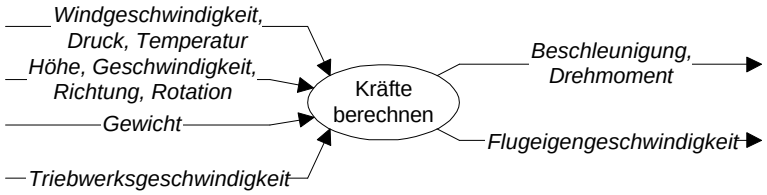


Abbildung 20: Beispiel funktionales Modell – Flugsimulator

An dem Prozeß „Kräfte berechnen“ wird deutlich, daß komplexe Berechnungen notwendig sind, um aus den numerischen Eingabewerten die numerischen Ausgabewerte zu erhalten. Weiterhin wird deutlich, daß die berechneten Ausgabewerte des Prozesses massiv von den Eingabewerten abhängig sind, und daß die Eingabewerte bestimmten Einschränkungen unterliegen. Zuletzt ist festzustellen, daß sich dieser abstrakte Prozeß in weitere Prozesse aufbrechen läßt, bis hin zu konkreten Prozessen die physikalischen Formeln entsprechen. Die funktionale Modellierung zeigt dabei auch die Abhängigkeiten zwischen den Ergebnissen der verfeinerten Prozesse auf. Insgesamt besitzt eine komplette funktionale Modellierung dieses Prozesses eine hohe Aussagekraft. Ermittelte Methoden, Datenflüsse, Abhängigkeiten und Einschränkungen unterstützen deutlich eine spätere Implementierung.

Zum Vergleich soll jetzt der in PEP für die Generierung von Dokumenten zuständige Prozeß betrachtet werden. Eine abstrakte Beschreibung für den Generierungsvorgang befindet sich bereits im dynamischen Modell als Zustandsdiagramm (vgl. Abbildung 19).

Beispiel 2: PEP – HTML Dokument erzeugen
Abbildung 21 zeigt den Prozeß der Dokumenterzeugung von PEP mit allen benötigten Eingabewerten und dem Ausgabewert Dokument. Die Abstraktionsebene dieser Abbildung ist mit der von Abbildung 20 in etwa vergleichbar.

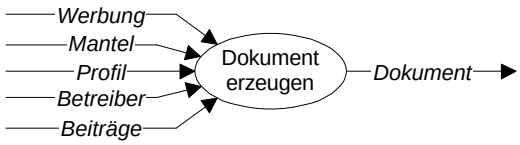


Abbildung 21: Beispiel funktionales Modell – PEP

[1] Dieser Prozeß ist Teil eines funktionalen Modells, das einen Flugsimulator beschreibt [Rumbaugh94, S.136]. Beginnend mit den Eingabewerten des Piloten (Gas, Steuerelemente) über Wetterdaten bis hin zu den Bildschirmausgabedaten werden darin eine Fülle von funktionalen Abhängigkeiten zwischen diesen Werten beschrieben.

Im Gegensatz zum ersten Beispiel finden in den verfeinerten Prozessen von „Dokument erzeugen“ keine „Berechnungen“ statt. Die an „Dokument erzeugen“ übergebenen Werte werden lediglich organisiert und in ein bestimmtes Format, z.B. HTML, gebracht. Sie werden dabei nicht verändert und es werden auch keine neuen Werte aus ihnen abgeleitet. Die gerade im funktionalen Modell entscheidenden Datenflüsse zwischen den verfeinerten Prozessen, beschränken sich hier im wesentlichen auf ein Weiterreichen von Werten bis zu ihrem Bestimmungsort im Dokument. Transformationen von Werten finden innerhalb dieser Prozesse nur marginal statt. Das erfolgreiche Erzeugen eines Dokuments hängt fast ausschließlich von dem Vorhandensein der Eingabewerte ab. Wenn ein benötigter Wert vorhanden ist, wird er unverändert in das Dokument eingebaut. Andernfalls ist zu entscheiden, welches Folgeereignis das Fehlen dieses Wertes auslöst. Genau dieser Vorgang hat jedoch wenig mit funktionalen Abhängigkeiten und Berechnungen zu tun, als vielmehr mit Kontrollinformationen. Eine aussagekräftige Verfeinerung des Prozesses „Dokument erzeugen“ ist demzufolge effektiver mit dem dynamischen Modell möglich. Darin kann genau bestimmt werden, wann und in welcher Reihenfolge die Werte für das Dokument benötigt bzw. in das Dokument einzubauen sind.

Wie an diesem Beispiel deutlich wurde, ist es **nicht** sinnvoll, den Prozeß "Dokument erzeugen" im funktionalen Modell weiter zu verfeinern bzw. aufzubrechen. Diese Feststellung trifft nicht nur auf den Kernprozeß von PEP ("Dokument erzeugen“) zu, sondern auf die meisten der zu modellierenden Bereiche von PEP. Vorgänge oder Operationen, die nur mit dem Verwalten bzw. Organisieren von Daten zu tun haben, geben dem funktionalen Modell nicht die nötige Substanz für eine Modellierung. Eine äquivalente Feststellung in bezug auf Datenbanken wird auch in [Rumbaugh94] getroffen:

> *Dagegen ist das funktionale Modell von Datenbanken häufig trivial, weil ihre Aufgabe darin besteht, Daten zu speichern und zu organisieren, nicht sie umzuwandeln* (S.149).

Aus diesen Gründen beschreibt das gesamte funktionale Modell von PEP das System lediglich auf einer extrem hohen Abstraktionsebene.

Nach [Rumbaugh94] erfolgt die Modellierung des funktionalen Modells nach der Top-down-Methode. Dabei werden im ersten Entwicklungsschritt alle Ein- und Ausgabewerte zwischen dem System (PEP) und der Außenwelt identifiziert und in einem sehr abstrakten Modell dargestellt (siehe Abbildung 22).

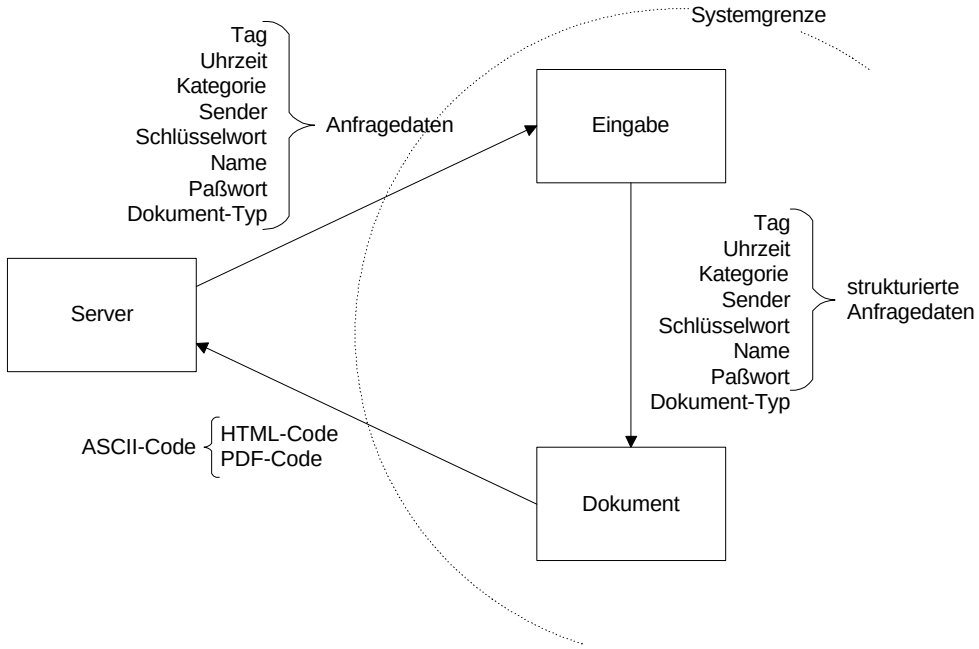


Abbildung 22: Ein- und Ausgabewerte von PEP

Die Systemgrenze von PEP verläuft dabei für alle Eingabewerte (Eingabeereignisse) zwischen den Klassen *Eingabe und Server* und für die Ausgabewerte zwischen *Dokument und Server* [1]. Aufbauend auf Abbildung 22 wurde jetzt das erste Datenflußdiagramm konstruiert, das beschreibt, wie aus den Eingabewerten die Ausgabewerte „berechnet“ werden (siehe Abbildung 23).

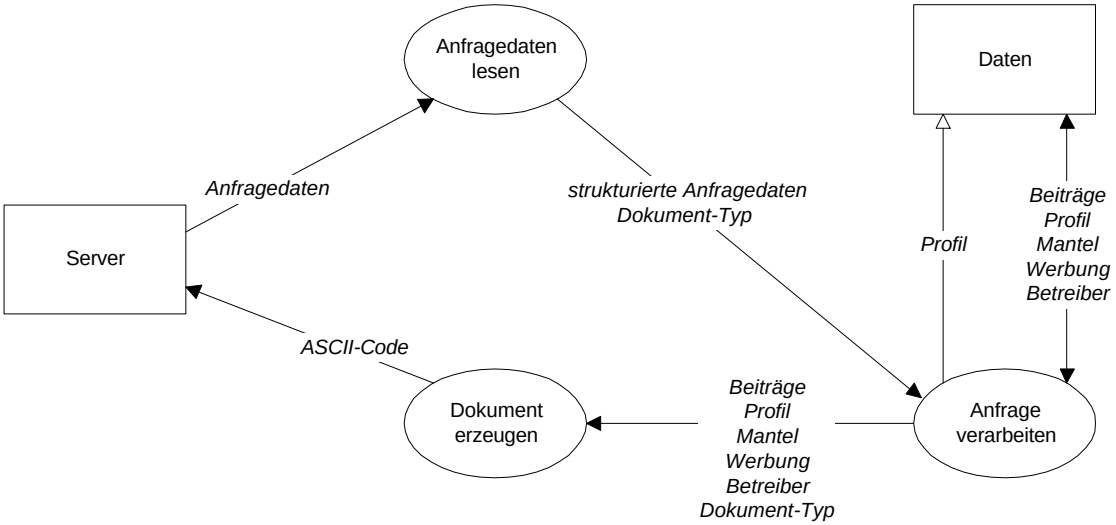


Abbildung 23: Oberstes Datenflußdiagramm

[1] Die drei Klassen *Server*, *Eingabe*, *Dokument* wurden dem Klassendiagramm in Abbildung 12 der Objektmodellierung entnommen. Alle Interaktionen zwischen der Außenwelt (*Server*) und dem System laufen nur über die beiden Klassen *Eingabe* und *Dokument* ab (vgl. dazu auch Seite 44 ff.).

In diesem „obersten Datenflußdiagramm“ wurden jetzt anstatt der Klassen *Eingabe* und *Dokument* Prozesse eingeführt, welche die Verarbeitung der Daten beschreiben. Dabei wurde die Klasse *Eingabe* durch den Prozeß „Anfragedaten lesen“ und die Klasse *Dokument* durch die Prozesse „Anfrage verarbeiten“ und „Dokument erzeugen“ ersetzt. „Anfragedaten lesen“ erhält vom Server einen String mit Anfragedaten, strukturiert diese und übergibt sie und den ermittelten Dokument-Typ an den Prozeß „Anfrage verarbeiten“. Der Prozeß „Anfrage verarbeiten“ ermittelt dann anhand der Anfragedaten und dem Dokument-Typ die für die Erzeugung des Dokuments benötigten Werte und holt sie aus der Datenbank. Dieser Prozeß ist bereits in Abbildung 19 des dynamischen Modells modelliert worden. Anschließend werden die akquirierten Werte an den Prozeß „Dokument erzeugen“ übergeben. Dieser erzeugt anhand der übergebenen Werte HTML- oder PDF-Code und übermittelt diesen an den Server.

Nach OMT ist jetzt jeder nicht triviale Prozeß dieser obersten Schicht des funktionalen Modells in einem neuen Datenflußdiagramm weiter aufzubrechen. „Anfrage verarbeiten“ beschränkt sich im wesentlichen auf Kontrollflüsse und wurde deshalb bereits im dynamischen Modell modelliert. Gleiches gilt für den Prozeß „Dokument erzeugen“, der zu Beginn dieses Abschnitts ausgiebig erörtert wurde. Die Modellierung dieses Prozesses ist Bestandteil des dynamischen Modells in Kapitel 5.4.2. Der letzte verbleibende Prozeß „Anfragedaten lesen“ beschränkt sich nur auf das Lesen und die Umstrukturierung der Anfragedaten. Eine funktionale Modellierung erübrigt sich daher.

Da die funktionale Modellierung nach der Top-down-Methode zu realisieren ist, und kein Prozeß des obersten Datenflußdiagramms sinnvoll weiter aufgebrochen werden kann, ist die funktionale Modellierung für PEP hiermit abgeschlossen. Die für eine spätere Implementierung benötigten Methoden werden im dynamischen Modell erarbeitet. Die zu verarbeitenden Werte können dem Objektmodell entnommen werden.

Bevor jetzt das Analysemodell im Objektentwurf in ein implementierungsfähiges Modell überführt wird, müssen im folgenden Kapitel zuerst einige grundlegende Entscheidungen für den Systementwurf getroffen werden.

5.3 Systementwurf

Nach Abschluß der Problemanalyse wird im Systementwurf damit begonnen, die Realisierung des Systems im Detail zu planen. Das heißt, die realen Bedingungen der Softwareentwicklung werden schrittweise in die abstrakte Sichtweise der Objektmodellierung eingearbeitet.

5.3.1 Aufbrechen des Systems in Teilsysteme

Die Modularisierung des Systems, die am Ende der Objektmodellierung stand (vgl. Kapitel 5.2.1), ist im Systementwurf leicht verändert für die Definition der Teilsysteme übernommen worden. In Abbildung 24 ist die Modularisierung noch einmal in vereinfachter Form wiedergegeben.

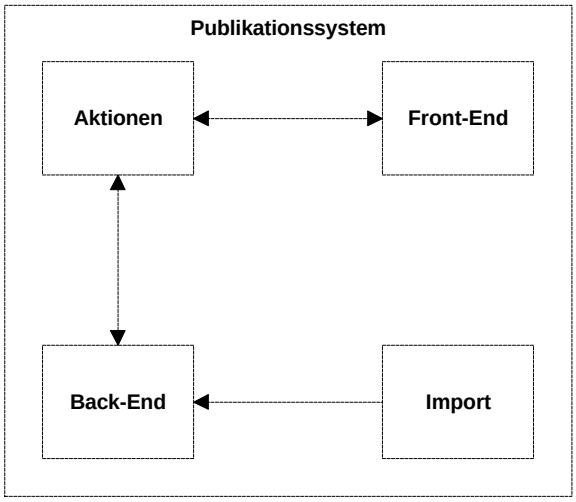


Abbildung 24: Modularisierung von PEP

In dieser Aufteilung wird die ausschließliche Verbindung der Systeme „Front-End“ und „Back-End“ über das System „Aktionen“ deutlich. Durch diese Teilung wird auch die physische Trennung des Front-End vom Back-End möglich. Beide Teilsysteme können, wie in Kapitel 5.2.1 gefordert, gegebenenfalls auf unterschiedlichen Rechnern laufen. Über das vermittelnde Teilsystem „Aktionen“ ist bei dieser Aufteilung allerdings keine eindeutige Aussage darüber zu treffen, auf welchem Rechner dessen Funktionalitäten und Strukturen zur Verfügung stehen sollen. Daher ist eine weitere Aufspaltung des Teilsystems „Aktionen“ in „Front-End Aktionen“ und „Back-End Aktionen“ nötig (s. Abbildung 25).

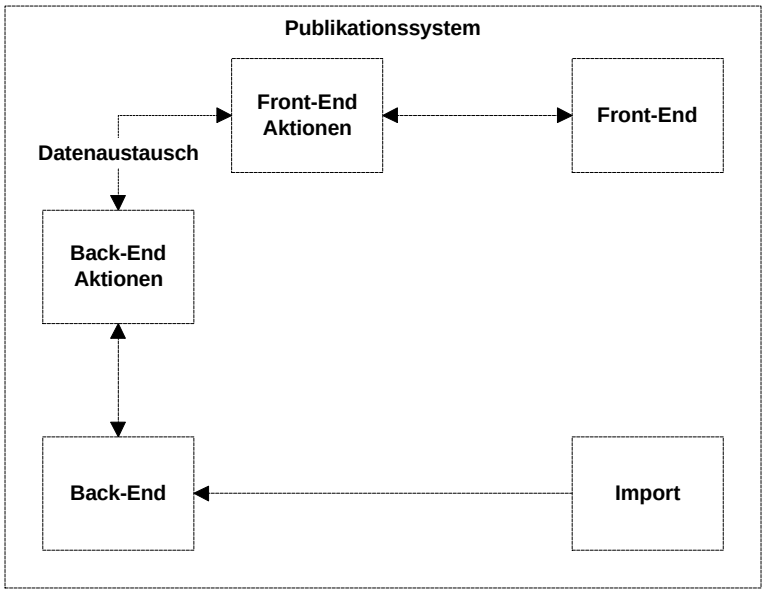


Abbildung 25: Teilsysteme von PEP

Die eindeutige Zuordnung der beiden Aktionen-Teilsysteme zum Front- bzw. Back-End, ermöglicht nun auch die physische Trennung der entsprechenden Funktionalitäten und Strukturen des Teilsystems „Aktionen“ aus Abbildung 24. Die Verbindung zwischen „Front-End Aktionen“ und „Back-End Aktionen“ beschränkt sich nun auf das reine

Austauschen von Daten. Die beiden Teilsysteme könnten somit auch auf verschiedenen Rechensystemen installiert werden. Der Datenaustausch könnte dann beispielsweise über CORBA oder eine andere Schnittstelle geschehen.

Wird von der Möglichkeit der Installation der Aktionen auf zwei Rechnern kein Gebrauch gemacht, stehen zwei zusätzliche Möglichkeiten zur Verfügung, Daten auszutauschen.

1. Der Datenaustausch findet über eine Parameter-Übergabe bei einem direkten Methoden-Aufruf statt.
2. Der Datenaustausch findet über Interprozeß-Kommunikation statt.

Fall 1 hebt die Trennung der Aktionen de facto auf, da beide Systeme in einem Programm zusammengeführt werden müssen. Fall 2 hätte die aufwendige Implementierung sogenannter „Pipes“, „Sockets“ oder „shared memory“ Bereiche zur Kommunikation zwischen den Prozessen zur Folge. Einerseits würde dies zu Geschwindigkeitseinbußen beim Datenaustausch führen, andererseits ist die Idee der Interprozeß-Kommunikation näher am CORBA-Modell, was eine Berücksichtigung beider Möglichkeiten (CORBA und Interprozeß-Kommunikation) während der Implementierung vereinfachen würde. Letztendlich ist über eine geeignete Kapselung der Kommunikations-Funktionalitäten sicher zu stellen, daß „Front-End“ und „Back-End“ vom zugrundeliegenden Kommunikationsmodell abstrahieren können.

Das Teilsystem „Import“ ist für die Aktualisierung bzw. die Erneuerung der Beitragsdaten verantwortlich. „Import“ verwendet Funktionalitäten des „Back-End“. „Back-End“ greift aber nicht auf Funktionalitäten von „Import“ zurück. Für den täglichen Betrieb des Publikationssystems wird „Import“ nicht benötigt. „Import“ wird als separates Programm in regelmäßigen Abständen gestartet, um die aktualisierten Beitragsdaten zu importieren.

Bei einer Aufteilung des Systems nach Abbildung 25 ergibt sich der in Abbildung 26 dargestellte „globale“ Datenfluß für die Kommunikation von PEP mit einem WWW-Server.

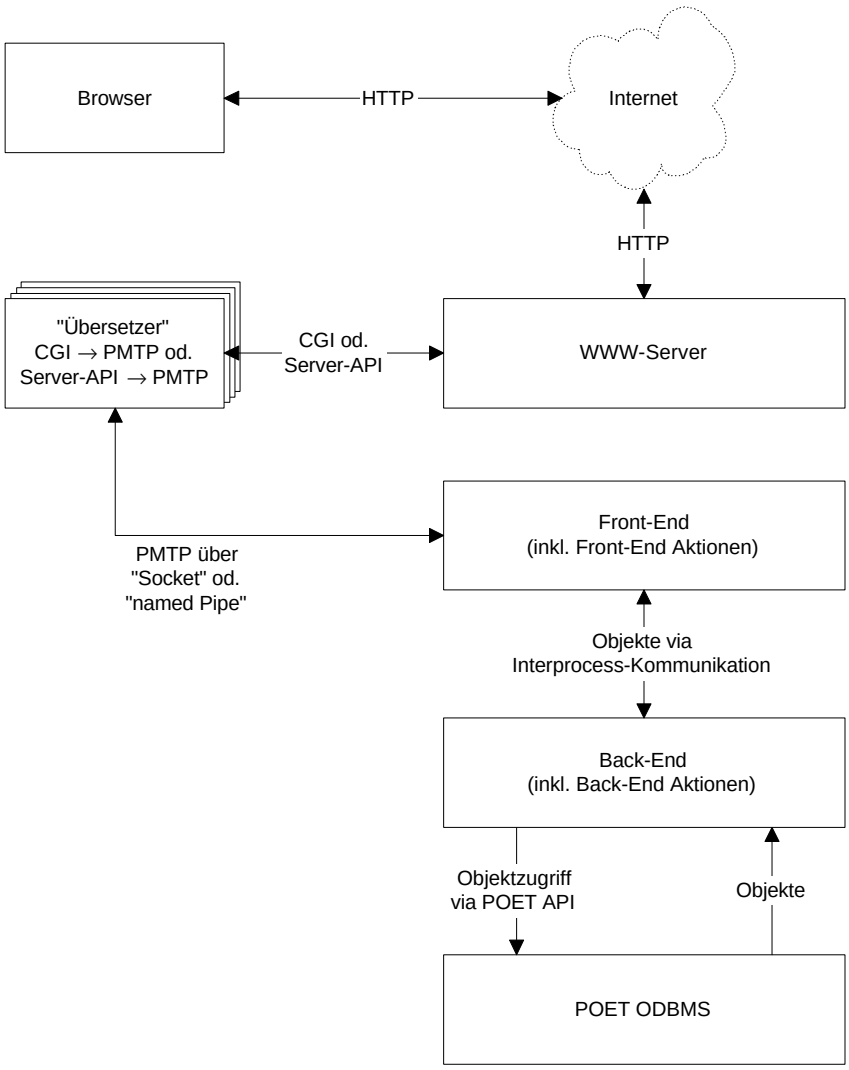


Abbildung 26: Globaler Datenfluß

Die beiden Teilsysteme „Front-End“ und „Back-End“ werden als unabhängige Applikationen gestartet. Beide übernehmen Serverfunktionen, wobei „Front-End“ die Kommunikation mit dem WWW-Server über ein Hilfsprogramm aufbaut. Dieses kleine Hilfsprogramm (hier „Übersetzer“) wird zum Beispiel via CGI vom WWW-Server gestartet. Das Programm liest den Anfrage-String vom WWW-Server, bzw. aus dem Umgebungsspeicher, und reicht ihn an „Front-End“ weiter. Der „Übersetzer“ nutzt dabei das Protokoll **PMTP**[1], um mit „Front-End“ zu kommunizieren. Dieses Hilfsprogramm ist nicht Bestandteil der Modellierung. Es ist für die Realisierung von PEP unerheblich, wie dieses Programm implementiert wird, da es ausschließlich dem Weiterreichen von unstrukturierten Daten dient. Es ist vorstellbar, daß diese Funktionalität per CGI zur Verfügung gestellt wird oder aber auch über ein sogenanntes Server-Modul, daß die API

[1] PMTP (**P**EP **M**ini **T**ransfer **P**rotokoll) ist eine Eigenentwicklung, die im Rahmen dieser Diplomarbeit erstellt worden ist. Es beschränkt sich auf wenige Befehle, die zum Beispiel den Beginn, das Ende oder den Abbruch der Kommunikation mit dem Front-End regeln. Eine vollständige Auflistung aller Befehle des Protokolls befindet sich in Abschnitt 9.8.

des jeweiligen WWW-Servers nutzt. Es ist ebenfalls unerheblich, ob das Hilfsprogramm auf dem selben Rechner läuft, wie „Front-End“, da der Datenaustausch zwischen Hilfsprogramm und „Front-End“ über eine Netzwerk-Verbindung realisiert wird. Die einzigen Voraussetzungen, die für das Hilfsprogramm getroffen werden müssen, sind die vollständige und korrekte Unterstützung von PMTP zur Kommunikation mit dem Front-End und die vollständige unverfälschte Übertragung aller vom WWW-Server zur Verfügung gestellten Daten.

Sinn und Zweck dieses „Umweges“, ist die daraus resultierende Möglichkeit, Front-End und Back-End als Server-Prozeß starten zu können. Auf diese Weise kann beispielsweise von „Back-End“ eine ständige Verbindung zur Datenbank gehalten werden. Würde statt dessen die komplette Applikation als CGI implementiert werden, müßte bei jedem Programmstart eine aufwendige Login-Prozedur durchgeführt werden, um das Programm bei der Datenbank anzumelden. Dieses Datenbank-Login entfällt, bei der obigen Architektur. Ferner ermöglicht dieser Systemaufbau das Vorhalten wiederkehrender Daten im „Front-End“. Bei der Generierung eines Gast-Dokuments müßte das System zum Beispiel jedesmal wieder das Gast-Profil von der Datenbank anfordern. Wird das Gast-Profil jedoch einmal bei Systemstart akquiriert und dann ständig im „Front-End“ vorgehalten, kann die Zeit, die für das Login und die Query benötigt wird, bei jeder Generierung eines Gast-Dokuments eingespart werden. Gleiches gilt für das Vorhalten der *Betreiber* und *Werbungen* innerhalb des Front-End. Da die Inhalte der Instanzen dieser drei Klassen sich selten ändern und sie außerdem keine schützenswerten benutzerspezifischen Informationen enthalten kann dieses Vorgehen sinnvoll sein.[1] Die Inhalte der Instanzen der Klassen *Betreiber* und *Werbung* stehen so bei jedem Generierungsvorgang ohne weitere Datenbank-Zugriffe zur Verfügung.

Abschießend ist zu erwähnen, daß die Architektur des Systems eine weite „Streuung“ der einzelnen Teilsysteme und Systemkomponenten über mehrere Computersysteme ermöglicht. So ist es beispielsweise vorstellbar, über mehrere WWW-Server auf ein zentrales Publikationssystem zuzugreifen. Auch der Zugriff von einem WWW-Server auf mehrere Publikationssysteme ist durch eine geeignete Implementierung des Hilfsprogramms realisierbar. Ferner kann durch die Verwendung des Hilfsprogramms vom verwendeten Server abstrahiert werden, da die über das PMTP-Protokoll übertragenen Daten in beliebige andere Kommunikations-Protokolle integriert werden können.

5.3.2 Datenspeicher-Strukturen

Wie in Kapitel 4.3 ausgeführt, wurde für die Realisierung des Systems das Objekt Datenbank Management System (ODBMS) POET des gleichnamigen Herstellers verwendet. Zum Einsatz kam das POET ODBMS in der Version 5.0 für Solaris 2.6.

Bei der Betrachtung der Datenspeicher-Strukturen kommt einer der Vorteile eines ODBMS zum Tragen. Die Festlegung der Strukturen der Datenbank-Tabellen eines

[1] Das Gast-Profil enthält zwar Profil-Einstellungen, da aber dieses Profil keinem Benutzer zugeordnet wird, handelt es sich nicht um benutzerspezifische Informationen.

relationalen Datenbanksystems entfällt. Vielmehr wird der persistente Teil des Objektmodells unverändert in eine Datenbank-Struktur übernommen. Die endgültige Festlegung, welcher Teil des Objektmodells innerhalb der Datenbank abgelegt werden soll, erfolgt in Kapitel 6.1. Die folgende Betrachtungen erörtern die Verzeichnisstrukturen, die sich aus der Verwendung des POET-Systems sowie aus der Berücksichtigung des Analysemodells und des Systementwurfs ergeben.

Alle Beitrags- und Benutzerdaten werden in einer objektorientierten Datenbank gespeichert. Die Architektur des POET ODBMS sieht ein Verzeichnis für die Speicherung der eigentlichen Daten innerhalb einer Datenbankdatei vor („pepbase“). Ein anderes Verzeichnis beinhaltet das sogenannte Dictionary der Datenbank („pepdict“). In diesem Dictionary sind die strukturellen Informationen der einzelnen Klassen gespeichert (Attribute, Vererbungen etc.).

/pep	Basisverzeichnis (beliebig)
/bin	PEP Programme („binaries“)
/pepbase	Datenbank
/pepdict	Datenbank-Dictionary
/bilder	Bilder (Beiträge, Werbung)
/log	PEP System-Logdateien
/hoerzu	Betreiber-Verzeichnis (hier Betreiber „HÖRZU“)
/vorlagen	Dokument-Vorlagen des Betreibers
/log	Betreiberspezifische Logdateien
/bilder	Betreiberspezifische Grafikdateien
/gast_html	Bereits generierte HTML-Dokumente aus dem Gast-Bereich

Tabelle 7: Verzeichnisstruktur

Die Grafiken der einzelnen Beiträge bzw. Werbungen werden innerhalb der Verzeichnisstruktur abgelegt. Für jeden Betreiber des Publikationssystems wird ein Unterverzeichnis angelegt, das betreiberspezifische Dateien enthält. Dies sind vor allem sogenannte Dokument-Vorlagen, die als Schablonen für die Generierung von Dokumenten dienen. Sie enthalten das Grundlayout eines Dokuments und Platzhalter, die zur Laufzeit durch aktuelle Werte aus der Datenbank ersetzt werden. Diese Dokument-Vorlagen werden im Unterverzeichnis „vorlagen“ gespeichert. Weiterhin enthält ein Betreiber-Verzeichnis das Unterverzeichnis „log“, in dem Logdateien abgelegt werden. Diese Logdateien enthalten Meldungen über betreiberspezfische Fehler, die zur Laufzeit des Systems auftreten (zum Beispiel Generierungsfehler, die auf fehlerhafte Vorlagen zurückzuführen sind). Das Unterverzeichnis „bilder“ enthält zum Beispiel Logo-Grafiken des Betreibers. Innerhalb des Unterverzeichnisses „gast_html“ werden bereits generierte Gast HTML-Dokumente abgelegt. Diese Speicherung von wiederholt verwendeten HTML-Dokumenten dient der Optimierung des Systems, da anstatt einer erneuten Generierung lediglich das Lesen der fertigen Dokumente aus der Verzeichnisstruktur erforderlich ist.

5.3.3 Sicherheit

Die Hauptaufgabe des in dieser Diplomarbeit beschriebenen Publikationssystems ist die personalisierte Aufbereitung allgemein zugänglicher Informationen für einen begrenzten Personenkreis – die Abonnenten. Um diese Aufgabe erfüllen zu können, werden innerhalb verschiedener Datenbanken sowohl öffentliche als auch personenbezogene Daten gespeichert. Für die personenbezogenen Daten sind **Vertraulichkeit**, **Authentizität** und **Integrität** zu gewährleisten. Da die oben genannten Leistungen des Publikationssystems kostenpflichtig gestaltet werden können, ist außerdem die **Verfügbarkeit** des Systems sicherzustellen.[1] Ferner muß verhindert werden, daß „Nicht-Abonnenten" in den Genuß der Abonnenten-Leistungen des Publikationssystems kommen.

Erforderliche Schutzmaßnahmen unter Berücksichtigung des IuKDG
Das am 1. August 1997 in Kraft getretene Informations- und Kommunikationsdienste-Gesetz (IuKDG) setzt u.a. den für die Verarbeitung personenbezogener Daten maßgeblichen Gesetzesrahmen. Nach Artikel 1, §2 Abs. 2 Satz 2 und §2 Abs. 2 Satz 5 des IuKDG stellt PEP einen sogenannten „Teledienst" dar[2]. Eine Berücksichtigung des IuKDG bei der Realisierung des Systems ist somit geboten. Wie in [Münch97, Kapitel 6.3] erörtert, sind „ [...] die personenbezogenen Daten der Benutzer des PEP-Systems gegen Kenntnisnahme Dritter zu schützen [...]" ([Münch97, S. 75]). Um diese Kenntnisnahme zu verhindern, gilt es u.a. die Profil-Datenbank vor einem unbefugten, direkten Zugriff zu schützen. Dies geschieht in erster Linie durch den Schutz des Servers, auf dem die Datenbank mit den personenbezogenen Daten physikalisch abgelegt ist. Kapitel 3.1.2 aus [Münch97] erörtert verschiedene Mechanismen, diesen Schutz zu realisieren.

Versagen diese Schutzmaßnahmen, muß die Zuordnung eines Benutzer-Profils zu einer natürlichen Person verhindert, zumindest aber massiv erschwert werden. Eine Möglichkeit, eine solche Zuordnung vorzunehmen, wird durch die im Benutzer-Profil gespeicherte E-Mail-Adresse eines Benutzers eröffnet. Folglich muß ein unbefugter Zugriff auf die E-Mail-Adresse verhindert werden. Dies wird durch eine Verschlüsselung der E-Mail-Adresse mit einem geeigneten Verschlüsselungsverfahren (zum Beispiel DES [Garfinkel96, S. 154]) erreicht. Ferner muß verhindert werden, daß Dritte das Paßwort des Abonnenten in Erfahrung bringen, um über eine reguläre Anmeldung beim System die E-Mail-Adresse des Abonnenten zu erhalten. Aus diesem Grund muß auch das Paßwort im Benutzer-Profil verschlüsselt werden.

Die Forderung nach Schutz der personenbezogenen Daten gilt auch für die Kommunikation des Benutzers mit dem Publikationssystem über das Internet. Hier ist sicherzustellen, daß bei der Übertragung der relevanten Daten ein sicheres Übertragungsprotokoll Verwendung findet (zum Beispiel S-HTTP). Die Kommunikation zwischen den einzelnen Teilsystemen des Publikationssystems bedarf, sofern sie über ein Netzwerk geschieht, ebenfalls geeigneter Schutzmaßnahmen. Wird das in Kapitel 5.3.1 erläuterte Hilfsprogramm („Übersetzer") auf einem anderen Server betrieben als das

---

[1] Definitionen zu Vertraulichkeit, Authentizität, Integrität und Verfügbarkeit finden sich u.a. in [Münch97], Kapitel 3.1.2.

[2] vgl. [Münch97], Kapitel 6.2 und [IuKDG97].

Publikationssystem, muß die Kommunikation zwischen Front-End und Hilfsprogramm gegebenenfalls durch eine Verschlüsselung gesichert werden. Entsprechendes gilt für die Kommunikation zwischen Front-End und Back-End und zwischen Back-End und ODBMS.

Mit den oben aufgeführten Maßnahmen soll die **Vertraulichkeit** der personenbezogenen Daten gewährleistet werden.

Authentifikation eines Abonnenten
Bevor ein Benutzer Abonnent von PEP werden kann, muß er sich über eine Anmeldeprozedur beim System bekannt machen. Der zukünftige Abonnent kann dabei einen Benutzernamen und ein Paßwort frei wählen. Die spätere Änderung des Paßwortes ist möglich, nicht jedoch die Änderung des Benutzernamens. Damit ein Abonnent in den Genuß der regelmäßigen personalisierten Zeitschrift kommen kann, muß er ferner über eine gültige E-Mail-Adresse verfügen und diese dem Publikationssystem bei der Erstanmeldung mitteilen. Nach der ersten Anmeldung eines Abonnenten und einer eventuellen Zahlung zur Freischaltung des Abonnements, kann der somit registrierte Benutzer persönliche Einstellungen an seinem Benutzer-Profil vornehmen. Nach einer erfolgreichen Erstanmeldung kann sich der Abonnent jederzeit mit seinem Benutzernamen und seinem Paßwort gegenüber dem Publikationssystem authentifizieren.

Die Verwendung des HTTP-Protokolls birgt einige Schwierigkeiten bezüglich der Authentifikation. Da es sich beim HTTP-Protokoll um ein zustandsloses Protokoll handelt, geht eine einmal vorgenommene Identifikation eines Benutzers bei jedem erneuten Zugriff verloren. Um zu verhindern, daß sich ein Abonnent bei jedem Seitenabruf erneut beim System anmelden muß, ist die Speicherung des durch einen erfolgreichen Anmeldevorgang erreichten Status des Abonnenten geboten. Diese Speicherung wird durch **Session-IDs** erreicht.

Bei jedem Anmeldevorgang erhält ein Abonnent eine Session-ID. Diese Session-ID ist unique und beinhaltet implizit den Anmeldestatus eines Abonnenten. Die aktuelle, an den Abonnenten vergebene, Session-ID wird in der Datenbank gespeichert und bei jedem Seitenabruf des Abonnenten auf Gültigkeit geprüft. Die Verwendung von Session-IDs bietet den Vorteil, daß nicht auf bestehende Authentifizierungsmechanismen[1] zurückgegriffen werden muß. Zum Schutz der Übertragung des Benutzernamens und des dazu gehörigen Paßworts können beliebige Verschlüsselungs- und Authentifizierungsmechanismen verwendet werden.

Eine Session-ID verliert ihre Gültigkeit grundsätzlich nach 24 Stunden, unabhängig davon, wie häufig sie während dieser Zeit verwendet wurde. Schon im Front-End kann auf diese Weise überprüft werden, ob eine vom Client übermittelte Session-ID gültig ist. Für diese Prüfung ist kein Zugriff auf die Datenbank nötig, da die Information über das Alter in der Session-ID festgehalten ist. Die Session-ID verfällt zusätzlich nach Ablauf von 30 Minuten, in denen der Abonnent keinen Zugriff auf das System getätigt hat. Bei jedem Zugriff eines Abonnenten auf das System muß demnach die Uhrzeit des Zugriffs

[1] zum Beispiel Basic Authentification [RFC1521].

gespeichert werden. Verläßt ein beim Publikationssystem angemeldeter Abonnent seinen Rechner und befindet sich innerhalb des Browsers eine Seite aus dem Abonnenten-Bereich, kann ein Unbefugter ohne Anmeldung die Abonnenten-Leistungen des Publikationssystems in Anspruch nehmen. Allerdings muß der Zugriff innerhalb von 30 Minuten nach Verlassen des Rechners durchgeführt werden. Nach Ablauf dieser Frist ist eine erneute Anmeldung notwendig.

Sind Benutzername und Paßwort eines Abonnenten vom System bei der Anmeldung verifiziert worden, wird eine neue Session-ID generiert. Die Session-ID beinhaltet folgende Daten:

- Datum und Uhrzeit der Anmeldung
- die ID des Profil-Objektes des Abonnenten
- clientseitige Informationen (beispielsweise die IP des Benutzer-Clients in maskierter Form)

Datum und Uhrzeit dienen, wie bereits erwähnt, der Überprüfung der Gültigkeit einer Session-ID. Die ID des Profil-Objektes soll einerseits sicherstellen, daß eine Session-ID unique ist.[1] Andererseits dient diese Objekt-ID dem schnellen Zugriff auf das in der Datenbank gespeicherte Profil-Objekt.[2] Die clientseitigen Informationen sollen verhindern, daß eine Session-ID auf ihrem Weg durch das Internet abgefangen und von Dritten verwendet werden kann. Die Integration möglichst eindeutiger, identifizierender Daten des Client-Rechners verhindert eine Nutzung der Session-ID von einem anderen Client als dem Abonnenten-Client. Allerdings ist sicherzustellen, daß anhand der Session-ID keine Zuordnung zu einem konkreten Computer hergestellt werden kann. Die Prüfung auf die Identität des Abonnenten würde sich auf eine positive Prüfung beschränken. Der clientseitige Anteil der Session-ID würde bei jedem Zugriff eines Abonnenten erneut generiert und dann mit der übermittelten Session-ID abgeglichen werden. Vorstellbar wäre hierfür eine maskierte Verwendung der Client-IP. Die Maskierung der Client-IP soll dabei die Rückgewinnung der IP aus der Session-ID verhindern. Allerdings wäre bei diesem Vorgehen die Möglichkeit nicht berücksichtigt, daß innerhalb eines lokalen Netzwerks Session-IDs abgefangen werden. Nutzen beide Clients, Abonnenten- und Angreifer-Client, beispielsweise den selben Proxy-Server, ist eine Unterscheidung über die IP nicht möglich.[3] Dementsprechend sollten noch weitere Client-Informationen für die Session-ID verwendet werden. An dieser Stelle sollte jedoch das Verhältnis zwischen dem Aufwand für die Verwendung einer Session-ID mit dem daraus entstehenden Nutzen betrachtet werden.

Die Verwendung von Session-IDs soll die **Authentizität** der vom Client an den Server übermittelten Daten gewährleisten. Die Nutzungsrechte, für die ein Benutzer über die Identifikation mittels einer Session-ID autorisiert ist, sind sehr beschränkt. Er kann

---

[1] Die Kombination aus Datum/Uhrzeit und Objekt-ID liefert in jedem Fall eine unique Session-ID, da die Objekt-ID innerhalb der Datenbank und die Kombination Datum/Uhrzeit jeweils unique sind.

[2] Der Zugriff auf ein Profil-Objekt wird in Kapitel 5.4.1 näher erläutert.

[3] Vorrausetzung für diesen Fall ist, daß der Proxy-Server IP-Masquerading verwendet.

lediglich die persönlich aufbereiteten öffentlichen Informationen (TV-Programminformationen) betrachten. Die Änderung der persönliche Einstellungen, die u.a. das Erscheinungsbild der personalisierten Zeitschrift beeinflussen, erfordert eine zusätzliche Authentifikation des Abonnenten. Die Verwendung von Session-IDs kann **Vertraulichkeit** und **Integrität** der personenbezogenen Daten nur auf einem niedrigen Sicherheitsniveau gewährleisten. Um das Sicherheitsniveau in diesem Zusammenhang zu erhöhen, ist es notwendig, bei der Ansicht und Änderung der persönlichen Einstellungen eines Abonnenten eine erneute Authentifikation vorzunehmen. Die im Anschluß an diese Authentifikation über das Internet übermittelten Daten müssen dann unter Verwendung eines sicheren Übertragungsprotokolls gesichert werden.

Ausfallsicherheit und Systemlast
Da ein Benutzer für die Leistungen des Publikationssystems unter Umständen bezahlt hat, ist sicherzustellen, daß die bezahlten Leistungen vom Publikationssystem erbracht werden. Mit anderen Worten: die **Verfügbarkeit** des Systems ist zu gewährleisten. Die Erfüllung der Forderung nach Verfügbarkeit wird durch die in Kapitel 5.3.1 eingeführte Architektur des Gesamtsystems unterstützt. Durch die mögliche Verteilung der Teilsysteme auf unterschiedliche Rechner, können während des laufenden Betriebes ein oder mehrere Backup-Systeme parallel betrieben werden. Fällt das Hauptsystem aus, kann eines der Backup-Systeme den Betrieb ohne Verzögerung übernehmen. Der Zugriff beispielsweise des „Übersetzers“ auf das alternative System wird durch eine einfache „Umleitung“ der Netzwerkverbindung zwischen „Übersetzer“ und Front-End geregelt.

Um lange Wartezeiten eines Benutzers zu vermeiden, können auch mehrere Systeme als Hauptsysteme ausgestattet werden. Ist die Systemlast eines Hauptsystems sehr hoch, könnte ein „intelligentes Routing“ eine Netzwerkverbindung zu dem System herstellen, das eine geringere Systemlast aufweist. Dieses Routing könnte auch die einzelnen Anfragen eines Servers auf verschiedene Systeme verteilen, um die Last gleichmäßig zwischen den Systemen aufzuteilen.

Das folgende Kapitel beschäftigt sich mit Aspekten des Systementwurfs, die ebenfalls Einfluß auf die Sicherheit des Systems haben. Die Behandlung von Grenzbedingungen dient u.a. der Erhaltung der Integrität der in den Datenbanken gespeicherten Informationen.

5.3.4 Behandlung von Grenzbedingungen

Initialisierung, Terminierung und Absturz des Systems sind Grenzbedingungen, die vor der weiteren Verfeinerung des OMT-Modells berücksichtigt werden müssen. Die Einführung einer umfassenden Ausnahmebehandlung ist nicht Bestandteil dieses Kapitels. Vielmehr sollen hier die Probleme bei Systemstart und Terminierung sowie die möglichen Auswirkungen eines Systemabsturzes erörtert werden.

Systemstart und Terminierung
Um den beim Systemstart zu betrachtenden Übergang des Systems vom Anfangs- in den Normalzustand zu erörtern, wird zunächst der Normalzustand des Publikationssystems umgangssprachlich formuliert:

Im Normalzustand wartet das Publikationssystem auf eingehende Socket-Verbindungen auf einem dedizierten Port. „Front-End“ ist in Verbindung zu „Back-End“, das wiederum eine ständige Verbindung zum Datenbank-Server hält. „Front-End“ hält das Gast-Profil, alle Betreiber-Objekte und alle Werbung-Objekte aus der Datenbank im Speicher

Diese oben beschriebenen Socket-Verbindungen finden sich in der Relation zwischen dem Hilfsprogramm und „Front-End“ in Abbildung 26. Um das System vom Anfangszustand in den Normalzustand zu überführen, müssen folgende Schritte chronologisch abgearbeitet werden:

1. Aufbau der Verbindung von „Back-End“ zum Datenbank-Server.
2. Aufbau der Verbindung von „Front-End“ zu „Back-End“.
3. Holen des Gast-Profils, der Betreiber-Objekte und der Werbung-Objekte aus der Datenbank.
4. Initialisierung des Sockets.

Um das Publikationssystem aus dem Normalzustand zu terminieren, müssen folgende Schritte chronologisch abgearbeitet werden:

1. Deinitialisierung des Sockets.
2. Löschen des Gast-Profils, der Betreiber-Objekte und der Werbung-Objekte aus dem Speicher.
3. Trennen der Verbindung von „Front-End“ zu „Back-End“.
4. Trennen der Verbindung von „Back-End“ zum Datenbank-Server.

Um das Publikationssystem aus einem anderen Zustand als dem Normalzustand zu terminieren, muß zuerst die Trennung aller bestehenden Socket-Verbindungen vom Hilfsprogramm zu „Front-End“ erfolgen. Danach wird verfahren wie bei der Terminierung aus dem Normalzustand.

Systemabsturz

Die Behandlung von Grenzbedingungen dient auch der Einhaltung der Integrität der gespeicherten Daten. Der Absturz des System sollte diese Integrität nicht in Frage stellen. Folgende Vorgänge müssen im Zusammenhang mit einem Systemabsturz näher betrachtet werden.

Durchführung einer Zahlung seitens eines Benutzers

Bei der Durchführung einer Zahlung, ist sicherzustellen, daß ein abgeschlossener Zahlungsvorgang des Kunden in jedem Fall einen entsprechenden Eintrag des bezahlten Abonnement-Zeitraums in der Datenbank zur Folge hat. Zu diesem Zweck muß der Zahlungsvorgang in chronologisch aufeinander folgende Schritte eingeteilt werden: [1]

[1] Die einzelnen Schritte des Zahlungsvorgangs werden an dieser Stelle als atomar angesehen. Da aber zum Beispiel die Übermittlung der Daten über ein Netzwerk geschieht, muß eine adäquate Ausnahmebehandlung auf mögliche Übertragungsfehler entsprechend reagieren. Gleiches gilt für den Eintrag des Abonnement-Zeitraums in die Datenbank, dessen Konsistenz durch die Verwendung von

1. Initialisierung des Zahlungsvorgangs bei PEP durch den Benutzer.
2. Initialisierung des Zahlungsvorgangs von PEP beim Abrechnungsanbieter.
3. Übermitteln der Benutzerdaten an den Abrechnungsanbieter.
4. Überprüfung der Bonität des Benutzers durch den Abrechnungsanbieter.
5. Endgültige Bestätigung des Benutzers zur Durchführung des Zahlungsvorgangs.
6. Meldung über die Bonität und das Einverständnis des Benutzers vom Abrechnungsanbieter an PEP.
7. Eintrag des Abonnement-Zeitraums in die Datenbank.
8. Nachricht von PEP an den Abrechnungsanbieter, daß der Zahlungsvorgang vorgenommen werden kann.
9. Durchführung der Zahlung mit abschließender Erfolgsmeldung an den Benutzer.

Die Schritte 3 bis 6 und der Schritt 9 werden vom externen Abrechnungsanbieter durchgeführt, so daß für die Betrachtung der Datenkonsistenz lediglich die Schritte 1, 2, 7 und 8 von Bedeutung sind. Ein Absturz des Systems zwischen Schritt 1 bis 3 bzw. zwischen Schritt 6 und 7 würde einen Abbruch des Zahlungsvorgangs hervorrufen. Ein Absturz zwischen Schritt 7 und 8 würde den ungerechtfertigten Eintrag des Abonnement-Zeitraums in die Datenbank hervorrufen, da die Durchführung der eigentlichen Zahlung nicht initialisiert werden konnte. Im schlimmsten Fall würde ein Benutzer also ein Abonnement nutzen können, für das keine Zahlung vorgenommen wurde. Um die entstehenden Nachteile für diesen Fall zu reduzieren, sollte der Eintrag des Abonnement-Zeitraums als „vorläufig“ gekennzeichnet werden und der ursprüngliche Abonnement-Zeitraum sollte innerhalb des Profils zwischengespeichert werden. Die Löschung dieser Kennzeichnung muß dann nach Eingang des Geldes auf das Konto des Betreibers vorgenommen werden. Nach Ablauf eines bestimmten Zeitraums, in dem kein Geld beim Betreiber eingegangen ist, führt das Vorhandensein der Markierung „vorläufig“ zum Zurücksetzen des Abonnement-Zeitraum-Eintrags auf den vorherigen Wert.

Änderungen der Profileinstellungen eines Benutzers

Ein mittels einer Zahlung freigeschaltetes Profil kann von einem Benutzer über ein HTML-Front-End seinen Bedürfnissen entsprechend angepaßt werden. Zu diesem Zweck werden die aktuellen Einstellungen des Benutzer-Profils zunächst benutzt, um ein Profileinstellungsformular im HTML-Format zu generieren. Dieses HTML-Formular wird via HTTP (bzw. S-HTTP) an den Browser des Benutzers übertragen. In diesem Formular kann der Benutzer dann seine individuellen Einstellungen vornehmen und sie anschließend an das System zurück übertragen.

HTTP (bzw. S-HTTP) ist ein zustandsloses Protokoll. Daher ist ein Absturz zwischen der Anforderung des Benutzerprofils für die Generierung des HTML-Formulars und dem Zurücksenden der geänderten Einstellungen für die Integrität der Daten unkritisch. Der Benutzer erhält dann vom System keine Rückmeldung, bzw. eine Fehlermeldung des Hilfsprogramms, daß keine Verbindung zum Publikationssystem aufgebaut werden konnte. Die kritische Phase bei der Änderung von Profileinstellungen ist der Zeitraum zwischen dem Eintreffen der geänderten Einstellungen des Benutzers und der Rückmeldung über die

---

Transaktionsmechanismen sichergestellt werden muß. Zur Vermeidung von Inkonsistenzen werden die Transaktionsmechanismen des POET ODBMS bei schreibenden Zugriffen auf die Datenbank genutzt.

Eintragung der Änderung in die Datenbank. Diese Phase kann in folgende Schritte aufgeteilt werden:

1. Eintreffen der geänderten Benutzer-Einstellungen.
2. Anforderung einer Instanz der Klasse Profil mit den entsprechenden Einstellungen des Benutzers zwecks Eintragung der Änderungen.
3. Eintragen der Änderungen in das Objekt „Benutzer-Profil".
4. Speichern des geänderten Benutzer-Profils in der Datenbank.
5. Rückmeldung über die Speicherung an den Benutzer.

Durch den einmaligen schreibenden Zugriff auf die Datenbank in Schritt 4 wird die Speicherung der Änderungen aus der Sicht des Publikationssystems atomar. Für die Konsistenz, bzw. Integrität der zur Speicherung übergebenen Daten ist das ODBMS verantwortlich. Auf einen Absturz des Datenbank-Servers zu diesem Zeitpunkt muß vom Publikationssystem mit einer entsprechenden Fehlermeldung an den Benutzer reagiert werden.

Ein Absturz des Publikationssystems birgt in diesem Szenario jedoch nicht die einzige Quelle möglicher Inkonsistenz. Wird vom Benutzer vor Beendigung des Schrittes 4 – beispielsweise über eine zweite lokale Browser-Instanz – ein weiterer Änderungsvorgang initialisiert, ist nicht vorhersehbar, welche der beiden angestoßenen Änderungen in welcher Reihenfolge welche Änderungen in der Datenbank einträgt. Diese Konstellation kann zu Inkonsistenzen im Datenbestand führen. Aus diesem Grund wird vor Beginn des Schrittes 2 eine Transaktion gestartet, die mit Schritt 4 beendet wird. Diese Transaktion verhindert einen parallel schreibenden Zugriff auf das Profil-Objekt.

Regelmäßige Dokument-Generierung
Die einzige Leistung des Publikationssystems, die von Abonnenten bezahlt werden muß, ist der regelmäßige Versand der personalisierten TV-Programmzeitschrift. Es ist daher sicherzustellen, daß alle Abonnenten ihre personalisierte Zeitschrift erhalten. Der Versand wird für jeden Benutzer, je nach Wunsch, wöchentlich oder täglich durchgeführt. Abbildung 27 zeigt den Vorgang der Versendung der TV-Programmzeitschriften anhand eines Zustandsdiagramms.[1]

[1] In Abbildung 27 wird das Holen der Beiträge und der Werbung in der Aktion „generiere Dokument" vereint, da diese Vorgänge für die Ausführungen über einen Systemabsturz an dieser Stelle uninteressant sind.

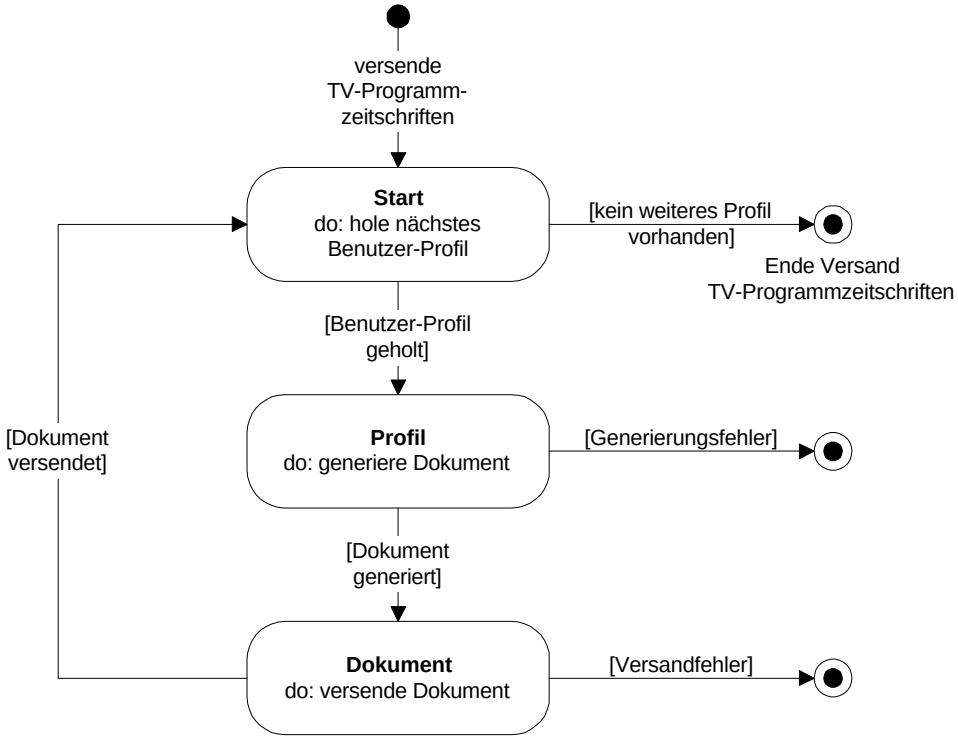


Abbildung 27: Zustandsdiagramm „versende TV-Programmzeitschriften“

Wird der Endzustand „Ende Versand TV-Programmzeitschriften“ nicht erreicht, so hat eine Teilmenge der Abonnenten keine Zeitschrift erhalten. Dies kann durch einen Absturz des Systems oder durch einen der in Abbildung 27 dargestellten Fehler geschehen. Um sicherzustellen, daß nach einem Systemabsturz bzw. nach einem Fehler alle noch nicht berücksichtigten Abonnenten ihre Zeitschrift erhalten, muß jeder erfolgreiche abgeschlossene Versand einer TV-Programmzeitschrift festgehalten werden. Dies kann durch das Schreiben der Versandvorgänge in eine Logdatei geschehen oder durch ein zusätzliches Attribut der Klasse Profil, das das Datum des letzten erfolgreichen Versands enthält. Der aus den oben erhaltenen Bedingungen entstehende Zustandsautomat ist in Abbildung 28 dargestellt.

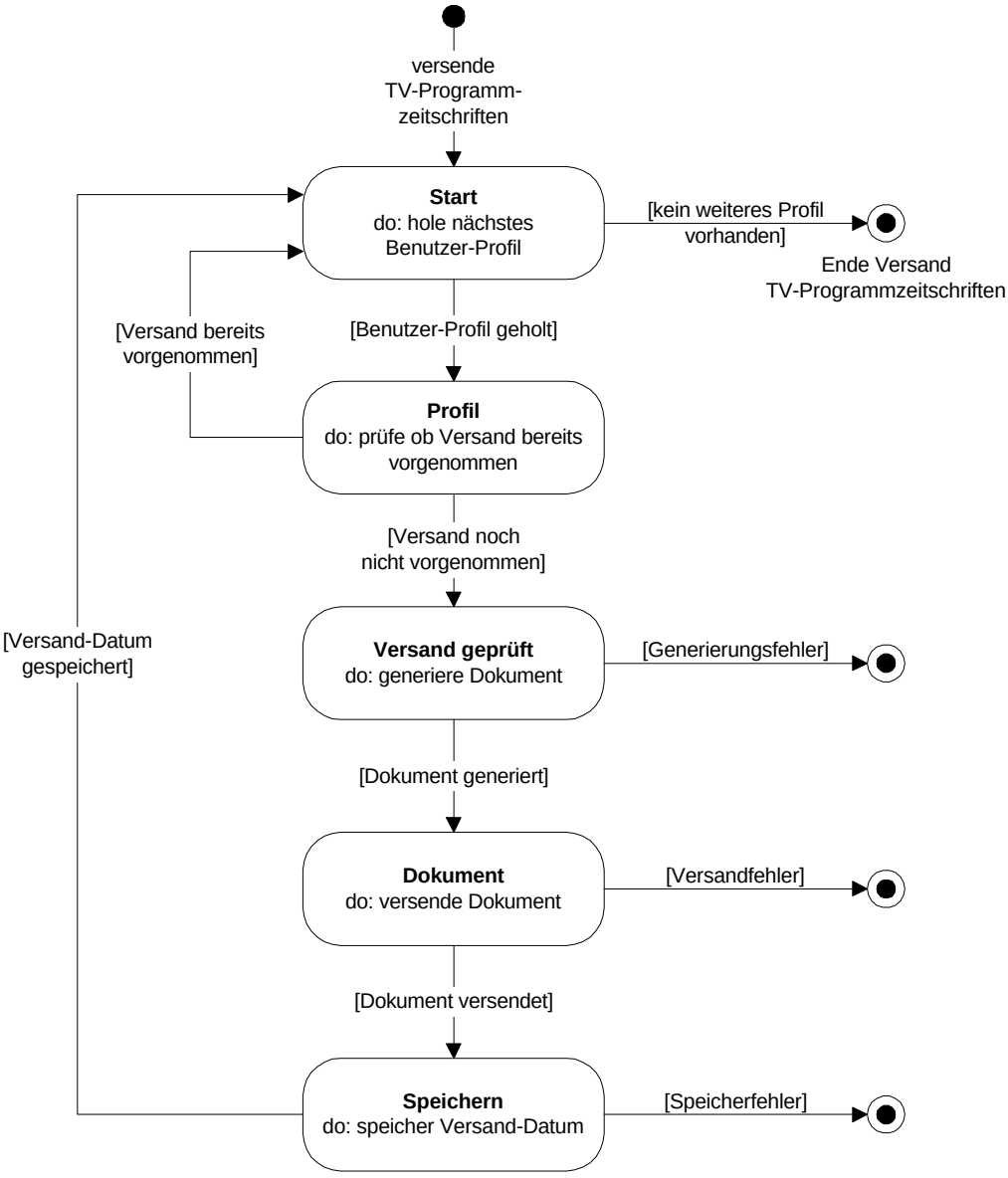


Abbildung 28: Zustandsdiagramm „versende TV-Programmzeitschriften - mit Überprüfung des Versand-Datums“ (Systementwurf)

Die im Abschnitt Systementwurf getroffenen Entscheidungen müssen in die während der Analyse entwickelten Modelle integriert werden. Dies geschieht durch die Einführung neuer Klassen und Attribute sowie durch die Modellierung ergänzender Methoden. Das folgende Kapitel erörtert diese Verfeinerungen anhand der bereits in Kapitel 5.2 verwendeten Modelle.

5.4 Objektentwurf

Im Objektentwurf wird das komplette Analysemodell von PEP durch Verfeinerungen, Erweiterungen und unter Berücksichtigung der im Systementwurf festgelegten Architekturentscheidungen in ein implementierungsfähiges Entwurfsmodell überführt. Der Überführungsvorgang ist dabei ein iterativ zu durchlaufender Prozeß, indem die drei

Modelle des Analysemodells von Abstraktionsebene zu Abstraktionsebene weiter verfeinert werden (vgl. Abbildung 10). Neben der Verfeinerung der Modelle, z.B. durch hinzufügen neuer Attribute, Operationen und Assoziationen, spielt im Objektentwurf auch die Identifizierung neuer interner Klassen eine wichtige Rolle. Die im Entwurf getroffenen Entscheidungen und Veränderungen am System fließen nicht nur in die Modelle von PEP ein, sondern führen auch zu einer ausführlicheren Dokumentation im Data-Dictionary.

In diesem Kapitel sollen die Details aus dem Entwurfsergebnis näher erörtert werden. Dazu werden grundlegende Bereiche aus dem Objektmodell und dem dynamischen Modell vorgestellt. Das funktionale Modell von PEP wurde im Objektentwurf nicht weiter verfeinert[1]. Der vollständige Objektentwurf, bestehend aus den Klassendiagrammen, Ereignispfaddiagrammen, Ereignisflußdiagrammen, Zustandsdiagrammen, Datenflußdiagrammen und dem Data-Dictionary, befindet sich im Band 2 dieser Diplomarbeit.

5.4.1 Objektmodell

Das Objektmodell ist für die Beschreibung des Systems PEP der wichtigste Bezugsrahmen und bildet daher auch die Grundlage für den Objektentwurf. Ausgehend von dem abstrakten Objektmodell der Analysephase wurden im Entwurf schrittweise Assoziationen und Attribute sowie Objekte für die interne Verarbeitung hinzugefügt. Neben diesen Verfeinerungen fanden auch grundlegende Umstrukturierungen am Objektmodell statt. So wurden zum Beispiel die im Systementwurf getroffenen Entscheidungen in das Modell eingearbeitet. Ziel des Objektentwurfs ist es, die logische Struktur des Analysemodells in eine physikalische Programmorganisation zu überführen. Dazu mußten aus den zum Teil noch recht abstrakten Klassen, Attributen und Assoziationen des Analysemodells spezifische Datenstrukturen für die Implementierung entwickelt werden.

In diesem Abschnitt werden jetzt einige Teilbereiche des endgültigen Objektmodells von PEP erläutert. Teilbereiche deshalb, weil das gesamte Objektmodell zu umfangreich für eine detaillierte Beschreibung ist. Die einzelnen Teilbereiche wurden jedoch so ausgewählt, daß sie einen möglichst guten Einblick/Einstieg in das Entwurfsmodell von PEP vermitteln. Dabei wird folgendermaßen vorgegangen: Zuerst wird anhand eines Übersicht-Klassendiagramms die endgültige Struktur von PEP erläutert. Im Anschluß daran werden, ausgehend von dieser groben Übersicht, die statischen Strukturen der Teilbereiche beschrieben, die für den Vorgang der Dokument-Generierung benötigt werden.

Klassendiagramm Übersicht

Das in Abbildung 29 auf Seite 74 dargestellte Klassendiagramm liefert einen Überblick auf das Entwurfsergebnis der Objektmodellierung von PEP. Dieses „Übersichtsdiagramm“ verdeutlicht das Zusammenwirken der wesentlichen Komponenten des Systems. Alle weiteren Klassendiagramme des Objektmodells entsprechen Verfeinerungen zu den in

[1] Das funktionale Modell von PEP besitzt für das System keine ausreichende Aussagekraft, so daß die Modellierung auf einer sehr hohen Abstraktionsebene in der Analyse beendet wurde (vgl. Kapitel 5.2.3)

diesem Übersichtsdiagramm dargestellten Komponenten. Die Grundlage für dieses Übersichtsdiagramm wurde bereits mit dem in der Analysephase modellierten und beschriebenen „Klassendiagramm Übersicht (Analyse)“ geschaffen (vgl. Kapitel 5.2.1, Abbildung 12). Die später im Systementwurf (Kapitel 5.3.1) getroffenen Entscheidungen zum Systemdesign und zur Optimierung von PEP führten zu grundlegenden Veränderungen innerhalb dieses ersten Übersichtsdiagramms, die im folgenden erläutert werden.

Die Klasse *Server*

Die Klasse *Server* in Abbildung 29 ist nicht mehr, wie in der Analyse definiert, eine Superklasse von *Web_Server* und *Mail_Server* (vgl. Kapitel 5.2.1, Abbildung 12 und Tabelle 4). Es handelt sich vielmehr um eine Abstraktion, die beliebige, reale Server[1] und den im Systementwurf eingeführten „Übersetzer“ beschreibt (vgl. Kapitel 5.3.1, Abbildung 26). Dieser „Übersetzer“ dient als Schnittstelle zwischen dem Front-End und dem jeweiligen realen Server. Für jeden „Dienst“ (realen Server) existiert ein eigenes Übersetzer-Programm, das auf der einen Seite mittels PMTP[2] mit dem System PEP und auf der anderen Seite mit dem jeweiligen „Dienst“ kommuniziert. Damit muß das System PEP ausschließlich das Protokoll PMTP verarbeiten können. Für das System PEP sind zwei „Übersetzer“ zu erstellen. Ein „WWW-Übersetzer“, der die Kommunikation mit dem jeweiligen HTTP-Server ermöglicht, und ein „Mail-Übersetzer“, der für die Kommunikation mit dem verwendeten Mail-Server zuständig ist.

Die Klasse *Eingabe_Server*

Die neu hinzugekommene Klasse *Eingabe_Server* ist das Ergebnis der im Systementwurf getroffenen Entscheidung, Instanzen der Klassen *Betreiber*, *Werbung* und Gast-*Profil* im Front-End vorzuhalten (vgl. Kapitel 5.3.1, S. 60). Da diese Klassen für die Generierung von nahezu allen Dokumenten benötigt werden, können durch das Vorhalten von Kopien im *Eingabe_Server* die Datenbankzugriffe erheblich minimiert werden. Eine detaillierte Beschreibung der Klasse *Eingabe_Server* erfolgt am Ende dieses Kapitels (S. 85 und Abbildung 34).

Die Klassen *Abrechnungsanbieter* und *Abrechnung_Gateway*

Die neu hinzugekommenen Klassen *Abrechnungsanbieter* und *Abrechnung_Gateway* erweitern das Modell für die bereits in der Analyse eingeführte Klasse *Zahlung*. Diese drei Klassen und deren Assoziationen haben, wie bereits in der Analyse, rein logischen Charakter. Auf eine weitere Verfeinerung wurde verzichtet, da zum Zeitpunkt der Modellierung noch keine Entscheidung über die Zusammenarbeit mit einem speziellen Abrechnungsanbieter[3] gefallen war. Vorläufig ist in diesem Zusammenhang deshalb nur wichtig, daß ein externer Abrechnungsanbieter für PEP ein Abrechnungs-Gateway zur Verfügung stellt, und daß PEP über eine Klasse *Zahlung* verfügt, die mit dem Abrechnungs-Gateway kommuniziert. Wichtig ist ebenfalls, daß PEP für den eigentlichen Zahlungsvorgang nicht zuständig ist. PEP bzw. die Klasse *Zahlung* initialisiert lediglich

---

[1] HTTP-Server, Mail-Server

[2] Die Befehle und Struktur von PMTP finden sich in Kapitel 9.8 dieser Arbeit.

[3] z.B. TeleCash, Cyber Cash, Ecash

den Zahlungsvorgang eines Benutzers beim Abrechnungsanbieter. Alle für die Abrechnung benötigten Daten werden anschließend ausschließlich zwischen dem Benutzer und dem Abrechnungsanbieter ausgetauscht. Bestätigt der Benutzer beim Abrechnungsanbieter seine Zahlungsabsicht und erfolgt eine positive Bonitätsprüfung, so erhält die Klasse *Zahlung* vom Abrechnungsanbieter die Meldung, daß der Zahlungsvorgang ausgeführt werden kann. Die Klasse *Zahlung* speichert daraufhin den Abonnement-Zeitraum des Benutzers in der Datenbank und bestätigt den Zahlungsauftrag an den Abrechnungsanbieter. Der Abrechnungsanbieter nimmt dann die Zahlung vor, bestätigt dies dem Benutzer und initiiert ein „Redirect" des Benutzers zu PEP. Der Vorteil dieser Methode ist, daß PEP keine personenbezogenen Daten des Abonnenten speichern, verwalten, übermitteln oder auch nur erfahren muß.[1]

Die Klasse *Sicherheit*
Die neu hinzugefügte Klasse *Sicherheit* stellt geeignete Ver- und Entschlüsselungs-methoden für die personenbezogenen Daten in der Klasse *Profil* zur Verfügung. Da PEP bis auf die E-Mail-Adresse keine personenbezogenen Daten speichert, beschränkt sich die Verwendung der Klasse *Sicherheit* lediglich auf die im Profil abgelegte E-Mail-Adresse und das Paßwort.

Die Container- und Aktions-Klassen
Speziell die Festlegung auf die Verwendung des POET ODBMS führte zu grundlegenden Veränderungen bezüglich der für die Dokumenten-Generierung benötigten Daten-Klassen (*Beitrag*, *Profil*, *Werbung*, *Betreiber*, *Mantel*). Um zu vermeiden, daß für den Zugriff auf die aufgeführten Klassen „POET API" verwendet werden muß, wurde für jede Daten-Klasse eine dedizierte Container-Klasse eingeführt (*Beitrag_Container*, *Profil_Container*, *Werbe_Container*, *Betreiber_Container*, *Mantel_Container*). Diese Container-Klassen dienen der Kapselung der POET API und der Zugriffsmechanismen auf die einzelnen Attribute der Daten-Klassen. Das heißt, Zugriffe auf die Attribute von Daten-Klassen geschehen **immer** über die entsprechende Container-Klasse. Die jeweilige Container-Klasse entscheidet zur Laufzeit, ob der Zugriff auf ein Attribut einer Daten-Klasse über Funktionalitäten des „Front-End" (*Aktion_FE*) oder des „Back-End" (*Aktion_BE*) zu erfolgen hat.[2]

Um die strukturellen Eigenschaften der verschiedenen Daten-Klassen zu berücksichtigen, wurden die Aktionen *Aktion_FE* und *Aktion_BE* weiter aufgebrochen, so daß letztendlich jeweils fünf Aktionstypen für Front- und Back-End definiert wurden. Jede dieser insgesamt zehn Aktionstypen hat eine spezifische Ziel-Daten-Klasse und ist innerhalb des Systems ausschließlich und vollständig für die „Betreuung" dieser dedizierten Daten-Klasse zuständig. Dementsprechend haben zum Beispiel die Klassen *Beitrag_Aktion_BE* und *Beitrag_Aktion_FE* als einzige Klassen Zugriff auf die Inhalte der Daten-Klasse *Beitrag*. Der Zugriff auf die Attribute der Klasse *Beitrag* geschieht trotzdem, aus Gründen der Datenkapselung, nur über Methoden der Klasse *Beitrag*.

---

[1] Ausgenommen der E-Mail-Adresse, die für den Versand der TV-Programmzeitschrift benötigt wird.

[2] Die Aufteilung der Aktionen in Front-End- und Back-End-Aktionen wurde bereits in Kapitel 5.2.1 vorgenommen, um Generierungs- von Datenbank-Funktionalitäten zu trennen.

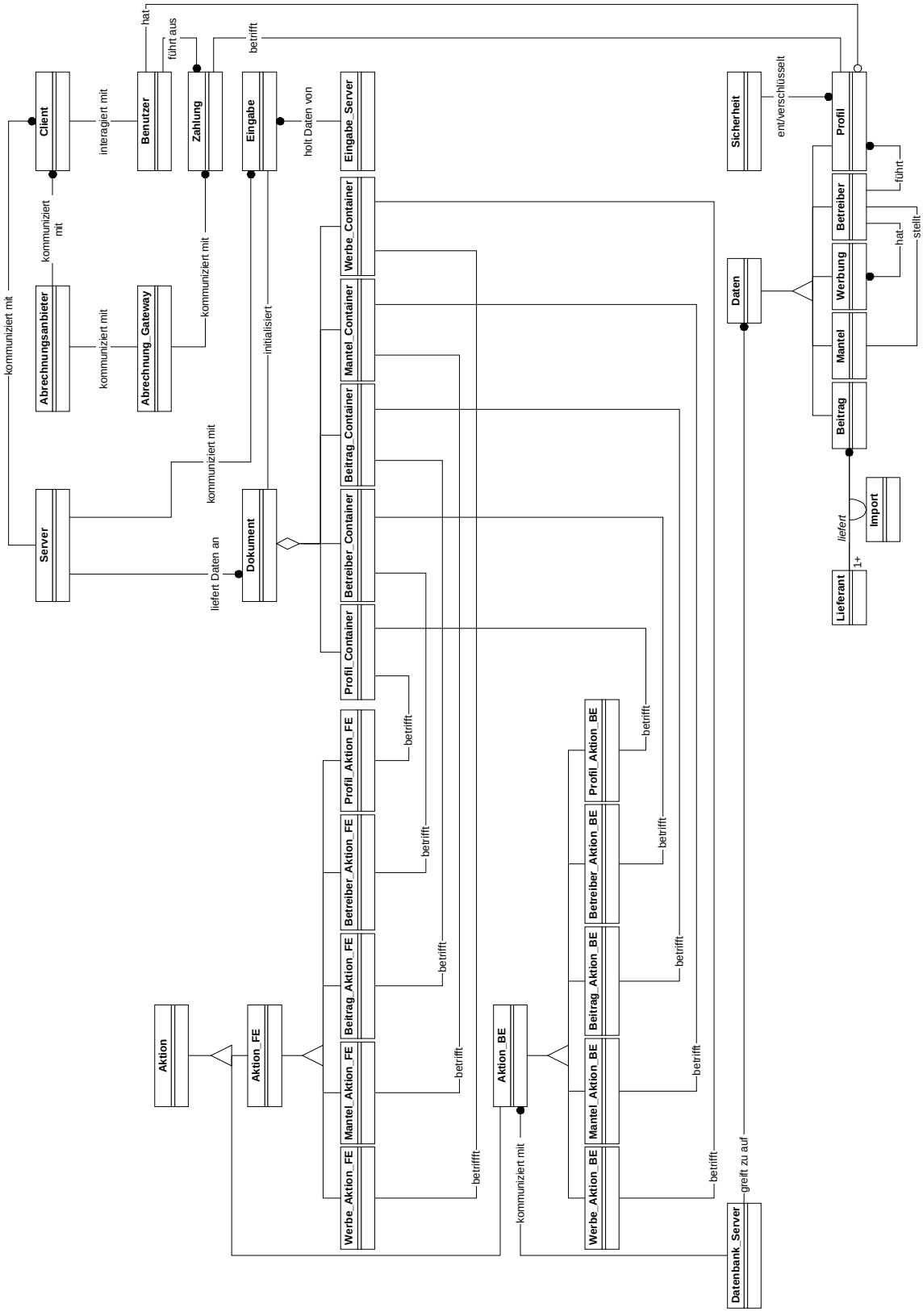


Abbildung 29: Klassendiagramm Übersicht

Im folgenden soll jetzt die Verfeinerung der Klasse *Beitrag* mit dem *Beitrag_Container* und den Aktionen *Beitrag_Aktion_FE* und *Beitrag_Aktion_BE* näher betrachtet werden.

Klassendiagramm Beitrag_Container

Das bereits in der Analyse vorgestellte Klassendiagramm Beitrag (Abbildung 13, S. 42) wurde im Objektentwurf zu dem in Abbildung 30 auf S. 77 dargestellten Klassendiagramm Beitrag_Container weiterentwickelt. Aus Gründen der Übersichtlichkeit sind in diesem Klassendiagramm zusätzlich zu der Klasse *Beitrag* und den aggregierten Klassen (*Charakter*, *Sparte*, *Sender*, *Darsteller*, *Detail_Text*, *PEPBild*) auch die beiden Aktionsklassen und der entsprechende *Beitrag_Container* dargestellt. Neu hinzugekommen sind die Methoden zu den einzelnen Klassen. Die Definition dieser Methoden erfolgte verteilt über den gesamten Modellierungsprozeß, wobei diese im wesentlichen aus der Modellierung des dynamischen Modells gewonnen wurden.

Die Klasse *Beitrag*

Für jeden vom Lieferanten gelieferten Beitrag wird von der Klasse *Import* eine neue Instanz der Klasse *Beitrag* erzeugt und in der Datenbank gespeichert. Ebenso verhält es sich mit den aggregierten Klassen *Darsteller*, *Detail_Text*, und *PEPBild*. Für die Klassen *Charakter*, *Sparte* und *Sender* werden nur dann neue Instanzen erzeugt, wenn nicht bereits eine Instanz der jeweiligen Klasse mit dem entsprechenden Namen in der Datenbank vorhanden ist. Damit wird gewährleistet, daß jeder Sender, Charakter und jede Sparte nur einmal in der Datenbank gespeichert wird. Die Attribute in der Klasse *Beitrag* und in den aggregierten Klassen (*Charakter*, *Sparte*, *Sender*, *Darsteller*, *Detail_Text*, *PEPBild*) haben sich im Entwurf, bis auf das Attribut **sortier_reihenfolge** in der Klasse *Sender*, nicht verändert. Eine Beschreibung findet sich bereits in der Analyse. Das Attribut **sortier_reihenfolge** wurde eingefügt, um eine vom Betreiber vorgegebene Reihenfolge der Sender in der Programmübersicht von PEP zu ermöglichen. Ein Fehlen dieses Attributs hätte zur Folge, daß die Sender nur in alphabetischer oder willkürlicher Reihenfolge in der TV-Programmübersicht ausgegeben werden können. Der erste Fall würde zum Beispiel dazu führen, daß der Sender ZDF nicht wie gewohnt an zweiter Stelle sondern erst am Ende der TV-Programmtabelle erscheint.

Folgende Methoden werden für den Zugriff auf die Attributwerte der Klasse *Beitrag* bereitgestellt: Die Methode ***gib_wert*** ist eine universelle Methode, um Werte von Attributen der Klasse *Beitrag* zu liefern. Sie wird mit zwei Parametern aufgerufen: Mit dem Namen des zu liefernden Attributwerts und einer Referenz auf eine Variable in der der Attributwert gespeichert werden soll. Die Methode ***setze_wert*** ist eine universelle Methode um Werte von Attributen der Klasse *Beitrag* zu verändern. Sie wird ebenfalls mit zwei Parametern aufgerufen: Mit dem Namen des zu verändernden Attributwerts und dem zu setzenden Wert. Die beiden Methoden ***gib_wert*** und ***setze_wert*** werden, damit sie auf alle Attribut-Typen der Klasse *Beitrag* angewendet werden können, überladen. Im folgenden wird erläutert, wie der Zugriff von der Klasse *Dokument* (vgl. Abbildung 29) auf die Attribute der Klasse *Beitrag* erfolgt.

Die Klassen *Beitrag_Container, Beitrag_Aktion_FE, Beitrag_Aktion_BE*

Wie bereits erwähnt, erfolgen alle Zugriffe auf die Klasse *Beitrag* ausschließlich über den *Beitrag_Container* und die *Beitrag_Aktion_FE* bzw. *Beitrag_Aktion_BE*. Generell ist der

Ablauf dabei folgendermaßen: Die Klasse *Dokument* erzeugt für das Generieren eines Dokuments zuerst eine neue Instanz der Klasse *Beitrag_Container*. Anschließend wird von *Dokument* die Methode ***fuell_dich*** aufgerufen. Als Parameter werden Instanzen der Klassen *Anfrage_Daten* und *Profil_Container* übergeben.[1] Bis zu diesem Zeitpunkt spielen sich alle Vorgänge im Front-End ab. Die Methode ***fuell_dich*** erfordert jetzt einen Wechsel in das Back-End, da nur *Beitrag_Aktion_BE* Beiträge aus der Datenbank lesen kann. *Beitrag_Container* ruft also die Methode ***fuell_mich*** in *Beitrag_Aktion_BE* mit denselben Parametern auf. *Beitrag_Aktion_BE* fordert dann vom *Datenbank_Server* (vgl. Abbildung 29) anhand der übergebenen Parameter die entsprechenden Instanzen der Klasse *Beitrag* an. Die erhaltenen Instanzen der Klasse *Beitrag* werden im Attribut **beitrag_set** der Klasse *Beitrag_Container* gespeichert. Alle lesenden und schreibenden Zugriffe von *Dokument* auf die Attribute der erhaltenen Instanzen der Klasse *Beitrag* erfolgen anschließend ausschließlich im Front-End über die Klasse *Beitrag_Aktion_FE*. Nur wenn die Klasse *Dokument* die Methode ***speicher_beitrag*** in *Beitrag_Container* aufruft, erfolgt wiederum ein Wechsel in das Back-End. Die Methode ***speicher_beitrag*** in *Beitrag_Container* und *Beitrag_Aktion_BE* wird mit nur einem Parameter aufgerufen. Dieser Parameter (Zähler) erlaubt es der Klasse *Dokument*, gezielt einen Beitrag des Beitrags-Set anzusprechen. Die Methoden ***gib_wert*** und ***setze_wert*** in *Beitrag_Container* und *Beitrag_Aktion_FE* entsprechen, bis auf den ebenfalls hinzugefügten dritten Parameter Zähler, den schon beschriebenen in *Beitrag*. Die Klasse *Dokument* erhält auch hier durch die Angabe des Parameters Zähler die Möglichkeit, gezielt einen Beitrag aus dem Beitrags-Set anzusprechen.

[1] Eine Beschreibung der Klassen *Profil_Container* erfolgt ab Seite 77, *Anfrage_Daten* ab Seite 85 dieses Abschnitts.

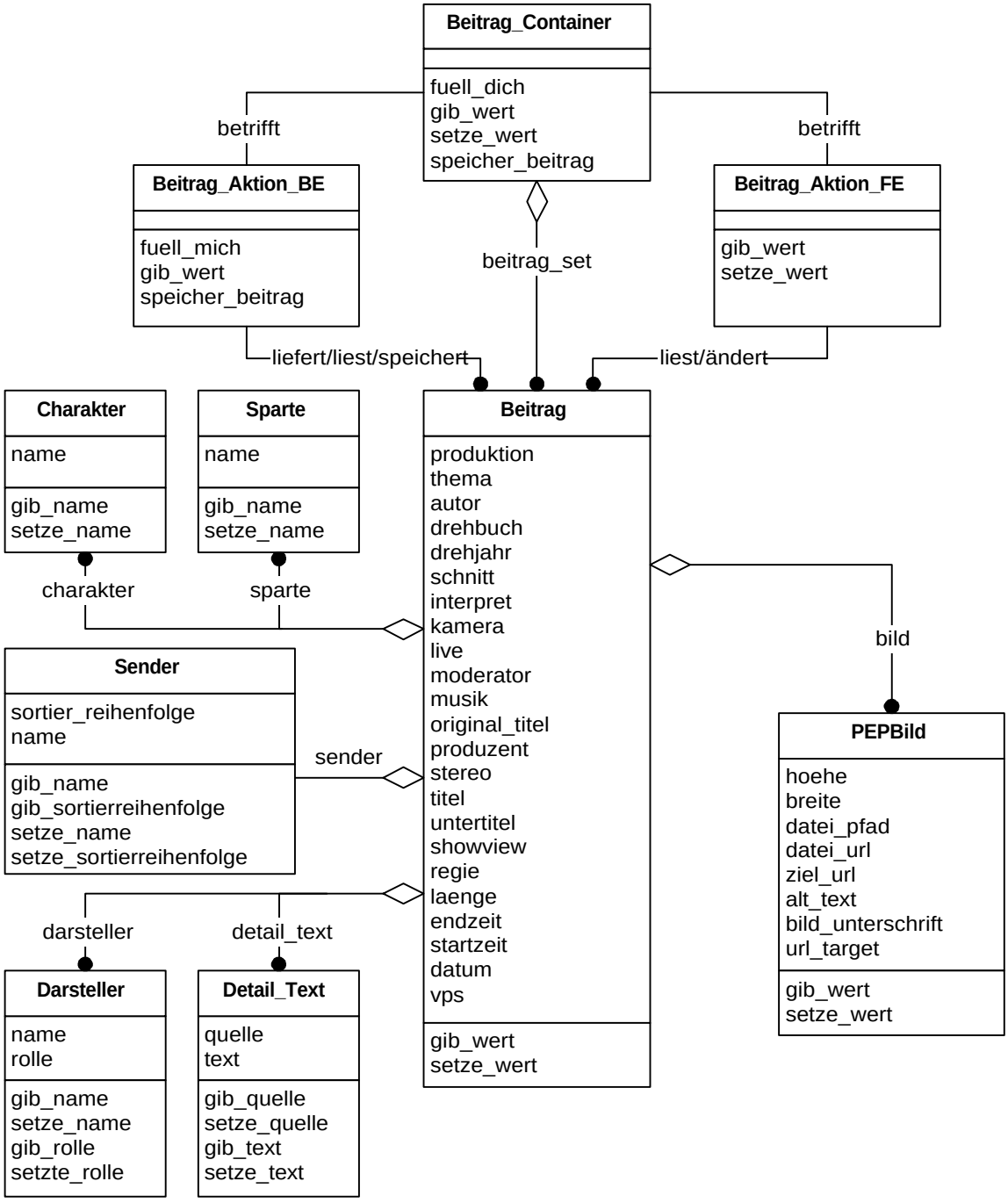


Abbildung 30: Klassendiagramm Beitrag_Container

Im folgenden wird jetzt noch exemplarisch die statische Struktur der Klasse *Profil_Container* mit dessen Aktionen und dem aggregierten Profil näher betrachtet. Die Beschreibung der drei weiteren Container-Klassen *Betreiber_Container*, *Mantel_Container* und *Werbe_Container* sowie deren Aktionen befindet sich in Band 2 dieser Diplomarbeit.

Klassendiagramm Profil_Container

Das in Abbildung 31 auf Seite 80 dargestellte Klassendiagramm Profil_Container zeigt das Entwurfsergebnis der Klasse *Profil* und dessen aggregierte Klassen *PEPZeit*, *PEPText*,

Sparte, *Charakter*, *Sender* und *Fernbedienung*. Der Zugriff auf eine Instanz der Klasse *Profil* erfolgt, ebenso wie in der bereits beschrieben Klasse *Beitrag*, über eine Container-Klasse (*Profil_Container*) und die beiden Aktions-Klassen (*Profil_Aktion_BE*, *Profil_Aktion_FE*).

Die Klasse *Profil*

Für jeden Abonnenten von PEP wird bei der Neu-Anmeldung eine Instanz der Klasse *Profil* erzeugt und in der Datenbank gespeichert. Zusätzlich zu diesen „Abonnenten-Profilen" kann der Betreiber ein „Gast-Profil" in PEP einrichten, das den Inhalt bzw. Aufbau des öffentlich zugänglichen TV-Programms beeinflußt. Die Attribute von *Profil* haben folgende Bedeutung: Das Attribut **email_empfang** dient als Vermerk, ob die E-Mail wöchentlich oder täglich versendet werden soll. In den Attributen **vps** und **showview** wird vermerkt, ob die VPS-Zeit und ShowView-Nummer in den generierten Programminformations-Dokumenten angezeigt werden soll oder nicht. Die aktuelle Session-ID des Abonnenten wird im Attribut **session_id** gespeichert.[1] Die Attribute **benutzer_name**, **passwort** und **email_adresse** enthalten den beliebig gewählten Benutzernamen des Abonnenten, sein Paßwort und die E-Mail-Adresse.[2] In dem Attribut **letzter_besuch** wird das Datum und die Uhrzeit des letzten Zugriffs des Abonnenten auf PEP vermerkt.[3] Das Attribut **bundesland** dient dazu, spezielle, lokal ausgestrahlte Beitragsinformationen liefern zu können (z.B. Schleswig-Holstein Magazin). Die Attribute der aggregierten Klassen *Sparte*, *Charakter* und *Sender* sind bereits im vorherigen Klassendiagramm Beitrag_Container (Abbildung 30) beschrieben worden. In den Attributen der aggregierten Klassen *PEPZeit* und *PEPText* werden die vom Abonnenten bevorzugten Fernsehzeiten und ergänzende Schlüsselworte für die Beitragsauswahl gespeichert. Eine Besonderheit stellt die aggregierte Klasse *Fernbedienung* dar. In ihr wird die vom Abonnenten angegebene Tastennummer der Fernbedienung seines TV-Gerätes zu jedem Sender eingetragen. Diese Nummer wird in den von PEP generierten Dokumenten zu jedem aufgelisteten Sender mit angegeben. Weiterhin wird eine Sortierung der in der Programmübersicht aufgelisteten Sender nach den Tastennummern vorgenommen. Die Methoden ***gib_wert*** und ***setze_wert*** in der Klasse *Profil* entsprechen in den übergebenen Parametern und der Funktionsweise exakt den bereits beschriebenen in der Klasse *Beitrag*.

Die Klassen Profil_Container, Profil_Aktion_FE, Profil_Aktion_BE

Der Zugriff auf die Klasse *Profil* ist, wie für die Klasse *Beitrag*, in Back-End- und Front-End-Aktionen aufgeteilt. Der Ablauf ist dabei folgendermaßen: Die Klasse *Dokument* erzeugt zuerst eine neue Instanz der Klasse *Profil_Container*. Anschließend ruft Dokument die Methode ***fuell_dich*** in *Profil_Container* mit Instanzen der Klassen *Anfrage_Daten* und *Betreiber_Container* als Parameter auf.[4] *Profil_Container* ruft dann

---

[1] vgl. dazu Kapitel 5.3.1 und Kapitel 5.4.2.

[2] Die Attribute **passwort** und **email_adresse** werden von der Klasse *Sicherheit* verschlüsselt in *Profil* abgelegt (vgl. dazu Kapitel 5.3.3).

[3] vgl. dazu Kapitel 5.3.3.

[4] Da Profile jeweils einem bestimmten Betreiber zugeordnet sind (siehe Abbildung 29), können Profile nur im Kontext eines bestimmten Betreibers (*Betreiber_Container*) ermittelt werden.

die Methode ***fuell_mich*** in *Profil_Aktion_BE* mit denselben Parametern auf. *Profil_Aktion_BE* fordert nun von *Datenbank_Server* (vgl. Abbildung 29) anhand der übergebenen Parameter die entsprechende Instanz der Klasse *Profil* an. Wenn die jeweils benötigten Überprüfungen der Zugriffsberechtigungen (***vergleiche_ids***, ***vergleiche_zeiten***, ***vergleiche_pw***) zwischen *Anfrage_Daten* und *Profil* ein positives Ergebnis liefern, wird die erhaltene Instanz der Klasse *Profil* im Attribut **profil_set** der Klasse *Profil_Container* gespeichert.

Die Klasse *Profil_Aktion_BE* enthält noch zwei weitere Methoden. Die Methode ***wechsel_in_abo_session_id*** erzeugt für einen Abonnenten, der sich beim System PEP anmeldet, aus der Gast-Session-ID eine Abonnenten-Session-ID und speichert diese im *Profil*. Eine Abonnenten-Session-ID enthält unter anderem die Objekt-ID (OID) des Profil-Objekts (vgl. Kapitel 5.3.3). Für jeden weiteren Zugriff auf das Profil eines bereits angemeldeten Abonnenten wird diese OID aus der in den Anfrage-Daten übermittelten Abonnenten-Session-ID über die Methode ***extrahiere_oid*** ermittelt.

Alle lesenden und schreibenden Zugriffe von *Dokument* auf die erhaltene Instanz der Klasse *Profil* erfolgen, wie beim *Beitrag*, ausschließlich im Front-End über die Klasse *Profil_Aktion_FE*. Die Methoden ***gib_wert***, ***setze_wert*** und ***speicher_profil*** in *Profil_Container* und *Profil_Aktion_FE* entsprechen den schon beschriebenen in *Beitrag_Container und Beitrag_Aktion_FE*. Neu sind damit nur die Methoden ***gib_profil_kopie*** und ***aender_profil***. Die Methode ***gib_profil_kopie*** liefert eine Kopie einer an *Profil_Container* angehängten Instanz der Klasse *Profil*. Als Parameter wird beim Aufruf lediglich den *Betreiber_Container* übergeben. Weitere Parameter sind für diese Methode nicht notwendig, da sie nur für das Kopieren des Gast-Profils eines Betreibers zuständig ist. Die Methode ***aender_profil*** erhält als Parameter die *Anfrage_Daten* übergeben. Diese Methode entspricht im Prinzip der Methode ***setze_wert***, jedoch verarbeitet sie nicht nur einen zu ändernden Wert pro Aufruf, sondern alle vom Abonnenten übertragenen Werte einer geänderten Profil-Einstellungsseite. Diese Methode wurde notwendig, um zum Beispiel eine Änderung eines Abonnenten-Paßworts zu verifizieren. In *Anfrage_Daten* wird in diesem Fall unter anderem das alte Paßwort und zweimal das neue Paßwort übertragen. Die Methode ***aender_profil*** prüft in diesem Fall zuerst die Korrektheit des alten Paßworts, vergleicht dann die beiden neuen Paßworte auf Übereinstimmung und trägt bei positivem Ergebnis das neue Paßwort in die Instanz der Klasse *Profil* ein. Anschließend wird die geänderte Instanz der Klasse *Profil* über *Profil_Aktion_BE* und ***speicher_profil*** in die Datenbank zurückgeschrieben.

Werden die statischen Strukturen der Klassendiagramme Profil_Container und Beitrag_Container verglichen, so ist festzustellen, daß sie sich konzeptionell kaum unterscheiden. Ebenso verhält es sich mit den hier nicht beschriebenen Klassendiagrammen Betreiber_Container, Werbe_Container und Mantel_Container. Aus diesem Grund wird auf eine weitere Beschreibung verzichtet. In den folgenden zwei Klassendiagrammen wird die Verfeinerung der Klasse *Dokument*, die die Container-Klassen für die Dokumenten-Generierung verwendet, näher betrachtet.

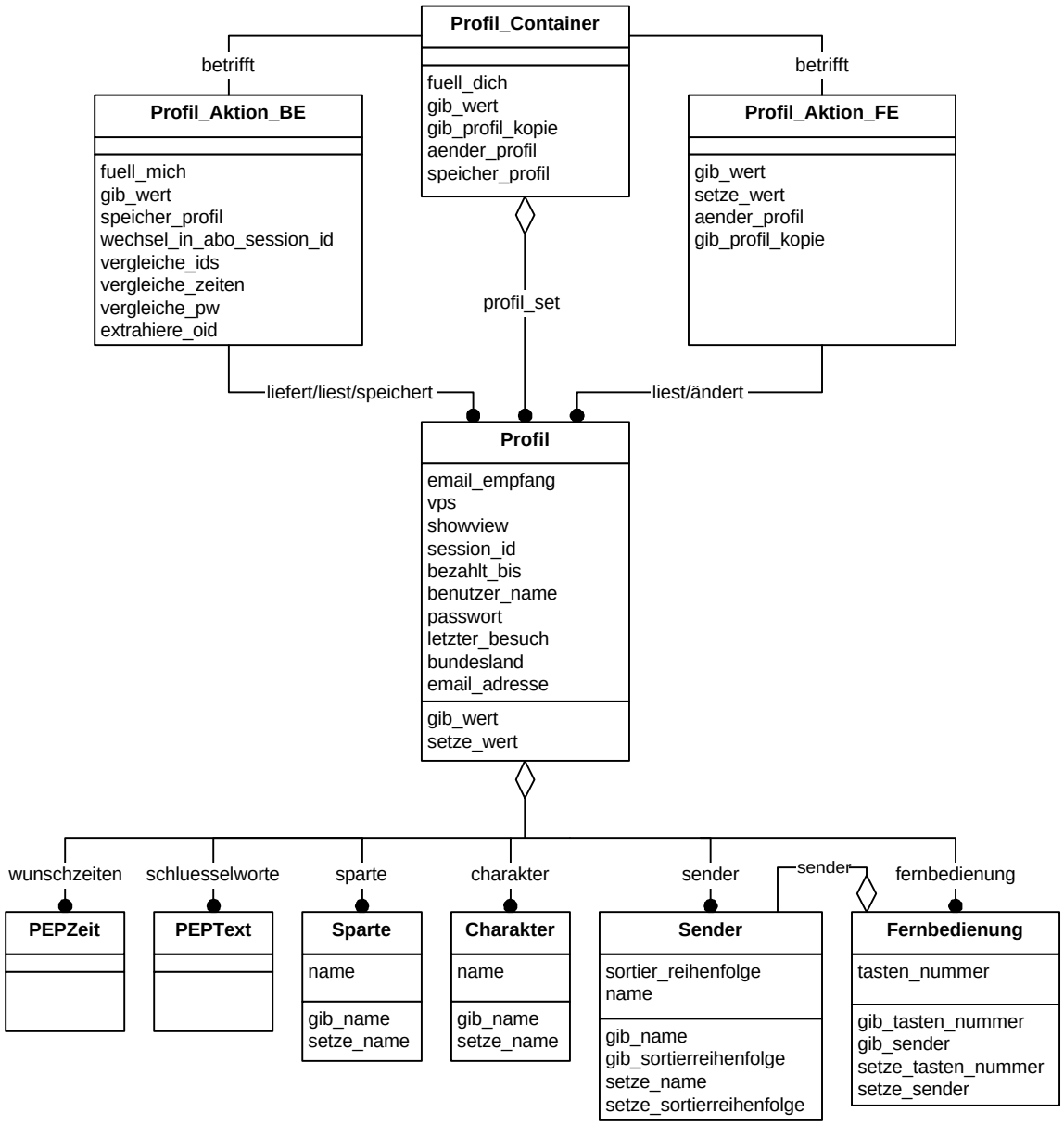


Abbildung 31: Klassendiagramm Profil_Container

Klassendiagramm Dokument

Das in Abbildung 32 auf Seite 82 dargestellte Klassendiagramm Dokument erweitert die statische Struktur der Klasse *Dokument* um die zwei Unterklassen *HTML_Kneter* und *PDF_Kneter*. Jede dieser Unter- bzw. abgeleiteten Klassen ist nur für die Generierung eines bestimmten Dokumenten-Typs zuständig (hier HTML bzw. PDF). Die in der Ober- bzw. Basisklasse *Dokument* aggregierten Attribute **profil_container**, **betreiber_container**, **mantel__container**, **beitrag_container** und **werbe_container** werden an die Unterklassen *HMTL_Kneter* und *PDF_Kneter* vererbt, so daß der jeweilige Kneter einen direkten Zugriff auf die benötigten Container-Klassen erhält. Ebenso verhält es sich mit der einzigen Methode der Klasse *Dokument* (***init_redirect***), die für ein „Redirect“ benötigt wird (z.B. für die Initialisierung eines Zahlungsvorgangs beim Abrechnungsanbieter). Die Klasse *Dokument* ist, wie deutlich wurde, lediglich eine

abstrakte Klasse. Wird zum Beispiel auf höheren Abstraktionsebenen dieses PEP OMT-Modells beschrieben, daß die Klasse *Dokument* ein Dokument generiert, so ist real immer eine der beiden Kneter-Klassen als eigentlich generierende Klasse zu sehen. In den verfeinerten Diagrammen der Modellierung wird in der Regel direkt die Klasse *HTML_Kneter* bzw. *PDF_Kneter* verwendet.

Die Klasse *HTML_Kneter*
Die Methoden der Klasse *HTML_Kneter* werden in der Regel, außer der internen Methode ***pruefe_uebersicht_vorhanden***, von der Klasse *Eingabe* aufgerufen (vgl. Abbildung 29). Die Methode ***init_html_beitraege*** dient zur Generierung jeglicher HTML-Seiten, die Beitrags-Informationen enthalten werden. Sie wird sowohl für Gast- als auch für Abonnenten-Dokumente verwendet. Als Parameter werden Instanzen der Klassen *Betreiber_Container*, *Werbe_Container* und falls benötigt *Profil_Container* mit den enthaltenen Kopien der Klassen *Betreiber*, *Werbung* und Gast-*Profil* übergeben.[1] Außerdem werden noch die *Anfrage_Daten* und der Typ des zu generierenden Dokuments übergeben (der Typ ist z.B. die Übersicht-Seite). Die Methode ***init_html_profil*** dient zur Generierung der Profil-Einstellungsseiten für Abonnenten. Als Parameter werden Instanzen der Klassen *Betreiber_Container* und *Werbe_Container* mit den enthaltenen Kopien der Klassen *Betreiber* und *Werbung* sowie die *Anfrage_Daten* und der Typ des zu generierenden Dokuments übergeben. Die Methode ***init_html_fehler*** dient der Generierung von HTML-Seiten, die Fehlermeldungen des Systems an den Benutzer weiterleiten. Als Parameter wird die Art des Fehlers übergeben. Die interne Methode ***pruefe_uebersicht_vorhanden*** wird ausschließlich für die zu generierenden Gast TV-Programm-Dokumente verwendet. Als Parameter werden die *Anfrage_Daten* und der Typ des zu generierenden Dokuments übergeben. Die Idee ist, daß zur Optimierung des Systems bereits generierte Gast HTML-Dokumente innerhalb einer Verzeichnisstruktur[2] gespeichert werden (z.B. Alle Dokumente für die Gast TV-Programmübersicht). Bei einer erneuten Anfrage auf eines dieser Dokumente wird das entsprechende von ***pruefe_uebersicht_vorhanden*** lokalisiert, anschließend wird es geöffnet und an den Server gesandt. Wird das gewünschte Dokument nicht aufgefunden, so findet der übliche Generierungsprozeß statt.

Die Klasse *PDF_Kneter*
Die Methode ***init_pdf_beitraege*** der Klasse *PDF_Kneter* wird ebenfalls von der Klasse *Eingabe* aufgerufen. Als Parameter werden Instanzen der Klassen *Betreiber_Container* und *Werbe_Container* mit den enthaltenen Kopien der Klassen *Betreiber* und *Werbung* sowie die *Anfrage_Daten* übergeben.

Als nächster Verfeinerungsschritt werden nun die vom *HTML_Kneter* für die Dokumenten-Generierung verwendeten Klassen betrachtet.

---

[1] Wie bereits erwähnt, werden aus Optimierungsgründen Instanzen der Klassen *Betreiber*, *Werbung* und Gast-*Profil* im *Eingabe_Server* vorgehalten (vgl. Abbildung 29 und Text).Eine Instanz der Klasse Gast-*Profil* wird nur dann als Parameter übergeben, wenn es sich um einen Gast-Zugriff handelt (erkennbar an der Session-ID).

[2] Zur Verzeichnisstruktur siehe Kapitel 5.3.2, Tabelle 7.

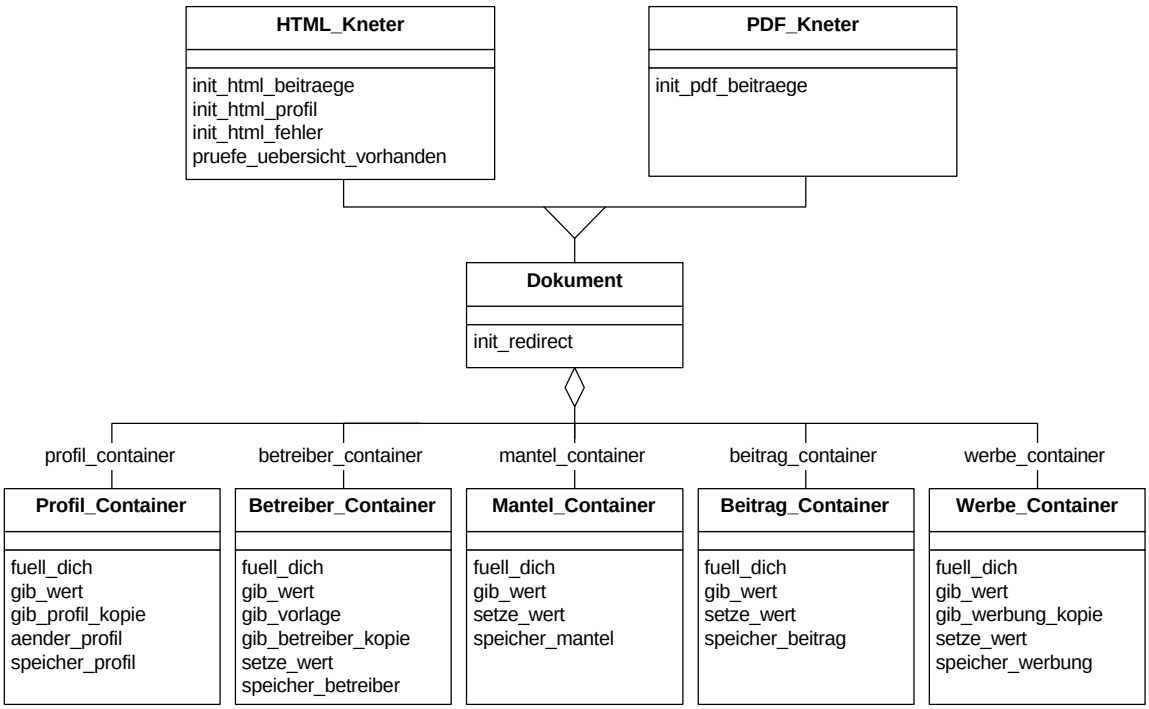


Abbildung 32: Klassendiagramm Dokument

Klassendiagramm *HTML_Kneter*

Die Klasse *HTML_Kneter* verwendet für die Generierung von HTML-Dokumenten verschiedene Dokument-Vorlagen. Jede Instanz der Klasse *Vorlage* repräsentiert die Dokument-Vorlage für einen zu generierenden Dokumenten-Typ (Attribut **typ** der Klasse *Vorlage*). Innerhalb der Klasse *Vorlage* wird die Struktur des Dokuments abgebildet. Abbildung 33 auf Seite 85 zeigt die für die Abbildung dieser Dokument-Struktur verwendeten Klassen *Sektion*, *SektionGlied* und *Constraint* sowie die Klasse *Platzhalter*. Die Instanzen der Klasse *Vorlage* und der aggregierten Klassen sind eindeutig einem Betreiber zugeordnet. Das heißt, die Klasse *Betreiber* verfügt über ein Attribut (**vorlage_set**), das die einzelnen *Vorlagen*-Objekte enthält. Über die Methode ***gib_vorlage*** der Klasse *Betreiber_Container* wird der Zugriff auf die einzelnen *Vorlagen*-Objekte realisiert.[1] Wird zum Beispiel die Methode ***init_html_beiträge*** aufgerufen, fordert *HTML_Kneter* vom *Betreiber_Container* das dem Dokument-Typ entsprechende *Vorlage*-Objekt an.

Die Klasse Vorlage

Wie bereits erwähnt, repräsentiert jede Instanz der Klasse Vorlage einen Dokument-Typ. Jede Vorlage ist in mehrere Teile (Sektionen) aufgegliedert. Jede Sektion repräsentiert einen Abschnitt eines Dokuments (zum Beispiel „Kopf“, „Beitragsinformationen“ und „Fuß“). Über die Methode ***naechste_sektion*** wählt *HTML_Kneter* die einzelnen *Sektionen* aus. Der Generierungsvorgang beginnt mit der ersten *Sektion* und endet wenn alle *Sektionen* einer *Vorlage* abgearbeitet sind. Eine *Sektion* besteht wiederum aus

[1] Das entsprechende Klassendiagramm ist Bestandteil des Ergänzungsbands (Klassendiagramm Betreiber_Container).

SektionGliedern, *Constraints* und *Platzhaltern*. Im folgenden werden diese vier Klassen beschrieben.

Die Klassen *Sektion*, *Platzhalter*, *Constraint* und *SektionGlied*
Die einzelnen *Sektionen* einer *Vorlage* sind über das Attribut **naechste** aneinander gereiht. So entsteht eine verkettete Liste von *Sektionen*. Über die Methode ***naechste_sektion*** kann innerhalb der verketteten Liste die nächste *Sektion* ausgewählt werden. Jede einzelne *Sektion* besteht aus *Platzhaltern*, *SektionGliedern* und *Constraints*.

In der aggregierten Klasse *Platzhalter* werden Standard-Werte gespeichert, die innerhalb der gesamten Sektion verwendet werden können. Standard-Werte sind zum Beispiel die Hintergrundfarben der Tabellenzellen in der TV-Programmtabelle (vgl. Abbildung 2, Seite 10). Mögliche Werte der Attribute von *Platzhalter* sind beispielsweise „**name**=farbe_nachrichten“, „**wert**=DDFFDD“.

Die aggregierte Klasse *SektionGlied* bildet den eigentlichen Inhalt der *Sektion* ab. Das Attribut **typ** der Klasse *SektionGlied* kann den Wert „platzhalter“ oder „text“ annehmen. Der Wert „platzhalter“ gibt an, daß der Wert des Attributs **inhalt** einen Wert aus der Datenbank referenziert. Ein möglicher Wert des Attributs **inhalt** ist „Beitrag.startzeit“. Hat das Attribut **typ** der Klasse *SektionGlied* den Wert „text“, so beinhaltet das Attribut **inhalt** den zur Generierung zu verwendenden Wert. Ein möglicher Wert des Attributes **inhalt** ist dann zum Beispiel „<html>\n<body bgcolor=#FFFFFF>“. Durch eine Aneinanderreihung einzelner *SektionGlieder* (Attribut **naechstes**) innerhalb der *Sektion* ist somit die Abbildung einer beliebigen Reihenfolge von HTML-Code und aktuellen Werten aus der Datenbank möglich. *HTML_Kneter* arbeitet die einzelnen *SektionGlieder* einer *Sektion* der Reihe nach ab. Dabei sendet *HTML_Kneter* den Inhalt des *SektionGliedes* direkt an den Server, wenn es sich um ein *SektionGlied* vom Typ „text“ handelt. Handelt es sich um ein SektionGlied vom Typ „platzhalter“, so wird der Wert des Attributs **inhalt** von *HTML_Kneter* ausgelesen. Anschließend ermittelt *HTML_Kneter* anhand des ausgelesenen Werts, an welche Container-Klasse die Wertanforderung zu erfolgen hat. Für das Beispiel „Beitrag.startzeit“, würde damit der Methoden-Aufruf ***gib_wert*** an die Klasse *Beitrag_Container* mit dem Parameter „startzeit“ erfolgen.

Die drei Klassen *Sektion*, *Platzhalter* und *SektionGlied* reichen aus, um einfach strukturierte HTML-Dokumente zu generieren. Um komplexere Dokumente generieren zu können, die zum Beispiel Wiederholungen von Elementen des selben Typs beinhalten, wurde die Klasse *Constraint* eingeführt. Beinhaltet ein Beitrag beispielsweise mehr als einen Darsteller, so läßt sich über die Klasse *Constraint* steuern, daß das *SektionGlied* mit dem Inhalt „Beitrag.darsteller.name“ solange für die Generierung verwendet wird, bis alle Darsteller-Namen verarbeitet wurden.

Die von *Sektion* aggregierte Klasse *Constraint* repräsentiert einen einfachen Kontrollfluß-Mechanismus für Teile von *Sektionen*. Ein *Constraint* wird dabei einem *SektionGlied* „vorgeschaltet“ (Attribut **kette** der Klasse *Constraint*). Dieses *SektionGlied* kann als Nachfolger ein weiteres *SektionGlied* (Attribut **naechstes**) oder ein *Constraint* (Attribut **constraint**) oder beides haben. Auf diese Weise entsteht eine Baumstruktur, wobei die Klasse *Constraint* als Nachfolger nur ein *SektionGlied* haben kann. Ein *Constraint* bezieht

sich dabei immer auf den gesamten „nachgeschalteten“ Teilbaum von *SektionGliedern* und *Constraints*. Das Attribut **typ** der Klasse Constraint kann die Werte „if“, „while more“ oder „while equal“ annehmen. Der Wert „if“ gibt an, daß der nachfolgende Teilbaum nur dann für die Generierung verwendet werden soll, wenn die im Attribut **inhalt** enthaltene Bedingung erfüllt ist. Zum Beispiel ist ein möglicher Wert des Attributs **inhalt** dieses *Constraints* „Profil.vps“. Dieses *Constraint* ist einem *SektionGlied* vorgeschaltet, das vom Typ „platzhalter“ ist und den Inhalt „Beitrag.vps“ hat. An dieser Kombination erkennt *HTML_Kneter*, daß die VPS-Zeit des Beitrags nur dann in der Generierung zu berücksichtigen ist, wenn im Profil des Abonnenten das Flag „VPS-Zeit anzeigen“ gesetzt ist. Ist das *Constraint* vom Typ „while more“ und vom Inhalt „Beitrag.darsteller.name“, wird der nachgeschaltete Teilbaum solange in der Generierung berücksichtigt, bis alle Darsteller des Beitrags verarbeitet worden sind. Ein *Constraint* vom Typ „while equal“ und vom Inhalt „Beitrag.sender.name“ veranlaßt den *HTML_Kneter*, den nachgeschalteten Teilbaum solange in der Generierung zu berücksichtigen, bis alle Beiträge eines Senders verarbeitet worden sind.

Für die Abarbeitung der Baumstruktur aus *SektionGliedern* und *Constraints* stellt die Klasse *Sektion* die Methode ***gib_naechstes*** zur Verfügung. Die Methode ***gib_naechstes*** ist für den Zugriff auf die Klassen *SektionGlied* und *Constraint* zuständig. Sie liefert den Wert des Attributs **typ** und den Wert des Attributs **inhalt** des nächsten *SektionGliedes* bzw. *Constraints* zurück. Die Methode ***gib_wert*** der Klasse *Sektion* ist nur für den Zugriff auf die Klasse *Platzhalter* notwendig. Mittels dieser Methode kann der Wert des Attributs **wert** der aggregierten Klasse *Platzhalter* ermittelt werden. Dazu wird der Name des gewünschten *Platzhalters* als Parameter übergeben. Dementsprechend dient die Methode ***setze_wert*** dem Verändern des Attributs **wert** der Klasse Platzhalter, gesteuert über den als Parameter übergebenen Namen und den zu setzenden Wert.

In diesem Kapitel wurde beschrieben, wie die Strukturen der in der Datenbank abgelegten *Beitrag*- und *Profil*-Objekte beschaffen sind. Ferner wurden mit den Container-Klassen die Strukturen beschrieben, die nötig sind, um auf Objekte, die in der Datenbank gespeichert sind, zugreifen zu können. Im Anschluß wurde erläutert, welche Klassen, Attribute und Methoden von *Dokument* bzw. *HTML_Kneter* verwendet werden, um ein Dokument zu generieren. Der folgende Abschnitt erörtert nun, wie eine von *Server* abgesetzte Anfrage vom System empfangen und wie der daraus resultierende Vorgang der Dokumenten-Generierung angestoßen wird.

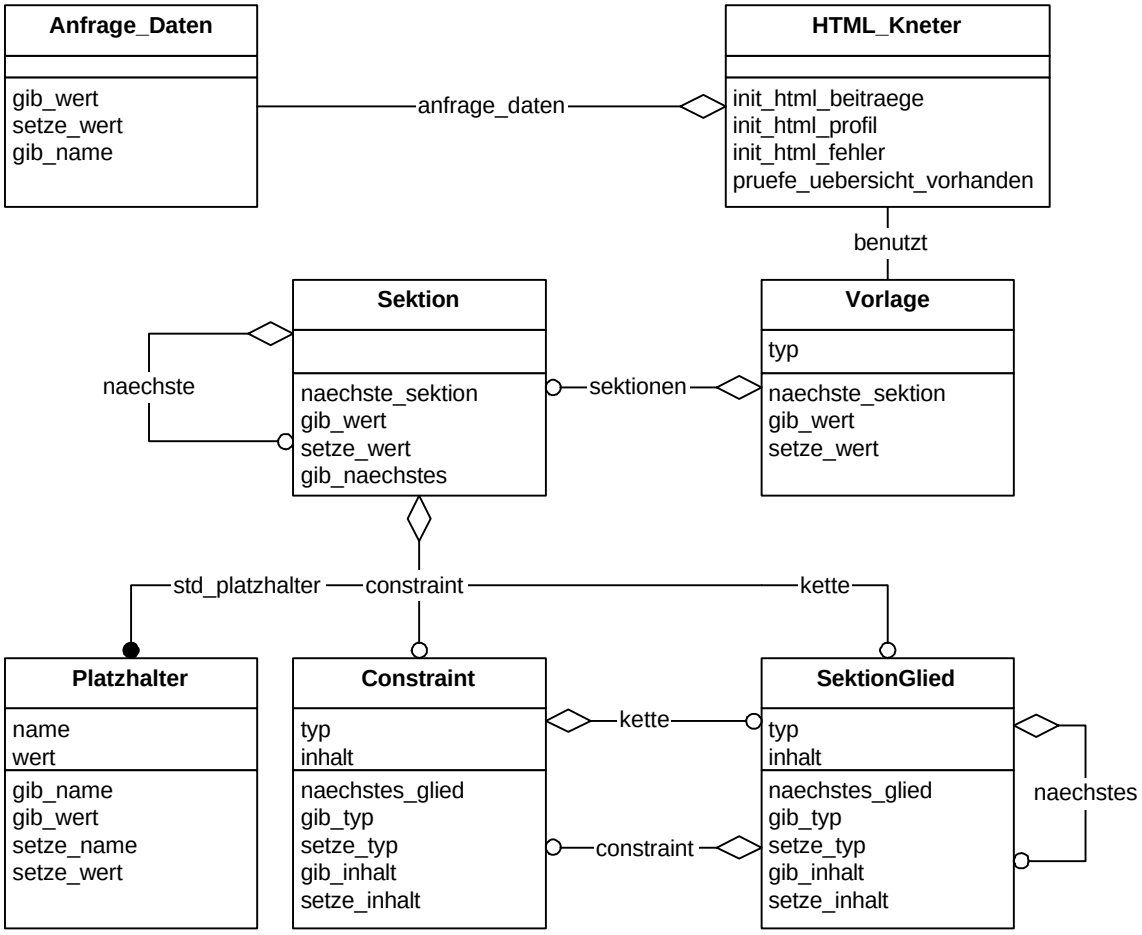


Abbildung 33: Klassendiagramm HTML_Kneter

Klassendiagramm Eingabe

In dem auf Seite 87 abgebildeten Klassendiagramm Eingabe werden nun die statischen Strukturen der Klassen *Eingabe*, *Eingabe_Server* und *Anfrage_Daten* näher erörtert. Die Klasse *Eingabe* wertet die von der Klasse *Server* mittels PMTP übertragenen Informationen aus und speichert sie in strukturierter Form in einer Instanz der Klasse *Anfrage_Daten*. Anhand dieser *Anfrage_Daten* ermittelt Eingabe im Anschluß daran, ob Instanzen der Klassen *Betreiber*, *Werbung* oder Gast-*Profil* für die Dokumenten-Generierung benötigt werden und fordert diese gegebenenfalls von *Eingabe_Server* an.[1] Am Ende ruft *Eingabe* je nach Anforderung die Klasse *HTML_Kneter* oder *PDF_Kneter* zur Dokumenten-Generierung mit den entsprechenden Parametern auf (vgl. Abbildung 32: Klassendiagramm Dokument).

Die Klasse *Eingabe*

Die Methode ***parse_request*** überführt den von *Server* übermittelten Anfrage-String in eine strukturierte Form, indem die im Anfrage-String enthaltenen Informationen als Name/Wert-Paare in der aggregierten Klasse *Anfrage_Daten* gespeichert werden. In

[1] Die zur Minimierung von Datenbankzugriffen eingeführte Klasse *Eingabe_Server* wurde bereits zu Beginn dieses Kapitels kurz beschrieben (vgl. S. 72 und Abbildung 29).

diesen *Anfrage_Daten* sind anschließend unter anderem die Werte für den Dokument-Typ und den Betreiber-Identifizierer enthalten. Der Dokument-Typ gibt Auskunft darüber, welches Dokument zu generieren ist (z.B. Gast-TV-Programmübersicht). Der Betreiber-Identifizierer gibt Auskunft darüber, von welchem Web-Auftritt die Anfrage zur Generierung einer Seite erfolgt ist (z.B. HÖRZU oder *TVneu*).[1] Mit dem Betreiber-Identifizierer fordert *Eingabe* eine Instanz der Klasse *Betreiber_Container* von *Eingabe_Server* an. Der *Betreiber_Container* beinhaltet eine Kopie des zu dem Betreiber-Identifizierer passenden *Betreiber*-Objekts. Dieses wiederum beinhaltet eine Instanz der Klasse *Vorlage*, die dem in *Anfrage_Daten* enthaltenen Dokument-Typ entspricht. Anhand der Vorlage wird nun über die Methode ***pruefe_profil*** ermittelt, ob zur Generierung des Dokuments das Gast-Profil benötigt wird. Ist dies der Fall, fordert *Eingabe* eine Instanz der Klasse *Profil_Container* mit einer Kopie des entsprechenden Gast-Profils von *Eingabe_Server* an. Wird zur Generierung Werbung benötigt (Methode ***pruefe_werbung***), fordert Eingabe zudem den *Werbe_Container* bei *Eingabe_Server* an.

Die Methode ***pruefe_session_id*** prüft die Gültigkeit bzw. das Vorhandensein der in den *Anfrage_Daten* abgelegten Session-ID (vgl. Kapitel 5.3.3). Ist die in *Anfrage_Daten* gespeicherte Session-ID abgelaufen oder keine Session-ID darin enthalten, so wird von der Methode ***generiere_session_id*** eine neue Gast Session-ID generiert und in den *Anfrage_Daten* gespeichert.

Die Klasse Eingabe_Server
Beim Systemstart wird eine Instanz der Klasse *Eingabe_Server* initialisiert. *Eingabe_Server* legt dann Instanzen der Klassen *Profil_Container*, *Betreiber_Container* und *Werbe_Container* an. Über die in den Container-Klassen enthaltene Methode ***fuell_dich*** werden alle Instanzen der Klassen *Betreiber* und *Werbung* sowie alle Gast-Profile von der Datenbank angefordert und an die entsprechenden Container-Klassen angehängt. Über die Methoden ***gib_profil_container***, ***gib_betreiber_container*** und ***gib_werbe_container*** liefert *Eingabe_Server* auf Anfrage Instanzen der Container-Klassen mit Kopien der angeforderten Objekte.

Die Klasse Anfrage_Daten und NameWertPaar
Die Klasse *Anfrage_Daten* dient der strukturierten Speicherung eines von *Server* an *Eingabe* übermittelten Anfrage-Strings in Name/Wert-Paaren. Zu diesem Zweck wird der Anfrage-String in seine Bestandteile zerlegt und über die Methode ***speicher_nw_paar*** in einer Instanz der Klasse *NameWertPaar* gespeichert. Beim ersten Aufruf der Methode ***speicher_nw_paar*** werden die Attribute des bereits bei der Initialisierung von *Anfrage_Daten* angelegten *NameWertPaar*-Objekts mit den entsprechenden Werten gefüllt. Der nächste Aufruf der Methode veranlaßt das erste *NameWertPaar*-Objekt eine neue Instanz der Klasse *NameWertPaar* zu initialisieren und dessen Methode ***speicher_nw_paar*** aufzurufen. Anschließend wird diese neue Instanz im Attribut **next** aggregiert. Jeder weitere Aufruf der Methode ***speicher_nw_paar*** wird bis zum letzten

[1] Das System PEP kann von verschiedenen Betreibern genutzt werden. Jeder Betreiber verfügt über einen individuell gestalteten Web-Auftritt. Die TV-Programmdaten und Funktionalitäten von PEP werden von allen Betreibern gemeinsam genutzt (vgl. Beschreibung der Klasse Betreiber im Data-Dictionary der Analyse – Kapitel 5.3.2).

NameWertPaar-Objekt durchgereicht und dort entsprechend verarbeitet. Die Methoden ***gib_wert***, ***setze_wert*** und ***gib_name*** ermöglichen den späteren Zugriff auf die Attribute der einzelnen *NameWertPaare*.

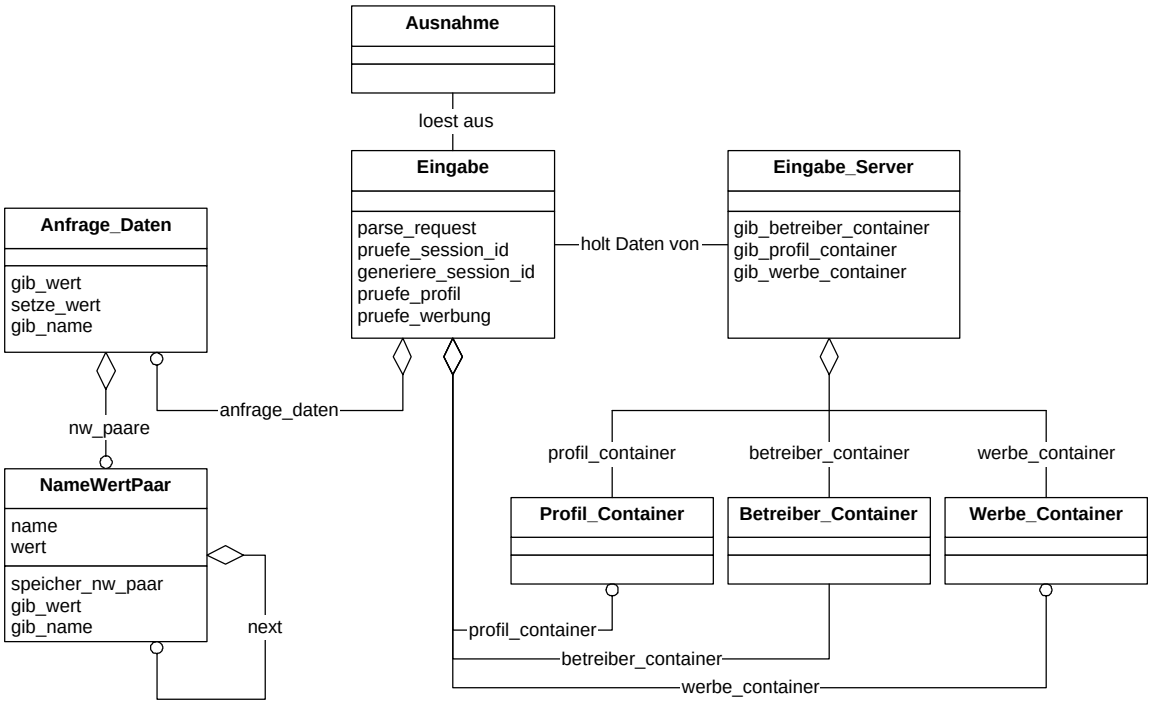


Abbildung 34: Klassendiagramm Eingabe

Das Klassendiagramm Eingabe bildet den Abschluß der Beschreibung der statischen Strukturen wesentlicher Teilbereiche des Systems. Im folgenden Kapitel werden nun die dynamischen Vorgänge zwischen den in diesem Kapitel beschriebenen statischen Strukturen des Systems erörtert.

5.4.2 Dynamisches Modell

Aufbauend auf dem im vorhergehenden Kapitel erläuterten Objektmodell und den im Systementwurf getroffenen Entscheidungen soll in diesem Kapitel das dynamische Modell erläutert werden. Zu diesem Zweck wird der Ablauf der Generierung eines Dokuments exemplarisch dargestellt, um die dynamischen Vorgänge innerhalb des Systems zu veranschaulichen. Zur Erläuterung dieser Generierung werden in diesem Kapitel ausschließlich Ereignisfluß- und Zustandsdiagramme verwendet, da die Erstellung der entsprechenden Ereignispfaddiagramme lediglich einen Zwischenschritt auf dem Weg zur Erstellung der Zustandsdiagramme darstellt. Dabei wurden Diagramme, die ein relativ einfaches dynamisches Verhalten repräsentieren nicht in dieses Kapitel integriert. Die relevanten dynamischen Aspekte dieser Diagramme werden innerhalb des Textes erörtert. An dieser Stelle sei noch einmal auf den Ergänzungsband zu dieser Diplomarbeit verwiesen, der das komplette dynamische Modell, inklusive der Ereignispfaddiagramme, enthält.

Identifizierung und Behandlung der auslösenden Ereignisse
Äquivalent zum Vorgehen der dynamischen Modellierung in Kapitel 5.2.2 sollen zunächst die „auslösenden Ereignisse“ identifiziert werden. Die Generierung eines Dokuments beginnt mit dem Eintreten des in Kapitel 5.2.2 eingeführten Ereignisses „*Server* **fordert Dokument** von *Eingabe* **an**“. Dieses auslösende Ereignis hat weiterhin seine Gültigkeit behalten, wurde jedoch auf „verarbeite Request“ vereinfacht. Die Ereignisfolgen, die dieses Ereignis innerhalb des Publikationssystems auslöst, sollen jetzt erläutert werden. Im weiteren Verlauf dieses Kapitels wird die Betrachtung der Ereignisfolgen durch die Klassenhierarchie Schritt für Schritt vervollständigt.

Die Klasse *Eingabe* ist nach wie vor der Empfänger des von *Server* ausgelösten Ereignisses „verarbeite Request“. Die dynamischen Vorgänge zwischen den Klassen *Server* und *Eingabe* werden gemeinsam mit den in Kapitel 5.4.1 eingeführten Klassen *HTML_Kneter*, *PDF_Kneter* und *Eingabe_Server* erläutert. Abbildung 35 zeigt das zu den oben aufgeführten Klassen passende Ereignisflußdiagramm und die zwischen den Klassen auftretenden Ereignisse.

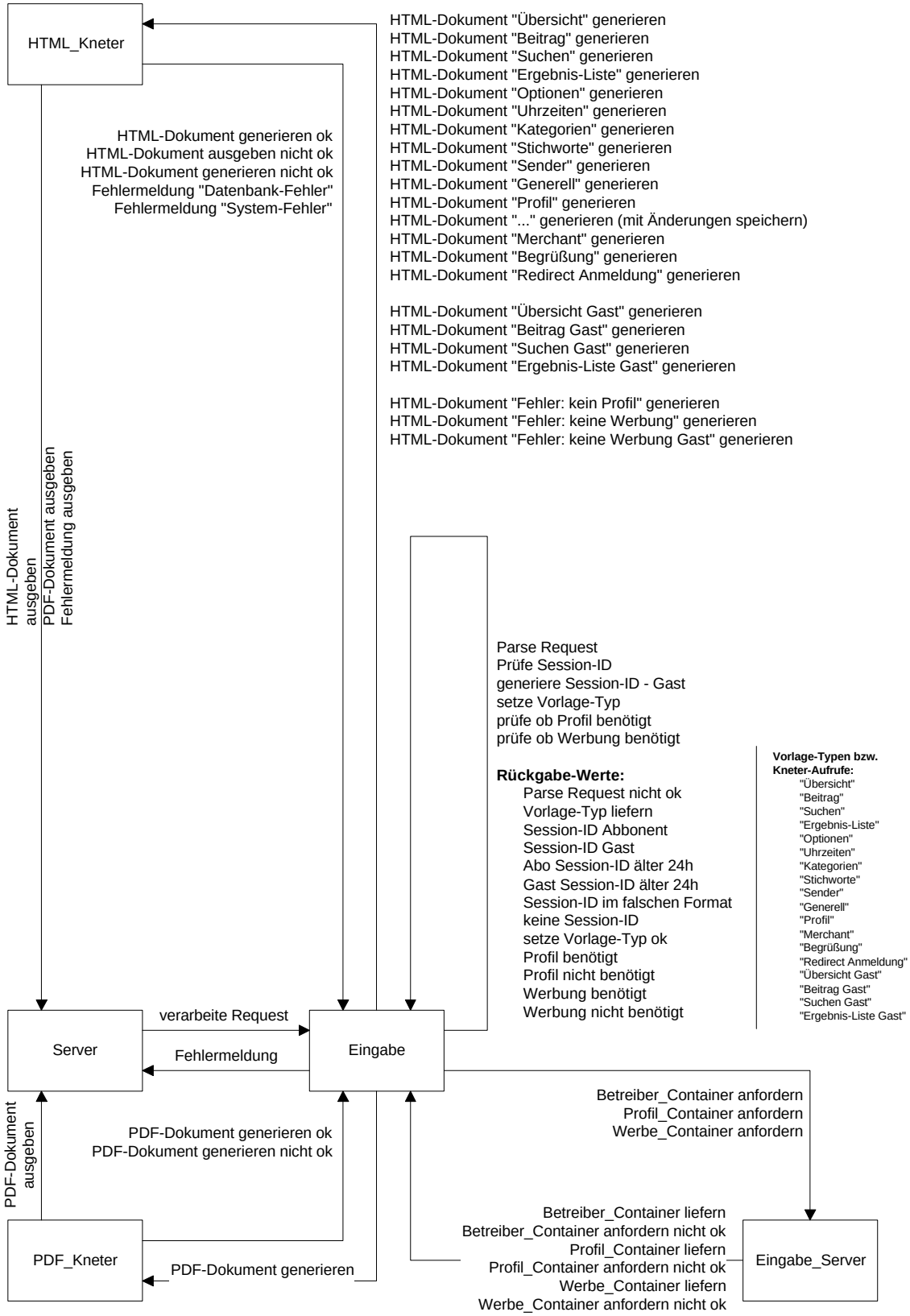


Abbildung 35: Ereignisflußdiagramm Server – Eingabe – Eingabe_Server – HTML_Kneter – PDF_Kneter

Server sendet zunächst eine Anfrage (Ereignis „verarbeite Request") an *Eingabe*. *Eingabe* wertet diese Anfrage aus (Ereignis „parse request"), um alle vom *Server* übermittelten Informationen in eine strukturierte Form zu überführen. Die strukturierten Daten werden in einer Instanz der Klasse *Anfrage_Daten* gespeichert. Mit Hilfe der Klasse *Anfrage_Daten* ermittelt *Eingabe* den Typ des zu generierenden Dokuments und den Identifizierer des betreffenden *Betreibers*.[1] Diese Werte werden von *Eingabe* benötigt, um die Generierung eines Dokuments anstoßen zu können. Für die erfolgreiche Generierung eines Dokuments sind außerdem noch weitere Komponenten nötig, die innerhalb der Datenbank gespeichert und über das ODBMS aus der Datenbank geladen werden. Wie in Kapitel 5.4.1 ausgeführt, wird das Laden aus der Datenbank und der anschließende Zugriff auf die Bestandteile der einzelnen Komponenten über sogenannte Container-Klassen realisiert. Diese Container-Klassen entscheiden für jeden Vorgang, ob für dessen Ausführung eine Front- oder Back-End-Aktion zuständig ist. So werden beim Laden eines Objekts aus der Datenbank Back-End-Funktionalitäten verwendet, während für das Lesen von Attribut-Inhalten zur Laufzeit grundsätzlich Front-End-Aktionen zuständig sind.

Eingabe_Server liefert an *Eingabe* verschiedene Container-Klassen. Diese Container-Klassen werden später von *Eingabe* verwendet, um *HTML_Kneter* bzw. *PDF_Kneter* zu der Generierung eines zur Anfrage passenden Dokumentes zu veranlassen. Abbildung 36 und Abbildung 37 zeigen das Zustandsdiagramm, für das Ereignis „verarbeite Request", aus Sicht der Klasse *Eingabe*.

[1] Der Identifizierer eines *Betreibers* ist üblicherweise dessen URL, die innerhalb der Klasse *Betreiber* im Attribut **name** gespeichert wird.

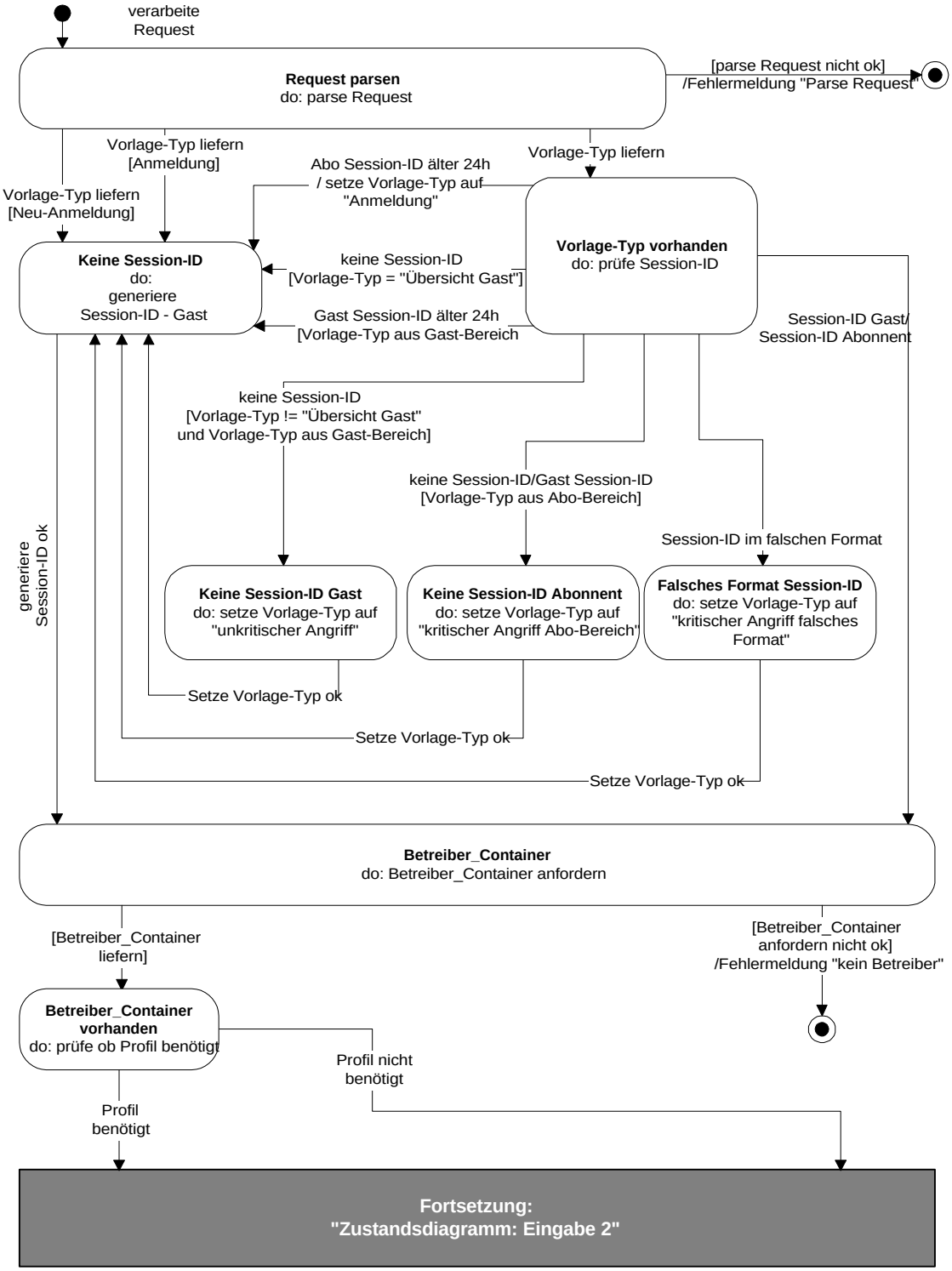
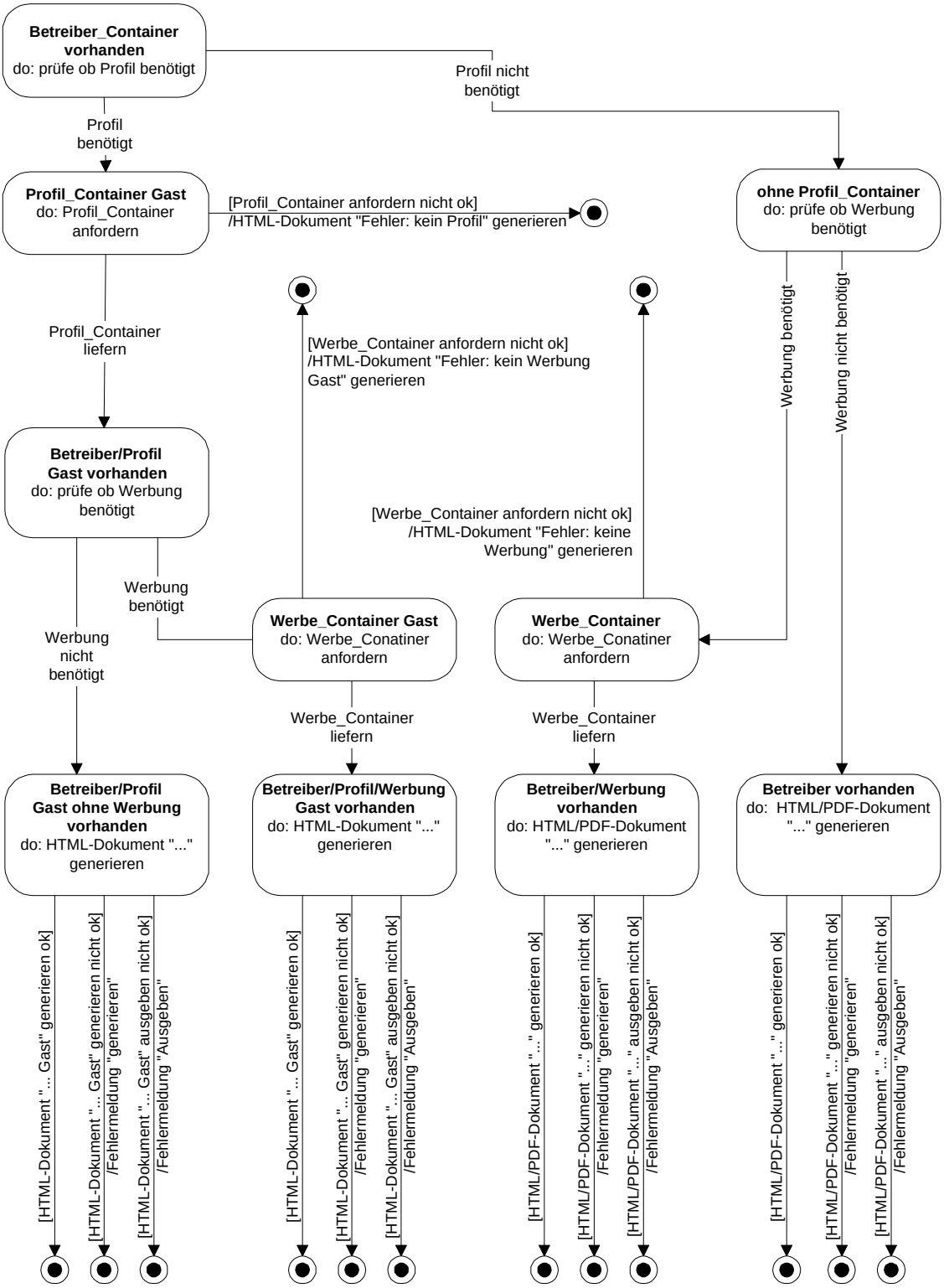


Abbildung 36: Zustandsdiagramm Eingabe



Bemerkung:
Die einzelnen Ereignisse "HTML/PDF-Dokumente "..." generieren" bzw. "HTML-Dokumente "... Gast" generieren" sind unterscheidbare Aktionen mit jeweils drei unter scheidbaren Endzuständen. Aus Platzgründen wurden sie in diesem Zustandsdiagramm zusammengefaßt.

Abbildung 37: Zustandsdiagramm Eingabe (Fortsetzung)

Die Anfragedaten werden von *Server* als Zeichenkette an *Eingabe* geliefert, deren einzelne Bestandteile durch ein dediziertes Trennzeichen getrennt sind.[1] Die Aktion „parse Request“ im Startzustand der Abbildung 36 strukturiert die Anfragedaten des *Servers*, indem es die einzelnen Bestandteile voneinander trennt und als Name/Wert-Paare in einer Instanz der Klasse *Anfrage_Daten* speichert. Anhand dieser strukturierten Anfragedaten kann *Eingabe* nun die Dokument-Vorlage für die Generierung des Dokuments ermitteln.

Zugriffs-Kontrollmechanismen
Je nach ermitteltem Vorlage-Typ, wird in einem der beiden Folgezustände die übermittelte Session-ID geprüft oder eine neue Session-ID generiert (siehe Kapitel 5.3.3). Die Prüfung der Session-ID beschränkt sich hier auf eine Überprüfung des Formats und des Alters der ID. Wenn keine Session-ID übermittelt worden ist oder wenn die übermittelte Session-ID älter als 24 Stunden ist oder wenn die übermittelte Session-ID nicht das korrekte Format hat, dann werden jeweils alternative Zustandsübergänge gewählt. Die auf diesem Wege erreichbaren Zustände „Keine Session-ID – Gast“, „keine Session-ID – Abo“ und „Falsches Format Session-ID“ markieren potentielle „Angriffe“ auf das System. Alle diese „Angriffe“ werden in einer Logdatei protokolliert.

Der Zustand „Keine Session-ID – Gast“ wird als „unkritischer Angriff“ eingestuft, da keine sicherheitsrelevanten Bestandteile des Systems attackiert werden. Das zu generierende Dokument beinhaltet ausschließlich Informationen aus dem öffentlichen Gast-Bereich von PEP. Trotzdem wird das Eintreten dieses Ereignisses protokolliert, da es sich um eine Vorstufe zu einem „kritischen Angriff“ handeln kann. Bei einer „korrekten“ Bedienung des Systems kann dieses Ereignis nicht eintreten. Der Angreifer könnte statt dessen einen Versuch gestartet haben, über eine Manipulation der URL in geschützte Bereiche des Systems vorzudringen. Gelingt es einem Angreifer letztendlich über die gezielte Manipulation der URL den Vorlage-Typ eines Abonnenten-Dokumentes herauszufinden, muß er zusätzlich über eine gültige Abonnenten-Session-ID verfügen, um in einen geschützten Bereich vordringen zu können. Hat der Angreifer keine gültige Abonnenten-Session-ID, wird beim Versuch ein Abonnenten-Dokument generieren zu lassen, einer der beiden Zustände „keine Session-ID – Abo“ oder „Falsches Format Session-ID“ erreicht. Diese Zustände repräsentieren „kritische Angriffe“, die ebenfalls protokolliert werden. Jeder Betreiber kann für jeden der möglichen Angriffsfälle jeweils eine Dokument-Vorlage definieren, die auf das Erreichen der entsprechenden „Angriffs-Zustände“ reagieren. So bleibt es dem Betreiber überlassen, ob er einem Angreifer den Mißerfolg seines Angriffes explizit mitteilt oder ob er ein reguläres Dokument generieren läßt (zum Beispiel die TV-Programm-Übersicht aus dem Gast-Bereich).

Anfordern der benötigten Container-Klassen von *Eingabe_Server*
Im Anschluß an die oben geschilderten Zugriffs-Kontrollmechanismen fordert *Eingabe* die benötigten Daten von *Eingabe_Server* an, um die Generierung des angeforderten Dokuments anzustoßen. Mit Hilfe der aus *Anfrage_Daten* ermittelten Werte Dokument-

[1] Als Beispiel soll hier die Übertragung des „Querystring“ eines HTTP-Requests mittels CGI dienen. Hier werden die Inhalte der einzelnen Formularfelder durch das Zeichen „&“ getrennt. Sonderzeichen werden durch ein Prozentzeichen gefolgt von ihrer Ordnungszahl im hexadezimalen Format übermittelt (Beispiel: "%F6" entspricht "ö"). Leerzeichen werden bei der Übertragung durch "+" ersetzt.

Typ und Betreiber-Identifizierer fordert *Eingabe* nun eine Instanz der Klasse *Betreiber_Container* mit einer Kopie des *Betreiber*-Objekts, das den entsprechenden Identifizierer enthält, beim *Eingabe_Server* an (Ereignis „Betreiber_Container anfordern“). Die Kopie des *Betreiber*-Objekts enthält unter anderem die Dokument-Vorlage, die zur Generierung des gewünschten Dokuments benötigt wird. Handelt es sich bei dem zu generierenden Dokument um ein Gast-Dokument (Ereignis „prüfe ob Profil benötigt“), fordert *Eingabe* anschließend das Gast-Profil des *Betreibers* beim *Eingabe_Server* an (Ereignis „Profil_Container anfordern“). Dieses Gast-Profil ist, äquivalent zum *Betreiber*, an die Container-Klasse *Profil_Container* gebunden. Nach der Prüfung, ob *Werbung* benötigt wird (Ereignis „prüfe ob Werbung benötigt“), wird die Kette der Anforderungen von *Eingabe* an *Eingabe_Server* durch die Anforderung des *Werbe_Containers* bei *Eingabe_Server* abgeschlossen (Ereignis „Werbe_Container anfordern“). Dieses Sammeln der benötigten Daten ist Bestandteil des folgenden Abschnitts, der auf die Besonderheiten des Systems im Zusammenhang mit der Klasse *Eingabe_Server* eingeht.

Das Ereignisflußdiagramm in Abbildung 38 stellt die auftretenden Ereignisse zwischen den Klassen *Eingabe_Server*, *Betreiber_Container*, *Betreiber_Aktion_FE* und *Betreiber* dar. Das in der vorhergehenden Beschreibung erwähnte Ereignis „Betreiber_Container anfordern“ wird neben anderen Ereignissen im Ereignisflußdiagramm in Abbildung 38 aufgeführt.

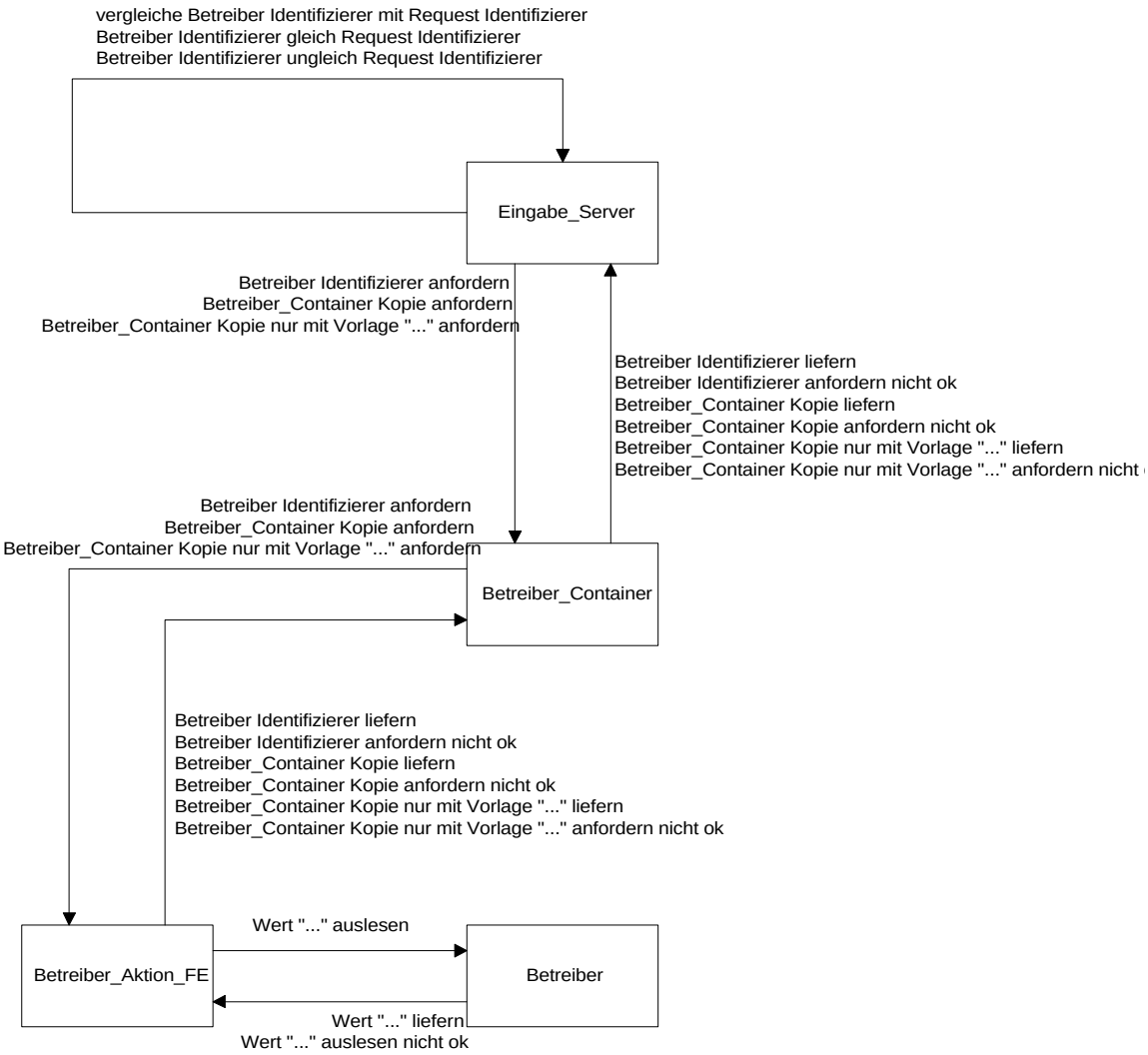


Abbildung 38: Ereignisflußdiagramm Eingabe_Server – Betreiber_Container – Betreiber_Aktion_FE – Betreiber

Wie im „Systementwurf (vgl. Kapitel 5.3.1) gefordert, werden u.a. alle persistenten Instanzen der Klasse *Betreiber* im „Front-End“ vorgehalten, um unnötige Datenbankzugriffe zu vermeiden. Die Klasse *Eingabe_Server* übernimmt diese Aufgabe. Zu diesem Zweck wird beim Systemstart eine Instanz der Klasse *Betreiber_Container* initialisiert, die alle *Betreiber*-Objekte aus der Datenbank über *Betreiber_Aktion_BE* anfordert und innerhalb einer Zeigerstruktur bereithält. Auf Anfrage von *Eingabe_Server* liefert *Betreiber_Container* den Identifizierer eines angehängten *Betreibers*. *Eingabe_Server* entscheidet anhand des von *Betreiber_Container* gelieferten Werts, ob es sich bei dem *Betreiber* um den geforderten handelt. Ist dies nicht der Fall, fordert *Eingabe_Server* den Identifizierer des nächsten *Betreibers* bei *Betreiber_Container* an. Dieser Vorgang wird wiederholt, bis der geforderte *Betreiber* identifiziert worden ist. Anschließend fordert *Eingabe_Server* eine neue Instanz der Klasse *Betreiber_Container* an, die eine Kopie des ermittelten *Betreibers* enthält. Diese Kopie des *Betreibers* enthält nur die zur Generierung des Dokuments benötigte Vorlage.

Container-Klassen

Die oben geschilderte Ermittlung der Identifizierer der einzelnen *Betreiber*-Instanzen wird nicht von *Betreiber_Container* selbst vorgenommen. Vielmehr reicht *Betreiber_Container* die Anfrage von *Eingabe_Server* an *Betreiber_Aktion_FE* weiter. *Betreiber_Aktion_FE* ruft eine Methode der Klasse *Betreiber* auf, die dessen Identifizierer zurückliefert. Anschließend reicht *Betreiber_Aktion_FE* den von *Betreiber* erhaltenen Wert an *Betreiber_Container* zurück. *Betreiber_Container* übergibt den Wert wiederum an *Eingabe_Server*. Abbildung 39 zeigt diesen Vorgang in Form eines Zustandsdiagramms aus Sicht der Klasse *Eingabe_Server*.

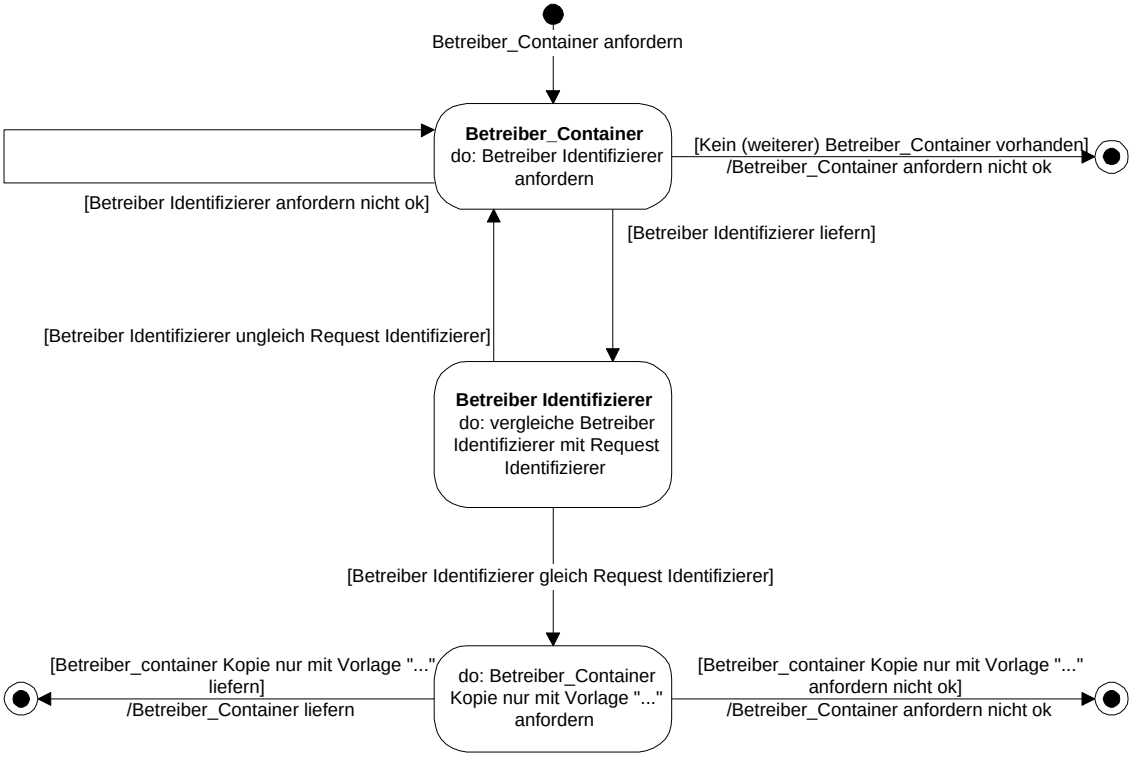


Abbildung 39: Zustandsdiagramm Eingabe_Server → Betreiber_Container anfordern

Bei der Anforderung des Gast-Profils und der Werbung wird äquivalent verfahren. Die Ermittlung des korrekten *Profil*-Objekts geschieht dabei allerdings nicht über einen Identifizierer. Wie in Abbildung 29 aus Kapitel 5.4.1 zu erkennen ist, verfügt die Klasse *Betreiber* jeweils über eine Assoziation zu den beiden Klassen *Profil* und *Werbung*. Über diese Assoziation ist eine eindeutige Zuordnung der beiden Klassen zu einem Betreiber möglich. Nachdem *Eingabe* nun die zur Generierung benötigten Daten ermittelt und erhalten hat, wird die Generierung des HTML-Dokuments angestoßen. Die akquirierten Container-Klassen und die strukturierten Anfragedaten werden an *HTML_Kneter* übergeben. Um während des laufenden Betriebes Generierungszeit zu sparen, werden einige Dokumente aus dem Gast-Bereich nach der Generierung innerhalb der Verzeichnisstruktur gespeichert. Dies sind Dokumente, die die verschiedenen Zeitfenster der TV-Programm-Übersicht repräsentieren und die diversen Einzelansichten zu den einzelnen Beiträgen. *HTML_Kneter* prüft anhand der übergebenen Anfragedaten, ob das angeforderte Dokument bereits generiert wurde. Ist dies der Fall, wird die in der

Verzeichnisstruktur abgelegte HTML-Datei geöffnet und an den *Server* übertragen. Andernfalls fährt *HTML_Kneter* mit der Generierung fort.

Anforderung eines Benutzer-Profils unter Berücksichtigung von zusätzlichen Zugriffs-Kontrollmechanismen

Um die persönlichen Interessen eines Benutzers berücksichtigen zu können, muß vor der eigentlichen Anforderung von Beiträgen aus der Datenbank das *Profil* des betreffenden Benutzers aus der Datenbank geladen werden. Die Auswahl des korrekten *Profils* geschieht durch *Profil_Aktion_BE*. *Profil_Aktion_BE* ermittelt anhand der in *Anfrage_Daten* enthaltenen Session-ID des Benutzers das entsprechende *Profil*-Objekt. Dazu wird aus der Session-ID die Objekt-ID des *Profil*-Objekts extrahiert, um mit dieser Objekt-ID das *Profil*-Objekt beim *Datenbank_Server* anzufordern. Bei der Anforderung des Benutzerprofils müssen mehrere Fehlerfälle berücksichtigt werden. Abbildung 40 zeigt die Generierung eines Abonnenten-Dokuments aus Sicht der Klasse *HTML_Kneter*. Fast alle Endzustände markieren dabei mögliche Fehler. Lediglich der Zustand, der über den Zustandsübergang „HTML-Dokument ausgeben ok“ erreicht wird, repräsentiert einen fehlerfreien Generierungsvorgang.

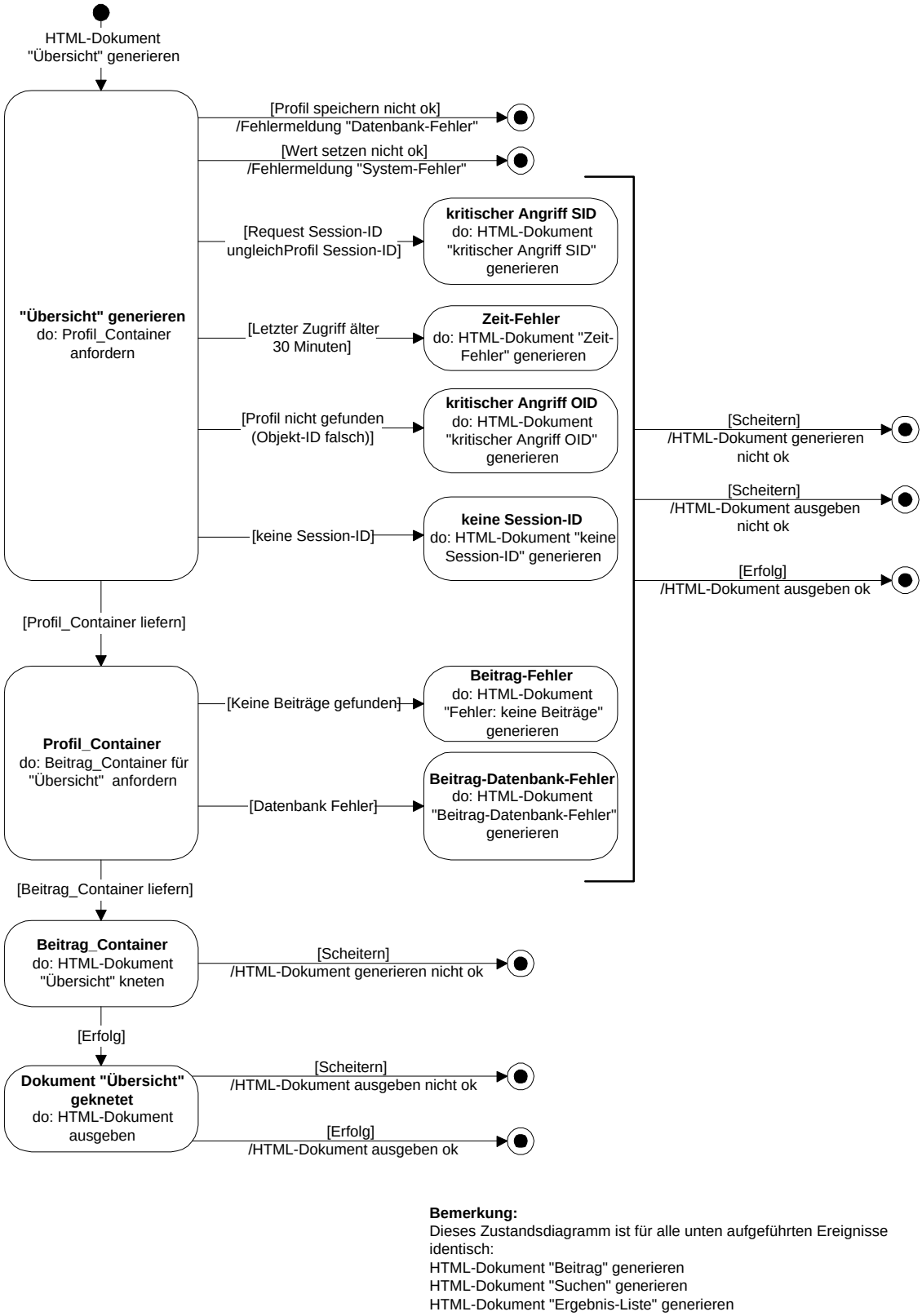


Abbildung 40: Zustandsdiagramm HTML_Kneter → Abonnenten-Dokument generieren

Zu Beginn der Anforderung eines Benutzer-Profils durch *HTML_Kneter* steht die Ermittlung der Objekt-ID des dem Benutzer zugeordneten *Profil*-Objekts. Kann anhand dieser Objekt-ID kein *Profil*-Objekt vom *Datenbank_Server* geliefert werden, ist davon auszugehen, daß die Session-ID von einem Angreifer manipuliert wurde. Dies hat das Erreichen des Zustandes „kritischer Angriff OID“ (OID entspricht **O**bjekt-**ID**) zur Folge. In diesem Fall wird eine entsprechende Fehlermeldung erzeugt und die Generierung des angeforderten Dokumentes wird abgebrochen.

Nach der erfolgreichen Anforderung des *Profil*-Objekts wird von *Profil_Aktion_BE* die in den *Anfrage_Daten* enthaltene Session-ID mit der im *Profil*-Objekt gespeicherten Session-ID verglichen. Stimmen beide IDs nicht überein, erreicht *HTML_Kneter* den Zustand „kritischer Angriff SID“ (SID entspricht **S**ession-**ID**), der ebenfalls eine Fehlermeldung und den Abbruch der Generierung zur Folge hat. Beide oben aufgeführten „Angriffszustände“ stellen jeweils ein mögliches Resultat der in Kapitel 5.3.3 eingeführten Zugriffs-Kontrollmechanismen dar.

Ein weiterer Bestandteil der Zugriffs-Kontrollmechanismen ist die Überprüfung des Zeitpunkts des letzten aktiven Zugriffs eines Benutzers auf das Publikationssystem. Liegt dieser Zugriff länger als 30 Minuten zurück, wird die bei der Anmeldung an einen Benutzer vergebene Session-ID ungültig (vgl. Kapitel 5.3.3). Tritt dieser Fall ein, erreicht *HTML_Kneter* den Zustand „Zeit-Fehler". *HTML_Kneter* produziert eine Seite, die den Benutzer auffordert, sich erneut beim System anzumelden. Ist der letzte Zugriff des Benutzers innerhalb der vergangenen 30 Minuten vorgenommen worden, wird bei der Anforderung des *Profil*-Objekts aus der Datenbank die aktuelle Uhrzeit im *Profil*-Objekt eingetragen. Anschließend wird das Benutzer-Profil in der Datenbank gespeichert.

Tritt beim Eintragen der aktuellen Uhrzeit oder beim Speichern des *Profil*-Objekts ein Fehler auf, so erreicht *HTML_Kneter* einen Endzustand und initiiert das Ereignis „Fehlermeldung "Datenbank-Fehler"“ bzw. „Fehlermeldung "System-Fehler"“. Der oben erwähnte Datenbank-Fehler wird durch die Ausnahmebehandlung des Systems behandelt und führt zu einer geordneten Terminierung des Systems. Diese Terminierung wird durchgeführt, da die permanente Verbindung zur Datenbank offensichtlich unterbrochen ist. Ohne diese Verbindung ist ein ordnungsgemäßer Betrieb des Publikationssystems nicht mehr gewährleistet.

Sollte bei der Eintragung der aktuellen Uhrzeit in das Profil-Objekt ein Fehler auftreten, so ist dies nur darauf zurückzuführen, daß für die Anforderung des Speicherplatzes für eine Instanz der Klasse *PEPZeit* nicht genügend Arbeitsspeicher zur Verfügung steht. Dieser Fehler führt ebenfalls zur sofortigen Terminierung des Systems. Allerdings wird auf die Verwendung der Ausnahmebehandlung verzichtet, da für die Initialisierung des *Ausnahme*-Objekts ebenfalls Arbeitsspeicher benötigt wird, der offensichtlich nicht zur Verfügung steht. Vielmehr werden alle zum Front-End bestehenden Socket-Verbindungen geschlossen und das Publikationssystem anschließend terminiert. Durch die Verwendung von Transaktionsmechanismen bei schreibenden Zugriffen auf die Datenbank, werden Dateninkonsistenzen bei dieser Terminierung verhindert.

Sechs der insgesamt acht in Abbildung 40 aufgeführten Angriffs- und Fehlerzustände bei der Anforderung einer Container-Klasse haben als auszuführende Folge-Aktion die

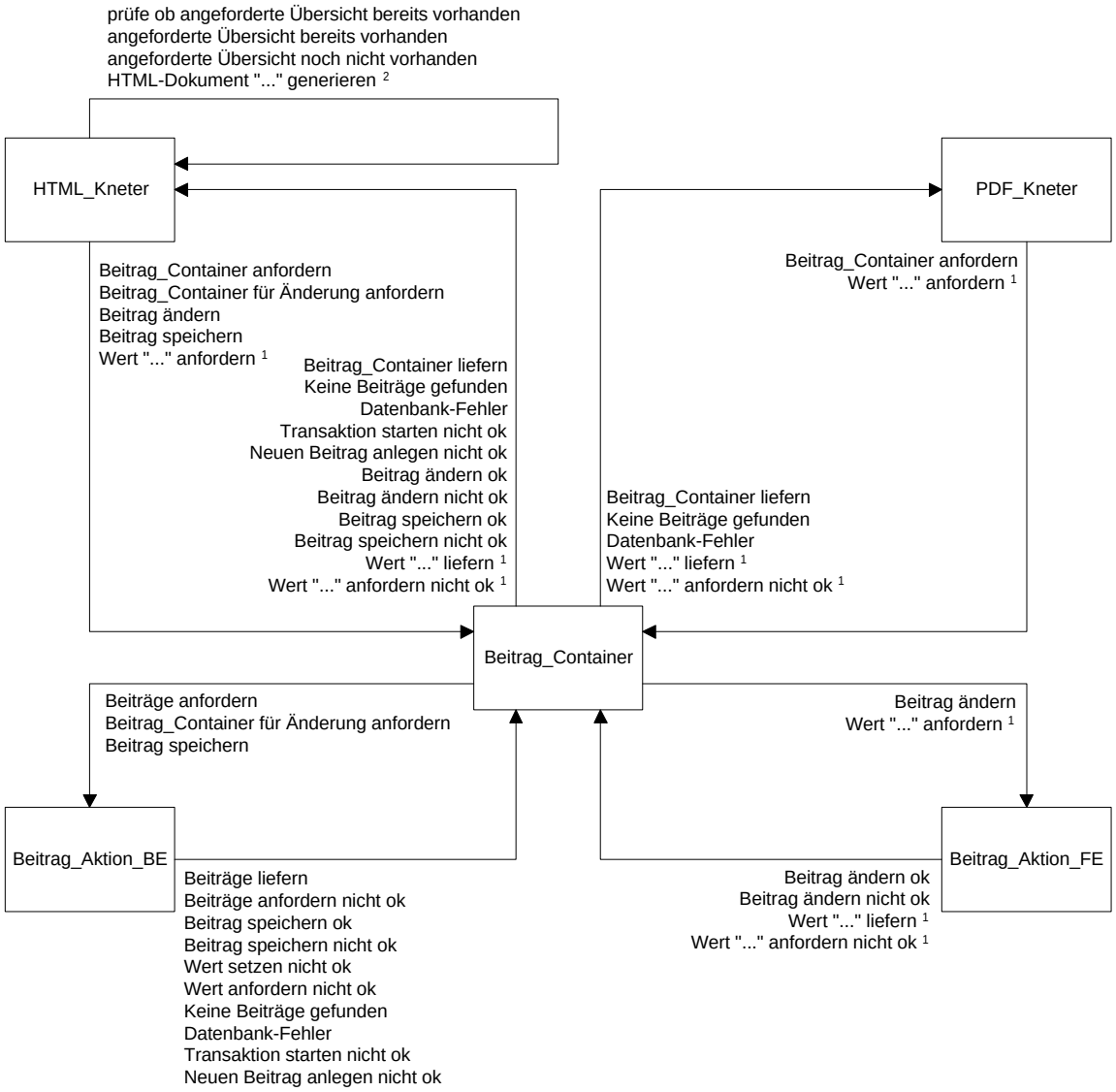
Generierung eines HTML-Dokuments. Diese Dokumente melden das Auftreten eines Fehlers an den Benutzer. Die möglichen Fehler, die bei der Generierung dieser Fehler-Dokumente auftreten können, sind in Abbildung 40 teilweise zusammengefaßt dargestellt. Die Folge-Aktionen können ihrerseits scheitern. Die drei Endzustände auf der äußersten rechten Seite des Diagramms repräsentieren den Erfolg und die möglichen Fehler bei der Erzeugung der Fehler-Dokumente. Weitere Unregelmäßigkeiten können bei der Erzeugung oder der Ausgabe des erzeugten Fehler-Dokuments auftreten (Übertragung des Dokumentes an *Server*). Alle auftretenden kritischen Fehler werden durch eine Ausnahmebehandlung protokolliert und der Güte der Fehler entsprechend behandelt. Ferner könnte, wie auch bei der Anforderung des Benutzer-Profils, die Verbindung zur Datenbank unterbrochen sein. Dies würde ebenfalls die Terminierung des Publikationssystems nach sich ziehen, da ein weiterer Betrieb des Publikationssystems ohne eine Verbindung zur Datenbank nicht sinnvoll ist.[1]

Nachdem *HTML_Kneter* das benötigte Benutzer-Profil erhalten hat, wird die Generierung des angeforderten Dokuments fortgesetzt. Im folgenden Abschnitt wird vom Inhalt des Profil-Objekts abstrahiert. Ab diesem Zeitpunkt ist es unerheblich, ob das Profil einem Benutzer zugeordnet wurde oder ob es sich um das Gast-Profil handelt.

Anfordern der Beiträge aus der Datenbank

Um das zu generierende HTML-Dokument mit Inhalten zu füllen, müssen zunächst die zu verarbeitenden *Beiträge* aus der Datenbank angefordert werden (vgl. Abbildung 41).

[1] Die Terminierung des Publikationssystems wird generell bei einem Auftreten eines Datenbank-Fehlers durchgeführt.



1 "..." steht für alle Werte, die in einem Beitrag eingetragen sein können.
2 "..." steht für die Werte "Fehler: Beitrag ändern", "Fehler: Beitrag speichern".

Abbildung 41: Ereignisflußdiagramm HTML_Kneter – PDF_Kneter – Beitrag_Container – Beitrag_Aktion_BE – Beitrag_Aktion_FE

Zuerst wird eine neue Instanz der Klasse *Beitrag_Container* initialisiert. Anschließend wird *Beitrag_Container* aufgefordert, sich mit Beiträgen zu füllen (Ereignis „Beitrag_Container anfordern“). Bei der Auswahl der *Beiträge*, die von *Beitrag_Aktion_BE* vorgenommen wird, werden sowohl die *Anfrage_Daten* als auch das *Profil* berücksichtigt. Dementsprechend müssen beide Objekte bei der Anforderung der Beiträge mit übergeben werden.

Abbildung 42 zeigt den Vorgang der Generierung eines Gast-Dokuments aus Sicht der Klasse *HTML_Kneter*. Diese Abbildung stimmt mit dem unteren Teil der Abbildung 40 überein, da dieser Teil der Generierung für Abonnenten- und Gast-Dokumente identisch ist.

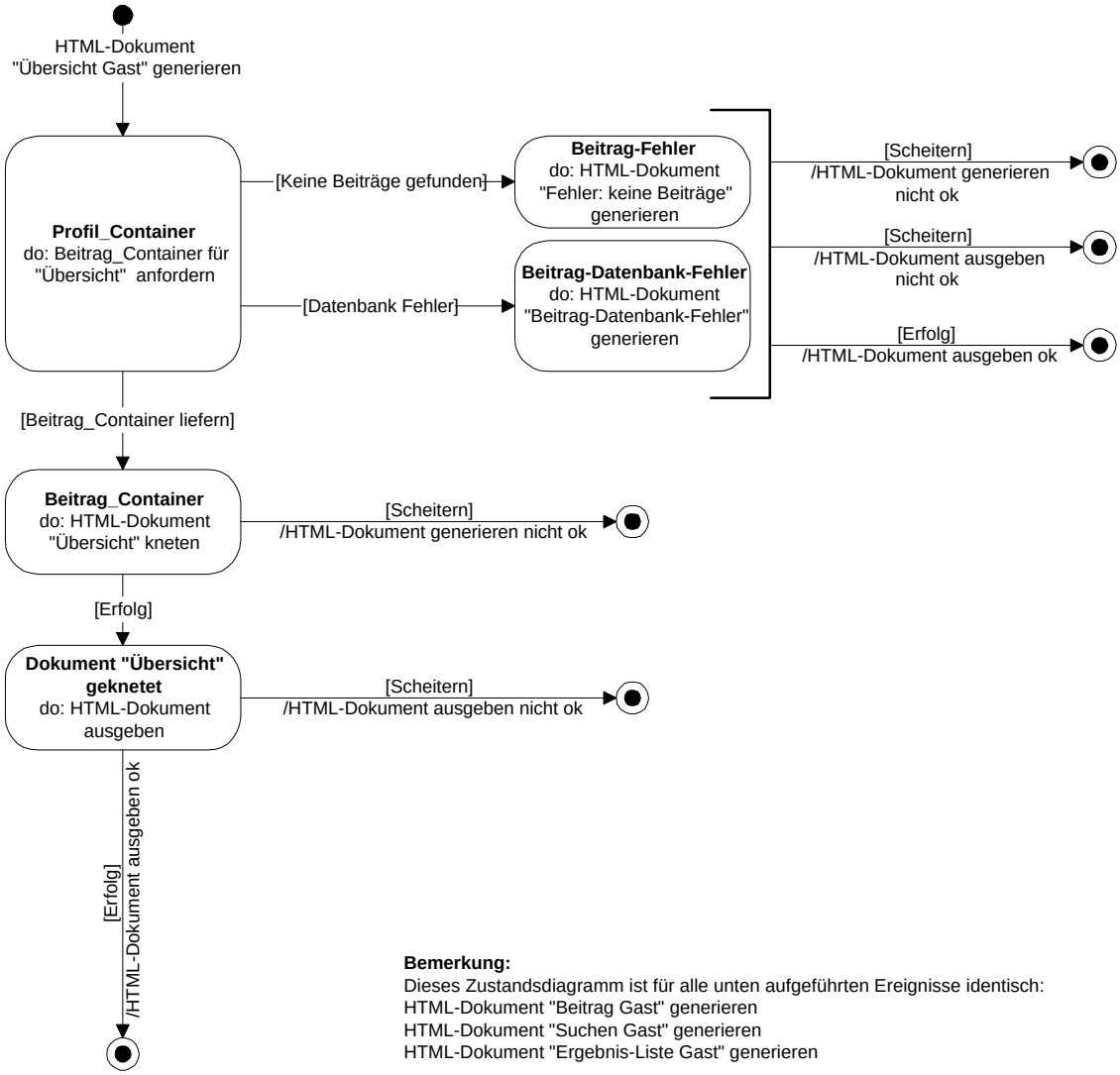


Abbildung 42: Zustandsdiagramm HTML_Kneter Gast-Dokument generieren

Die Anforderung aller benötigten Beiträge, die letztendlich von *Beitrag_Aktion_BE* realisiert wird, erfordert, wie erwähnt, eine Berücksichtigung der *Anfrage_Daten* und der Profil-Einstellungen. Wie in Kapitel 3.3 geschildert, wird beispielsweise die TV-Programm-Übersicht im Normalfall ausschließlich anhand der Profileinstellungen eines Benutzers erzeugt. Dem Benutzer werden aber auch Möglichkeiten geboten, über Selektionsfelder von seinem Profil bzw. vom Gast-Profil abweichende Einstellungen für den Ausschnitt der in der Übersicht anzuzeigenden Beiträge vorzunehmen. So kann er zum Beispiel statt der im Profil festgelegten Sparten nur eine einzige Sparte in der Übersicht anzeigen lassen. In diesem Fall wird die Auswahl der anzuzeigenden Beiträge über das Profil **und** über die *Anfrage_Daten* vorgenommen. Berücksichtigung findet in diesem Fall die Sparte aus den *Anfrage_Daten* und alle Einstellungen des Profils mit Ausnahme der Sparten-Auswahl. Grundsätzlich werden alle Einstellungen des Profils ignoriert, die von Werten aus den *Anfrage_Daten* überschrieben werden.

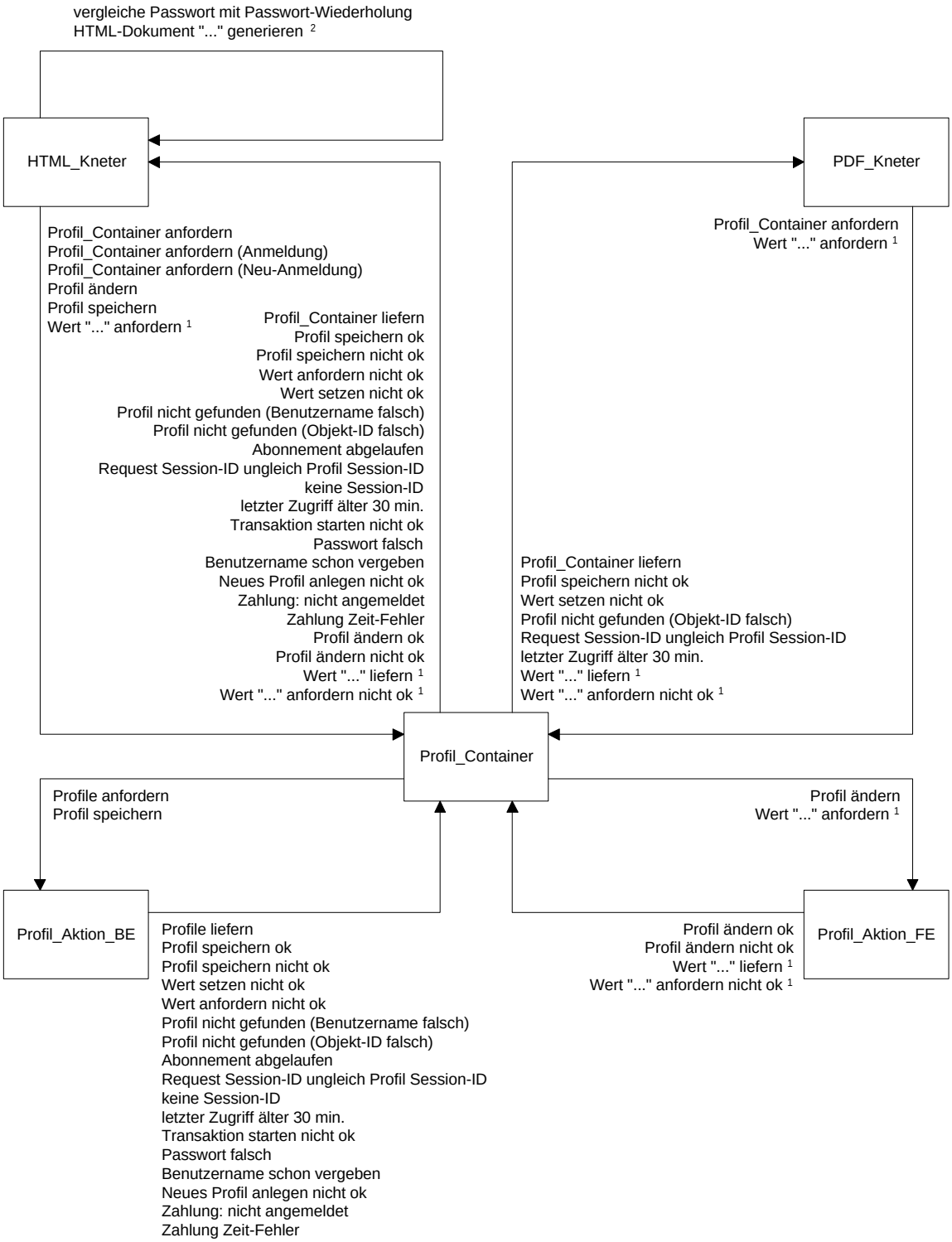
Beitrag_Aktion_BE extrahiert aus *Anfrage_Daten* und *Profil* die für die Anforderung relevanten Informationen und übergibt sie an *Datenbank_Server*. *Datenbank_Server* wiederum setzt aus den übermittelten Informationen ein Query-Statement zusammen und sendet es an das POET ODBMS. Das Ermitteln der relevanten Informationen durch *Beitrag_Aktion_BE* beschränkt sich dabei auf das Abarbeiten der in *Anfrage_Daten* gespeicherten Name/Wert-Paare und der beitragsspezifischen Einträge im Profil. Trifft *Beitrag_Aktion_BE* im Namenseintrag eines Name/Wert-Paares auf den Namen eines Beitrag-Attributs, wird das entsprechende Name/Wert-Paar als relevant vermerkt (zum Beispiel „sender=ARD“). Alle Profil-Einstellungen, die nicht von einem relevanten Name/Wert-Paar überschrieben werden, werden an *Datenbank_Server* übermittelt.

Dem Benutzer, unabhängig davon ob es sich um einen Gast oder Abonnenten handelt, soll außer den standardmäßig angebotenen Übersichten und Einzelansichten die Möglichkeit geboten werden, auf dem gesamten Beitrag-Datenbestand eine Volltextsuche durchzuführen. Bei der Nutzung dieser Suchfunktionalitäten des Systems werden alle Profil-Einstellungen ignoriert. Die Auswahl der anzuzeigenden Beiträge wird ausschließlich anhand der angegebenen Suchbegriffe vorgenommen. Es wäre allerdings auch denkbar, einem Abonnenten die Wahl zu lassen, ob seine Profil-Einstellungen bei der Suche berücksichtigt werden sollen. Auf diese Weise könnte die Ergebnismenge auf für den Benutzer relevante Beiträge reduziert werden.

Die Anforderung der *Beiträge* aus der Datenbank kann aus zwei Gründen scheitern. Erstens ist es möglich, daß keine Beiträge gefunden werden konnten. Dies wäre zum Beispiel dann der Fall, wenn die Einstellungen des Benutzer-Profils so unglücklich gewählt worden sind, daß die entsprechende Recherche in der Datenbank kein Ergebnis hervorbringt. Außerdem könnte die Berücksichtigung der *Anfrage_Daten* bzw. die Kombination der *Anfrage_Daten* mit den Profil-Einstellungen eine leere Ergebnismenge hervorrufen.

Zugriff auf Profil-Einstellungen über die Klasse *Profil_Container*

Um die persönlichen Einstellungen eines Benutzers berücksichtigen zu können wird über die Klasse *Profil_Container* der Zugriff auf ein Benutzer-Profil zur Verfügung gestellt. *Profil_Container* stellt dabei lediglich eine Hülle dar, die das Profil „umschließt“. Die eigentlichen Zugriffsmechanismen auf die Inhalte des Profils werden jedoch von den Klassen *Profil_Aktion_FE, Profil_Aktion_BE* und den Methoden der Klasse *Profil* zur Verfügung gestellt. Abbildung 43 zeigt die Ereignisse zwischen den Klassen *HTML_Kneter, PDF_Kneter, Profil_Container, Profil_Aktion_FE* und *Profil_Aktion_BE* in Form eines Ereignisflußdiagramms.



[1] "..." steht für alle Werte, die in einem Profil eingetragen sein können.
[2] "..." steht für die Werte "Beitrag-Datenbank-Fehler", "Fehler: keine Beiträge", "kritischer Angriff SID", "kritischer Angriff OID", "Zeit-Fehler", "falsches Passwort", "unbekannter Benutzername", "Abonnement abgelaufen", "Fehler: Profil ändern", "Fehler: Profil speichern", "Passwort-Wiederholung", "Benutzername vergeben", "Zahlung nicht angemeldet", "keine Session-ID".

Abbildung 43: Ereignisflußdiagramm HTML_Kneter – PDF_Kneter – Profil_Container – Profil_Aktion_BE – Profil_Aktion_FE

Bei der näheren Betrachtung von Abbildung 43 fällt zunächst auf, daß die Klasse *HTML_Kneter* lediglich sechs[1] verschiedene Ereignisse an *Profil_Container* senden kann. Als mögliche Folgeereignisse von *Profil_Container* an *HTML_Kneter* sind jedoch 21 Ereignisse im Diagramm erkennbar. Diese „überzähligen" Folgeereignisse werden von den Aktionsklassen *Profil_Aktion_FE* und *Profil_Aktion_BE* erzeugt und durch *Profil_Container* an *HTML_Kneter* weitergereicht. Zum größten Teil sind dies Ereignisse, die das Auftreten eines Fehlers repräsentieren. Diese Fehler sind teilweise bereits bei der Beschreibung des Zustandsdiagramms in Abbildung 40 erörtert worden.

Die Ereignisse zwischen *PDF_Kneter* und *Profil_Container* stehen in einem ähnlichen Verhältnis zueinander wie die zwischen *HTML_Kneter* und *Profil_Container*. Vergleicht man jedoch die Ereignisse des *HTML_Kneters* mit denen des *PDF_Kneters*, so wird deutlich, daß der *PDF_Kneter* lediglich einen lesenden Zugriff auf Profil-Inhalte realisiert. *HTML_Kneter* dagegen liest, schreibt und speichert Profil-Inhalte durch die Ereignisse „Wert "..." anfordern", „Profil ändern" und „Profil speichern". Dies liegt darin begründet, daß *HTML_Kneter* unter anderem für die Durchführung der Änderungen eines Benutzer-Profils zuständig ist. Während dieses Änderungsvorganges werden alle vom Benutzer im HTML-Formular vorgenommenen Änderungen über *Profil_Container* an *Profil_Aktion_FE* weitergereicht (Ereignis „Profil ändern"). *Profil_Aktion_FE* führt die vom Benutzer gewünschten Änderungen aus und meldet dies über *Profil_Container* an *HTML_Kneter* (Ereignis „Profil ändern ok"). Daraufhin initialisiert *HTML_Kneter* die Speicherung des Profils (Ereignis „Profil speichern"). Der Erfolg bzw. Mißerfolg der geforderten Speicherung wird von *Profil_Aktion_BE* zurückgemeldet (Ereignisse „Profil speichern ok" bzw. „Profil speichern nicht ok").

Aus den oben geschilderten Vorgängen wird ersichtlich, daß die Verwaltung bzw. Manipulation der Profil-Inhalte zur Laufzeit ausschließlich von den beiden Aktions-Klassen *Profil_Aktion_BE* und *Profil_Aktion_FE* vorgenommen wird. *Profil_Aktion_FE* ist dabei für die Änderung und das Liefern von Werten einer Instanz der Klasse *Profil* zuständig. *Profil_Aktion_BE* übernimmt alle Aktionen, die in irgendeiner Form einen Kontakt des Publikationssystems mit dem ODBMS erfordern. Diese schon in der Problemanalyse geforderte Trennung von Datenbank- und Generierungs-Funktionalitäten zieht sich durch die gesamte dynamische Modellierung der verschiedenen Container-Klassen und deren zugeordneten Aktions-Klassen.

Änderung von Profil-Einstellungen
Möchte ein Benutzer Änderungen an seinen Profil-Einstellungen vornehmen, muß das Publikationssystem eine entsprechende Oberfläche zur Verfügung stellen. Zur Generierung dieser Oberfläche wird erneut die Klasse *HTML_Kneter* verwendet. Anders als bei der Generierung eines Dokuments mit Beitragsinformationen werden hierfür keine

[1] Das Ereignis „Wert "..." anfordern" repräsentiert alle Ereignisse, die der Anforderung eines Attribut-Werts der Klasse Profil entsprechen (zum Beispiel „Wert "benutzername" anfordern"). Dementsprechend sind weitaus mehr als sechs Ereignisse von *HTML_Kneter* an *Profil_Container* möglich. Dieser Sachverhalt wird hier aus Gründen der Übersichtlichkeit vernachlässigt. Diese Vernachlässigung gilt für alle Ereignisse des Ereignisflußdiagramms in dem die Zeichen "..." verwendet werden.

Informationen aus der Beitrag-Datenbank benötigt. Die Generierung eines solchen Profil-Dokuments wird in Abbildung 44 aus Sicht der Klasse HTML_Kneter dargestellt.

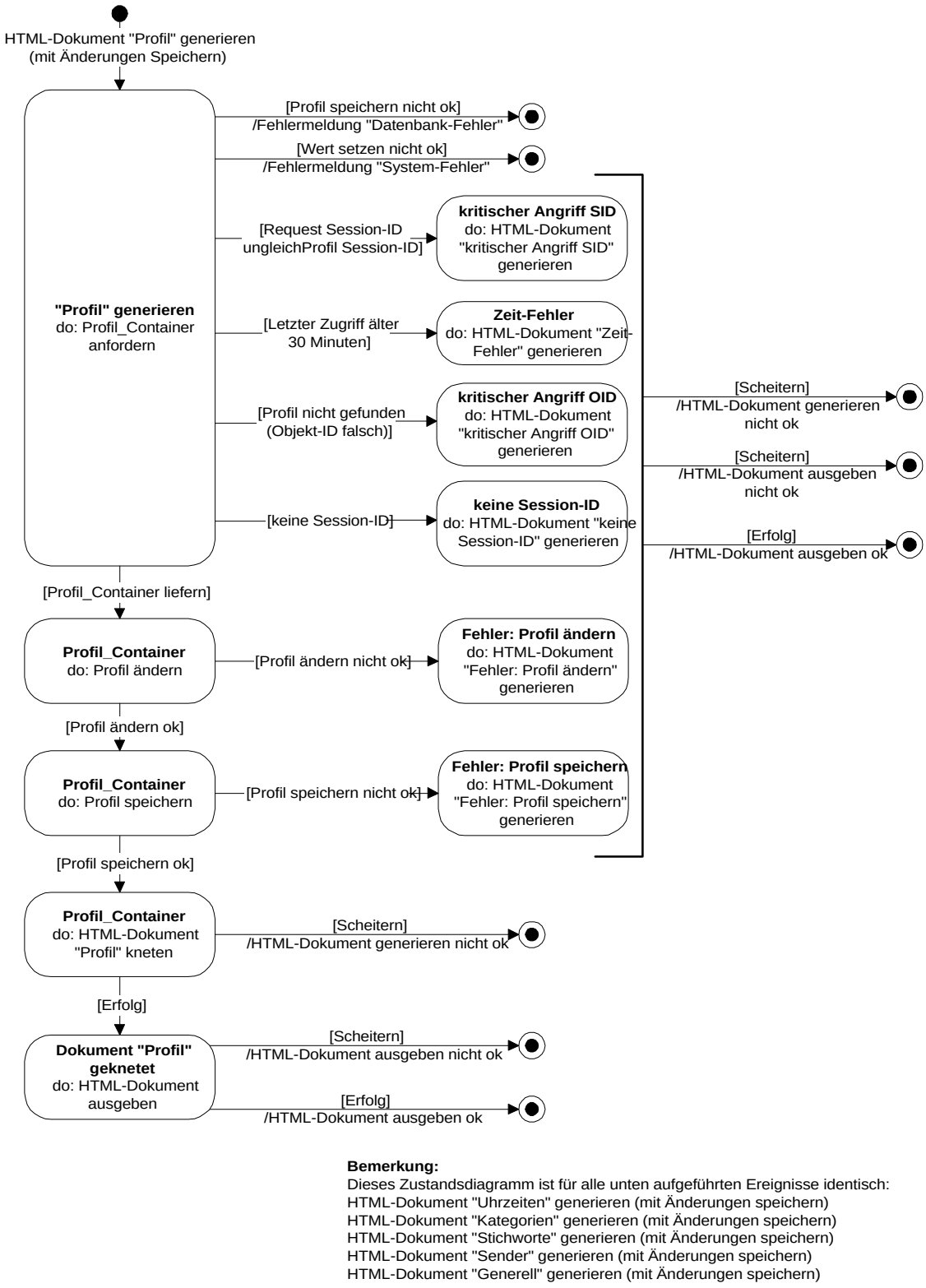


Abbildung 44: Zustandsdiagramm HTML_Kneter → Profil Änderungen speichern

Zunächst greifen die identischen Zugriffs-Kontrollmechanismen wie bei der Generierung eines normalen Abonnenten-Dokuments. Anschließend werden die vom Benutzer im HTML-Formular vorgenommenen Einstellungen in das Benutzer-Profil übernommen (Aktion „Profil ändern“). Nach der anschließenden Speicherung des Profils wird ein Antwort-Dokument erzeugt. Innerhalb der Anfragedaten wurde spezifiziert, welche Vorlage für die Generierung dieses Antwort-Dokuments verwendet werden soll. Grundsätzlich können nur sogenannte Profil-Dokumente verwendet werden. Also Vorlagen, die lediglich Profil-Informationen darstellen. Zusätzlich können auf diesen Seiten Werbe- und Betreiber-Informationen verwendet werden. Beitrags-Informationen werden nicht dargestellt .

Dieses Kapitel beleuchtet nur einen Ausschnitt aus dem kompletten OMT-Modell von PEP. Das gesamte Modell wird im Ergänzungsband zu dieser Diplomarbeit „Konzeption und Ansätze der Realisierung eines digitalen Publikationssystems für personalisierte TV-Programminformationen - Band 2: OMT-Modell“ dargestellt.

5.4.3 Entwurf von Algorithmen

Jede durch das funktionale Modell erhaltene Operation muß laut OMT-Methodologie als Algorithmus formuliert werden. Die Algorithmen für die Operationen aus dem dynamischen Modell werden bereits durch die entsprechenden Zustandsdiagramme definiert. Da für PEP kein funktionales Modell erstellt werden mußte (vgl. Kapitel 5.2.3) und die relevanten Algorithmen bereits im dynamischen Modell beschrieben sind, war ein weiterer Entwurf von Algorithmen nicht notwendig. Diese Feststellung wurde auch während der späteren Implementierung bestätigt. Die Zustandsdiagramme des dynamischen Modells konnten leicht in Programm-Code überführt werden.

6 Implementation

Unter Berücksichtigung der in Kapitel 4 getroffenen Entscheidungen wurde mit der Implementierung des Publikationssystems begonnen. Als Entwicklungsplattform stand eine SUN Ultra Sparc II mit der Betriebssystemversion Solaris 2.6 zur Verfügung. Als Compiler wurde der SUN C++ Compiler in der Version 4.2 verwendet. Das ODBMS POET in der Version 5.0 für Solaris 2.6 wurde als Datenbank Management System genutzt.

Zunächst wurde das im Kapitel 5.4.1 beschriebene Objektmodell in C++ Source-Code überführt. Das verwendete Modellierungswerkzeug Object Domain bot dazu über Generierungsfunktionalitäten die Möglichkeit, die gesamte Klassenstruktur mit allen Attributen und Methoden in Form von C++ Header-Dateien zu exportieren. Zusätzlich konnten mit Object Domain C++ Source-Dateien generiert werden, die bereits den Rahmen der Methoden der einzelnen Klassen enthielten. Diese Kombination aus C++ Header- und Source-Dateien stellte das grobe Gerüst für die Implementierung des Systems PEP dar.

Der erste Schritt für die Implementierung war das Aufsetzen der Datenbank und Schaffung der Datenbank-Strukturen. Im Kapitel 6.1 wird das Datenbankmodell anhand des persistenten Anteils des Objektmodells erörtert. Darauf aufbauend wird im folgenden Kapitel 6.2 dann am Beispiel der Klasse *Datenbank_Server* und *Profil* der Zugriff auf die Datenbank erläutert. Anschließend wird die Generierung eines Dokuments (Kapitel 6.3) betrachtet.

6.1 Datenbankmodell

Da für die Realisierung des Systems PEP ein objektorientiertes Datenbanksystem verwendet wird, entfällt die aufwendige Abbildung der Datenstrukturen in ein relationales Datenbankmodell. Es ist lediglich notwendig, den persistenten Teil des Objektmodells zu identifizieren und anschließend über die API des verwendeten ODBMS als persistent zu deklarieren.

Der persistente Teil des Objektmodells von PEP wird von den in Abbildung 45 dargestellten Klassen gebildet. Die Pfeile verdeutlichen logische Abhängigkeiten zwischen den einzelnen Klassen, die sich aus den im Objektmodell dargestellten Aggregationen ergeben. So werden z.B. die Klassen *Charakter*, *Sparte*, *Sender* und *Fernbedienung* von der Klasse *Profil* aggregiert (vgl. Abbildung 31, Seite 80).

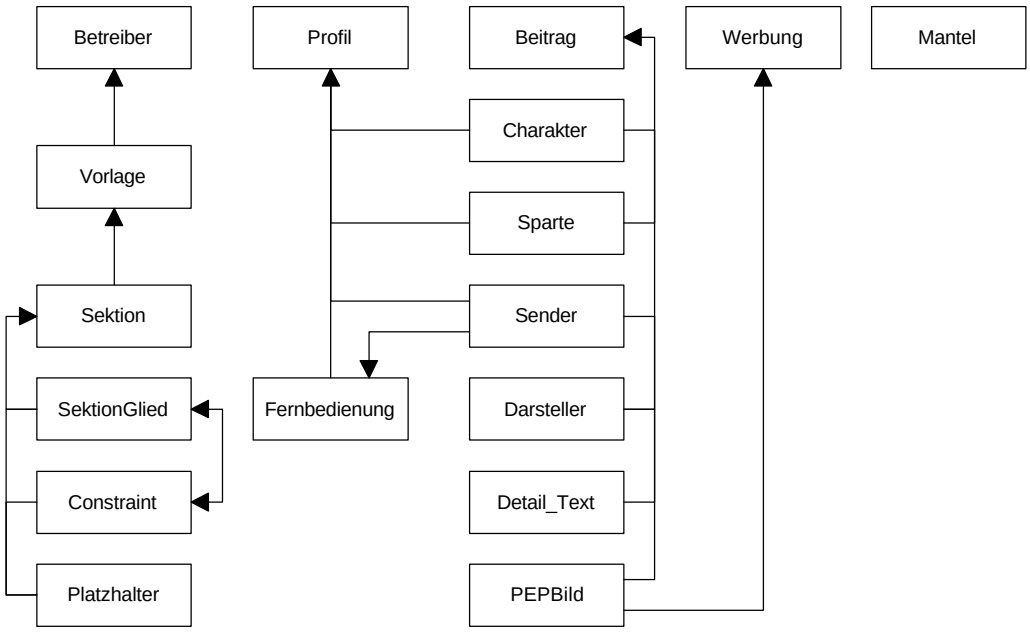


Abbildung 45: Persistente Klassen von PEP

Persistenz ist kein Bestandteil des C++-Standards. Die verschiedenen Anbieter objektorientierter Datenbanken schlagen aus diesem Grund unterschiedliche Wege bei der Umsetzung von Persistenz ein. Object Store überlädt beispielsweise den in C++ verwendeten „new“-Operator, der innerhalb des C++-Standards für die Anforderung von Arbeitsspeicher vorgesehen ist. Durch die Übergabe verschiedener Parameter an den überladenen „new“-Operator wird ein Objekt als persistent deklariert. Innerhalb der Applikation kann ein solches persistentes Objekt wie eine Instanz eines transienten (flüchtigen) Objektes verwendet werden. Auf diese Weise kann innerhalb der Applikation von der datenbankspezifischen Funktionalität abstrahiert werden. Durch die im folgenden Beispiel dargestellte Verwendung des überladenen „new“-Operators wird hier die Persistenz für das Objekt „sender“ erreicht.

Beispiel Object Store:

```
os_typespec *Sender_type = new os_typespec("Sender");
Sender      *sender = new(data_base, Sender_type) Sender("ARD");
```

Die POET-API verwendet dagegen Schlüsselworte[1] um eine Klasse als persistent zu deklarieren. Durch das Voranstellen des Schlüsselwortes „persistent“ vor eine Klassendeklaration, wird zum Beispiel allen Instanzen dieser Klasse die Möglichkeit gegeben persistent in der POET-Datenbank gespeichert zu werden. Da Header-Dateien, die Deklarationen persistenter Klassen beinhalten, den Suffix „HCD“ besitzen müssen (zum Beispiel „profil.hcd“), wird im folgenden von HCD-Dateien gesprochen. Das folgende Beispiel zeigt nun einen Auszug aus einer HCD-Datei, in der die Klasse *Sender* als persistent deklariert wird.

[1] Die POET-API stellt insgesamt 11 Schlüsselworte zur Verfügung, siehe [POET97].

Beispiel POET:

```
persistent class Sender
{
     PEPText   *sender_name;
     int       sortier_reihenfolge;
};
```

Da der SUN C++ Compiler die persistente Klassendeklaration nicht versteht, wird mittels des Precompilers PTXX[1] Standard C++ Code generiert. Dieser sogenannte Vanilla C++ Code kann vom C++ Compiler verarbeitet werden. Außerdem generiert der Precompiler sogenannte „class factory“ Dateien, die Routinen enthalten, die von der Applikation zur Erstellung und Pflege der persistenten Objekte in der Datenbank genutzt werden können. Weiterhin wird vom Precompiler das Datenbank-Schema erzeugt, das sich aus den persistenten Klassendeklarationen der HCD-Dateien ergibt. Abbildung 46 zeigt die bei einem Durchlauf des Precompilers entstehenden Dateien am Beispiel der in der Betriebs-Datenbank von PEP gespeicherten Klassen *Betreiber*, *Profil*, *Werbung* und *Mantel*.

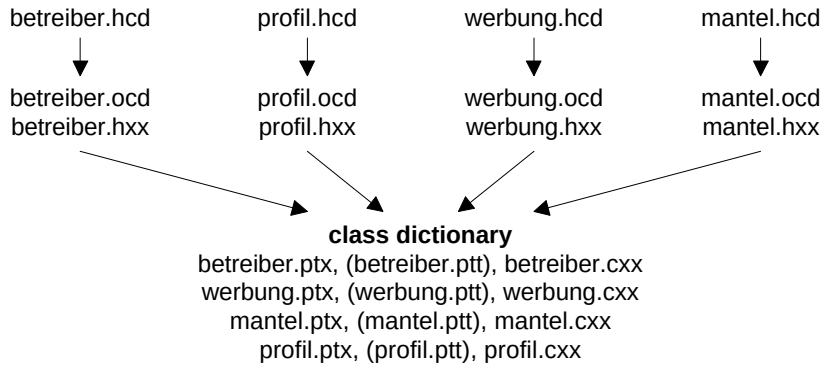


Abbildung 46: Kompilierung und Registrierung des Schemas der Betriebs-Datenbank

Die Deklarationen der oben genannten Klassen finden sich in den HCD-Dateien, die Standard C++ Code und die POET Schlüsselworte enthalten. Zu jeder dieser HCD-Dateien erstellt der Precompiler zunächst jeweils eine HXX-Datei und eine OCD-Datei. Die HXX-Dateien enthalten die Standard C++ Repräsentation der Deklarationen in den HCD-Dateien. Das heißt sämtliche POET-Schlüsselworte sind durch Standard C++ Code ersetzt worden. Die HXX-Dateien (zum Beispiel „profil.hxx“) ersetzen die standardmäßig „includierten“ Header-Dateien (zum Beispiel „profil.h“). Die Gesamtheit aller OCD-Dateien repräsentiert das Datenbank-Schema, anhand dessen das Programm PTXXREG[1] später die eigentliche Datenbank erstellt.

Anschließend erstellt der Precompiler CXX- und PTX-Dateien, die als „class factory“ Dateien bezeichnet werden. Die „class factory“ Dateien beinhalten den datenbankspezifischen Code, der benötigt wird, um auf die persistenten Klassen zugreifen zu können. Die CXX- und PTX-Dateien müssen separat kompiliert und anschließend an das Programm „gelinkt“ werden. Wenn in persistenten Klassen C++ Templates verwendet

[1] Die Programme PTXX und PTXXREG sind Bestandteil des POET C++ Software Development Kit (SDK), das für die Implementierung von Applikationen, die das POET ODBMS nutzen, benötigt wird.

werden, so werden vom Precompiler außerdem noch PTT-Dateien generiert. Diese Dateien enthalten den benötigten Code für die Unterstützung von Templates innerhalb persistenter Klassen.

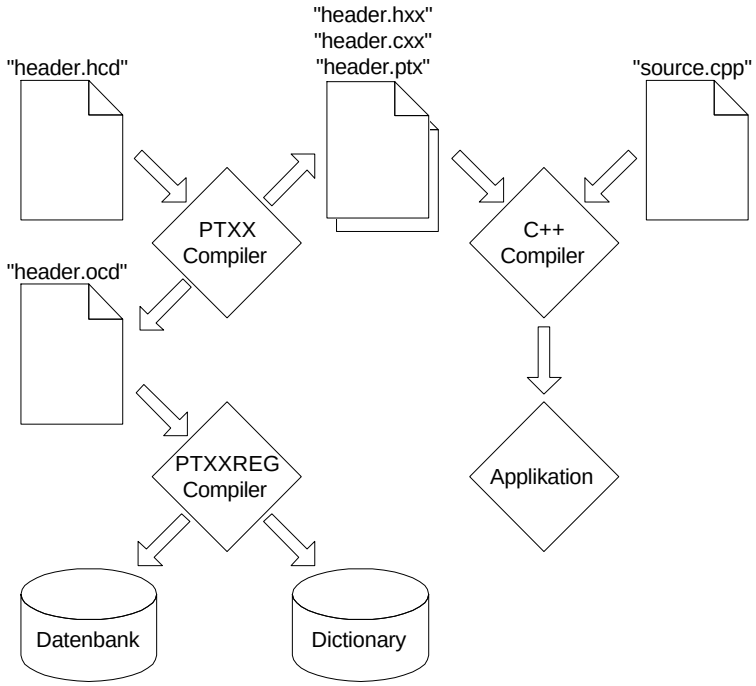


Abbildung 47: Kompilierung und Registrierung einer POET Datenbank

Mit Hilfe des Datenbank-Schemas (OCD-Dateien) erstellt das Programm PTXXREG die eigentlichen Datenbank-Dateien und das Datenbank-Dictionary. Das Dictionary enthält Informationen über die Struktur der persistenten Klassen (Vererbungen, Attribute etc.), die zur Laufzeit ausgewertet werden, um auf Objekte der Datenbank zugreifen zu können. Das POET Dictionary speichert dabei das DB-Schema unabhängig von Applikationen, Betriebssystemen, verwendeter Datenbank-Technik und Programmiersprachen. Die Kombination aus Dictionary und Datenbank-Dateien stellt die eigentliche Datenbank dar. Abbildung 48 zeigt die Struktur der PEP-Datenbank mit Betriebs-Datenbank und Beitrags-Datenbank.

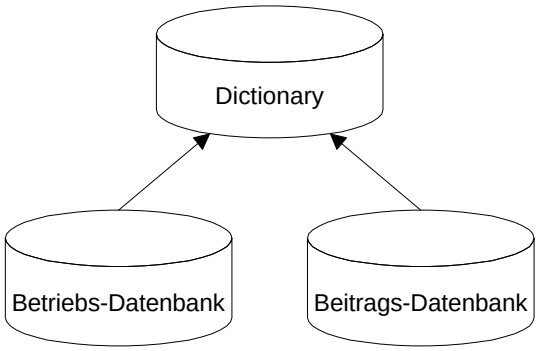


Abbildung 48: Datenbanken mit gemeinsamen Dictionary

Die in Abbildung 45 dargestellten Aggregationen zwischen den einzelnen Klassen, lassen sich zwischen verschiedenen Datenbanken über sogenannte „cross database references“ realisieren. So stellt das POET ODBMS Funktionalitäten zur Verfügung, mit denen Objekte der Beitrags-Datenbank (zum Beispiel *Sender*-Objekte) auch für Objekte der Betriebs-Datenbank (zum Beispiel *Profil*-Objekte) zur Verfügung gestellt werden können.

6.2 Datenbankzugriff

In diesem Abschnitt werden verschiedene Methoden der Klasse *Datenbank*[1] zum Öffnen und Schließen einer Datenbank sowie zum Speichern und Laden persistenter Objekte beschrieben. Außerdem wird das Transaktionsmodell des POET-ODBMS erläutert und in die Klasse *Datenbank* integriert. Diese Klasse dient als Datenbank-Abstraktions-Schicht für die Kommunikation zwischen dem POET ODBMS und den Back-End-Aktionen von PEP.

Öffnen und Schließen der Datenbank
Damit das System PEP persistente Objekte aus der Beitrags- und Betriebs-Datenbank lesen bzw. in diese Datenbanken schreiben kann, muß zuvor die POET-API initialisiert und die Datenbanken müssen geöffnet werden. Beides geschieht beim Start des Back-Ends. Dieses baut zuerst eine Verbindung des Systems zu der von allen Betreibern benötigten Beitrags-Datenbank und im Anschluß zu den Betreiber-Datenbanken auf. Das Initialisieren der Beitrags-Datenbank erfolgt dabei über den Konstruktor der Klasse *Datenbank*. Beim Start des Back-Ends wird eine Instanz dieser Klasse mit den Parametern **_s_dbhost** und **_s_dbname** initialisiert. Der Konstruktor der Klasse *Datenbank* ruft die Methode ***OeffneDB*** mit denselben Parametern auf.

```
1. DatenBank::DatenBank(const char* const _s_dbhost, const char* const
                        _s_dbname)
2. {
3.       OeffneDB(_s_dbhost, _s_dbname);
4. }
```

Die Methode ***OeffneDB*** stellt dann die Verbindung zu der Beitrags-Datenbank her. Zuerst prüft ***OeffneDB***, ob ein Host und ein Name der zu öffnenden Datenbank übergeben wurde. Ist dies nicht der Fall, so wird eine Datenbank-Ausnahme geworfen. Da der Zugriff auf die Beitrags-Datenbank für den Betrieb des Systems von existentieller Bedeutung ist, führt das Werfen dieser Datenbank-Ausnahme zum Abruch des Systemstarts.

```
1.   long DatenBank::OeffneDB(const char *_s_dbhost,
                              const char *_s_dbname) throw(DBAusnahme)
2.   {
3.       if(!(_s_dbhost && _s_dbname))
4.       throw DBAusnahme(SYS_FEHLER_KEINE_DATENBANK,
                          TXT_FEHLER_KEINE_DATENBANK,
                          LOG_FEHLER_KEINE_DATENBANK);
5.
```

[1] Die im Objektmodell und im Data-Dictionary beschriebe Klasse *Datenbank_Server* wurde innerhalb der Implementierung in *Datenbank* umbenannt, so daß in diesem Kapitel nur von der Klasse *Datenbank* gesprochen wird.

```
6.       err_no = InitPOET(PtTransactionByThread);
7.
8.       err_no = PtBase::POET()->GetBase(_s_dbhost, _s_dbname,
                                          beitrags_datenbank);
9.
10.      if(err_no < 0)
11.      {
12.      PtBase::POET()->UngetBase(beitrags_datenbank);
13.      DeinitPOET();
14.
15.      throw DBAusnahme(SYS_FEHLER_OEFFNE_DATENBANK,
                          TXT_FEHLER_OEFFNE_DATENBANK,
                          LOG_FEHLER_OEFFNE_DATENBANK);
16.      }
17.
18.      return err_no;
19.  }
```

Wurde an ***OeffneDB*** der Host und der Name der zu öffnenden Datenbank übergeben, so muß vor dem eigentlichen Öffnen der Datenbank zuerst die globale Funktion InitPOET() der POET API aufgerufen werden. Die Funktion InitPOET() initialisiert die POET API für den Zugriff auf das POET ODBMS, wobei als Parameter das von der Applikation gewünschte Transaktionsmodell angegeben werden muß.[1] Zudem erzeugt InitPOET() ein PtRoot Objekt, daß für die Ressourcen-Verwaltung sogenannter PtBase Objekte zuständig ist.[2] Jedes PtBase Objekt repräsentiert dabei genau eine Datenbank und stellt für die Applikation unter anderem Methoden zum Öffnen und Schließen der Datenbank zur Verfügung. Im Anschluß daran kann die Methode GetBase() aufgerufen werden, die eine Verbindung zum Datenbank-Server (Parameter: **_s_dbhost**) herstellt und die Beitrags-Datenbank (Parameter: **_s_dbname**) öffnet. Die Methode GetBase() initialisiert daraufhin eine Instanz der Klasse PtBase, die die geöffnete Datenbank repräsentiert. Die Speicheradresse dieser Instanz wird im dritten übergebenen Parameter **beitrags_datenbank** (Attribut der Klasse *Datenbank*) gespeichert.

Mißlingt der Verbindungsaufbau zur Datenbank, so muß zuerst über die Methode UngetBase() die Datenbank geschlossen und dann über DeinitPOET() die POET API für den Zugriff auf das POET ODBMS deinitialisiert werden. Weiterhin wird eine Datenbank-Ausnahme geworfen, die zum Abruch des Systemstarts führt. Über die Methoden UngetBase() und DeinitPOET() wird eine geöffnete Datenbank auch im „Normalfall“, zum Beispiel bei der Systemterminierung, geschlossen.

Nachdem die Datenbank erfolgreich geöffnet wurde, kann über das Attribut **beitrags_datenbank** der Klasse *Datenbank* auf die Objekte der Beitrags-Datenbank zugegriffen werden.

---

[1] POET unterstützt die drei Transaktionsmodelle PtTransactionByBase, PtTransactionByThread, PtTransactionByObject, siehe [POET97]. Für die „multithreaded“ Applikation PEP wird PtTransactionByThread genutzt.

[2] PtRoot verwaltet unter anderem eine Liste aller von der Applikation geöffneten Datenbanken (repräsentiert durch PtBase Objekte) und stellt sicher, daß jede Datenbank von der Applikation höchstens einmal geöffnet wird.

Das POET ODBMS erlaubt es, mehrere PtBase Objekte gleichzeitig geöffnet zu haben, so daß mehrere Datenbanken von einer Applikation genutzt werden können. Die Verwaltung dieser PtBase Objekte übernimmt das über InitPOET() erzeugte PtRoot Objekt. Zum Öffnen der einzelnen vom System PEP benötigten Betreiber-Datenbanken (Betriebs-Datenbanken) wird die Methode ***OeffneDB*** überladen. Diese zweite Implementation erhält zur Authentifikation des Systems gegenüber dem Datenbank-Server zusätzlich zu den Parametern **_s_dbname** und **_s_dbhost** die zwei Parameter **_s_benutzer_name** und **_s_passwort** übergeben.

Transaktionen
POET unterscheidet generell zwei verschiedene Arten von Transaktionen. Für die datenbankinterne Konsistenzsicherung werden sogenannte System-Level-Transaktionen verwendet. Die Mechanismen dieser System-Level-Transaktionen sind für Applikationen transparent. Weiterhin bietet POET Transaktionen, die aus einer Applikation heraus genutzt werden können. Bei diesen Transaktionen unterscheidet POET folgende drei Modelle. Eines dieser Modelle muß zwingend verwendet werden und wird bei der Initialisierung der POET API der Methode InitPOET() als Parameter übergeben:

- **PtTransactionByBase**
  PtTransactionByBase wird für Applikationen verwendet, die nicht explizit Transaktionen verwenden. Es existiert genau eine Transaktion pro Datenbank.

- **PtTransactionByThread**
  PtTransactionByThread ist für Applikationen, die Transaktionen verwenden und für Applikationen die mehrere Threads und/oder mehrere Transaktionen pro Datenbank verwenden. Außerdem wird dieses Modell für „multithreaded“ Applikationen verwendet, die ein Locking der persistenten Objekte zwischen den einzelnen Threads benötigen.

- **PtTransactionByObject**
  PtTransactionByObject wird für Applikationen verwendet, die mehrere Transaktionen pro Thread verwenden.

Für jede Transaktion existiert ein eigener Transaktionspuffer, der Änderungen an persistenten Objekten puffert (vgl. Abbildung 49). Bei einer von der Applikation ausgeführten Speicherung (Store) eines oder mehrerer Objekte, wird der Speichervorgang zunächst nur innerhalb des Transaktionspuffers durchgeführt. Die Daten werden erst beim Festschreiben der Transaktion (Commit) vollständig in der Datenbank gespeichert. Bei einem Abbruch (Abort) werden die gepufferten Änderungen nicht in die Datenbank übernommen. Nach einem Abbruch einer Transaktion kann über ein Rücksetzen (Refresh) der ursprüngliche Zustand der Objekte im Speicher der Applikation wiederhergestellt werden. Diese Vorgehensweise gilt für alle drei Transaktionsmodelle.

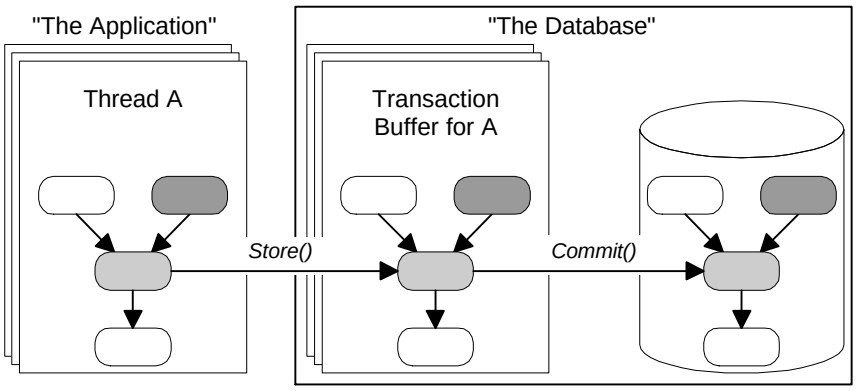


Abbildung 49: POET Transaktions-Puffer [POET97]

Die Unterschiede zwischen den drei Transaktions-Modellen sind im sogenannten Transaktions-Kontext zu finden. Für das Modell PtTransactionByBase ist der Transaktions-Kontext stets die gesamte Datenbank und die gesamte Applikation. Bei Verwendung des Modells PtTransactionByThread wird eine Transaktion immer an einen Thread und an eine Datenbank gebunden. Werden mehrere Transaktionen innerhalb eines Threads verwendet, muß vor Änderungen an persistenten Objekten die entsprechende Transaktion gewählt werden. Der Transaktions-Kontext wird somit durch einen Thread, eine Datenbank und eine Transaktion bestimmt.

Beim Modell PtTransactionByObject dagegen wird, abweichend von Abbildung 49, eine Transaktion an genau ein Objekt und an eine Datenbank gebunden. Bei Änderungen an einem persistenten Objekt wird die an ein Objekt gebundene Transaktion automatisch gewählt. Der Transaktions-Kontext wird demzufolge von einem Objekt, einer Datenbank und einer Transaktion bestimmt. Bei der Verwendung dieses Modells wird somit die komplette Ausführung einer Transaktionen nur für **ein einziges** Objekt sichergestellt. Die Applikation ist dafür verantwortlich Änderungen eines anderen Threads an referenzierten Objekten unabhängig von der Transaktion zu berücksichtigen oder gegebenenfalls über ein Locking der referenzierten Objekte zu verhindern. Bei Nutzung dieses Modells kann POET die Datenkonsistenz nicht sicherstellen. Die POET Dokumentation empfiehlt aus diesem Grund die Verwendung des Modells PtTransactionByThread, wenn die Applikation immer den aktuellen Transaktions-Kontext kennt.

Da das System PEP eine „multithreaded“ Applikation ist und weil der Transaktions-Kontext zu jedem Zeitpunkt bekannt ist, wird für den Datenbank-Zugriff das Modell PtTransactionByThread verwendet. Damit von PEP Transaktionen verwendet werden können, wurden die drei *Datenbank*-Methoden ***transaktion_starten***, ***transaktion_beenden*** und ***transaktion_abbrechen*** eingeführt. Um eine Transaktion zu starten, wird die Methode ***transaktion_starten*** mit dem Namen der Datenbank aufgerufen an die die Transaktion gebunden werden soll (Parameter **_s_dbname**).

```
1.  Transaktion* DatenBank::transaktion_starten(const char *_s_dbname)
                                                    throw(DBAusnahme)
2.  {
3.       PtBase       *__p_datenbank;
4.       Transaktion  *__op_transaktion;
5.
6.       try {
7.            __op_transaktion = new Transaktion;
8.       } catch(DBAusnahme &a) {
9.            if(__op_transaktion)
10.                delete __op_transaktion;
11.
12.           throw;
13.      }
14.
15.      if(!(__p_datenbank = gib_DB(_s_dbname)) {
16.           delete __op_transaktion;
17.           return (Transaktion *) NULL;
18.      }
19.
20.      (__op_transaktion->current)->SetRefreshOnAbort(PtTRUE);
21.      (__op_transaktion->current)->RegisterResource(__p_datenbank);
22.
23.      Transaktion->previous = __p_datenbank->SetCurrentTransaction
                                (__op_transaktion->current);
24.
25.      if(!((__op_transaktion->current)->Begin())) {
26.           __p_datenbank->SetCurrentTransaction
             (__op_transaktion->previous);
27.
28.           delete __op_transaktion;
29.           return (PtTransaction*) NULL;
30.      }
31.
32.      return __op_transaktion;
33. }
```

Um eine Transaktion beginnen zu können, muß zunächst eine Instanz der Klasse PtTransaction initialisiert und anschließend als aktuelle Transaktion festgelegt werden (SetCurrentTransaction()). Mit der Methode SetCurrentTransaction() wird ein neuer Transaktions-Kontext hergestellt. Damit nach Beendigung der jetzt aktuellen Transaktion, der vor der Ausführung der Methode SetCurrentTransaction() gültige Transaktions-Kontext wiederhergestellt werden kann, wurde die Klasse *Transaktion* eingeführt. Diese Klasse dient dabei als Hülle für die Zwischenspeicherung zweier PtTransaction-Objekte. Diese Objekte werden in den öffentlichen (public) Attributen **current** und **previous** der Klasse *Transaktion* gespeichert und repräsentieren die aktuelle und vorherige Transaktion. Im Konstruktor der Klasse *Transaktion* wird das Attribut **current** mit einer neuen Instanz der Klasse PtTransaction gefüllt. Der Destruktor der Klasse *Transaktion* löscht diese Instanz wieder.

Um den neuen Transaktions-Kontext herstellen zu können, muß zuerst anhand des übergebenen Parameters **_s_dbname** die Datenbank ermittelt werden, an die die neue Transaktion gebunden werden soll (Zeile 15: ***gib_DB***(**_s_dbname**)). Anschließend wird über die Methode SetRefreshOnAbort() für die neue Transaktion festgelegt, daß bei einem Abbruch derselben ein Rücksetzen (Refresh) erfolgen soll (Zeile 20). Mit der Methode RegisterResource() wird die neue Transaktion an die entsprechende Datenbank gebunden

(Zeile 21). In Zeile 23 wird die derzeit aktuelle Transaktion im Attribut *Transaktion*.**previous** gespeichert und die neue Transaktion (*Transaktion*.**current**) als aktuelle festgelegt. In Zeile 25 wird die neue Transaktion begonnen (Begin()). Kann die Transaktion nicht gestartet werden, so wird in Zeile 26 der alte Transaktions-Kontext wiederhergestellt. Bei einem erfolgreichen Beginn der Transaktion, wird das neue *Transaktion*-Objekt zurückgeliefert, um die Transaktion damit später beenden bzw. abbrechen zu können.

```
1.  long DatenBank::transaktion_beenden(Transaktion *_op_transaktion)
2.  {
3.       long __l_err=0;
4.
5.       __l_err = (_op_transaktion->current)->Commit();
6.       PtBase::POET()->SetCurrentTransaction
                        (__op_transaktion->previous);
7.
8.       delete _op_transaktion;
9.
10.      return __l_err;
11. }
```

Für die Festlegung von innerhalb einer Transaktion vorgenommenen Änderungen an persistenten Objekten wird die Methode ***transaktion_beenden*** verwendet. Diese Methode führt die POET Methode Commit() aus. Ab diesem Commit() übernehmen die System-Level-Transaktion des POET ODBMS die Verantwortung für die vollständige Speicherung aller vorgenommenen Änderungen vom Transaktions-Puffer in die Datenbank. Anschließend wird in Zeile 6 der ursprüngliche Transaktions-Kontext wiederhergestellt (SetCurrentTransaction()). Die hier nicht mehr aufgeführte Methode ***transaktion_abbrechen*** führt anstatt der Methode Commit() in Zeile 5 die Methode Abort() aus, um eine Transaktion abzubrechen. Ansonsten sind die beiden Methoden identisch.

Speichern eines Objekts in der Datenbank

Das Speichern eines Objekts soll anhand der Klasse *Profil* erläutert werden. Der Methode ***speichern*** der Klasse *Datenbank* wird das zu speichernde Objekt und ein Tiefen-Modus für die Speicherung übergeben. Dieser Tiefen-Modus legt fest, ob und welche von dem übergebenen Objekt referenzierten persistenten Objekte mitgespeichert werden sollen (vgl. Abbildung 50). Innerhalb der Methode ***speichern*** wird dieser Tiefen-Modus zusammen mit dem zu speichernden Objekt an die POET Methode Store() übergeben.

```
1.  long DatenBank::speichern(Profil * _o_profil,
                              PEPSpeicherModus _i_speicher_modus)
2.  {
3.       long __l_err = _o_profil->Store(_i_speicher_modus);
4.
5.       return __l_err;
6.  }
```

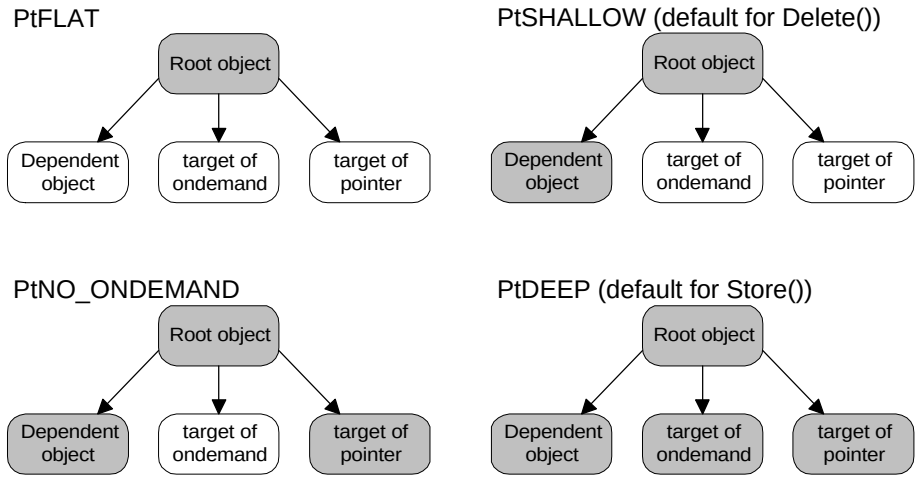


Abbildung 50: Tiefen-Modi für den Zugriff auf POET Objekten [POET97]

Mögliche POET Tiefen-Modi die nicht nur für das Speichern (Store) sondern auch für das Löschen (Delete), Sperren (Lock) und Beobachten (Watch) von Objekten eingesetzt werden:

- **PtFLAT**
  Nur das übergebene Objekt ist betroffen. Dependent Objekte, Ondemand[1] Objekte, und über Zeiger referenzierte Objekte werden nicht berücksichtigt.

- **PtSHALLOW**
  Das übergebene Objekt und Dependent Objekte sind betroffen. Ondemand Objekte und über Zeiger referenzierte Objekte werden nicht berücksichtigt. PtSHALLOW ist der Standard-Wert für die Methode Delete() der POET API.

- **PtNO_ONDEMAND**
  Das übergebene Objekt, Dependent Objekte und über Zeiger referenzierte Objekte sind betroffen. Ondemand Objekte werden nicht berücksichtigt.

- **PtDEEP**
  Das übergebene Objekt und alle referenzierten Objekte werden berücksichtigt. PtDEEP ist der Standard-Wert für die Methode Store() der POET API.

Laden eines Objekts aus der Datenbank
Das POET ODBMS gibt jedem Objekt innerhalb einer Datenbank eine eindeutige Objekt-ID (OID). Beim Speichern eines Objekts werden dazu alle Zeiger auf persistente Objekte in OIDs umgewandelt. Jede Objekt-ID hat eine Länge von 47 Bit und wird genau einmal innerhalb einer Datenbank verwendet. Der schnellste Zugriff auf ein persistentes

[1] Ein Objekt kann, bedingt durch seine Klassendefinition, Objekte beinhalten, die von ihm abhängig sind (Dependent Object), d.h. die nicht ohne das beinhaltende Objekt existieren können. Außerdem kann ein Objekt Referenzen auf andere Objekte beinhalten (target of pointer und target of ondemand). Ondemand Objekte sind referenzierte Objekte, die nur aus der Datenbank geladen werden, wenn sie explizit angefordert werden.

Objekt ist über seine OID möglich. Dies hat vor allem Vorteile für die Generierung der TV-Programmtabelle, in der zum Beispiel die OIDs der *Beitrag*-Objekte in den die Einzelansicht referenzierenden Hyper-Link eingetragen werden können. Klickt ein Benutzer auf einen solchen Hyper-Link, kann das entsprechende *Beitrag*-Objekt direkt über die OID in der Datenbank referenziert werden. Aber auch für das Laden eines Abonnenten-Profils kann so, anhand der in einer Session-ID enthaltenen OID, sehr schnell das entsprechende Profil aus der Datenbank geladen werden.

Für das Suchen eines Objekts bzw. von Objekten in einer Datenbank bietet die POET API mehrere Möglichkeiten an. Angeboten werden POET-Queries, OQL-Queries und Filter-Abfragen. POET hat bis heute nur einen Teil der im ODMG-Standard dokumentierten Merkmale von OQL implementiert. Für die Nutzung von POET-Queries generiert der PTXX-Compiler für jede persistente Klasse eine eigene Query-Klasse, die Methoden für die Suche auf Objekten dieser persistenten Klasse und deren Attributen bereitstellt. Außerdem werden vom PTXX-Compiler für jede persistente Klasse sogenannte Set-Klassen generiert, die die Ergebnisse einer POET-Query aufnehmen bzw. den Suchraum für die Query definieren. Für die Klasse Profil existieren zum Beispiel folgende von PTXX generierte Klassen: ProfilQuery, ProfilSet und ProfilAllSet. ProfilSet und ProfilAllSet können die Ergebnisse einer Query aufnehmen. Zusätzlich dient die Klasse ProfilAllSet der Beschränkung des Suchraums einer Query auf Objekte des Typs Profil. Die Klasse ProfilQuery beinhaltet Methoden für die Suche auf allen Attributen der Klasse Profil, wie zum Beispiel **benutzer_name**. Die entsprechende Methode der Klasse ProfilQuery heißt Set**benutzer_name**. Über diese ProfilQuery-Methoden können Werte definiert werden, die dem Wert eines Attributs eines persistenten Objekts entsprechen müssen, damit dieses Objekt als Teil der Ergebnismenge einer POET-Query geliefert wird. POET-Queries können über Boolsche Werte und Klammerungen zu komplexen Queries kombiniert werden bevor sie an das ODBMS abgesetzt werden.

Da POET-Queries immer das gesamte Suchergebnis zurückliefern, besteht keine Möglichkeit das erste gefundene Objekt zu verarbeiten bevor nicht alle Objekte gefunden wurden. Dies ist für bestimmte Anwendungen nicht brauchbar, so daß POET zusätzlich zu den POET-Queries noch sogenannte Filter eingeführt hat. Diese entsprechen weitestgehend den Funktionalitäten der Queries, es kann jedoch angeben werden, daß z.B. jedes Objekt direkt nach dem Auffinden zurückgeliefert werden soll. Die folgende Tabelle soll die Unterschiede zwischen Queries und Filtern verdeutlichen.

Queries		**Filter**	
Finde Objekt 1 Finde Objekt 2 ... Finde Objekt X	 Verarbeite Objekt 1 Verarbeite Objekt 2 ... Verarbeite Objekt X	Finde Objekt 1 Finde Objekt 2 ... Finde Objekt X	 Verarbeite Objekt 1 Verarbeite Objekt 2 Verarbeite Objekt X

Tabelle 8: Queries versus Filter [POET97]

Im System PEP wird zum Beispiel zur Datenbank-Abfrage eines Benutzerprofils anhand des Benutzernamens ein Filter eingesetzt. Dazu wird die Methode ***gib_profil_set*** der Klasse *Datenbank* mit den Parametern **_s_benutzer_name** und **_s_dbname** aufgerufen.

```
1.  void* DatenBank::gib_profil_set(PEPText _s_benutzer_name,
                                    PEPText _s_dbname) throw(DBAusnahme)
2.  {
3.       ProfilAllSet  *__p_alle_profil;
4.       ProfilQuery   __q_profil_query;
5.       long           __l_err = 0;
6.       PtBase         *__p_betriebs_datenbank;
7.       Profil         *__o_profil;
8.
9.       if(!(__p_betriebs_datenbank = gib_DB(_s_dbname))
10.           return (void*) NULL;
11.
12.      __p_alle_profil = new ProfilAllSet(__p_betriebs_datenbank);
13.
14.      if(!__p_alle_profil)
15.           throw AbbruchAusnahme(SYS_FEHLER_NEW,
                                   TXT_FEHLER_NEW,
                                   LOG_FEHLER_NEW);
16.
17.      __q_profil_query.Setbenutzer_name(_s_benutzer_name);
18.      __l_err = __p_alle_profil->SetFilter(&__q_profil_query);
19.
20.      if(__l_err || !(__p_alle_profil->Get(__o_profil, 0, PtSTART)))
         {
21.           delete __p_alle_profil;
22.           return (void*) NULL;
23.      } else __p_alle_profil->Unget(__o_profil);
24.
25.      return (void*) __p_alle_profil;
26. }
```

Zunächst wird in Zeile 9 über die Methode ***gib_DB*** die entsprechende Betriebs-Datenbank angefordert. Anschließend wird eine Instanz der Klasse ProfilAllSet mit der ermittelten Datenbank initialisiert. In Zeile 17 wird die Methode Setbenutzer_name der Klasse ProfilQuery aufgerufen, um den übergebenen Benutzernamen in die Query zu integrieren. Danach wird die ProfilAllSet-Methode SetFilter mit der erstellten Query aufgerufen. Diese Methode füllt das Objekt __p_alle_profil mit den aus der Datenbank ermittelten Instanzen der Klasse *Profil*. Da ein Benutzername immer genau einem Profil in der Datenbank zugeordnet ist, enthält das Objekt __p_alle_profil immer null oder ein Profil-Objekt. In Zeile 20 wird überprüft, ob das Setzen des Filters einen Fehler (__l_err) hervorgerufen hat oder ob das Objekt __p_alle_profil kein *Profil*-Objekt enthält (`__p_alle_profil->Get()`). Tritt eine der beiden Möglichkeiten ein, wird von ***gib_profil_set*** ein Null-Zeiger zurückgeliefert. Andernfalls wird das mit der Methode Get() geholte Profil-Objekt mit der Methode Unget() wieder freigegeben (Zeile 23). Als Rückgabewert liefert die Methode ***gib_profil_set*** dann das gefüllte ProfilAllSet-Objekt __p_alle_profil.

Über die Methoden Get() und Unget() realisiert die POET-API den Zugriff auf Objekte, die in einem AllSet-Objekt enthalten sind. Diesen beiden Methoden kann als zusätzlicher Parameter eine Instanz der Klasse PtLockSpec übergeben werden, die ein Locking bzw.

Unlocking des angeforderten Objektes in der Datenbank ermöglicht. Ein Locking eines Profil-Objektes ist immer dann nötig, wenn ein Profil angefordert wird, auf dem Änderungen vorgenommen werden sollen. Zusätzlich ist über die weiter oben eingeführten Methoden ***transaktion_starten*** und ***transaktion_beenden*** der gesamte Vorgang „Anforderung → Änderung → Speicherung“ des Profil-Objekts in einer Transaktion zu schachteln.

6.3 Dokumenten-Generierung

Im vorhergehenden Kapitel wurde der Zugriff auf den Datenbankserver beschrieben. Dieses Kapitel befaßt sich mit der Aufbereitung der aus der Datenbank akquirierten Daten. Um den Betreibern des TV-Programminformationssystems möglichst umfangreiche Freiheiten bei der gestalteten Darstellung dieser Daten zu geben, wird der HTML- und PDF-Code der einzelnen Seiten des TV-Programms über Vorlagen gesteuert und erst zur Laufzeit generiert.

Die HTML-Generierung
Für die Generierung eines HTML-Dokuments muß ein Betreiber für jede WWW-Seite (*physikalische Seite*) eine Vorlage (*logische Seite*) erstellen, die dann von der Applikation verwendet wird, um den vollständigen HTML-Code zu generieren. Diese Vorlagen enthalten Informationen über das **Grundlayout** der zu generierenden Seiten sowie **Platzhalter** die auf Datenbankinhalte verweisen. Beim Generieren der Seiten werden der Vorlage die Platzhalter entnommen und durch entsprechende Inhalte aus der Datenbank ersetzt.

Betrachtet man die TV-Programmtabelle von PEP (siehe Abbildung 2, Seite 10), so wird deutlich, daß die Struktur des Grundlayouts relativ einfach ist. Trotzdem ließe sich diese einfache Struktur unter der ausschließlichen Verwendung von Platzhaltern innerhalb von regulärem HTML-Code nicht generieren. Das Problem liegt einerseits in der variablen Breite der einzelnen Tabellenzellen mit Beitragsinformationen, andererseits an der variablen Anzahl der bei der Generierung zu berücksichtigenden Sender. Die einzelnen Zellenbreiten zum Beispiel richten sich nach der zeitlichen Länge der Beiträge, die erst zum Generierungszeitpunkt bekannt sind. Das Grundlayout muß folglich mehr als reinen HTML-Code beinhalten, um diesem Sachverhalt Rechnung zu tragen. Zur Veranschaulichung dieses Problems folgt eine nähere Betrachtung der Struktur der TV-Programmtabelle.

Der HTML-Code beginnt mit einem recht umfangreichen Kopf, der diverse Navigations- und Gestaltungselemente enthält und einige allgemeine Informationen sowie einer Tabelle, die die eigentlichen Programminformationen beinhaltet. Nach der Tabelle folgt ein zum Kopf analoger Fuß. Der Aufbau der einzelnen Zeilen der TV-Programmtabelle läßt sich wiederum folgendermaßen zusammenfassen. In der ersten Spalte findet sich der Name eines TV-Senders. Es folgen mehrere Tabellenspalten mit jeweils einem Hyperlink auf zusätzliche Informationen zu einem Sendungsbeitrag. Abgeschlossen wird die Zeile wiederum durch eine Spalte mit dem Namen eines Senders.

Zunächst fällt auf, daß der Inhalt der ersten und der letzten Spalte einer Zeile identisch ist. Sie enthalten lediglich den Namen eines Senders. Weniger offensichtlich aber um so bedeutender ist der Umstand, daß vor der eigentlichen Generierung über die Anzahl und Breite der Spalten mit den Beitragsinformationen keine Aussage getroffen werden kann, da die exakte Anzahl der Beiträge pro Zeile erst zur Generierung bekannt wird. Folgende Überlegung soll helfen dieses Problem zu lösen.

Zunächst wird festgelegt, daß ein Beitrag eine Mindestdauer besitzen muß. **Y** sei die Mindestdauer eines Beitrags in Minuten. **X** sei der von der gesamten Zeile überstrichene Zeitraum in Minuten. Ferner sei es möglich, daß von einem Sender während des gesamten darzustellenden Zeitraums **kein** Beitrag gesendet wird. Die Anzahl der Zellen **Z** für diese Tabellenzeile wäre in diesem Fall null. Die Anzahl **Z** der minimal und maximal in einer Zeile enthaltenen Spalten errechnet sich dann nach der folgenden einfachen Relation.

$$\mathbf{0 \leq Z \leq \frac{X\,min.}{Y\,min.}}$$

Gleichung 1: Anzahl der Tabellenzellen

Für die Anzeige aller möglichen Beiträge einer Zeile müssen somit, beispielsweise bei einem Zeitfenster von *X=135* (2 Stunden, 15 Minuten) und einer Mindestlänge *Y=5*, maximal 27 Spalten zur Verfügung stehen (135 Min. / 5 Min = 27). Da es sich bei obiger Formel um eine Relation handelt, kann über die exakte Anzahl der Spalten bei der Erstellung des Grundlayouts keine eindeutige Aussage getroffen werden. Es muß bei der Erstellung vielmehr die maximale Anzahl von Spalten berücksichtigt werden. Unter Zugrundelegung des HTML-Standards ließe sich dieses Problem relativ einfach lösen. Im HTML-Standard existiert die Möglichkeit, einer Tabelle eine beliebige Anzahl von Spalten zuzuweisen. Einzelne Tabellenzellen können sich dabei mit Hilfe des HTML-Tags *colspan* über mehrere Spalten erstrecken. Legt man für eine Tabellenzelle *colspan=5* fest, so nimmt sie den Platz von insgesamt 5 Spalten ein. Die Idee ist, innerhalb des Grundlayouts grundsätzlich die maximale Anzahl der Tabellspalten zur Verfügung zu stellen (*logische Zellen*). Die Anzahl der sichtbaren (*physikalischen*) Zellen einer Zeile läßt sich dann über den *colspan* Tag steuern. Da in der Tabelle jede logische Zelle die Mindestlänge von Y Minuten überstreicht, beinhaltet die im obigen Beispiel erwähnte physikalische Zelle mit *colspan=5* also einen Zeitraum von 5 x Y = 25 Minuten. Voraussetzung für diesen Lösungsweg ist allerdings, daß Y ein Teiler von X ist, da innerhalb des *colspan*-Eintrages nur naürliche Zahlen verwendet werden dürfen. Auf diese Weise könnte jede Ausprägung von Zellenbreiten und Zellenverteilung in Vorlagen berücksichtigt werden.

Betrachtet man die Anzahl der Permutationen der möglichen Ausprägungen für Zellenbreite und -verteilung wird jedoch schnell deutlich, daß es extrem aufwendig wäre, für jede mögliche Ausprägung der einzelnen Zellenbreiten und –verteilungen eine dedizierte Vorlage zu produzieren und zu verwalten. Für die Bestimmung der Permutationen sind die Grenzen zwischen den Zellen zu betrachten. Abhängig davon ob die Grenze zwischen zwei Zellen von einem *colspan*-Eintrag überschrieben wird gilt, daß die Grenze innerhalb der Zeile erscheint oder nicht. Daher ergibt sich die Anzahl der Permutationen der Ausprägungen einer Tabellenzeile $\mathbf{P}_Z$ durch $\mathbf{P}_Z = \mathbf{2^{Z-1}}$.

Für das obige Beispiel wäre $\mathbf{P}_Z = 2^{26} = 67.108.864$.[1] Jede Änderung des Layouts bzw. das Hinzufügen eines Senders würde **alle** Vorlagen betreffen. Es mußte also ein Weg gefunden werden die dynamisch ermittelten Ausprägungen der Zellenbreiten und Anzahl der Sender im Grundlayout zu berücksichtigen. Zu diesem Zweck wurden Vorlagen in Abschnitte (Sektionen) eingeteilt, die in beliebiger Reihenfolge innerhalb der Vorlage plaziert werden können. Innerhalb der Sektionen stehen Platzhalter zur Verfügung, die durch aktuelle Werte aus der Datenbank ersetzt werden. Der Vorteil dieser Aufteilung besteht in der möglichen Verwendung wiederkehrender, bis auf den dynamischen Inhalt identischer, HTML-Code Fragmente. Diese Fragmente können dann beliebig häufig verwendet werden, um zum Beispiel die einzelnen Tabellenzeilen zu generieren.

Eine Sektion aus einer solchen Vorlage hat zum Beispiel das Aussehen, wie in Listing 1 dargestellt. Die führenden Nummern sind lediglich zur besseren Lesbarkeit eingefügt und nicht Bestandteil des eigentlichen Codes.

```
1. <#sektion start#>
2. <html>
3. <head>
4. <title>TV-Programm vom <#wochentag#> den <#datum#></title>
5. </head>
6. <body bgcolor="#FFFFFF">
7. <#sektion ende#>
```

Listing 1: Beispiel einer Vorlagen-Sektion

Die beiden Zeichen "<#" in Zeile 1 markieren den Beginn eines Schlüsselwortes, in diesem Fall das Wort "`sektion`", das Informationen über eine **Sektion** ankündigt. Dem Schlüsselwort folgt ein Parameter, der den Typ der Information festlegt. In diesem Fall ist es das Wort "`start`", das den Beginn eines neuen Abschnitts markiert. Abgeschlossen wird die Zeile durch die Zeichen "#>", die das Ende eines Schlüsselworts markieren. Innerhalb einer Sektion kann regulärer HTML-Code für das Layout der zu generierenden HTML-Seite verwendet werden (siehe Zeile 2, 3, 5 und 6). Zusätzlich zu diesem HTML-Code können innerhalb einer Sektion Platzhalter positioniert werden (hier `wochentag` und `datum`), die ebenfalls durch die Zeichen "<#" und "#>" eingerahmt werden. Darauf folgt ein weiteres Schlüsselwort, das das Ende des in Zeile 1 begonnenen Abschnitts markiert, gekennzeichnet durch den Parameter "`ende`".

---

[1] In dieser Berechnung ist der Sachverhalt nicht berücksichtigt, daß die Anzeige der einzelnen Zeilen über die Einstellungen eines Benutzer-Profils gesteuert werden kann. Setzt man voraus, daß dabei nur die Anzahl der Sender **S** pro Vorlage von Bedeutung ist, so muß für jede Sender-Zeile die Größe $\mathbf{P}_Z$ berücksichtigt werden. Die Anzahl der Permutationen $\mathbf{P}_S$ einer Vorlage mit S Sender-Zeilen ergibt sich durch $\mathbf{P}_S = \mathbf{P}_Z{}^S$. Für die Anzahl aller Permutationen der TV-Programmtabelle $\mathbf{P}_T$ erhält man dann eine geometrische Reihe, die folgendermaßen formuliert werden kann:

$$P_T = 1 + P_Z{}^1 + P_Z{}^2 + K + P_Z{}^S - 1 = \sum_{i=0}^{S} P_Z{}^i - 1 = \frac{1 - P_Z{}^{S+1}}{1 - P_Z} - 1$$

Für Z=27 und S=46 ergibt sich $\mathbf{P}_T \cong 1{,}08 \cdot 10^{360}$.

Betrachtet man das Klassendiagramm in Abbildung 33 auf Seite 85 und bildet die oben aufgeführte Sektion innerhalb der dort beschriebenen Klassenstruktur ab, so entsteht eine lineare Struktur, die folgenden Aufbau hat.

SektionGlied: typ="text", inhalt="<html>\n<title>TV-Programm vom "
SektionGlied: typ="platzhalter", inhalt="wochentag"
SektionGlied: typ="text", inhalt=" den "
SektionGlied: typ="platzhalter", inhalt="datum"
SektionGlied: typ="text", inhalt="</title>\n</head>\n<body bgcolor='#FFFFFF'>"

Die Sektion besteht ausschließlich aus linear aufeinanderfolgenden *SektionGliedern*. Auf jedes Text-*SektionGlied* folgt ein Platzhalter-*SektionGlied*. Tatsächlich ist es so, daß die Grenzen zwischen den *SektionGliedern* genau durch diese Übergänge zwischen Platzhaltern und Texten gebildet werden. Wäre also in Listing 1 kein Platzhalter vorhanden, würde die gesamte Sektion nur aus einem SektionGlied bestehen.

Mit einer solchen linearen Abfolge von *SektionGliedern* könnte beispielsweise der Kopf der TV-Programmübersicht generiert werden, da der Kopf wie erwähnt lediglich Navigationselemente und eventuell ein paar allgemeine Informationen enthält. Die Realisierung der TV-Programmtabelle ist in Listing 2 beispielhaft dargestellt.

```
01. <#sektion start#>
02. <table>
03. <#while more Beitrag.sender.name#>
04.       <tr>
05.           <td><#Beitrag.sender.name#></td>
06.           <#while equal++ Beitrag.sender.name#>
07.           <td colspan=<#Beitrag.colspan#>><#Beitrag.titel#></td>
08.           <#while end#>
09.       </tr>
10. <#while end#>
11. </table>
12. <#sektion ende#>
```

Listing 2: Vorlagen-Sektion „TV-Programmtabelle“

Die Abbildung dieser Sektion auf die Klassenstruktur ergibt eine komplexere Struktur. Abbildung 51 zeigt die entstehende Sektionsbaum-Struktur. Die beiden *Constraints* („C“) in der Abbildung des Sektionsbaumes repräsentieren die Zeilen "`while more Beitrag.sender.name`" und "`while equal++ Beitrag.sender.name`". Die *SektionGlieder* („SG“) repräsentieren die Folge von Text- und Platzhalter-Bestandteilen des Listing 2. Ein Sektionsbaum wird von *HTML_Kneter* durch einen „Preorder-Durchlauf“ verarbeitet. Die Abfolge der von *HTML_Kneter* besuchten Knoten des Sektionsbaumes in Abbildung 51 ist demnach: S_A, SG_B, C_C, SG_E, SG_F, SG_G, C_H, SG_K, SG_M, SG_N, SG_O, SG_P, SG_J, SG_L, SG_D.

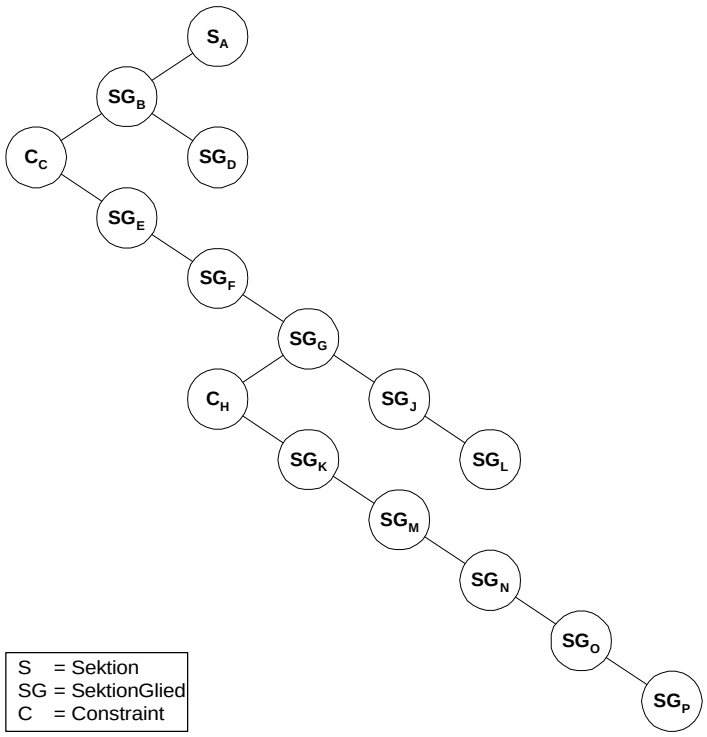


Abbildung 51: Sektionsbaum „TV-Programmtabelle“

Das Constraint („C_C“) „`while more Beitrag.sender.name`“ veranlaßt *HTML_Kneter*, den auf das Constraint folgenden Teilbaum solange bei der Generierung zu berücksichtigen, bis keine Beiträge mit unterschiedlichem Sender-Namen mehr zu verarbeiten sind. Das Constraint („C_H“) „`while equal++ Beitrag.sender.name`“ teilt *HTML_Kneter* mit, die Iteration so lange zu durchlaufen, bis sich der Name des Senders des aktuellen Beitrages ändert. Durch die beiden Pluszeichen hinter dem Wort „equal“ wird *HTML_Kneter* aufgefordert, den nächsten Beitrag in der Beitragsliste auszuwählen, nachdem ein Iterationsschritt beendet wurde. Nach Beendigung eines jeden „while equal++“ Schritts, wechselt *HTML_Kneter* also zum nächsten Beitrag in der Liste der zu verarbeitenden Beiträge. Auf diese Weise werden nach und nach alle Beiträge verarbeitet.

Durch Groß- und Kleinschreibung kann das „Ziel“ des „++“ definiert werden. Würde der Eintrag "while more++ Beitrag.**D**arsteller.name" lauten, so würde *HTML_Kneter* das „++“ auf das Beirags-Attribut *Darsteller* beziehen. *HTML_Kneter* würde also die Iteration solange durchlaufen, bis alle Darsteller eines Beitrags mit unterschiedlichem Namen verarbeitet worden sind.

Die PDF-Generierung

Die PDF-Generierung basiert, wie die HTML-Generierung, auf der Verarbeitung von Vorlagen. Die PDF-Vorlagen beinhalten ebenfalls *Sektionen*, *SektionGlieder* und *Constraints*, die analog zur HTML-Generierung verarbeitet werden. Bezüglich der PDF-Generierung gilt es jedoch eine Betrachtung der PDF-Spezifikation [Adobe96] einzufügen.

Der PDF-Standard schreibt die Verwendung bestimmter Teile eines PDF-Dokuments in einem festgelegten Format vor. Ein PDF-Dokument hat dabei immer folgende Grundstruktur:

1. header
2. body
3. cross-reference table
4. trailer

Der „header“ beinhaltet dabei die Versionsnummer der verwendeten PDF-Spezifikation (zum Beispiel „%PDF-1.2“). Ohne diese Versionsangabe, kann ein PDF-Dokument nicht interpretiert werden. Bei der Erstellung einer PDF-Vorlage ist also darauf zu achten, daß die Versionsangabe grundsätzlich in der ersten Zeile der ersten Sektion der Vorlage aufgeführt wird. Die „cross-reference table“ ist ein Verzeichnis aller im PDF-Dokument enthaltenen Objekte. Dieses Verzeichnis beinhaltet für jedes PDF-Objekt den Versatz (Offset) zum Beginn des Dokuments. Damit ist es möglich, PDF-Objekte (zum Beispiel Seiten des PDF-Dokuments) schneller anzuwählen, da eine Suche nach dem gewünschten Objekt innerhalb des Dokuments entfällt. Durch die Angabe des Offset kann das gewünschte Objekt direkt angewählt werden. Der „trailer“ schließlich beinhaltet eine Referenz auf das sogenannte Root-Objekt des Dokuments. Stellt man sich die Objekt-Struktur eines PDF-Dokuments als Baum vor, so ist das Root-Objekt die Wurzel des Baums. Weiterhin enthält der „trailer“ unter anderem das Offset der „cross-reference table“, die Anzahl der im Dokument enthaltenen Objekte und die Zeichenkette „%%EOF“, die das Ende der Datei markiert.

Der „body“ wird aus einzelnen PDF-Objekten gebildet, die den eigentlichen Inhalt des PDF-Dokuments ausmachen. Objekte sind zum Beispiel die einzelnen Seiten eines PDF-Dokuments oder die einzelnen Bestandteile einer Seite, wie Texte und Bilder. Über Referenzen auf die einzelnen Objekte wird die Struktur eines PDF-Dokuments bestimmt. PDF-Objekte können dabei beliebig auf einer Seite positioniert werden. Die Reihenfolge, in der die PDF-Objekte innerhalb der PDF-Datei aufgeführt werden spielt für das Layout des Dokuments allerdings keine Rolle. Allein die Reihenfolge, in der die Objekte über ihre Referenzen „aufgerufen“ werden, legt das endgültige Aussehen eines PDF-Dokuments fest. Hier besteht auch der größte Unterschied zu HTML-Dokumenten, die linear aufgebaut sind. Das heißt, die Reihenfolge in der die einzelnen Elemente innerhalb eines HTML-Dokumentes aufgeführt werden, ist für das Aussehen des Dokuments von entscheidender Bedeutung.

Um ein PDF-Dokument generieren zu können muß zumindest ein Ausschnitt aus dem Objektmodell der PDF-Spezifikation übernommen werden. Die Umsetzung der gesamten PDF-Spezifikation wäre Thema einer kompletten Diplomarbeit. Aus diesem Grund wurden die dynamischen Gestaltungsmöglichkeiten auf die Bedürfnisse der Generierung einer personalisierten Zeitschrift, wie in Abbildung 4 auf Seite 13 dargestellt, beschränkt. Um Strukturen für die Repräsentation von PDF-Dokumenten und deren Objekten zu

schaffen, wurden vier zusätzliche Klassen eingeführt: *PDFVorlage*, *PDFSektion*, *PDFObjekt* und *Absatz*.[1]

Einer der entscheidendsten Vorteile bei der Verwendung von PDF für die Generierung der personalisierten Zeitschriften ist die „Layout-Festigkeit“ eines PDF-Dokuments. Ein einmal erstelltes PDF-Dokument, wird auf verschiedenen Druckern in einem identischen Layout reproduziert. Zum Beispiel ist dabei die optische Struktur des Dokuments unabhängig davon, welche Auflösung der jeweilige Drucker unterstützt. Um diesen Vorteil nutzen zu können, werden innerhalb einer PDF-Vorlage die Maße des zu generierenden Dokumentes festgelegt. Dies betrifft die Größe der Seite und die Seitenränder. Um *PDF_Kneter* diese Maße mitteilen zu können wurde ein Einstellungen-Abschnitt in eine Vorlage eingefügt:

```
1.   <#einstellungen start#>
2.   seite 595 841
3.   raender 10 10 10 10
```

Mit der Zeichenkette `<#einstellungen start#>` wird dieser Abschnitt eröffnet. In Zeile 2 werden die Maße der Seite in Breite und Höhe festgelegt. Zeile 3 legt die Seitenränder in der Reihenfolge links, rechts, oben, unten fest. Die Maße werden dabei korrespondierend zu der typographischen Einheit *points* angegeben. Ein *point* hat in x- und y-Richtung jeweils eine Ausdehnung von $^{1}/_{72,27}$ inch. Die in der PDF-Spezifikation verwendete Einheit hat eine Ausdehnung von $^{1}/_{72}$ inch (entspricht 72dpi oder ca. 0,353 mm).[2] Der Einfachheit halber wird im folgenden die „Einheit“ *q-point* („quasi points“) verwendet, um dieses Maß zu beschreiben. Die oben gesetzten Werte werden innerhalb der Klasse *PDFVorlage* in entsprechenden Attributen gespeichert.

Im Anschluß an diese globalen Einstellungen für das zu generierende PDF-Dokument, können einzelne PDF-Objekte definiert werden. Diese Definitionen werden während der Generierung als Schablonen verwendet, um zur Laufzeit PDF-Objekte mit aktuellen Werten aus der Datenbank zu generieren. Schablonen haben eine feste Breite. Die Höhe eines PDF-Objektes wird von *PDF_Kneter* zur Laufzeit berechnet. Dazu verwendet *PDF_Kneter* die festgelegte Breite der Schablone, die innerhalb der Schablone definierten Texte und Bilder und die Anzahl der zu verwendenden Spalten[3]. Der folgende Code-Ausschnitt setzt den obigen dreizeiligen Code fort und zeigt an einem Beispiel die Definition einer PDF-Schablone für die Darstellung von Beitragsinformationen.

```
4.   %PDF-Objekt für die Darstellung der Beiträge
5.   <#objekt 6001 start#> %Beginn der Schablonen-Definition
6.   <#attribute start#> %Festlegen der einzelnen Schablonen-Attribute
```

---

[1] Die Klasse *PDFSektion* ist eine Kind-Klasse von *Sektion* und die Klasse *PDFVorlage* ist eine Kind-Klasse von *Vorlage*. Das entsprechende Klassendiagramm findet sich im Ergänzungsband.

[2] Die hier angegebenen Maße beziehen sich auf den sogenannten „Default User Space“, der üblicherweise für die Programmierung von PostScript- und PDF-Dokumenten verwendet wird. 595 x 841 Einheiten entsprechen dabei exakt den Abmaßen einer DIN A4 Seite.

[3] Die Anzahl der Spalten kann innerhalb einer Sektionsdefinition festgelegt werden (siehe weiter unten in diesem Abschnitt).

```
7.   breite 190
8.   bilder Beitrag.detail_text.text
9.   yspacing 5 %Zu berücksichtigender Platz zwischen zwei Objekten
10.  <#attribute ende#>
11.  <#absaetze start#> %Festlegen der einzelnen Absatz-Einstellungen
12.  Profil.fernbedienung.tastennummer F1 9 46 3
13.  Beitrag.startzeit F1 14 46 3
14.  Beitrag.sender.name F1 11 46 3
15.  Beitrag.showview F1 9 46 3
16.  Beitrag.titel F1 12 145 3 1
17.  Beitrag.dauer F1 9 145 3 1
18.  Beitrag.untertitel F1 9 145 3 1
19.  Beitrag.sparte.name F1 9 145 3 1
20.  Beitrag.detail_text.text F1 9 180 3 1
21.  <#absaetze ende#>
```

Die Syntax der PDF-Vorlagen-Definition entspricht teilweise, der Syntax der Definition einer HTML-Vorlage. Für die Schablonen-Definitionen sind allerdings einige Schlüsselworte hinzugekommen. Eine Schablonen-Definition beginnt mit dem neu eingeführten Schlüsselwort `<#objekt 6001 start#>`. Dieser konkreten Schablone wird die Nummer 6001 zugeordnet, über die eine spätere Referenzierung der Schablone realisiert wird. Anschließend folgt das ebenfalls neue Schlüsselwort `<#attribute start#>`, das den Beginn der Festlegung der Attribute der PDF-Schablone definiert. So kann zum Beispiel die Breite der Schablone definiert werden. Der Eintrag `bilder Beitrag.detail_text.text` teilt *PDF_Kneter* mit, daß innerhalb der Schablone Bilder verwendet werden und daß die Absatzbreite des Beitrag-Attributs **detail_text.text** von den Maßen der verwendeten Bilder beeinflußt wird. Dies ist notwendig, damit *PDF_Kneter* die Höhe eines Absatzes berechnen kann. Diese Höhe kann in einer PDF-Vorlage verwendet werden, um weitere PDF-Objekte zu positionieren. *PDF_Kneter* errechnet zur Laufzeit nach jeder Positionierung eines PDF-Objekts die X- und Y-Koordinaten für die Plazierung des folgenden PDF-Objekts. Diese Koordinaten können über die Schlüsselworte <#xpos#> und <#ypos#> in einer Vorlage verwendet werden. Ferner sind einfache Rechenoperationen mit diesen Koordinaten möglich, wie Addition und Subtraktion. In diese Berechnungen können zusätzlich anhand von Klassen-Attributen berechnete Zahlenwerte einbezogen werden (zum Beispiel *Beitrag*.**detail_text**.**text**.hoehe).

Um diese Berechnungen durchführen zu können, muß *PDF_Kneter* davon in Kenntnis gesetzt werden, welche Schrift für einen bestimmten Absatz verwendet werden soll und welche Breite der Absatz überstreichen darf. Diese Einstellungen werden in Zeile 12 bis 20 vorgenommen. Mit dem Eintrag `Beitrag.titel F1 12 145 3 1` beispielsweise wird festgelegt, das für den Text, der im Attribut *Beitrag*.**titel** hinterlegt ist, die Schrift F1 in der Schriftgröße 12pt verwendet werden soll. Ferner werden 145 *q-points* für die Absatzbreite und 3 *q-points* für den Zeilenabstand definiert. Abschließend wird *PDF_Kneter* durch den Wert 1 am Ende der Zeile mitgeteilt, daß die Werte dieses Absatzes für die Berechnung der Gesamthöhe des zu generierenden PDF-Objektes verwendet werden soll. Die Summe der Höhen aller Absätze, die für die Berechnung markiert wurden, bestimmt die Gesamthöhe des PDF-Objektes.

Nachdem alle Einstellungen für die Vorlage und die PDF-Objekte vorgenommen worden sind, beginnt die Definition des Layouts des PDF-Objekts:

```
22.  << >>
23.  stream
24.  q
25.  <#Beitrag.sparte.farbe#> rg 1 w <#xpos + 2#> <#ypos#> 50
     <#Beitrag.titel.hoehe + Beitrag.dauer.hoehe +
     Beitrag.sparte.name.hoehe + Beitrag.untertitel.hoehe#> re f
26.  <#Beitrag.sparte.farbe#> RG 1 w <#xpos + 2#> <#ypos#> 50
     <#Beitrag.titel.hoehe + Beitrag.dauer.hoehe +
     Beitrag.sparte.name.hoehe + Beitrag.untertitel.hoehe#> re s
27.  1 w <#xpos + 54#> <#ypos#> 136 <#Beitrag.titel.hoehe +
     Beitrag.dauer.hoehe + Beitrag.sparte.name.hoehe +
     Beitrag.untertitel.hoehe#> re s
28.  BT /F1 14 Tf 0 0 0 rg
29.  1 0 0 1 <#xpos + 5#> <#ypos - 14#> Tm <#Beitrag.startzeit#>
30.  ET
31.  BT /F1 11 Tf
32.  1 0 0 1 <#xpos + 5#> <#ypos - 29#> Tm <#Beitrag.sender.name#>
33.  ET
34.  <#if Beitrag.showview#>
35.  BT /F1 9 Tf
36.  1 0 0 1 <#xpos + 5#> <#ypos - 41#> Tm <#Beitrag.showview#>
37.  ET
38.  <#if ende#>
39.  <#if Profil.fernbedienung.tastennummer#>
40.  BT /F1 9 Tf
41.  1 0 0 1 <#xpos + 5#> <#ypos - 53#> Tm <#'Taste '
     Profil.fernbedienung.tastennummer#>
42.  ET
43.  <#if ende#>
44.  BT /F1 14 Tf
45.  1 0 0 1 <#xpos + 56#> <#ypos - 14#> Tm <#Beitrag.titel#>
46.  ET
47.  BT /F1 9 Tf
48.  1 0 0 1 <#xpos + 56#> <#ypos - Beitrag.titel.hoehe - 9#> Tm
     <#Beitrag.dauer 'Min.'#>
49.  ET
50.  BT /F1 9 Tf
51.  1 0 0 1 <#xpos + 56#> <#ypos - Beitrag.titel.hoehe -
     Beitrag.dauer.hoehe - 9#> Tm <#Beitrag.sparte.name#>
52.  ET
53.  <#if Beitrag.untertitel#>
54.  BT /F1 9 Tf
55.  1 0 0 1 <#xpos + 56#> <#ypos - Beitrag.titel.hoehe -
     Beitrag.dauer.hoehe - Beitrag.sparte.name.hoehe - 9#> Tm
     <#Beitrag.untertitel#>
56.  ET
57.  <#if ende#>
58.  <#if Beitrag.detail_text.text#>
59.  BT /F1 9 Tf
60.  1 0 0 1 <#xpos + 2#> <#ypos - Beitrag.titel.hoehe -
     Beitrag.dauer.hoehe - Beitrag.sparte.name.hoehe -
     Beitrag.untertitel.hoehe - 9#> Tm <#Beitrag.detail_text.text#>
61.  ET
62.  <#if ende#>
63.  <#if Beitrag.bild#>
64.  <#Beitrag.bild.breite#> 0 0 <#Beitrag.bild.hoehe#>
65.  <#xpos + 2 + Beitrag.detail_text.text.breite#>
66.  <#ypos - Beitrag.titel.hoehe - Beitrag.dauer.hoehe -
     Beitrag.sparte.name.hoehe - Beitrag.untertitel.hoehe - 9 -
     Beitrag.bild.hoehe#> cm
67.  BI /W <#Beitrag.bild.breite#> /H <#Beitrag.bild.hoehe#>
```

```
68.  /BPC 8 /CS /RGB /F [/DCT]
69.  ID <#Beitrag.bild#>
70.  EI
71.  <#if ende#>
72.  Q
73.  endstream
74.  <#objekt ende#>
```

Dieser Teil des Codes beinhaltet unter anderem Berechnungen zur Positionierung der Inhalte eines PDF-Objekts relativ zu den von *PDF_Kneter* berechneten absoluten Koordinaten. Diese absoluten Koordinaten sind über die Schlüsselworte <#xpos#> und <#ypos#> referenzierbar. In Zeile 60 beispielsweise werden diese Koordinaten benutzt um den Detailtext eines Beitrags zu positionieren.[1] Außerdem werden in dieser Zeile Werte verwendet, die von *PDF_Kneter* anhand von Absatz-Einstellungen berechnet werden, wie zum Beispiel **`Beitrag.titel.hoehe`**. Hier wird die relative Position des Detailtextes eines Beitrags in Abhängigkeit der Absatzhöhen von *Beitrag***.titel**, *Beitrag***.dauer**, *Beitrag***.sparte.name** und *Beitrag***.untertitel** berechnet. Da der Ursprung des Koordinatensystems links unten auf der Seite zu finden ist, werden die einzelnen berechneten Höhen, von der aktuellen Y-Koordinate subtrahiert. Somit wird der Detailtext 2 *q-points* (`<#xpos + 2#>`) rechts von der aktuellen X-Koordinate und so viele *q-points* unterhalb der aktuellen Y-Koordinate positioniert, wie die Summe der einzelnen oben aufgeführten Absatzhöhen ergibt.

Um die innerhalb eines PDF-Objekts anzuzeigenden Informationen steuern zu können, werden, analog zur HTML-Generierung, Constraints verwendet. Die Überführung der innerhalb der PDF-Schablone verwendeten Constraints in eine Baumstruktur ergibt eine zur Abbildung 51 auf Seite 126 äquivalente Struktur. Eine einmal definierte PDF-Schablone kann beliebig oft innerhalb einer Sektion referenziert werden.

Der folgende Ausschnitt zeigt eine weitere Definition einer PDF-Schablone und den Abschluß der Vorlagen-Einstellungen und Schablonen-Definitionen.

```
75.  %PDF-Objekt für die Darstellung des Logos und der Werbung:
76.  <#objekt 6002 start#>
77.  << >>
78.  stream
79.  q
80.  0.6 0.6 0.6 RG 2 w 12 760 130 61 re s
81.  0.6 0.6 0.6 rg
82.  130 0 0 61 12 760 cm
83.  BI /W <#Betreiber.logo.breite#> /H <#Betreiber.logo.hoehe#>
84.  /BPC 8 /CS /RGB /F [/DCT]
85.  ID <#Betreiber.logo#>
86.  EI
87.  Q
88.  q
89.  <#Werbung.banner.breite#> 0 0 <#Werbung.banner.hoehe#> 145 <#831 -
     Werbung.banner.hoehe#> cm
90.  BI /W <#Werbung.banner.breite#> /H <#Werbung.banner.hoehe#>
91.  /BPC 8 /CS /RGB /F [/DCT]
92.  ID <#Werbung.banner#>
```

[1] Die für Berechnungen verwendeten Teile der Zeile 60 wurden fett formatiert.

```
93.  EI
94.  Q
95.  q
96.  0.6 0.6 0.6 rg 1 w 12 735 571 21 re f
97.  BT /F1 14 Tf 1 1 1 rg
98.  25 745 Td <#'Guten Tag, Herr ' Profil.benutzer_name#>
99.  ET
100. Q
101. endstream
102. <#objekt ende#>
103. <#einstellungen ende#>
```

Zeile 76 bis 102 definieren eine weitere PDF-Schablone, die allerdings hauptsächlich statische Koordinaten enthält. Lediglich die Maße des Werbe-Banners werden bei der Generierung berücksichtigt (Zeile 89 bis 90). Die Definition aller Vorlagen-Einstellungen (Maße und PDF-Schablonen) wird durch Zeile 103 abgeschlossen. Nun beginnt der Teil der PDF-Vorlage, der die Abfolge von Sektionen und den darin verwendeten PDF-Schablonen definiert:

```
104. %Sektionen. Beginnt mit PDF-Code für Mantel und Fonts:
105. <#sektion start#>
106.
107. 7 0 obj
108. << /Type /Font /Subtype /Type1 /Name /F1 /BaseFont /Helvetica
     /Encoding /WinAnsiEncoding >>
109. endobj
110.
111. <#Mantel.titel#>
112. <#if Mantel.vorspann#><#Mantel.vorspann#><#if ende#>
113. <#if Mantel.nachspann#><#Mantel.nachspann#><#if ende#>
114. <#objekt 6002#>
115. <#sektion ende#>
116.
117. <#sektion spalten 3 start#>
118. <#while equal++ Beitrag.datum#>
119. <#object 6001#>
120. <#while end#>
121. <#sektion ende#>
```

Listing 3: Beispiel für eine PDF-Vorlage

Zunächst wird in Zeile 107 bis 109 die innerhalb der PDF-Schablonen verwendete Schrift definiert. Anschließend wird der PDF-Code für den redaktionellen Mantel eingefügt. (Zeile 111 bis 113). Ist für einen Betreiber kein redaktioneller Mantel definiert worden, so wird ein Standard-Titel in das PDF-Dokument eingefügt. Dieser Standard-Titel beinhaltet zum Beispiel immer das Root-Objekt des PDF-Dokuments und eine Referenz auf das zu verwendende erste Seiten-Objekt. Im Anschluß daran wird am Ende der ersten Sektion die Schablone mit der Nummer 6002 eingefügt, die das Betreiber-Logo, ein Werbe-Banner und die Anrede für den Abonnenten enthält. Den Abschluß der Vorlage bildet eine Sektion, die innerhalb eines „while equal“-Constraints alle für den Abonnenten ermittelten Beiträge eines Tages darstellt (Zeile 118 bis 120). Dabei werden die einzelnen PDF-Objekte dreispaltig angeordnet (Zeile 117 „`sektion` **`spalten 3`** `start`“).

Abbildung 52 zeigt ein mögliches Ergebnis der PDF-Generierung bei Verwendung der PDF-Vorlage in Listing 3.

Abbildung 52: Beispiel für ein generiertes PDF-Dokument

Das in Abbildung 52 dargestellte PDF-Dokument hat eine Dateigröße von 3443 Byte. Wird diese PDF-Seite vollständig mit Beitragsinformationen gefüllt, so muß die Dateigröße in etwa verfünffacht werden (ca. 17,5 KByte). Da dieses Dokument aus reinen Textdaten besteht, müssen bei der Berechnung der Dateigröße noch das Betreiber-Logo, das Werbe-Banner und die Beitragsbilder berücksichtigt werden. Wird eine durchschnittliche Anzahl von sieben Bildern pro PDF-Seite zugrunde gelegt, ergibt sich eine Dateigröße von ca. 87 KByte (bei ca. 10 KByte pro Bild). Diese Zahlen gelten für ein nicht komprimiertes PDF-Dokument.

Mit der Beschreibung der PDF-Generierung werden die Ausführungen über Modellierung und Implementation des Publikationssystems abgeschlossen. Anhand der Schilderung in Kapitel 5 und 6 sowie unter Verwendung des Ergänzungsbandes ist das System umfassend beschrieben worden. Der folgende Abschnitt soll zeigen, wie die Funktionalität des beschriebenen Systems nachgewiesen werden kann.

7 Nachweis der Funktionalität

Da für die Modellierung des Systems PEP keine formalen Methoden verwendet wurden, ist für den Nachweis der Funktionalität ein umfangreicher Test zu konzeptionieren. Dabei ist auf das objektorientierte Paradigma der Applikation Rücksicht zu nehmen. Im Rahmen dieser Diplomarbeit ist nur ein sehr geringer Teil des Systems implementiert worden. Aus diesem Grund beschränkt sich der in diesem Kapitel beschriebene Nachweis der Funktionalität auf eine Betrachtung der grundlegenden Verfahrensweise bezüglich der Verifizierung und Validierung des Systems (Kapitel 7.1). Aufbauend auf diese Verfahrensweise wird in Kapitel 7.2 ein Modell für das Vorgehen beim Testen des Systems PEP erläutert.

7.1 Verfahrensweise

Die hier beschriebene Verfahrensweise für den Test objektorientierter Software basiert auf dem GMD-Bericht Nr. 260 „Test, Analyse und Verifikation von Software“ [GMD260]. Der Schwerpunkt liegt dabei auf der Betrachtung des Artikels von Harry M. Sneed, der auf Seite 25 des Berichts beginnt. Dieser Artikel betrachtet die Unterschiede zwischen Tests konventioneller und objektorientierter Software und erarbeitet ein objektorientiertes Testverfahren.

Innerhalb des Artikels wird erläutert, daß sich die Ziele des Testens von objektorientierter Software weitgehend mit denen eines konventionellen Software-Tests decken. Beim Test objektorientierter Software ergeben sich allerdings „völlig neue Fehlerquellen, zum Beispiel in der Objektinitialisierung, der Nachrichtenübermittlung, der Vererbung falscher Versionen und der Beseitigung toter Objekte“ [GMD260, Seite 26]. Der Test dient dabei folgenden Zielen:[1]

- **Funktionsnachweis**: Das Anwendungssystem erfüllt die spezifizierten Anforderungen.
- **Normierungsnachweis**: Die Software erfüllt die herrschenden Normen z.B. für die Netzkommunikation und die Objektverwaltung.
- **Robustheitsnachweis**: Unkorrekte Eingaben werden abgefangen. Das System bricht nicht unkontrolliert ab.
- **Sicherheitsnachweis**: Sicherung von Zwischenergebnissen.
- **Ergebnisnachweis**: Das Anwendungssystem liefert korrekte Ausgaben.
- **Plattformnachweis**: Das Anwendungssystem läuft stabil in der Zielumgebung.

Zur Erbringung dieser Nachweise für objektorientierte Software, werden die drei konventionellen Testansätze „White-Box-Test“, „Black-Box-Test“ und „Grey-Box-Test“

[1] Die für den Test von PEP relevanten Ziele sind einer Aufzählung aus [GMD260, Seite 26] entnommen worden.

in die OO-Testmethoden „Regelbasierter Test“, „Nachrichtenbasierter Test“ und „Nutzungsbasierter Test“ überführt.[1]

7.1.1 Regelbasierter Test

Der Regelbasierte Test entspricht einem White-Box-Test der einzelnen Methoden. Ziel dieses Tests ist es, jeden möglichen Entscheidungspfad einer Methode zu testen. Dazu gehören nicht nur die regulären Verarbeitungszweige sondern auch die durch Fehlerbedingungen und Ausnahmebehandlung entstehenden Pfade. Sneed schlägt vor, die Ableitung der einzelnen Testfälle direkt aus dem Source-Code vorzunehmen. Der Regelbasierte Test ist nur für den Unit-Test von Methoden, Klassen und Moduln zu verwenden.

Klassenverifikation
Aus der Sicht des Testers ergeben sich für den Regelbasierten Test die drei zu betrachtenden Klassenarten Standardklassen, Basisklassen und abgeleitete Klassen.

Als erstes ist eine Validierung der Standardklassen durchzuführen. Dies sind gekaufte oder sonst übernommene Klassen, die bereits spezifiziert, programmiert, kompiliert und verifiziert worden sind. Eine Validierung dieser Klassen in der Systemumgebung ist jedoch noch nicht vorgenommen worden. Deshalb müssen diese Klassen in den Nutzungstest integriert werden (vgl. Kapitel 7.1.3). Um das Vertrauen in die Korrektheit von Standardklassen zu erhöhen, können diese vor der Entwicklung der eigenen Klassen einem Prototyptest unterzogen werden.

Im Anschluß daran sind die Basisklassen zu testen. Da Basisklassen lediglich eine nicht ausführbare Basis für abgeleitete Klassen darstellen, ist für jede Basisklasse mindestens eine reale abgeleitete Testklasse zu konstruieren. Diese Testklassen sollten alle Operationen der Basisklasse enthalten. Mit Hilfe der Testklassen sind alle Variationen sämtlicher Basisoperationen zu verifizieren.

Abgeleitete Klassen sollten nur für getestete Basisklassen implementiert werden. Da sie Attribute, Methoden und Schnittstellen von der Basisklasse erben, darf eine Implementierung erst nach der Verifizierung der Basisklasse erfolgen. Für jede abgeleitete Klasse müssen alle Methoden nacheinander kodiert und getestet werden. Anschließend sollte das Zusammenspiel der einzelnen Methoden untereinander getestet werden. Anhand der Methodenspezifikationen werden dazu regelbasierte Testfälle entwickelt, die in Form von Pre- und Postzusicherungen in Testprozeduren abgespeichert werden. Die einzelnen Testprozeduren werden von sogenannten Methodentesttreibern ausgeführt.

[1] Die Begriffe „White-Box“ und „Black-Box“ wurden von E. Miller [Miller77] geprägt. Der Begriff „Grey-Box-Test“ stammt von W. Howden [Howden87].

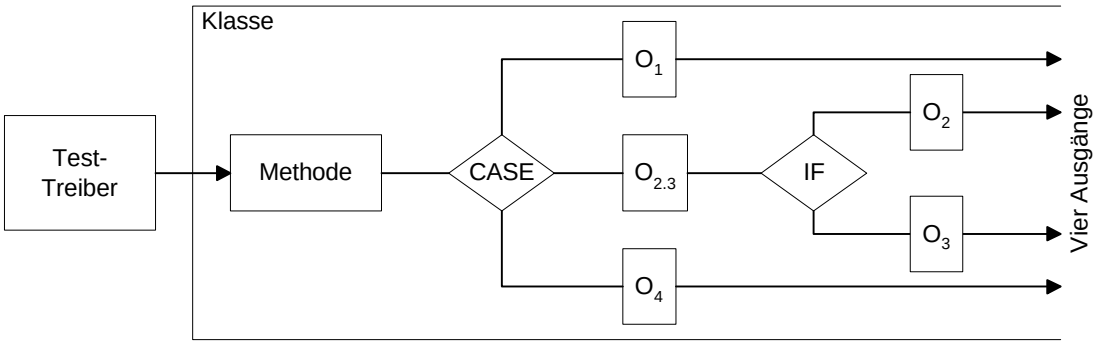


Abbildung 53: Regelbasierter Test („White-Box-Test“) [GMD260]

Generell ist jede Methode sofort nach ihrer Codierung zu testen und jede Klasse ist sofort nach dem Test ihrer Methoden zu testen. Die Vorgehensweise und Testergebnisse sind zu dokumentieren.

Modulverifikation
Ist eine Basisklasse mit ihren sämtlichen abgeleiteten Klassen implementiert und verifiziert, so kann mit der Modulverifikation begonnen werden. Nach Sneed wird in diesem Testverfahren unter einem Modul eine Menge abgeleiteter Klassen mit der gleichen Basisklasse verstanden. Entsprechend dem Test der Basisklassen werden für Module Testklassen entwickelt, um die Vererbungsmechanismen zu verifizieren.

Jedes Modul ist sofort nach der Fertigstellung seiner letzten Klasse zu testen. Klassenverifikation und Modulverifikation sind direkt von den Entwicklern durchzuführen.

7.1.2 Nachrichtenbasierter Test

Der Nachrichtenbasierte Test (Schnittstellenvalidation) entspricht einem Black-Box-Test der einzelnen Schnittstellen zwischen den Objekten und kann sofort nach Fertigstellung der ersten Module begonnen werden. Grundlage für diesen Test bilden die zwischen den Objekten ausgetauschten Nachrichten. Für jede innerhalb der Spezifikation festgehaltene relevante Nachricht sollte dabei ein eigener Testfall entwickelt werden. Jeder Eingangsnachricht folgt eine entsprechende Ausgangsnachricht, die zu validieren ist. Außerdem ist es wichtig, bei Nachrichtenketten zwischen mehreren Objekten die einzelnen „Zwischennachrichten“ abzufangen und deren Inhalte auf Korrektheit zu überprüfen. Durch diese Vorgehensweise werden alle Ausgangsnachrichten validiert.

Für die Durchführung dieses Tests müssen zuerst nachrichtenbasierte Testfälle in Form von Eingabe/Ausgabe-Zusicherungen erfaßt werden. Ein Generator erstellt anschließend anhand der Eingabe-Zusicherungen verschiedene Eingangsnachrichten. Die entsprechenden Ausgangsnachrichten werden von einem Validator anhand der Ausgabe-Zusicherungen überprüft. Die Vorgehensweise und Testergebnisse sind zu dokumentieren.

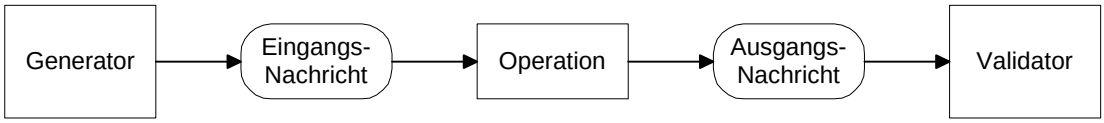


Abbildung 54: Nachrichtenbasierter Test („Black-Box-Test“) [GMD260]

7.1.3 Nutzungsbasierter Test

Der Nutzungsbasierte Test entspricht einem Grey-Box-Test der einzelnen Nutzungsfälle der zu testenden Applikation. Für einen vollständigen Test werden alle spezifizierten Nutzungsfälle von der Benutzeroberfläche „gestartet“. Die einzelnen Endergebnisse der Bearbeitung der Nutzungsfälle werden auf Korrektheit überprüft. Ziel dieses Tests ist die Feststellung der Einhaltung der Spezifikation aus funktionaler Sicht. Mit anderem Worten, es wird überprüft, ob das System die spezifizierten Leistungen erbringt. Der Autor spricht in diesem Zusammenhang deshalb auch von einem „Eignungstest“ [GMD260, Seite 36]. Neben der Validierung der direkt an der Benutzeroberfläche sichtbaren Endergebnisse, sind außerdem die Zustände der Objekte in der Datenbank nach jeder Ausführung eines Nutzungsfalls zu überprüfen.

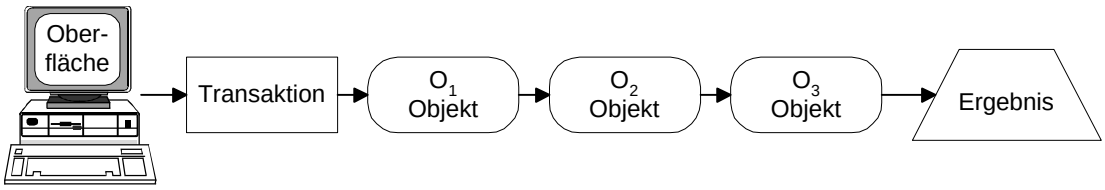


Abbildung 55: Nutzungsbasierter Test („Grey-Box-Test“) [GMD260]

Da der Nutzungsbasierte Test das gesamte System prüft, beginnt dieser Test erst nach dem Test aller Module. Die Vorgehensweise und Testergebnisse sind zu dokumentieren.

7.1.4 Systemabnahme

Die Systemabnahme erfolgt nach Beendigung des nutzungsbasierten Tests und bildet den Abschluß der Testphase. Dabei arbeiten Benutzer auf dem gesamten System unter realen Bedingungen. Das heißt, das System wird in der echten Zielumgebung getestet. In diesem Punkt besteht kein Unterschied zum Abnahmetest konventioneller Systeme. Die Vorgehensweise und Testergebnisse sind zu dokumentieren.

Um die Anforderungen des in diesem Kapitel beschriebenen Testverfahrens zu erfüllen, ist, insbesondere bei umfangreichen Projekten, der Einsatz von Testwerkzeugen erforderlich. Teilweise sind Werkzeuge am Markt erhältlich, die einen systematischen OO-Testprozeß unterstützen.[1] Für die Umsetzung des gesamten Testverfahrens, sind jedoch unter Umständen eigene, projektspezifische Werkzeuge zu entwickeln.

---

[1] Zum Beispiel für den Black-Box- und White-Box-Test das Produkt „IDAS TESTAT“ der Firma IDAS GmbH (http://www.idas.de/) und für den Grey-Box-Test das Produkt „WinRunner“ der Firma Mercury Interactive GmbH (http://www.merc-int.com/).

Die in diesem Kapitel erläuterte Testverfahrensweise stellt ein Grundgerüst für die Entwicklung eines Vorgehensmodells für den Nachweise der Funktionalität des Systems PEP dar. Das folgende Kapitel erörtert ein solches Modell anhand der in diesem Kapitel beschriebenen Verfahrensweise für den Test objektorientierter Software.

7.2 Testvorgehensmodell

Um die Anforderungen an einen systematischen OO-Test erfüllen zu können, ist neben dem eigentlichen Entwicklungsprojekt ein separates Testprojekt einzuführen. Dies erlangt insbesondere durch den Sachverhalt Bedeutung, daß etwa 40% des gesamten Projektaufwandes auf das Testen entfällt [Duden-Inf93]. In [GMD260] wird sogar ein Aufwand von mindestens 50% angenommen.

Laut [GMD260, Seite 26] sollte ein OO-Testprojekt die folgenden neun Phasen durchlaufen:

1. Erstellung eines Testplans nach ANSI/IEE 829-1983 [IEEE829-83].
2. Testfälle bzw. Nutzungsfälle spezifizieren (Testfallspezifikation).
3. Testprozeduren für Methoden, Klassen, Moduln und Schnittstellen entwerfen (Testprozedurerstellung).
4. Testumgebung mit Testtreibern, Teststubs (Test-Programmstümpfe) und Testdaten aufbauen (Testrahmengenerierung).
5. Einzelne Methoden, Klassen und Module testen (vgl. Kapitel 7.1.1 Regelbasierter Test).
6. Schnittstellen und Nachrichten testen (vgl. Kapitel 7.1.2 Nachrichtenbasierter Test).
7. Nutzungsfälle testen (vgl. Kapitel 7.1.3 Nutzungsbasierter Test).
8. Das System in seiner Gesamtheit testen (vgl. Kapitel 7.1.4 Systemabnahme).
9. Ergebnisse der Tests entsprechend der Testkonvention dokumentieren (Abschlußbericht).

Für das Testprojekt zur Durchführung des OO-Tests für das System PEP muß zunächst ein Testplan nach [IEEE829-83] erarbeitet werden. Dieser Testplan legt den Umfang, die Vorgehensweise, die benötigten Ressourcen und den Terminplan für die einzelnen Testaktivitäten fest. Außerdem identifiziert er die zu testenden Einheiten, den zu testenden Funktionsumfang, die einzelnen Testaktivitäten und die personellen Verantwortlichkeiten für jede Testaktivität. Ferner gibt der Testplan Auskunft über Art und Umfang der während des Tests zu erstellenden Dokumentation.

Da das zu testende System PEP eine objektorientierte Anwendung ist, müßte der Testplan als Vorgehensweise zur Durchführung des Tests die oben genannten Phasen 2 bis 9 festlegen. Dies impliziert, daß die in Kapitel 7.1 vorgestellte Verfahrensweise zum Test objektorientierter Software zur Anwendung kommt.

Im Anschluß an die Erstellung des Testplans müssen die einzelnen Testfälle bzw. Nutzungsfälle spezifiziert werden. Hierzu ist die durch diese Diplomarbeit repräsentierte Beschreibung des Systems zu verwenden. Nutzungsfälle für das System PEP sind zum Beispiel die Generierung einer personalisierten TV-Programmzeitschrift im PDF-Format oder die Anmeldung eines Abonnenten. Die ermittelten Nutzungsfälle werden für den Nutzungsbasierten Test in Phase 7 des Vorgehensmodells benötigt. Für die Spezifikation der einzelnen Testfälle werden die Methodenbeschreibungen des Data-Dictionaries verwendet. Diese Testfälle müssen in Phase 3 in Form von Pre- und Postzusicherungen als Testprozeduren implementiert werden. Die so erstellten Testprozeduren bilden die Grundlage für den Aufbau der Testumgebung in Phase 4.

Eine komplette Spezifikation aller Testfälle wird in Phase 2 des Vorgehensmodells wahrscheinlich nicht erreicht werden können, da während der Entwicklung des Systems immer neue Testfälle hinzukommen. Die Spezifikation kann vielmehr immer nur den aktuellen Entwicklungsstand widerspiegeln (vgl. [GMD260, Seite 41]). Entsprechendes gilt für die Testprozeduren.

In den Phasen 5 und 6 wird im Anschluß die Testumgebung für den systematischen Test der Methoden, Klassen und Module sowie deren Schnittstellen verwendet. Für die Klassen Eingabe und Eingabe_Server müßte dieser Test beispielsweise folgendermaßen aufgebaut werden: Zunächst müssen alle Methoden der beiden Klassen unabhängig voneinander getestet werden, anschließend wird jede Klasse als Ganzes getestet. Damit ist Phase 5 für die beiden Klassen abgeschlossen. In Phase 6 werden dann die Schnittstellen zwischen den Klassen getestet. Dies sind die Methoden ***gib_betreiber_container***, ***gib_profil_container*** und ***gib_werbe_container***.

Um den Testaufwand für die Phasen 2 bis 6 zu minimieren und um Fehler bei der Erstellung des Testrahmens zu vermeiden, ist der Einsatz eines kommerziellen Test-Tools zur Unterstützung des Tests in Betracht zu ziehen.

Nach Abschluß der Phasen 5 und 6 muß das Gesamtsystem unter Verwendung der in Phase 2 ermittelten Nutzungsfälle einem Nutzungsbasierten Test unterzogen werden. Für diesen Test kann ein sogenanntes Capture/Replay-Tool eingesetzt werden. Für den Nutzungsbasierten Test von PEP ist der Einsatz eines solchen Tools wahrscheinlich jedoch nicht sinnvoll. Dies liegt darin begründet, das die vom System generierten Ausgaben, in Form von dynamischen HTML- und PDF-Dokumenten, nur bedingt durch feste Regeln prüfbar sind.

Abschließend ist das System unter realen Bedingungen von einem größeren Benutzerkreis zu testen. Das Ergebnis dieses Abschlußtests und die Ergebnisse aller vorher durchgeführten Tests sind in standardisierter Form zu dokumentieren und in einem Abschlußbericht zusammenzufassen.

8 Ausblick

In dieser Diplomarbeit wurde ein System modelliert, das personalisierte Programmzeitschriften für einzelne Benutzer produziert. Diese Zeitschriften können von einem Benutzer auf S/W- oder Farbdruckern ausgedruckt und anschließend wie herkömmliche Programmzeitschriften vor dem Fernseher verwendet werden. Den Verlagen eröffnet sich damit, neben der traditionellen Massenproduktion von TV-Programmzeitschriften, eine neue Produktionsform – die Individualproduktion von Programmzeitschriften.

Die Idee einer „Individualproduktion von Programmzeitschriften“ läßt sich auch auf beliebige Print-Titel übertragen. Diese Übertragung ist bei dem in dieser Arbeit entwickelten **P**ersonalisierten **E**lektronischen **P**rogrammberater (PEP) problemlos möglich, da die gesamte benutzerspezifische Ausgabe des Systems unabhängig von der Struktur der zu verarbeitenden Daten ist. Erreicht wird dies über die PDF- und HTML-Vorlagen, welche die gesamte benutzerspezifische Ein- und Ausgabe des Systems steuern. Das heißt, in den Vorlagen wird festgelegt, welche Daten bei der Generierung eines Dokuments verwendet werden und welche Profileinstellungsmöglichkeiten dem Benutzer angeboten werden, um die Inhalte seiner Zeitschrift zu konfigurieren. Aus diesem Grund ist es für die Funktionalität des Systems nicht relevant, welche und wieviele Attribute eines Beitrags bzw. eines Profils effektiv genutzt werden. Wird das System für die Aufbereitung redaktioneller Inhalte verwendet, werden für die Beschreibung eines Beitrags zum Beispiel nur noch Titel, Untertitel, Detailtext, Charakter, Sparte und Autor benötigt. Die Einstellungsmöglichkeiten eines Benutzerprofils beinhalten dementsprechend nur noch diese Eigenschaften zur Konfiguration der personalisierten Zeitschrift. Daraus ergibt sich, daß die redaktionellen Inhalte sämtlicher Zeitschriften und Zeitungen eines Verlages auf demselben Wege, wie er für PEP konzeptioniert wurde, individualisiert an den Benutzer vermittelt werden können.

In einer Weiterentwicklung könnten die Einsatzmöglichkeiten des Systems auf zusätzlich zu generierende Formate, zum Beispiel für mobile, digitale Medien erweitert werden. Vorstellbar ist hier beispielsweise das Bereitstellen von TV-Programminformationen für **P**ersönliche **D**igitale **A**ssistenten (PDA)[1]. Der Besitzer eines PDAs lädt das personalisierte TV-Programm zum Beispiel in den Terminkalender seines digitalen Assistenten. Da PDAs üblicherweise eine Alarmfunktion bieten, kann der Besitzer so rechtzeitig über den Beginn einer für ihn relevanten Sendung informiert werden. Ebenso wäre zum Beispiel die Generierung einer personalisierten Tageszeitung für das Rocket eBook[2] denkbar.

Die aller Voraussicht nach zunehmende technische Integration von Computer und Fernsehempfänger wird auch entscheidenden Einfluß auf zukünftige Entwicklungen von

---

[1] Zum Beispiel das Gerät „PalmPilot“ der Firma 3Com und die Produktserie „Psion“ des gleichnamigen Herstellers.

[2] Das von der Firma NuvoMedia entwickelte elektronische Buch Rocket eBook entspricht in seinen Maßen einem herkömmlichen Buch. Bücher und Texte können in digitaler Form zum Beispiel über das Internet bezogen und über eine Docking-Station in das Rocket eBook überspielt werden. Technische Details sind unter http://www.rocket-ebook.com/ zu finden.

TV-Programminformationssystemen haben. Die Verschmelzung der Darstellung herkömmlicher TV- und Internet-Angebote in einem Gerät bietet auch für elektronische Programmführer (EPG) ganz neue Möglichkeiten. Denkbar wäre hier ebenfalls ein personalisierter TV-Programmberater, der seine TV-Programminformationen über das Internet bezieht. Zusätzlich kann der Benutzer die von diesem EPG empfohlenen Beiträge direkt – ohne Medienbruch – anwählen. Auf diese Weise können zum Beispiel Previews der angewählten Filme übertragen werden. Werden in dem Gerät zusätzlich sichere Zahlungsmöglichkeiten integriert, so ist auf demselben Wege die Realisierung einer Programmberatung mit anschließendem „pay per view“ oder „video on demand“ denkbar.

9 Anhang

9.1 Allgemeines Literaturverzeichnis

[Booch94] Grady Booch
Object-oriented analysis and design : with applications
ISBN 0-8053-5340-2
1994 (Benjamin/Cummings)

[Burkhardt97] Rainer Burkhardt
UML - Unified Modelling Language:
Objektorientierte Modellierung für die Praxis
ISBN 3-8273-1226-4
1997 (Addison-Wesley)

[Garfinkel96] Simson Garfinkel und Gene Spafford
Titel: Practical UNIX and Internet Security, Second Edition
971 pages, Paperback
ISBN: 1-56592-148-8
1996 (O'Reilly & Associates, Inc.)

[Hughes97] Cameron and Tracy Hughes
Object-Oriented Multithreading Using C++
495 pages, Paperback
ISBN 0-471-18012-2
1997 (Published by John Wiley & Sons, Inc.)

[Jacobsen92] Ivar Jacobsen, Magnus Christerson, Patrik Jonsson, Gunnar Övergaard
Object-Oriented Software Engineering:
A Use Case Driven Approach
ISBN 0-201-54435-0
1992 (Addison-Wesley)

[Merz98] Thomas Merz
Titel: Mit Acrobat ins World Wide Web:
effiziente Erstellung von PDF-Dateien und ihre Einbindung ins Web
285 Seiten, 1 CD
ISBN: 3-9804943-1-4
1998 (tm-Verlag, München)

[Pfleeger97] Charles P. Pfleeger
Security in Computing
ISBN 0-13-185794-0
1997, 2. Auflage (Prentice-Hall International Editions)

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen
Object-Oriented Modeling and Design
ISBN 0-13-629841-9
1991, amerikanische Originalausgabe (Prentice-Hall International Inc.)

[Rumbaugh94] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen
Objektorientiertes Modellieren und Entwerfen
ISBN 3-446-17520-2
1994, dt. Ausgabe (Prentice-Hall International Inc.)

[Schneider91] Prof. Dr. Hans-Jochen Schneider [Hrsg.]
Lexikon der Informatik und Datenverarbeitung
ISBN 3-486-21514-0
1991, 3.Auflage (R. Oldenbourg Verlag GmbH, München)

[Stary96] Christian Stary
Interaktive Systeme, Softwareentwicklung und Software-Ergonomie
ISBN 3-528-15384-9
1996, 2. Auflage (Vieweg & Sohn Verlagsgesellschaft mbH)

[Siegel96] David Siegel
Web Site Design: Killer Web Sites der 3. Generation
ISBN 3-8272-518774-2
1996 (Markt&Technik Buch- und Software- Verlag GmbH)

[Schmidt87] P.C. Lockemann und J.W. Schmidt (Herausgeber)
Datenbank-Handbuch
ISBN 3-540-10741-X
1987 (Springer-Verlag Berlin Heidelberg New York)

[Uxxxx] siehe URL Verzeichnis in Abschnitt 9.3.

9.2 Spezielles Literaturverzeichnis

[Adobe96] Adobe Systems Incorporated
Portable Document Format Reference Manual
Version 1.2
27. Nov. 1996

[ASV95] Relations "Programmzeitschriften und Fernsehen"
"Die Beziehungen zwischen Programmzeitschriften und Fernsehen heute"
Random Stichprobe von 1.661 deutschsprachigen TV-Haushalten (3.033 Personen über 14 Jahre) des Instituts: Infratest Kommunikationsforschung
1995, Axel Springer Verlag AG

[Chen76] P.P.S. Chen
The Entity-Relationship model - toward a unified view of data
ACM Transactions on Database Systems 1, (March 1976)

[DeMarco79] Tom DeMarco
Structured Analysis and Systems Specification
Englewood Cliffs, New Jersey
1979, Prentice Hall

[Gane78] Chris Gane and Trish Sarson
Structured Systems Analysis: Tools and Techniques
Englewood Cliffs, New Jersey
1978, Prentice Hall

[GMD260] M. Müllerburg, A. Spillner, P. Liggesmeyer (Herausgeber)
Test, Analyse und Verifikation von Software
ISBN 3-486-23845-0, GMD-Bericht Nr. 260
1996, R. Oldenburg Verlag

[Harel87] David Harel
Statecharts: a visual formalisms for complex systems. Science of Computer Programming 8 (1987), S. 231-274

[Howden87] W. Howden
Functional Program Testing
New York
1987, McGraw-Hill

[IDraft96a] E. Rescorla, A. Schiffman;
Terisa Systems, Inc.; The Secure HyperText Transfer Protocol;
Internet-Draft <draft-ietf-wts-shttp-03.txt>; Juli 1996

[IDraft96b] Freier, Karlton, Kocher
Netscape Communications Corp.; The SSL Protocol Version 3.0;
Internet-Draft <draft-ietf-tls-ssl-version3-00.txt>; 18 November 1996

[Duden-Inf93] Duden „Informatik“
Ein Sachlexikon für Studium und Praxis
ISBN 3-411-05232-5
1993, 2., vollst. überarb. und erw. Aufl.,
Mannheim; Leipzig; Wien; Zürich: Dudenverlag

[IEEE729-83] ANSI/IEEE Standard 829-1983
IEEE Standard Glossary of Engineering Terminology
New York
1983, The Institut of Electrical and Electronics Engineers, Inc.

[IEEE829-83] ANSI/IEEE Standard 829-1983
IEEE Standard for Software Test Documentation
New York
1983, The Institut of Electrical and Electronics Engineers, Inc.

[IEEE1008-87] ANSI/IEEE Standard 1008-1987
IEEE Standard for Software Unit Testing
New York
1987, The Institut of Electrical and Electronics Engineers, Inc.

[IEEE1012-86] ANSI/IEEE Standard 1012-1986
IEEE Standard for Software Verification and Validation Plans
New York
1986, The Institut of Electrical and Electronics Engineers, Inc.

[IEEE1059-93] ANSI/IEEE Standard 1059-1993
IEEE Guide for Software Verification and Validation Plans
New York
1993, The Institut of Electrical and Electronics Engineers, Inc.

[IuKDG97] Gesetz zur Regelung der Rahmenbedingungen für Informations- und Kommunikationsdienste
(Informations- und Kommunikationsdienste-Gesetz – IuKDG)
In der Fassung des Beschlusses des Deutschen Bundestages vom 13.06.97
(BT-Drs. 13/7934 vom 11.06.1997)
URL: http://www.iid.de/rahmen/iukdgbt.pdf

[Kollock90] Nicolai G. Kollock
PostScript richtig eingesetzt
ISBN 3-88322-247-X
1990, 2. Auflage, IWT Verlag GmbH

[Miller77] E. Miller
Program Testing – Art meets Theory
IEEE Computer
Juli 1977 Seite 42 bis 51

[Münch97] Nils Münch
Analyse bestehender digitaler TV-Informationssysteme in interaktiven Medien / Studienarbeit
Universität Hamburg, Fachbereich Informatik
Standort: FB Informatik [18/228], Signatur: R32045

[NIST93] U.S. Department of Commerce
National Institute of Standards and Technology (NIST)
Subject: Digital Signature Standard (DSS)
Federal Information Processing Standards Publication 186
19 May 1994

[NIST95] U.S. Department of Commerce
National Institute of Standards and Technology (NIST)
Subject: Secure Hash Standard (SHS)
Federal Information Processing Standards Publication 180-1
17 April 1995

[POET97] POET Software Corporation, San Mateo
POET C++ SDK Programmer´s Guide
Version 5.0
http://www.poet.com
http://www.poet.de

[POET96] POET Software Corporation, San Mateo
POET Generic Programming Guide
Version 4.0
http://www.poet.com
http://www.poet.de

[Renner91] Gerhard Renner
PostScript – Fortgeschrittene Programmierung
ISBN 3-89090-968-X
1991, Markt&Technik Verlag AG

[RFC 1521] MIME (Multipurpose Internet Mail Extensions) Part One:
Mechanisms for Specifying and Describing the Format of
Internet Message Bodies. N. Borenstein & N. Freed.
September 1993.

[RFC 1522] MIME (Multipurpose Internet Mail Extensions) Part Two:
Message Header Extensions for Non-ASCII Text. K. Moore.
September 1993.

[RFC1945] T. Berners-Lee MIT/LCS, R. Fielding UC Irvine, H. Frystyk MIT/LCS
Request for Comments: 1945, Category: Informational
Hypertext Transfer Protocol -- HTTP/1.0; May 19

[Uxxxx] siehe URL Verzeichnis in Abschnitt 9.3.

[VDZ98] Verband Deutscher Zeitschriftenverleger
Titel: Messung der Werbeträgerleistung von Online-Medien
Dezember 1996
http://www.pz-online.de/

[Yourdon89] Edward Yourdon
Modern Structured Analysis
Englewood Cliffs, New Jersey
1989, Yourdon Press

9.3 URL Verzeichnis

[URL-ASV98]

ASV Homepage	http://www.asv.de/
fussball.de	http://www.fussball.de/
ticker.de	http://www.ticker.de/
verreisen.de	http://www.verreisen.de/
Sport 1	http://www.sport1.de/
rallye racing online	http://www.rallyeracing.de/

[URL-OOC97] OO-CASE-Tools
http://www.informatik.fh-luebeck.de/~st/gz/CASETools/index.html

[URL-PDF-Pl] http://www.ep.cs.nott.ac.uk/pdf-pl/
alpha release (v1.0a24) of PDF on the Fly, a Perl5 library for creating PDF files

[URL-PDF-C] http://www.ifconnection.de/~tm/
PDFlib: C library for dynamically generating PDF
Thomas Merz

[URL-POET98] http://www.poet.com
http:///www.poet.de/

[URL-DENIC98] DEutsches-Network Information Center
Das Wachstum des Internet
http://www.nic.de/

[URL-NetWiz98] Network Wizards (Hrsg.)
Internet Domain Survey
http://www.nw.com/

[URL-TV98]	ARD und ZDF: TV-Lotse	http://www.ardzdf.de/
	Euro-TV (Belgien)	http://www.eurotv.com/
	GIST (USA)	http://www.gist.com/
	ProSieben Online TV-SPY	http://www.pro-sieben.de/
	rtv-online	http://www.rtv.de/
	STERN: tv-agent	http://stern.de/sterntv/agent/
	The TV Entertainment Magazine (USA)	http://www.tvguide.com/tv/
	TELE - Das Schweizer Medienmagazin	http://www.tele.ch/
	TELE Online (Österreich)	http://www.tele.at/
	TVCompass - TV-Infos per e-Mail	http://www.tvcompass.de/
	TV Movie online	http://www.tvmovie.de/
	TV Spielfilm	http://www.tvspielfilm.de/
	TV TODAY (THE NAVIGATOR)	http://www.tvtoday.de/
[URL-ZS98]	Planet Allegra	http://www.allegra.de/
	Sport BILD - Online	http://www.sportbild.de/
	Auto BILD - Online	http://www.autobild.de/
	Familie&Co	http://www.familie.de/
	FINANZEN	http://finanzenonline.de/Magazin/
[URL-ZT98]	BILD - Online	http://www.bild.de/
	Computer BILD online	http://www.computerbild.de/
	DIE WELT - Online	http://www.welt.de/
	Hamburger Abendblatt - Online	http://www.abendblatt.de/
	BERLINER MORGENPOST- Online	http://www.berliner-morgenpost.de/
	BZ auf Draht	http://www.bz-berlin.de/
	Kieler Nachrichten	http://www.kn-online.de/
	Lübecker Nachrichten Online	http://www.ln-online.de/

9.4 Tabellenverzeichnis

Tabelle 1: Wichtigste Informationsquellen für das Fernsehprogramm 6

Tabelle 2: Beispiel für den Inhalt eines Benutzerprofils 11

Tabelle 3: Vergleich objektorientierter CASE-Tools 18

Tabelle 4: Data-Dictionary (Analyse)....37
Tabelle 5: Assoziationen (Analyse)....38
Tabelle 6: Szenario „Server fordert Dokument von Eingabe an“ 46
Tabelle 7: Verzeichnisstruktur....61
Tabelle 8: Queries versus Filter [POET97] 120
Tabelle 9: Data-Dictionary 196
Tabelle 10: PMTP Befehle....199
Tabelle 11: OO Case-Tools 200
Tabelle 12: Beispiel für den Inhalt eines Fernsehbeitrages 201

9.5 Abbildungsverzeichnis

Abbildung 1: Aufbau von PEP.... 8
Abbildung 2: PEP TV-Programmtabelle 10
Abbildung 3: Personalisierte Online Version von PEP....11
Abbildung 4: Layout-Entwurf der personalisierten TV-Programmzeitschrift 13
Abbildung 5: Page-Server....21
Abbildung 6: Object-Server....21
Abbildung 7: Ausschnitt der OMT-Notation für das Objektmodell [Rumbaugh94].....26
Abbildung 8: Ausschnitt der OMT-Notation für das dynamische Modell [Rumbaugh94].29
Abbildung 9: OMT-Notation für das funktionale Modell [Rumbaugh94].....30
Abbildung 10: OMT Lebenszyklusphasen....32
Abbildung 11: Klassendiagramm Objekt Analyse 39
Abbildung 12: Klassendiagramm Übersicht (Analyse) 40
Abbildung 13: Klassendiagramm Beitrag (Analyse)....42
Abbildung 14: Module (Analyse)....43

Abbildung 15: Ausriß des Klassendiagramms aus Abbildung 12 45

Abbildung 16: Ereignispfaddiagramm Dokument generieren (Analyse) 47

Abbildung 17: Ereignispfaddiagramm Dokument erzeugen (Analyse) 49

Abbildung 18: Ereignisflußdiagramm Server – Eingabe – Dokument – Aktion_BE – Aktion_FE – Datenbank_Server 50

Abbildung 19: Zustandsdiagramm Dokument – Dokument generieren 51

Abbildung 20: Beispiel funktionales Modell – Flugsimulator 53

Abbildung 21: Beispiel funktionales Modell – PEP 53

Abbildung 22: Ein- und Ausgabewerte von PEP 55

Abbildung 23: Oberstes Datenflußdiagramm 55

Abbildung 24: Modularisierung von PEP 57

Abbildung 25: Teilsysteme von PEP 57

Abbildung 26: Globaler Datenfluß 59

Abbildung 27: Zustandsdiagramm „versende TV-Programmzeitschriften“ 69

Abbildung 28: Zustandsdiagramm „versende TV-Programmzeitschriften - mit Überprüfung des Versand-Datums“ (Systementwurf) 70

Abbildung 29: Klassendiagramm Übersicht 74

Abbildung 30: Klassendiagramm Beitrag_Container 77

Abbildung 31: Klassendiagramm Profil_Container 80

Abbildung 32: Klassendiagramm Dokument 82

Abbildung 33: Klassendiagramm HTML_Kneter 85

Abbildung 34: Klassendiagramm Eingabe 87

Abbildung 35: Ereignisflußdiagramm Server – Eingabe – Eingabe_Server – HTML_Kneter – PDF_Kneter 89

Abbildung 36: Zustandsdiagramm Eingabe 91

Abbildung 37: Zustandsdiagramm Eingabe (Fortsetzung) 92

Abbildung 38: Ereignisflußdiagramm Eingabe_Server – Betreiber_Container – Betreiber_Aktion_FE – Betreiber 95

Abbildung 39: Zustandsdiagramm Eingabe_Server → Betreiber_Container anfordern.....96

Abbildung 40: Zustandsdiagramm HTML_Kneter → Abonnenten-Dokument generieren98

Abbildung 41: Ereignisflußdiagramm HTML_Kneter – PDF_Kneter – Beitrag_Container – Beitrag_Aktion_BE – Beitrag_Aktion_FE.....101

Abbildung 42: Zustandsdiagramm HTML_Kneter Gast-Dokument generieren.....102

Abbildung 43: Ereignisflußdiagramm HTML_Kneter – PDF_Kneter – Profil_Container – Profil_Aktion_BE – Profil_Aktion_FE 104

Abbildung 44: Zustandsdiagramm HTML_Kneter → Profil Änderungen speichern.....106

Abbildung 45: Persistente Klassen von PEP.....110

Abbildung 46: Kompilierung und Registrierung des Schemas der Betriebs-Datenbank ..111

Abbildung 47: Kompilierung und Registrierung einer POET Datenbank.....112

Abbildung 48: Datenbanken mit gemeinsamen Dictionary.....112

Abbildung 49: POET Transaktions-Puffer [POET97] 116

Abbildung 50: Tiefen-Modi für den Zugriff auf POET Objekten [POET97] 119

Abbildung 51: Sektionsbaum „TV-Programmtabelle“ 126

Abbildung 52: Beispiel für ein generiertes PDF-Dokument 133

Abbildung 53: Regelbasierter Test („White-Box-Test“) [GMD260].....137

Abbildung 54: Nachrichtenbasierter Test („Black-Box-Test“) [GMD260].....138

Abbildung 55: Nutzungsbasierter Test („Grey-Box-Test“) [GMD260].....138

Abbildung 56: PMTP Paket.....197

9.6 Glossar

API

Application Programming Interface; Programmierschnittstelle.

Audiotex

Unter Audiotex wird ein interaktiver Telefonansagedienst, gestützt durch einen Audiotext-Sprachserver, verstanden.

Black-Box-Test

Die Bezeichnung „Black-Box“ wurde von E. Miller geprägt. Eine Black-Box ist nach [Ferretti96, Seite 707] eine „Einheit bekannter Schnittstellen und Funktion, aber unbekannter interner Ausführung“.

Nach der Black-Box-Methode werden für jede Konstellation der Datenstrukturen und Eingabedaten eines Programms verschiedene Testfälle konstruiert. Ziel der Methode ist die Validierung der Ausgabewerte eines Programms unter Berücksichtigung aller möglichen bzw. sinnvollen Wertekombinationen von Eingabedaten. Der Black-Box-Test ist demnach datenbezogen.

Siehe auch →*Grey-Box-Test* und →*White-Box-Test*.

CGI

Common Gateway Interface; Protokoll über das sich Web-Server mit externen Programmen koppeln lassen.

CORBA

Common Object Request Broker Architecture; Auf Grundlage der →*OMA* wurde der Kommunikationsmechanismus CORBA von der →*OMG* erarbeitet. CORBA ermöglicht die netzwerktransparente Kommunikation zwischen unterschiedlichen Software-Systemen über heterogene Hardware hinweg. Siehe auch →*IDL*.

DBMS

Datenbankmanagementsystem; System zum Aufbau, zur Kontrolle und Manipulation von Datenbanken. Es realisiert alle Funktionen der Datenbeschreibung und Datenmanipulation. [Schneider91]

ECash

Electronic Cash; elektronische Bezahlung per Internet über spezielle Dienstleister, die in der Regel eine Art Konto für Ihre Kunden führen.

Grey-Box-Test

Die Bezeichnung „Grey-Box“ wurde von E. Howden geprägt. Grundlage für die Grey-Box-Methode sind die funktionalen Spezifikationen eines Programms. Dabei dient die Ein- und Ausgabe der gesamten Applikation als Rahmen des Tests. Jeder spezifizierten Eingabe muß die entsprechende spezifizierte Ausgabe folgen. Der Grey-Box-Test ist demnach in dem Sinne funktionsbezogen, daß die spezifizierten, fachbezogenen Anforderungen einer Applikation erfüllt werden müssen.

Siehe auch →*Black-Box-Test* und →*White-Box-Test*.

IDL

Interface Definition Language; unabhängige Beschreibungssprache für die unter →*CORBA* verfügbaren Dienste.

Objektpersistenz

Objektpersistenz beschreibt die Möglichkeit, die Lebensdauer eines Objektes, über die Lebensdauer des Prozesses zu verlängern, der das Objekt erzeugt hat. Persistenz ist für das betroffene Objekt transparent, so daß der →*Persistenz Mechanismus* innerhalb einer →*Persistenz Schicht* gekapselt werden kann.

Antonym: →*Objekttransienz*

Objekttransienz

Der Sachverhalt, daß die Lebensdauer eines Objektes auf die Lebensdauer des erzeugenden Prozesses beschränkt ist.

Antonym: →*Objektpersistenz*

ODBMS

Objektorientiertes →*Datenbankmanagementsystem*.

OMA

Object Management Architecture; Referenzmodell der →*OMG* das die OMA die Grundzüge einer verteilten objektorientierten Architektur beschreibt, die aus Applikations- und verschiedenen System-Objekten besteht, die über einen gemeinsamen Kommunikationsmechanismus, den sogenannten „Object Request Broker“ (ORB), Nachrichten austauschen. Siehe auch →*CORBA*.

OMG

Object Management Group; unabhängige Standardisierungsorganisation für die netzwerktransparente Kommunikation heterogener Systeme auf Basis objektorientierter Software-Komponenten.

PageImpressions

(bisher PageViews) bezeichnen die Anzahl der Sichtkontakte beliebiger Benutzer mit einer potentiell werbeführenden HTML-Seite. Sie liefern ein Maß für die Nutzung einzelner Seiten eines Angebotes. Enthält ein Angebot Bildschirmseiten, die sich aus mehreren Frames zusammensetzen (Frameset), so gilt jeweils nur der Inhalt eines Frames als Content. Der Erstabruf eines Framesets zählt daher nur als ein PageImpression, ebenso wie jede weitere nutzerinduzierte Veränderung des entsprechenden Content-Frames. Demnach wird pro Nutzeraktion nur ein PageImpression gezählt. Zur definitionsgerechten Erfassung der PageImpressions verpflichtet sich der Anbieter, gekennzeichneten Content jeweils nur in einen Frame pro Frameset zu laden. [VDZ98]

PDF

Portable Document Format; plattformunabhängiges Dateiformat der Firma Adobe®. PDF basiert auf der Seitenbeschreibungssprache PostScript® und beschreibt Text und Grafik in einem geräte- und auflösungsunabhängigen Format. Zum Betrachten von PDF-Dateien gibt es für alle gängigen Betriebssysteme die kostenlose Software „Acrobat® Reader“.

Persistenz

Dauerhaftigkeit. →*Objektpersistenz*
Antonym: →*Transienz*

Persistenz Mechanismus

Die permanente Speicherfähigkeit Objekte →*Persistenz* zu verleihen. Der Persistenz Mechanismus ermittelt und speichert die transitive Hülle eines Objekts bezüglich dessen „enthält"-Beziehungen zur Laufzeit eines Prozesses. Auf Anforderung wird diese „Teilmenge" des Objekts einem Prozeß zugänglich gemacht.

Persistenz Schicht

Eine Sammlung von Klassen, die Objekten die Möglichkeit zur →*Objektpersistenz* gibt. Die Persistenz Schicht kapselt effektiv den →*Persistenz Mechanismus*.

q-point

Quasi Point; Ein q-point ist eine in dieser Diplomarbeit eingeführte Maßeinheit, die korrespondierend zu der typographischen Einheit *points* verwendet wird. Ein *point* hat in x- und y-Richtung jeweils eine Ausdehnung von $^{1}/_{72,27}$ inch. Ein q-point hat eine Ausdehnung von $^{1}/_{72}$ inch (entspricht 72dpi oder ca. 0,353 mm).

RDBMS

Relationales →*Datenbankmanagementsystem*.

SQL

Structured Query Language; Datenmanipulationssprache für das Wiederauffinden und für den Änderungsdienst relationaler Datenbanken.

Supplement

Eine Zeitschrift oder Zeitung, die einer gekauften Zeitschrift oder Zeitung kostenlos beiliegt. Zum Beispiel ein wöchentlich beigelegtes TV-Programmheft in einer Tageszeitung.

Transaktion

In [Schmidt87, Seite 398] wird der Begriff der Transaktion folgendermaßen erläutert: „"A transaction is a unit of work that is atomic from the view of the enterprise." Atomic bedeutet sowohl *unteilbar* in dem Sinne, daß alle zu einer Transaktion gehörenden Einzelaktionen zusammengehören. [...], als auch ununterbrechbar in dem Sinne, daß aus der Sicht des Benutzers die Transaktion entweder vollständig ausgeführt wird oder gar nicht."

Transienz

Vergänglichkeit. →*Objekttransienz*
Antonym: →*Persistenz*

Validierung

Bei dem klassischen Softwaredesign versteht man unter Validierung den Prozeß der Software-Evaluierung am Ende des Entwicklungsprozesses mit dem Ziel, die Übereinstimmung mit den Anforderungen sicherzustellen. [IEEE729-83]

Verifizierung

Bei der konventionellen Softwareerstellung versteht man unter Verifizierung die Untersuchung, ob ein Produkt in einer bestimmten Phase der Softwareentwicklung die Spezifikationen erfüllt oder nicht, die vorher festgestellt wurden. [IEEE729-83]

Visit

Ein Visit bezeichnet einen zusammenhängenden Nutzungsvorgang (Besuch) eines WWW-Angebots. Er definiert den Werbeträgerkontakt. Als Nutzungsvorgang zählt ein technisch erfolgreicher Seitenzugriff eines Internet-Browsers auf das aktuelle Angebot, wenn er von außerhalb des Angebotes erfolgt. [VDZ98]

White-Box-Test

Der Begriff „White-Box“ wurde von E. Miller geprägt. Ziel der Methode ist der Durchlauf jedes Programmzweiges. Dabei wird für jeden möglichen Zweig des Programms ein Testfall definiert. Der White-Box-Test ist demnach ablaufbezogen. Siehe auch →*Black-Box-Test* und →*Grey-Box-Test*.

9.7 Data-Dictionary

Anmerkung zur Notation innerhalb des Data-Dictionary:
An oberster Position eines Abschnitts wurde der Name der zu beschreibenden Klasse aufgeführt. Referenzen auf andere im Data-Dictionary beschriebene Klassen wurden mit einem Pfeil gekennzeichnet und die referenzierte Klasse in Kursivschrift dargestellt (zum Beispiel „→*Darsteller*“). Attribute einer Klasse wurden **fett**, Methoden ***fett-kursiv*** formatiert. Es sei an dieser Stelle ausdrücklich darauf hingewiesen, daß innerhalb des Data-Dictionary ausschließlich die jeweiligen Klassen des Systems beleuchtet werden. Da manche Klassen einen „umgangssprachlichen“ Namen (wie zum Beispiel „*Darsteller*“) erhalten haben, wurde die oben aufgeführte Notation eingeführt, um Verwechslungen zu vermeiden.

Abrechnung_Gateway	Abstrakte Interface-Klasse zwischen →*Zahlung* und →*Abrechnungsanbieter*. Bietet einen vom Anbieter einer ECash Zahlungsmöglichkeit unabhängigen Weg zur Durchführung von Zahlungen via Internet. Die Implementation der benötigten Funktionalitäten wird vom jeweiligen Abrechnungsanbieter geliefert.
Abrechnungsanbieter	Anbieter einer Online-Zahlungsmöglichkeit. *Abrechnungsanbieter* kommuniziert direkt mit dem →*Client* eines →*Benutzers*. Der eigentliche (physikalische) Zahlungsvorgang wird von *Abrechnungsanbieter* vorgenommen.

Absatz	Repräsentiert einen Absatz für die Generierung von PDF-Dokumenten. Für einen Absatz können Schrift, Schriftgröße, Absatzbreite und Zeilenabstand festgelegt werden. **Attribute:** - **inhalt** Der Wert aus der Datenbank, für den diese Absatzeinstellungen gelten (zum Beispiel *Beitrag*.**titel**). - **hoehen_berechnung** Gibt an, ob der Absatz für die Berechnung der Höhe eines PDF-Objektes verwendet werden soll. - **schrift_typ** Die Schriftart des Absatzes. - **schrift_groesse** Die Größe der Schrift. - **absatz_breite** Die Breite des Absatzes. - **zeilenabstand** Der Zeilenabstand des Absatzes. **Methoden:** - ***gib_wert*** Liefert Werte der Klassen-Attribute. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes 2. Referenz auf eine Variable, in der der zu liefernde Wert abgelegt werden soll. - ***setze_wert*** Setzt Werte der Klassen-Attribute. Als Parameter werden übergeben: 1. Name des zu setzenden Wertes 2. Der zu setzende Wert.
Aktion	Abstrakte Superklasse von →*Aktion_BE* und →*Aktion_FE*.
Aktion_BE	Abstrakte Superklasse von →*Werbe_Aktion_BE*, →*Mantel_Aktion_BE*, →*Beitrag_Aktion_BE*, →*Betreiber_Aktion_BE* und →*Profil_Aktion_BE*. **Attribute:** - **datenbank_server** Referenz auf →*Datenbank_Server* **Methoden:** - ***gib_wert*** Virtuelle Methode. Liefert Inhalte von Attributen der Daten-Klassen →*Beitrag*, →*Profil*, →*Werbung*, →*Mantel*, →*Betreiber*. Wird implementiert von →*Beitrag_Aktion_BE*, →*Betreiber_Aktion_BE*, →*Mantel_Aktion_BE*, →*Werbe_Aktion_BE*, →*Profil_Aktion_BE* - ***fuell_mich*** Virtuelle Methode. Veranlaßt das Laden der Daten-Klassen →*Beitrag*, →*Profil*, →*Werbung*, →*Mantel* und →*Betreiber* aus der Datenbank. Wird implementiert von →*Beitrag_Aktion_BE*, →*Betreiber_Aktion_BE*, →*Mantel_Aktion_BE*, →*Werbe_Aktion_BE*, →*Profil_Aktion_BE*

Aktion_FE	Abstrakte Superklasse von →*Werbe_Aktion_FE*, →*Mantel_Aktion_FE*, →*Beitrag_Aktion_FE*, →*Betreiber_Aktion_FE* und →*Profil_Aktion_FE*. **Methoden:** - ***gib_wert*** Virtuelle Methode. Liefert Inhalte von Attributen der Daten-Klassen →*Beitrag*, →*Profil*, →*Werbung*, →*Mantel*, →*Betreiber*. Wird implementiert von →*Beitrag_Aktion_FE*, →*Betreiber_Aktion_FE*, →*Mantel_Aktion_FE*, →*Werbe_Aktion_FE*, →*Profil_Aktion_FE* - ***setze_wert*** Virtuelle Methode. Verändert Inhalte von Attributen der Daten-Klassen →*Beitrag*, →*Profil*, →*Werbung*, →*Mantel*, →*Betreiber*. Wird implementiert von →*Beitrag_Aktion_FE*, →*Betreiber_Aktion_FE*, →*Mantel_Aktion_FE*, →*Werbe_Aktion_FE*, →*Profil_Aktion_FE*
Anfrage_Daten	Klasse für die Speicherung der während einer Anfrage anfallenden Client-Informationen. Wird von →*Eingabe* an →*HTML_Kneter* oder →*PDF_Kneter* übergeben, um die den Client-Informationen entsprechende Generierung anzustoßen. **Attribute:** - **nw_paare** Referenz auf →*NameWertPaar*. Beinhaltet die Name/Wert-Paare, die sich aus der Strukturierung der Anfragedaten durch *Eingabe* ergeben. **Methoden:** - ***gib_wert*** Liefert Inhalt des Attributs **wert** der Klasse *NameWertPaar*. Als Parameter wird der Name des zu liefernden Wertes übergeben. - ***setze_wert*** Verändert Inhalt des Attributs **wert** der Klasse *NameWertPaar*. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert - ***gib_name*** Liefert Inhalt des Attributs **name** der Klasse *NameWertPaar*. Als Parameter wird übergeben: 1. die Nummer des zu liefernden Namens.
Ausnahme	Eine in sich geschlossene Anforderung von Operationen, die Meldungen für zur Laufzeit auftretende Ausnahmen (Fehler) ausgibt. **Attribute:** - **fehler_nummer** Nummer des aufgetretenen Fehlers - **fehler** Textuelle Beschreibung des aufgetretenen Fehlers für die Ausgabe an den Benutzer - **log_fehler** Textuelle Beschreibung des aufgetretenen

	Fehlers für die Ausgabe in eine Logdatei - **log_datei** Pfad und Dateiname der Logdatei - **uhrzeit** Datum und Uhrzeit, zu der der Fehler aufgetreten ist **Methoden:** - ***drucke_fehler*** Gibt die Informationen über den aufgetretenen Fehler aus. - ***gib_fehler_nummer*** liefert die Nummer des aufgetretenen Fehlers **Anmerkung:** *Ausnahme* ist innerhalb des Objektmodells nur an die Klasse →*Eingabe* gekoppelt. Die Klasse *Ausnahme* wird allerdings von **allen** Klassen des Systems genutzt, in denen Fehler auftreten können.
Beitrag	Abgeschlossene Informationsblöcke, die die Gesamtheit der Daten eines einzelnen Beitrages ausmachen. Mehrere Beiträge werden von einem externen →*Lieferant*en geliefert und von →*Import* in die Beitrag-Datenbank importiert. **Attribute:** - **produktion** Produzierendes Unternehmen - **thema** - **autor** - **drehbuch** - **drehjahr** - **schnitt** - **interpret** - **kamera** - **live** Live-Übertagung oder Aufzeichnung - **moderator** - **musik** - **original_titel** - **untertitel** - **produzent** - **stereo** Stereo, Zweikanal oder Mono - **titel** Name des Beitrages - **showview** - **regie** - **laenge** Länge des Beitrages in Minuten - **endzeit** End-Uhrzeit des Beitrages - **startzeit** Beginn des Beitrages - **datum** Sendedatum - **vps** **V**ideo **P**rogramming **S**ystem, Nummerncode zur Unterstützung der Programmierung eines Video-Rekorders - **darsteller** Liste von Referenzen auf →*Darsteller*

	- **detail_text** Liste von Referenzen auf die beschreibenden Texte zu einem Beitrag, →*Detail_Text* - **bild** Referenz auf ein Bild, →*PEPBild* - **sender** Referenz auf den ausstrahlenden →*Sender* - **sparte** Liste von Referenzen auf zum Beitrag passende Kategorisierungen, →*Sparte* - **charakter** Ähnlich dem Attribut **sparte** eine Liste von Referenzen auf kategorisierende Inhalte, →*Charakter* **Methoden:** - ***gib_wert*** Liefert aktuelle Inhalte von Attributen der Klasse *Beitrag* bzw. Inhalte der Attribute von referenzierten Objekten (zum Beispiel Attribute der Klasse *Darsteller*). Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* (zum Beispiel Beitrag.titel) und int (zum Beispiel Beitrag.laenge). Als Parameter werden übergeben: 1. Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Beitrag_Container*.***gib_wert***, →B*eitrag_Aktion_FE*.***gib_wert***, →*Beitrag_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Beitrag*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu setzenden Typen →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert (→*Beitrag_Container*.***setze_wert***, →B*eitrag_Aktion_FE*.***setze_wert***) **Anmerkung:** Das Lesen und Schreiben der Attribute referenzierter Klassen geschieht über die jeweiligen Methoden der referenzierten Klasse.
Beitrag_Aktion_BE	Eine in sich geschlossene Anforderung von Operationen auf →*Beitrag*. *Beitrag_Aktion_BE* füllt eine Instanz der Klasse *Beitrag_Container* mit *Beitrag*-Objekten aus der Beitrag-Datenbank. Beauftragt →*Datenbank_Server* mit der Speicherung von Änderungen. Wird ausschließlich von einer Instanz der Klasse →*Beitrag_Container* verwendet. **Attribute:** - **beitrag_container** Referenz auf eine Instanz der Klasse

	Beitrag_Container **Methoden:** - ***fuell_mich*** Veranlaßt das Laden von Instanzen der Klasse *Beitrag* aus der Datenbank. Anschließend werden die geladenen *Beitrag*-Objekte an den →*Beitrag_Container* angehängt (Attribut *Beitrag_Container*.**beitrag_set**). Als Parameter werden übergeben: 1. Instanz der Klasse →*Anfrage_Daten* 2. Instanz der Klasse →*Profil_Container* 3. Flag, ob eine Änderung erfolgen soll (→*Beitrag_Container*.***fuell_dich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Beitrag*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Beitrag_Container*.***gib_wert***, →B*eitrag_Aktion_FE*.***gib_wert***, →*Beitrag*.***gib_wert***) - ***speicher_beitrag*** Veranlaßt das Speichern einer Instanz der Klasse →*Beitrag* in die Datenbank. Als Parameter werden übergeben: 1. Nummer des zu speichernden Beitrages
Beitrag_Aktion_FE	Eine in sich geschlossene Anforderung von Operationen auf →*Beitrag*. Liest und ändert Inhalte von Instanzen der Klasse *Beitrag*, die von →*Beitrag_Aktion_BE* an →*Beitrag_Container* angehängt wurden. **Attribute:** - **beitrag_container** Referenz auf eine Instanz der Klasse *Beitrag_Container* **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Beitrag*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Beitrag_Container*.***gib_wert***, →B*eitrag*.***gib_wert***, →*Beitrag_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse

	Beitrag. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert (→*Beitrag_Container*.***setze_wert***, →B*eitrag*.***setze_wert***)
Beitrag_Container	Container-Klasse für →*Beiträge*. Dient als Interface zwischen Beitrag-Aktionen und *Beitrag*. **Attribute:** - **beitrag_set** Zeigerstruktur mit Instanzen der Klasse *Beitrag* **Methoden:** - ***fuell_dich*** Veranlaßt das Laden von Instanzen der Klasse *Beitrag* aus der Datenbank. Anschließend werden die geladenen *Beitrag*-Objekte in **beitrag_set** gespeichert. Als Parameter werden übergeben: 1. Instanz der Klasse →*Anfrage_Daten* 2. Instanz der Klasse →*Profil_Container* (→*Beitrag_Aktion_BE*.***fuell_mich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Beitrag*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. 3. Nummer des gewünschten Beitrages. (→*Beitrag_Aktion_FE*.***gib_wert***, →B*eitrag*.***gib_wert***, →*Beitrag_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Beitrag*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes. 2. der zu setzende Wert. 3. Nummer des gewünschten Beitrages (→*Beitrag_Aktion_FE*.***setze_wert***, →B*eitrag*.***setze_wert***) - ***speicher_beitrag*** Veranlaßt das Speichern einer Instanz der Klasse →*Beitrag* in die Datenbank. Als Parameter werden übergeben: 1. Nummer des zu speichernden Beitrages

Benutzer	Abstraktion einer natürlichen Person. Ein *Benutzer* hat ein →*Profil*, das seine persönlichen Interessen beschreibt. Die Einrichtung dieses *Profils* ist kostenpflichtig und es muß mittels einer →*Zahlung* freigeschaltet werden. Der Benutzer interagiert mit dem System über →*Client*s.
Betreiber	Betreibt ein →*Publishing_System*. →*Benutzer* sind Kunden eines *Betreiber*s, der deren jeweilige →*Profil*e verwaltet, bzw. führt. **Attribute:** - **name** Identifizierer des Betreibers - **beitrag_db_host** Datenbank-Host der Beitrag-Datenbank - **beitrag_db_name** Name der Beitrag-Datenbank - **betriebs_db_host** Datenbank-Host der Betriebs-Datenbank - **betriebs_db_name** Name der Betriebs-Datenbank - **vorlage_set** Liste von Referenzen auf →*Vorlagen*, die zur Generierung von HTML-Dokumenten verwendet werden. - **pdfvorlage_set** Liste von Referenzen auf →*PDFVorlagen*, die zur Generierung von PDF-Dokumenten verwendet werden. Üblicherweise beinhaltet **pdfvorlage_set** nur ein einziges *PDFVorlage*-Objekt. **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Betreiber*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Betreiber_Container*.***gib_wert***, →*Betreiber_Aktion_FE*.***gib_wert***, →*Betreiber_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Betreiber*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert (→*Betreiber_Aktion_FE*.***setze_wert***, →*Betreiber*.***setze_wert***) - ***gib_vorlage*** Liefert eine Instanz der Klasse →*Vorlage*. Als Parameter werden übergeben: 1. Typ der zu liefernden Vorlage

	(→*Betreiber_Aktion_FE*.***gib_vorlage***, →*Betreiber_Container*.***gib_vorlage***) **Anmerkung:** Das Lesen und Schreiben der Attribute referenzierter Klassen geschieht über die jeweiligen Methoden der referenzierten Klasse.
Betreiber_Aktion_BE	Eine in sich geschlossene Anforderung von Operationen auf →*Betreiber*. *Betreiber_Aktion_BE* füllt eine Instanz der Klasse *Betreiber_Container* mit *Betreiber*-Objekten aus der Betriebs-Datenbank. Beauftragt →*Datenbank_Server* mit der Speicherung von Änderungen. Wird ausschließlich von einer Instanz der Klasse →*Betreiber_Container* verwendet. **Attribute:** - **betreiber_container** Referenz auf eine Instanz der Klasse →*Betreiber_Container* **Methoden:** - ***fuell_mich*** Veranlaßt das Laden von Instanzen der Klasse *Betreiber* aus der Datenbank. Anschließend werden die geladenen *Betreiber*-Objekte an den *Betreiber_Container* angehängt (Attribut *Betreiber_Container*.**betreiber_set**). Als Parameter werden übergeben: 1. Identifizierer des *Betreibers* (→*Betreiber_Container*.***fuell_dich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Betreiber*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Betreiber_Container*.***gib_wert***, →*Betreiber_Aktion_FE*.***gib_wert***, →*Betreiber*.***gib_wert***) - ***speicher_betreiber*** Veranlaßt das Speichern einer Instanz der Klasse →*Betreiber* in die Datenbank. Als Parameter werden übergeben: 1. Nummer des zu speichernden Betreibers
Betreiber_Aktion_FE	Eine in sich geschlossene Anforderung von Operationen auf →*Betreiber*. Liest und ändert Inhalte von Instanzen der Klasse *Betreiber*, die von →*Betreiber_Aktion_BE* an →*Betreiber_Container* angehängt wurden. **Attribute:** - **betreiber_container** Referenz auf eine Instanz der Klasse

	Betreiber_Container **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Betreiber*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Betreiber_Container*.***gib_wert***, →*Betreiber*.***gib_wert***, →*Betreiber_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Betreiber*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert (→*Betreiber_Container*.***setze_wert***, →*Betreiber*.***setze_wert***) - ***gib_vorlage*** Liefert eine Instanz der Klasse →*Vorlage*. Als Parameter werden übergeben: 1. Typ der zu liefernden Vorlage (→*Betreiber*.***gib_vorlage***, →*Betreiber_Container*.***gib_vorlage***) - ***gib_betreiber_kopie*** liefert eine Instanz der Klasse →*Betreiber*. Diese Instanz ist eine Kopie eines an *Betreiber_Container* angehängten *Betreiber*-Objektes. Als Parameter werden übergeben: 1. Nummer des gewünschten Betreibers Diese Methode wird ausschließlich von der Klasse →*Eingabe_Server* verwendet, um bei einer Anforderung eines *Betreiber_Containers* durch →*Eingabe* eine Kopie des geforderten *Betreibers* zu liefern.
Betreiber_Container	Container-Klasse für →*Betreiber*. Dient als Interface zwischen Betreiber-Aktionen und *Betreiber*. **Attribute:** - **betreiber_set** Zeigerstruktur mit Instanzen der Klasse *Betreiber* **Methoden:** - ***fuell_dich*** Veranlaßt das Laden von Instanzen der Klasse *Betreiber* aus der Datenbank. Anschließend werden die geladenen *Betreiber*-Objekte in **betreiber_set** gespeichert.

	Als Parameter werden übergeben: 1. Identifizierer des *Betreibers* (→*Betreiber_Aktion_BE.**fuell_mich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Betreiber*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird 3. Nummer des gewünschten Betreibers (→*Betreiber_Aktion_FE.**gib_wert***, →*Betreiber.**gib_wert***, →*Betreiber_Aktion_BE.**gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Betreiber*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert 3. Nummer des gewünschten Betreibers (→*Betreiber_Aktion_FE.**setze_wert***, →*Betreiber.**setze_wert***) - ***speicher_betreiber*** Veranlaßt das Speichern einer Instanz der Klasse *Betreiber* in die Datenbank. Als Parameter werden übergeben: 1. Nummer des zu speichernden Betreibers - ***gib_vorlage*** Liefert eine Instanz der Klasse →*Vorlage*. Als Parameter werden übergeben: 1. Typ der zu liefernden Vorlage 2. Nummer des gewünschten Betreibers (→*Betreiber.**gib_vorlage***, →*Betreiber_Aktion_FE.**gib_vorlage***) - ***gib_betreiber_kopie*** liefert eine Instanz der Klasse *Betreiber*. Diese Instanz ist eine Kopie eines an *Betreiber_Container* angehängten *Betreiber*-Objektes. Als Parameter werden übergeben: 1. Nummer des gewünschten Betreibers Diese Methode wird ausschließlich von der Klasse →*Eingabe_Server* verwendet, um bei einer Anforderung eines *Betreiber_Containers* durch →*Eingabe* eine Kopie des geforderten *Betreibers* zu liefern. (→*Betreiber_Aktion_FE.**gib_betreiber_kopie***)
Browser	Client-Software zur Kommunikation eines →*Benutzer*s mit

	→*Server* (WWW-Server).
Charakter	Klasse zur verfeinerten Kategorisierung eines Beitrages. Mögliche Werte sind z.B. Western, Krimi, Leichtathletik. Die grobe Kategorisierung wird durch die Klasse →*Sparte* vorgenommen. **Attribute:** - **name** Name des Charakters, z.B. Western **Methoden:** - ***gib_name*** liefert den Wert des Attributs **name**. Als Parameter werden übergeben: 1. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird - ***setze_name*** verändert den Wert des Attributs **name**. 1. Der zu setzende Wert.
Client	Abstrakte Superklasse von →*Browser* und →*E-Mail_Client*.
Constraint	Randbedingung für die Anzeige einer →*Sektion* oder eines →*SektionGlieds*. *Constraints* können verschiedenen, konditionalen Typs sein. Mögliche Typen sind „if“, „while more“ und „while equal“. Ein Constraint beinhaltet einen auf Wahrheit zu prüfenden Wert, der im Attribut **inhalt** gespeichert ist. Beinhaltet eine *Sektion* oder ein *SektionGlied* eine Instanz der Klasse *Constraint*, wird der Wert des Attributs **inhalt** mit dem Wert des Attributs **typ** gemeinsam betrachtet. Ist der Typ des *Constraints* „if“ wird die *Sektion* bzw. das *SektionGlied* nur ausgegeben, wenn die Prüfung des Wahrheitswerts des Attributs **inhalt** „wahr“ ergibt. Ist der Typ des *Constraints* „while [...]“, wird die *Sektion* bzw. das *SektionGlied* sooft wiederholt ausgegeben, bis die Prüfung des Wahrheitswertes des Attributs **inhalt** „falsch“ ergibt. Folgendes Beispiel soll die Funktion verdeutlichen: Die HTML-Vorlage für die Einzelansicht eines Beitrages enthält für ein auszugebendes *SektionGlied* ein *Constraint* vom Typ „if“ und im Attribut **inhalt** den Wert „*Profil*.**vps“**. *HTML_Kneter* überprüft nun, ob das Attribut **vps** im Benutzer-Profil den Eintrag „VPS-Nummer anzeigen“ enthält. Ist dies der Fall, wird das *SektionGlied* ausgegeben. Andernfalls wird die Ausgabe des *SektionGliedes* unterdrückt. **Attribute:** - **typ** Typ des Constraints - **inhalt** Auf Wahrheit zu überprüfender Wert - **kette** Referenz auf →*SektionGlied* **Methoden:** - ***naechstes_glied*** Liefert die Referenz auf das *SektionGlied*

	aus dem Attribut **kette** - ***gib_typ*** Liefert den Wert des Attributs **typ**. - ***setze_typ*** Verändert den Wert des Attributs **typ**. Als Parameter werden übergeben: 1. der zu setzende Wert - ***gib_inhalt*** Liefert den Wert des Attributs **inhalt**. - ***setze_inhalt*** Verändert den Wert des Attributs **inhalt**. Als Parameter werden übergeben: 1. der zu setzenden Wert
Container	Abstrakte Superklasse von →*Beitrag_Container*, →*Betreiber_Container*, →*Mantel_Container*, →*Werbe_Container* und →*Profil_Container*.
Daten	Abstrakte Superklasse von →*Beitrag*, →*Profil*, →*Werbung* und →*Betreiber*. Instanzen der Klasse *Daten* und deren Kinderklassen sind Bestandteil einer Datenbank.
Datenbank_Server	Datenbank-Abstraktions-Schicht für die Kommunikation zwischen dem POET-ODBMS und den Back-End-Aktionen. **Attribute:** - **betriebs_datenbanken** Liste von Referenzen auf Betriebs-Datenbanken. Eine Betriebs-Datenbank beinhaltet persistente Instanzen der Klassen →*Profil*, →*Werbung*, →*Mantel* und →*Betreiber*. - **beitrags_datenbank** Referenz auf eine Beitrags-Datenbank. Eine Beitrags-Datenbank beinhaltet persistente Instanzen der Klasse →*Beitrag*. **Methoden:** - ***oeffne_db*** Stellt eine Verbindung zur einer Datenbank her. Es gibt zwei Implementationen für diese Methode. Die erste Implementation öffnet die Beitrags-Datenbank und speichert eine Referenz auf die geöffnete Datenbank im Attribut **beitrag_datenbank**. Als Parameter werden übergeben: 1. Datenbank-Host 2. Name der zu öffnenden Datenbank Die zweite Implementation öffnet eine Betreiber-Datenbank und speichert eine Referenz auf die geöffnete Datenbank im Attribut **betreiber_datenbanken**. Als Parameter werden übergeben: 1. Datenbank-Host 2. Name der zu öffnenden Datenbank 3. Benutzername für den Zugriff auf die Datenbank 4. Paßwort für den Benutzernamen - ***schliesse_db*** Schließt die Verbindung zu einer Datenbank. Die Beitrags-Datenbank kann erst dann geschlossen

	werden, wenn alle Betreiber-Datenbanken bereits geschlossen worden sind. Als Parameter werden übergeben: 1. Name der Datenbank - ***transaktion_abbrechen*** Bricht eine Transaktion ab, restauriert den Zustand der betroffen Objekte im Arbeitsspeicher und stellt den vorherigen Transaktions-Kontext wieder her. Als Parameter werden übergeben: 1. Eine Instanz der Klasse →*Transaktion*, die die abzubrechende Transaktion enthält. - ***transaktion_starten*** Startet eine Transaktion und liefert eine Instanz der Klasse *Transaktion* zurück. Die Instanz der Klasse *Transaktion* wird zuvor mit der aktuellen und vorherigen Transaktion gefüllt. Als Parameter werden übergeben: 1. Name der Datenbank, an die die Transaktion gebunden werden soll. - ***transaktion_beenden*** Beendet eine Transaktion und stellt den vorherigen Transaktions-Kontext wieder her. Als Parameter werden übergeben: 1. Eine Instanz der Klasse *Transaktion*, die die zu beendende Transaktion enthält. - ***gib_beitraege*** Liefert eine Menge von Instanzen der Klasse →*Beitrag*. Als Parameter werden übergeben: 1. Instanz der Klasse →*Anfrage_Daten* 2. Instanz der Klasse →*Profil_Container* - ***gib_profile*** Liefert eine Menge von Instanzen der Klasse →*Profil*. Es gibt drei Implementationen dieser Methode. Die erste Implementation liefert die Profile aller Betreiber, die einen bestimmten Benutzernamen enthalten (zum Beispiel alle Gast-Profile eines Betreibers). Als Parameter werden übergeben: 1. Benutzername Die zweite Implementation liefert ein über den Benutzernamen identifiziertes Profil eines Benutzers bei einem Betreiber. Als Parameter werden übergeben: 1. Name des Benutzers 2. Instanz der Klasse *Betreiber_Container* Die dritte Implementation liefert ein über eine Objekt-ID identifiziertes Profil eines Benutzers bei einem Betreiber. Als Parameter werden übergeben: 1. Objekt-ID des Profil-Objektes 2. Instanz der Klasse *Betreiber_Container* - ***gib_betreiber*** Liefert eine Menge von Instanzen der Klasse →*Betreiber*. - ***gib_werbung*** Liefert eine Menge von Instanzen der Klasse

	→*Werbung*. - ***gib_maentel*** Liefert eine Menge von Instanzen der Klasse →*Mantel*. Als Parameter werden übergeben: 1. Instanz der Klasse *Betreiber_Container* - ***speicher_beitrag*** Speichert eine Instanz der Klasse *Beitrag*. Als Parameter werden übergeben: 1. die zu speichernde Instanz der Klasse *Beitrag* 2. der für die Speicherung zu verwendende Tiefen-Modus. - ***speicher_profil*** Speichert eine Instanz der Klasse *Profil*. Als Parameter werden übergeben: 1. die zu speichernde Instanz der Klasse *Profil* 2. der für die Speicherung zu verwendende Tiefen-Modus. - ***speicher_betreiber*** Speichert eine Instanz der Klasse *Betreiber*. Als Parameter werden übergeben: 1. die zu speichernde Instanz der Klasse *Betreiber* 2. der für die Speicherung zu verwendende Tiefen-Modus. - ***speicher_werbung*** Speichert eine Instanz der Klasse *Werbung*. Als Parameter werden übergeben: 1. die zu speichernde Instanz der Klasse *Werbung* 2. der für die Speicherung zu verwendende Tiefen-Modus. - ***speicher_mantel*** Speichert eine Instanz der Klasse *Mantel*. Als Parameter werden übergeben: 1. die zu speichernde Instanz der Klasse *Mantel* 2. der für die Speicherung zu verwendende Tiefen-Modus.
Detail_Text	Klasse zur Beschreibung eines →*Beitrages*. **Attribute:** - **text** Beschreibender Text zu einem Beitrag. - **quelle** Quelle des beschreibenden Textes (zum Beispiel „ifl“, für Internationales Filmlexikon). **Methoden:** - **gib_quelle** Liefert den Wert des Attributs **quelle**. - **setze_quelle** Verändert den Wert des Attributs **quelle**. Als Parameter werden übergeben: 1. der zu setzende Wert - **gib_text** Liefert den Wert des Attributs **text**. - **setze_text** Verändert den Wert des Attributs **text**. Als Parameter werden übergeben: 1. der zu setzende Wert.
Dokument	Oberklasse von →*HTML_Kneter* und →*PDF_Kneter*. Abgeschlossenes Dokument, das sich aus

	→*Mantel_Container*, →*Betreiber_Container*, →*Profil_Container*, →*Werbe_Container* und/oder →*Beitrag_Container* zusammensetzt und an einen →*Server* übermittelt wird. Es gibt verschiedene Formate für *Dokumente*, mindestens aber PDF und HTML. **Attribute:** - **profil_container** Instanz der Klasse *Profil_Container*. - **betreiber_container** Instanz der Klasse *Betreiber_Container*. - **mantel_container** Instanz der Klasse *Mantel_Container*. - **beitrag_container** Instanz der Klasse *Beitrag_Container*. - **werbe_container** Instanz der Klasse *Werbe_Container*. **Methoden:** - ***init_redirect*** Generiert eine Umleitung (Redirect) über den HTTP-Header „Location“. Als Parameter werden übergeben: 1. Die Adresse, auf die der Redirect vorgenommen werden soll.
Eingabe	„Zwischenklasse“ (kein Interface) zwischen →*Server* und →*Dokument*. Erhält und analysiert alle von einem Server übermittelten Daten, um eine den Anfrage-Daten entsprechende Generierung eines *Dokument*s anzustoßen. Klasse, die alle vom Benutzer an das System übermittelte Kontroll- und Dateneingaben [Stary96] verarbeitet. **Attribute:** - **anfrage_daten** Instanz der Klasse →*Anfrage_Daten* für die strukturierte Speicherung der von *Server* gelieferten Daten. - **profil_container** Instanz der Klasse →*Profil_Container* für die Zwischenspeicherung der von →*Eingabe_Server* gelieferten Instanz der Klasse *Profil_Container*. - **betreiber_container** Instanz der Klasse →*Betreiber_Container* für die Zwischenspeicherung der von *Eingabe_Server* gelieferten Instanz der Klasse *Betreiber_Container*. - **werbe_container** Instanz der Klasse →*Werbe_Container* für die Zwischenspeicherung der von *Eingabe_Server* gelieferten Instanz der Klasse *Werbe_Container*. **Methoden:** - ***parse_request*** strukturiert die von *Server* gelieferten Daten und speichert sie im Attribut **anfrage_daten**. - ***pruefe_session_id*** überprüft die Gültigkeit einer Session-ID. - ***generiere_session_id*** generiert eine Gast-Session-ID.

	- ***pruefe_profil*** prüft, ob das Gast-Profil vom *Eingabe_Server* angefordert werden muß. - ***pruefe_werbung*** prüft, ob Werbung vom *Eingabe_Server* angefordert werden muß.
Eingabe_Server	Stellt →*Eingabe* auf Anfrage die Container-Klassen →*Betreiber_Container*, →*Werbe_Container* und →*Profil_Container* zur Verfügung. Inhalt eines vom *Eingabe_Server* gelieferten *Profil_Container*s ist eine Instanz der Klasse →*Profil* die das Gast-Profil umfaßt. Ein *Werbe_Container* beinhaltet alle Werbungen eines →*Betreiber*s. Ein *Betreiber_Container* umfaßt genau eine Instanz der Klasse *Betreiber*. Die jeweiligen Container beinhalten immer eine Kopie der betreffenden Klassen. Bei Systemstart wird eine Instanz der Klasse *Eingabe_Server* initialisiert, die wiederum die drei Instanzen der Container-Klassen initialisiert. Anschließend ruft *Eingabe_Server* die Methode *fuell_dich* der drei Container-Klassen auf. Die zum Füllen der Container-Klassen aufgebaute Verbindung zur Datenbank über zu den Container-Klassen gehörenden Back-End-Aktionen bleibt bis zur Terminierung des Publikationssystems erhalten. Dies geschieht, um einen wiederholten Anmeldevorgang beim ODBMS zu verhindern. **Attribute:** - **profil_container** Instanz der Klasse →*Profil_Container*. Enthält alle in der Datenbank gespeicherten Instanzen der Gast-Profile aller *Betreiber*. - **betreiber_container** Instanz der Klasse →*Betreiber_Container*. Enthält alle in der Datenbank gespeicherten Instanzen der Klasse *Betreiber*. - **werbe_container** Instanz der Klasse →*Werbe_Container*. Enthält alle in der Datenbank gespeicherten Instanzen der Klasse *Werbung*. **Methoden:** - ***gib_betreiber_container*** liefert eine Instanz der Klasse *Betreiber_Container* mit einer Kopie des *Betreiber*-Objektes. Als Parameter werden übergeben: 1. Der Identifizierer des gewünschten Betreibers - ***gib_profil_container*** liefert eine Instanz der Klasse *Profil_Container* mit einer Kopie des *Profil*-Objektes. Als Parameter werden übergeben: 1. eine Instanz der Klasse *Betreiber_Container* - ***gib_werbe_container*** liefert eine Instanz der Klasse *Werbe_Container* mit einer Kopie des *Werbung*-Objektes. Als Parameter werden übergeben:

	1. eine Instanz der Klasse *Betreiber_Container*
E-Mail_Client	Abstrakte Klasse, die während der Analyse eingeführt wurde. Beschreibt eine Client-Software zur Kommunikation eines →*Benutzer*s mit dem System über →*Mail_Server*.
Fernbedienung	Speichert die vom Benutzer angegebene Belegung seiner Fernbedienung. Dabei wird jeder Instanz der Klasse Fernbedienung ein Sender und eine dazugehörige Tastennummer zugeordnet. **Attribute:** - **sender** Referenz auf ein Sender-Objekt. - **tasten_nummer** Die Nummer der Taste auf der Fernbedienung des Abonnenten, mit der der Sender erreicht wird. **Methoden:** - ***gib_tasten_nummer*** Liefert den Wert des Attributs **tasten_nummer**. - ***gib_sender*** Liefert den Wert des Attributs **sender**. - ***setze_tasten_nummer*** verändert den Wert des Attributs **tasten_nummer**. - ***setze_sender*** verändert den Wert des Attributs **sender**.
HTML_Kneter	Kindklasse von →*Dokument*, die für die Generierung von HTML-Dokumenten zuständig ist. **Attribute:** - **anfrage_daten** Instanz der Klasse →*Anfrage_Daten*. Beinhaltet die von →*Server* übermittelten und von →*Eingabe* strukturierten Anfragedaten. - **profil_container** Instanz der Klasse →*Profil_Container*. Beinhaltet entweder das von →*Eingabe_Server* an *Eingabe* gelieferte Gast-Profil oder das über die Methode *Profil_Container*.***fuell_dich*** aus der Datenbank angeforderte Benutzer-Profil. - **beitrag_container** Instanz der Klasse →*Beitrag_Container*. Beinhaltet Instanzen der Klasse →*Beitrag*, die über die Methode *Beitrag_Container*.***fuell_dich*** von der Datenbank angeforderten Beiträge zur Generierung eines HTML-Dokumentes. - **werbe_container** Instanz der Klasse →*Werbe_Container*. Beinhaltet die von →*Eingabe_Server* an *Eingabe* gelieferten *Werbung*-Objekte. - **mantel_container** Instanz der Klasse →*Mantel_Container*. Beinhaltet eine Instanz der Klasse →*Mantel*, den über die Methode

	Mantel_Container.***fuell_dich*** von der Datenbank angeforderten redaktionellen Mantel zur Generierung eines HTML-Dokumentes. - **betreiber_container** Instanz der Klasse →*Betreiber_Container*. Beinhaltet das von →*Eingabe_Server* an *Eingabe* gelieferte *Betreiber*-Objekt. **Methoden:** - ***init_html_beitraege*** Generiert HTML-Dokumente, die Beitrags-Informationen enthalten. Als Parameter werden übergeben: 1. Typ des zu generierenden HTML-Dokumentes (zum Beispiel die TV-Programmübersicht). 2. Eine Instanz der Klasse Anfrage_Daten. 3. Eine Instanz der Klasse Betreiber_Container. 4. Eine Instanz der Klasse Werbe_Container. 5. Eine Instanz der Klasse Profil_Container, sofern es sich um ein Gast-Dokument handelt. Die Berücksichtigung dieses Parameters erlaubt die Überladung der Methode ***init_html_beitraege***. Eine Implementation generiert dann Gast-Dokumente. Die andere Implementation generiert Abonnenten-Dokumente. - ***init_html_profil*** Generiert HTML-Dokumente, die Profil-Informationen enthalten. Als Parameter werden übergeben: 1. Typ des zu generierenden HTML-Dokumentes (zum Beispiel die Sender-Auswahl). 2. Eine Instanz der Klasse Anfrage_Daten. 3. Eine Instanz der Klasse Betreiber_Container. 4. Eine Instanz der Klasse Werbe_Container. - ***init_html_fehler*** Generiert HTML-Dokumente, die Fehler-Informationen enthalten. Als Parameter werden übergeben: 1. Nummer des Fehlers - ***pruefe_uebersicht_vorhanden*** Prüft ob ein angefordertes Gast-Dokument bereits im Vorwege generiert und innerhalb der Verzeichnisstruktur gespeichert worden ist. Als Parameter werden übergeben. 1. Typ des Dokumentes. - ***init_redirect*** Generiert eine Umleitung (Redirect) über den HTTP-Header „Location“. Als Parameter werden übergeben: 1. Die Adresse, auf die der Redirect vorgenommen werden soll.

Import	Eine Klasse, die Dateien mit →*Beiträgen* verschiedener →*Lieferanten* einliest und anschließend in eine Datenbank importiert. **Attribute:** - **import_verzeichnis** Verzeichnis, in dem sich die zu importierenden Dateien eines →*Lieferanten* befinden. **Methoden:** - ***dateien_einlesen*** Öffnet das Verzeichnis, das im Attribut **import_verzeichnis** hinterlegt ist. Anschließend ließt diese Methode die Namen der Dateien aus dem Verzeichnis der Reihe nach aus und initialisiert jeweils eine Instanz der Klasse →*Import_Datei* pro Datei-Name.
Import_Datei	Repräsentiert eine von einem →*Lieferanten* gelieferte Datei mit Beitrags-Infomationen. Jede Datei enthält alle Sendungsbeiträge genau eines Senders für genau einen Tag. Die erste Zeile der Datei enthält Informationen über den ausstrahlenden Sender sowie über das Sendedatum aller in der Datei enthaltenen Beiträge. Alle folgenden Zeilen enthalten die eigentlichen Beitrags-Informationen. **Attribute:** - **datei_name** Name der zu importierenden Datei. **Methoden:** - ***oeffne_datei*** Öffnet die Datei, deren Pfad und Name im Attribut **datei_name** gespeichert ist. - ***schliesse_datei*** Schließt die durch die Methode ***oeffne_datei*** geöffnete Datei. - ***lese_erste_zeile*** Liest die erste Zeile der Datei. - ***interpretiere_zeile*** Interpretiert alle Zeilen der Datei mit Ausnahme der ersten Zeile. **Anmerkung:** Das Verhalten dieser Klasse hängt extrem vom Format der gelieferten Beitrags-Dateien ab. Beispiel-Dateien, die während der Implementierung vorlagen, entsprachen dem Eingangs beschriebenen Aufbau.
Kompression	Komprimiert von →*PDF_Kneter* erzeugte PDF-Dokumente. **Methoden:** - ***komprimiere_pdf*** Komprimiert PDF-Dokumente.
Lieferant	Anbieter von TV-Programminformationen. Ein Lieferant liefert Dateien die Beitragsinformationen enthalten (→*Import_Datei*).
Mail_Server	Abstrakte Klasse, die eine Server-Software zur Kommunikation des Systems mit einem →*Benutzer* über einen

	→*E-Mail_Client* repräsentiert.
Mantel	Redaktionelles Rahmenangebot eines →*Betreibers*. Der *Mantel* kann nicht vom Profil eines Benutzers beeinflußt werden. **Attribute:** - **vorspann** Vorspann des redaktionellen Mantels. Beinhaltet zum Beispiel redaktionelle Texte zu einem in der personalisierten Programm-Zeitschrift erwähnten Beitrag. Der Vorspann wird zwischen Titelblatt und dem personalisierten Anteil der Zeitschrift eingefügt. - **nachspann** Nachspann des redaktionellen Mantels. Beinhaltet, wie der Vorspann, zum Beispiel redaktionelle Texte zu einem in der personalisierten Programm-Zeitschrift erwähnten Beitrag. Der Nachspann bildet das Ende der personalisierten Zeitschrift. - **titel_blatt** Titelblatt der personalisierten Zeitschrift. **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Mantel*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Mantel_Aktion_FE*.***gib_wert***, →*Mantel_Container*.***gib_wert***, →*Mantel_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Mantel*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte *PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert (→*Mantel_Aktion_FE*.***setze_wert***, →*Mantel_Container*.***setze_wert***)
Mantel_Aktion_BE	Eine in sich geschlossene Anforderung von Operationen auf →*Mantel*. *Mantel_Aktion_BE* füllt eine Instanz der Klasse *Mantel_Container* mit *Mantel*-Objekten aus der Betriebs-Datenbank. Beauftragt →*Datenbank_Server* mit der Speicherung von Änderungen. Wird ausschließlich von einer Instanz der Klasse →*Mantel_Container* verwendet. **Attribute:**

	- **mantel_container** Referenz auf eine Instanz der Klasse →*Mantel_Container* **Methoden:** - ***fuell_mich*** Veranlaßt das Laden von Instanzen der Klasse *Mantel* aus der Datenbank. Anschließend werden die geladenen *Mantel*-Objekte an den *Mantel_Container* angehängt (Attribut *Mantel_Container*.**mantel_set**). Als Parameter werden übergeben: 1. Eine Instanz der Klasse *Betreiber* (→*Mantel_Container*.***fuell_dich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Mantel*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Mantel_Container*.***gib_wert***, →*Mantel_Aktion_FE*.***gib_wert***, →*Mantel*.***gib_wert***) - ***speicher_mantel*** Veranlaßt das Speichern einer Instanz der Klasse →*Mantel* in die Datenbank. Als Parameter werden übergeben: 1. Nummer des zu speichernden *Mantels*.
Mantel_Aktion_FE	Eine in sich geschlossene Anforderung von Operationen auf →*Mantel*. Liest und ändert Inhalte von Instanzen der Klasse *Mantel*, die von →*Mantel_Aktion_BE* an →*Mantel_Container* angehängt wurden. **Attribute:** - **mantel_container** Referenz auf eine Instanz der Klasse *Mantel_Container* **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Mantel*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Mantel_Container*.***gib_wert***, →*Mantel*.***gib_wert***, →*Mantel_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Mantel*. Für diese Methode gibt es mindestens 2

	Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert (→*Mantel_Container*.***setze_wert***, →*Mantel*.***setze_wert***)
Mantel_Container	Container-Klasse für →*Mantel*. Dient als Interface zwischen *Mantel*-Aktionen und *Mantel*. **Attribute:** - **mantel_set** Zeigerstruktur mit Instanzen der Klasse *Mantel*. **Methoden:** - ***fuell_dich*** Veranlaßt das Laden von Instanzen der Klasse *Mantel* aus der Datenbank. Anschließend werden die geladenen *Mantel*-Objekte in **mantel_set** gespeichert. Als Parameter werden übergeben: 1. Eine Instanz der Klasse *Betreiber_Container*. (→*Mantel_Aktion_BE*.***fuell_mich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Mantel*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird 3. Nummer des gewünschten Mantels (→*Mantel_Aktion_FE*.***gib_wert***, →*Mantel*.***gib_wert***, →*Mantel_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Mantel*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. der Name des zu setzenden Wertes 2. der zu setzende Wert 3. Nummer des gewünschten Mantels (→*Mantel_Aktion_FE*.***setze_wert***, →*Mantel*.***setze_wert***) - ***speicher_mantel*** Veranlaßt das Speichern einer Instanz der Klasse *Mantel* in die Datenbank. Als Parameter werden übergeben: 1. Nummer des zu speichernden *Mantels*.
NameWertPaar	Klasse zur Speicherung von Name/Wert-Paaren in einer verketteten Liste von *NameWertPaar*-Objekten. Wird von

	→*Anfrage_Daten* verwendet, um die während einer Anfrage anfallenden Client-Informationen zu speichern. **Attribute:** - **name** - **wert** - **next** Referenz auf eine Instanz der Klasse *NameWertPaar*. **Methoden:** - ***speicher_nw_paar*** Speichert ein Name/Wert-Paar. Beim ersten Aufruf der Methode ***speicher_nw_paar*** werden die Attribute **name** und **wert** mit den übergebenen Werten gefüllt. Der nächste Aufruf der Methode veranlaßt *NameWertPaar* eine neue Instanz der Klasse *NameWertPaar* zu initialisieren und dessen Methode ***speicher_nw_paar*** aufzurufen. Anschließend wird diese neue Instanz im Attribut **next** aggregiert. Jeder weitere Aufruf der Methode ***speicher_nw_paar*** wird bis zum letzten *NameWertPaar*-Objekt durchgereicht, und dort entsprechend verarbeitet. Als Parameter werden übergeben: 1. Name des Name/Wert-Paares. 2. Wert des Name/Wert-Paares. - ***gib_wert*** Liefert den Wert eines Name/Wert-Paares. Für diese Methode existieren zwei Implementationen. Die erste Implementation referenziert das gewünschte Name/Wert-Paar über den als Parameter übergebenen Namen. Als Parameter werden also übergeben: 1. Name des gewünschten Name/Wert-Paares Die zweite Implementation der Methode referenziert das gewünschte Name/Wert-Paar über Position des Paares in der verketteten Liste. Als Parameter werden übergeben: 1. Position des gewünschten Name/Wert-Paares. ***gib_name*** Liefert den Namen eines Name/Wert-Paares. Die Methode referenziert das gewünschte Name/Wert-Paar über Position des Paares in der verketteten Liste. Als Parameter werden übergeben: 1. Position des gewünschten Name/Wert-Paares.
PDF_Kneter	Kindklasse von →*Dokument*, die für die Generierung von PDF-Dokumenten zuständig ist. Verwendet eine PDF-Vorlage, die Bestandteil eines Betreiber-Objektes ist. Ferner verwendet die Klasse *PDF_Kneter* Funktionalitäten der Klasse Kompression, um ein generiertes PDF-Dokument zu komprimieren. **Attribute:** - **anfrage_daten** Instanz der Klasse →*Anfrage_Daten*. Beinhaltet die von →*Server* übermittelten und von

	→*Eingabe* strukturierten Anfragedaten. - **profil_container** Instanz der Klasse →*Profil_Container*. Beinhaltet das über die Methode *Profil_Container*.***fuell_dich*** aus der Datenbank angeforderte Benutzer-Profil. - **beitrag_container** Instanz der Klasse →*Beitrag_Container*. Beinhaltet Instanzen der Klasse →*Beitrag*, die über die Methode *Beitrag_Container*.***fuell_dich*** von der Datenbank angeforderten Beiträge zur Generierung eines PDF-Dokumentes. - **werbe_container** Instanz der Klasse →*Werbe_Container*. Beinhaltet die von →*Eingabe_Server* an *Eingabe* gelieferten *Werbung*-Objekte. - **mantel_container** Instanz der Klasse →*Mantel_Container*. Beinhaltet eine Instanz der Klasse →*Mantel*, den über die Methode *Mantel_Container*.***fuell_dich*** von der Datenbank angeforderten redaktionellen Mantel zur Generierung eines PDF-Dokumentes. - **betreiber_container** Instanz der Klasse →*Betreiber_Container*. Beinhaltet das von →*Eingabe_Server* an *Eingabe* gelieferte *Betreiber*-Objekt. **Methoden:** - ***init_pdf_beiträge*** Generiert PDF-Dokumente, die Beitrags-Informationen enthalten. Als Parameter werden übergeben: 1. Eine Instanz der Klasse *Anfrage_Daten*. 2. Eine Instanz der Klasse *Betreiber_Container*. 3. Eine Instanz der Klasse *Werbe_Container*.
PDFObjekt	Repräsentiert ein PDF-Objekt. **Attribute:** - **nummer** Nummer zur Referenzierung des Objektes innerhalb einer Sektion. - **breite** Feste Breite des PDF-Objektes. - **hoehe** Berechnete Höhe des PDF-Objektes. - **bilder** Legt fest, wie innerhalb eines PDF-Objektes Texte dargestellt werden, wenn innerhalb dieses PDF-Objektes Bilder plaziert werden. Ist dies der Fall, beinhaltet **bilder** den Namen eines Attributs der Klasse →*Beitrag*. Bei der Darstellung des Wertes dieses Attributs, werden die Maße eines eventuell einzubindenden Bildes berücksichtigt. Das heißt, die Breite des Bildes wird von der für die Darstellung des Textes zur Verfügung stehenden Breite

	subtrahiert. Dies betrifft die Darstellung eines Beitragstextes, mit einem rechts oder links daneben liegenden Bild. - **xspacing** Horizontaler Abstand zwischen zwei PDF-Objekten in *q-points*. - **yspacing** Vertikaler Abstand zwischen zwei PDF-Objekten in *q-points*. - **constraint** Instanz der Klasse →*Constraint*. - **absaetze** Liste von Instanzen der Klasse →*Absatz*. - **kette** Instanz der Klasse →*SektionGlied*. **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klassen *PDFObjekt* und *Absatz*. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. - ***setze_wert*** Verändert Inhalte von Attributen der Klassen *PDFObjekt* und *Absatz*. Als Parameter werden übergeben: 1. Name des zu verändernden Wertes. 2. Der zu setzende Wert.
PDFSektion	Kindklasse von →*Sektion*. Unterteilt eine PDF-Vorlage in mehrere voneinander unabhängige Bestandteile (Sektionen). **Attribute:** - **spalten** Anzahl der bei der Generierung zu berücksichtigenden Spalten der *PDFSektion*. - **werbung** Legt fest, ob und wieviele Werbebanner innerhalb einer *PDFSektion* plaziert werden sollen. - **std_platzhalter** Liste von Referenzen auf Instanzen der Klasse →*Platzhalter*. In der aggregierten Klasse *Platzhalter* werden Standard-Werte gespeichert, die innerhalb der gesamten Sektion verwendet werden können. - **pdf_objekte** Liste von Referenzen auf Instanzen der Klasse →*PDFObjekt*. Beinhaltet Referenzen auf alle innerhalb der Sektion verwendete *PDFObjekte*. - **constraint** Referenz auf eine Instanz der Klasse →*Constraint*. - **kette** Referenz auf eine Instanz der Klasse →*SektionGlied*. - **naechste** Referenz auf eine Instanz der Klasse *PDFSektion*. Über dieses Attribut werden die einzelnen Sektionen einer PDF-Vorlage aneinander gereiht. **Methoden:** - ***naechste_sektion*** Liefert die *PDFSektion*, die im Attribut

	naechste gespeichert ist. - ***gib_wert*** Liefert Werte der Attribute der Klassen *PDFSektion*, *Platzhalter*, *PDFObjekt*, *Constraint*, *SektionGlied* und *Absatz*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. - ***setze_wert*** Verändert Werte der Attribute der Klassen *PDFSektion*, *Platzhalter*, *PDFObjekt*, *Constraint*, *SektionGlied* und *Absatz*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu verändernden Inhalte →*PEPText* und int. 1. Name des zu verändernden Wertes. 2. Der zu setzende Wert. - ***gib_naechstes*** Liefert das nächste Objekt des Sektionsbaumes und den Typ des gewählten Objektes. Als Parameter werden übergeben: 1. Referenz auf eine Variable, in die der Typ des gewählten Objektes geschrieben werden soll.
PDFVorlage	Kindklasse von →*Vorlage*. Repräsentiert eine PDF-Vorlage, die von →*PDF_Kneter* zur Generierung eines PDF-Dokumentes verwendet wird. **Attribute:** - **typ** Typ der PDF-Vorlage. Dient zur Unterscheidung verschiedener Vorlagen. - **rand_rechts** Bei der Generierung zu berücksichtigenden rechten Seitenrandes. - **rand_links** Bei der Generierung zu berücksichtigenden linken Seitenrandes. - **rand_oben** Bei der Generierung zu berücksichtigenden oberen Seitenrandes. - **rand_unten** Bei der Generierung zu berücksichtigenden unteren Seitenrandes. - **seiten_breite** Gesamtbreite des zu generierenden PDF-Dokumentes. - **seiten_hoehe** Gesamthöhe des zu generierenden PDF-Dokumentes. - **sektionen** Referenz auf eine Instanz der Klasse →*PDFSektion*. **Methoden:** - ***naechste_sektion*** Liefert die Instanz der Klasse

	PDFSektion, die im Attribut **sektionen** gespeichert ist. - ***gib_wert*** Liefert Werte der Attribute der Klassen *PDFVorlage*, *PDFSektion*, *Platzhalter*, *PDFObjekt*, *Constraint*, *SektionGlied* und *Absatz*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. - ***setze_wert*** Verändert Werte der Attribute der Klassen *PDFVorlage*, *PDFSektion*, *Platzhalter*, *PDFObjekt*, *Constraint*, *SektionGlied* und *Absatz*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu verändernden Inhalte →*PEPText* und int. 1. Name des zu verändernden Wertes. 2. Der zu setzende Wert.
PEPBild	Eine Klasse, die begleitende grafische Beitragsinformationen oder grafische Werbeinformationen repräsentiert. Zusätzlich enthält *PEPBild* einen alternativ anzuzeigenden Text (ALT-Text), das Grafik-Format der Bildinformation (GIF, JPG), eine URL (Unified Ressource Locator), die Abmessungen der grafischen Information (Breite, Höhe in Pixeln), die Auflösung der grafischen Information, eine Bildunterschrift und einen Dateinamen. **Attribute:** - **hoehe** Höhe des Bildes in Pixeln. - **breite** Breite des Bildes in Pixeln. - **datei_pfad** Pfad und Dateiname des im Dateisystem abgelegten Bildes. - **datei_url** URL, die beim „Anklicken“ des Bildes im →*Browser* angezeigt werden soll. - **alt_text** Alternativ anzuzeigender Text. Dieser Wert wird benötigt, wenn ein Benutzer in den Einstellungen seines Browsers die Anzeige von Grafiken deaktiviert hat. - **bild_unterschrift** Bildunterschrift. - **url_target** Name des Target-Frames, in dem die der Inhalt der URL beim Anklicken des Bildes dargestellt werden soll. **Methoden:** - ***gib_wert*** Liefert Werte der Attribute der Klasse *PEPBild*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter

	werden übergeben: 1. Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. - ***setze_wert*** Verändert Werte der Attribute der Klasse *PEPBild*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu verändernden Inhalte →*PEPText* und int. 1. Name des zu verändernden Wertes. 2. Der zu setzende Wert.
PEPText	Textuelle Informationen und String-Handling Routinen. Für die Implementation wurde die Klasse PtString der POET-API verwendet.
PEPDatumZeit	Klasse für die Verwendung von Datums- und Uhrzeit-Informationen. Für die Implementation wurde die Klasse PtDateTime der POET-API verwendet.
PEPZeit	Klasse für die Verwendung von Uhrzeit-Informationen. Für die Implementation wurde die Klasse PtTime der POET-API verwendet.
Platzhalter	Klasse zur Speicherung von Name/Wert-Paaren. Wird von →*Sektion* und →*PDFSektion* verwendet, um Standard-Werte zu speichern, die innerhalb einer Sektion verwendet werden können. **Attribute:** - **name** - **wert** **Methoden:** - ***gib_name*** Liefert den Wert des Attributs **name**. - ***gib_wert*** Liefert den Wert des Attributs **wert**. - ***setze_name*** Verändert den Wert des Attributs **name**. Als Parameter werden übergeben: 1. Der zu setzende Name - ***setze_wert*** Verändert den Wert des Attributs **wert**. Als Parameter werden übergeben: 1. Der zu setzende Wert
Profil	Individuelle Einstellungen eines →*Benutzer*s bei einem →*Betreiber*. Diese Einstellungen werden bei der Generierung von HTML- und PDF-Dokumenten verwendet (→*HTML_Kneter*, →*PDF_Kneter*). Profile werden einem bestimmten →*Betreiber* zugeordnet. Für jeden Betreiber existiert genau ein Gast-Profil und beliebig viele Abonnenten-Profile. **Attribute:**

	- **email_empfang** Hier legt ein Abonnent fest, in welchem Zeitintervall seine personalisierte Zeitschrift per E-Mail an ihn verschickt werden soll (täglich, wöchentlich, zweiwöchentlich). - **vps** Legt fest, ob VPS-Zeiten bei der Generierung von Dokumenten berücksichtigt und damit im Dokument angezeigt werden sollen. - **showview** Legt fest, ob der ShowView-Code eines Beitrages bei der Generierung von Dokumenten berücksichtigt und damit im Dokument angezeigt werden sollen. - **session_id** Beinhaltet die letzte Session-ID, die dem Benutzer vom System zugeteilt worden ist. Betrifft ausschließlich Abonnenten-Profile. - **bezahlt_bis** Bezahlter Abonnement-Zeitraum. Betrifft ausschließlich Abonnenten-Profile. - **benutzer_name** Name des Profils. Es gibt Abonnenten-Profile, für die der Abonnent den Benutzernamen frei wählen kann, sofern dieser Name noch nicht an einen anderen Abonnenten vergeben worden ist. Zusätzlich gibt es ein Profil mit dem Benutzernamen „gast“, das das Gast-Profil eines Betreibers repräsentiert. - **passwort** Frei wählbares Paßwort eines Abonnenten für sein Profil oder Paßwort eines Betreibers für sein Gast-Profil. - **letzter_besuch** Datum und Uhrzeit des letzten Zugriffs eines Abonnenten auf das System. - **bundesland** Hier kann ein Abonnent festlegen, welche regionalen Sendungen bevorzugt angezeigt werden sollen. Laufen zum Beispiel auf dem Sender N3 parallel die Beiträge „Schleswig-Holstein Magazin“, „Hamburg Journal“ und „Hallo Niedersachsen“, so kann ein Abonnent über den Wert dieses Attributes festlegen, daß beispielsweise nur der Beitrag „Hamburg Journal“ bei der Generierung eines Dokumentes berücksichtigt wird. - **email_adresse** Hier legt ein Abonnent fest, an welche E-Mail Adresse seine personalisierte Zeitschrift verschickt werden soll. - **wunschzeiten** Liste von Instanzen der Klasse →*PEPZeit*. Hier legt ein Abonnent fest, welche Uhrzeiten an welchen Wochentagen zur Auswahl von Beiträgen für die Generierung der personalisierten Zeitschrift verwendet werden sollen (zum Beispiel Wochentags nur zwischen 20 und 24 Uhr, am Wochenende den ganzen Tag). - **schluesselworte** Liste von Instanzen der Klasse →*PEPText*. Hier kann ein Abonnent besondere Interessen über Schlüsselworte definieren (zum Beispiel

	„Synchronschwimmen“, „Neufundlandschoner“ oder „Informatik“). Schlüsselworte haben eine höhere Ordnung als alle anderen Einstellungen des Profils. Das heißt, Beiträge, die diesen Schlüsselworten entsprechen, werden auch dann bei der Generierung berücksichtigt, wenn sie außerhalb der Wunschzeiten-, Sparten-, Charakter- und Sender-Einstellungen liegen. - **sparte** Liste von Referenzen auf Instanzen der Klasse →*Sparte*. Beinhaltet eine Liste von Sparten, von denen ein Beitrag mindestens eine beinhalten muß, damit er bei der Generierung von Dokumenten berücksichtigt wird (zum Beispiel „Spielfilm“). - **charakter** Liste von Referenzen auf Instanzen der Klasse →*Charakter*. Beinhaltet eine Liste von Charakteren, von denen ein Beitrag mindestens einen beinhalten muß, damit er bei der Generierung von Dokumenten berücksichtigt wird (zum Beispiel „Komödie“). - **sender** Liste von Referenzen auf Instanzen der Klasse →*Sender*. Beinhaltet eine Liste von Sendern, so daß nur Beiträge der angegebenen Sender für die Generierung von Dokumenten berücksichtigt werden (zum Beispiel: „ARD“, „TRTint“). - **fernbedienung** Liste von Instanzen der Klasse →*Fernbedienung*. Hier legt ein Abonnent fest, auf welcher Taste seiner Fernbedienung welcher Sender gespeichert ist. Diese Information wird bei der Generierung von Dokumenten verwendet, um für jeden Sender die entsprechende Tastennummer anzeigen zu können (zum Beispiel: „ARD=1“, „TRTint=38“). **Methoden:** - ***gib_wert*** Liefert Werte der Attribute der Klassen *Profil*, *PEPZeit*, *PEPText*, *Sparte*, *Charakter*, *Sender* und *Fernbedienung*. Für diese Methode gibt es mindestens 3 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte *PEPText*, *PEPZeit* und int. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. - ***setze_wert*** Verändert Werte der Attribute der Klassen *Profil*, *PEPZeit*, *PEPText*, *Sparte*, *Charakter*, *Sender* und *Fernbedienung*. Für diese Methode gibt es mindestens 3 Implementationen; jeweils eine Implementation für die zu verändernden Inhalte *PEPText*, *PEPZeit* und int. 1. Name des zu verändernden Wertes. 2. Der zu setzende Wert.

	Anmerkung: Das Lesen und Schreiben der Attribute aggregierter Klassen geschieht über die jeweiligen Methoden der aggregierten Klasse.
Profil_Aktion_BE	Eine in sich geschlossene Anforderung von Operationen auf →*Profil*. *Profil_Aktion_BE* füllt eine Instanz der Klasse *Profil_Container* mit *Profil*-Objekten aus der Betriebs-Datenbank. Beauftragt →*Datenbank_Server* mit der Speicherung von Änderungen. Wird ausschließlich von einer Instanz der Klasse →*Profil_Container* verwendet. **Attribute:** - **profil_container** Referenz auf eine Instanz der Klasse *Profil_Container* **Methoden:** - ***fuell_mich*** Veranlaßt das Laden von Instanzen der Klasse *Profil* aus der Datenbank. Anschließend werden die geladenen *Profil*-Objekte an den →*Profil_Container* angehängt (Attribut *Profil_Container*.**profil_set**). Als Parameter werden übergeben: 1. Instanz der Klasse →*Anfrage_Daten* 2. Instanz der Klasse →*Betreiber_Container* 3. Flag, ob eine Änderung erfolgen soll (→*Profil_Container*.***fuell_dich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Profil*. Für diese Methode gibt es mindestens 3 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText*, →*PEPZeit* und int. Als Parameter werden übergeben: 1. Der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Profil_Container*.***gib_wert***, →*Profil_Aktion_FE*.***gib_wert***, →*Profil*.***gib_wert***) - ***speicher_profil*** Veranlaßt das Speichern einer Instanz der Klasse →*Profil* in die Datenbank. Als Parameter werden übergeben: 1. Nummer des zu speichernden Profils - ***wechsel_in_abo_session_id*** Erzeugt für einen Abonnenten, der sich beim System PEP anmeldet, aus der Gast-Session-ID eine Abonnenten-Session-ID und speichert diese im Attribut *Profil*.**session_id** über die Methode *Profil*.***setze_wert***. Als Parameter werden übergeben: 1. Die zu verwendende Gast-Session-ID - ***vergleiche_ids*** Vergleicht die in der Instanz der Klasse

	Anfrage_Daten enthaltene Session-ID mit der Session-ID des angeforderten Profils (Zugriffskontrollmechanismus). - ***vergleiche_zeiten*** Prüft, ob der letzte Zugriff des Abonnenten innerhalb der letzten 30 Minuten erfolgt ist (Zugriffskontrollmechanismus). Wenn die Prüfung ein positives Ergebnis liefert, wird das aktuelle Datum und die aktuelle Uhrzeit in das Attribut *Profil*.**letzter_besuch** eingetragen. - ***vergleiche_pw*** Vergleicht bei der Anmeldung eines Abonnenten das in der Instanz der Klasse *Anfrage_Daten* enthaltene Paßwort mit dem Paßwort des angeforderten Profils (Zugriffskontrollmechanismus). Wenn die Prüfung ein positives Ergebnis liefert und der Abonnement-Zeitraum nicht abgelaufen ist, wird die Methode ***wechsel_in_abo_session_id*** aufgerufen. - ***extrahiere_oid*** Extrahiert die Objekt-ID eines anzufordernden Profils aus der Abonnenten-Session-ID. Mit Hilfe dieser Objekt-ID kann ein Benutzer-Profil vom Datenbank-Server direkt referenziert werden. Unter anderem stellt diese Methode auch einen Zugriffskontrollmechanismus dar. Kann in der Datenbank kein der Objekt-ID entsprechendes Profil-Objekt aufgefunden werden, dann handelt es sich um einen potentiellen Angriff auf das System.
Profil_Aktion_FE	Eine in sich geschlossene Anforderung von Operationen auf →*Profil*. Liest und ändert Inhalte von Instanzen der Klasse *Profil*, die von →*Profil_Aktion_BE* an →*Profil_Container* angehängt wurden. **Attribute:** - **profil_container** Referenz auf eine Instanz der Klasse *Profil_Container* **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Profil*. Für diese Methode gibt es mindestens 3 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText*, →*PEPZeit* und int. Als Parameter werden übergeben: 1. Der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Profil_Container*.***gib_wert***, →*Profil*.***gib_wert***, →*Profil_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Profil*. Für diese Methode gibt es mindestens 3 Implementationen; jeweils eine Implementation für die zu

	ermittelnden Inhalte →*PEPText*, →*PEPZeit* und int. Als Parameter werden übergeben: 1. Der Name des zu setzenden Wertes 2. Der zu setzende Wert (→*Profil_Container*.***setze_wert***, →*Profil*.***setze_wert***) - ***gib_profil_kopie*** liefert eine Instanz der Klasse →*Profil*. Diese Instanz ist eine Kopie eines an *Profil_Container* angehängten *Profil*-Objektes. Als Parameter werden übergeben: 1. Nummer des gewünschten Profils Diese Methode wird ausschließlich von der Klasse →*Eingabe_Server* verwendet, um bei einer Anforderung eines *Profil_Containers* durch →*Eingabe* eine Kopie des geforderten *Profils* zu liefern. - ***aender_profil*** Ändert Werte von Attributen der Klasse *Profil*. Als Parameter werden übergeben: 1. Eine Instanz der Klasse →*Anfrage_Daten*.
Profil_Container	Container-Klasse für →*Profil*. Dient als Interface zwischen Profil-Aktionen und *Profil*. **Attribute:** - **profil_set** Zeigerstruktur mit Instanzen der Klasse *Profil*. **Methoden:** - ***fuell_dich*** Veranlaßt das Laden von Instanzen der Klasse *Profil* aus der Datenbank. Anschließend werden die geladenen *Profil*-Objekte in **profil_set** gespeichert. Als Parameter werden übergeben: 1. Eine Instanz der Klasse →*Anfrage_Daten*. 2. Eine Instanz der Klasse →*Betreiber_Container*. (→*Profil_Aktion_BE*.***fuell_mich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Profil*. Für diese Methode gibt es mindestens 3 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText*, →*PEPZeit* und int. Als Parameter werden übergeben: 1. Der Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. 3. Nummer des gewünschten Profils (→*Profil_Aktion_FE*.***gib_wert***, →*Profil*.***gib_wert***, →*Profil_Aktion_BE*.***gib_wert***) - ***gib_profil_kopie*** Liefert eine Instanz der Klasse *Profil*. Diese Instanz ist eine Kopie eines an *Profil_Container* angehängten *Profil*-Objektes. Als Parameter werden übergeben: 1. Nummer des gewünschten Profils

	Diese Methode wird ausschließlich von der Klasse →*Eingabe_Server* verwendet, um bei einer Anforderung eines *Profil_Containers* durch →*Eingabe*, eine Kopie des geforderten *Profil* zu liefern. (→*Profil_Aktion_FE*.***gib_profil_kopie***) - ***aender_profil*** Ruft die Methode ***aender_profil*** der Klasse →*Profil_Aktion_FE* mit den selben Parametern auf. Als Parameter werden übergeben: 1. Eine Instanz der Klasse →*Anfrage_Daten*. 2. Nummer des gewünschten Profils - ***speicher_profil*** Ruft die Methode ***speicher_profil*** der Klasse →*Profil_Aktion_BE* mit denselben Parametern auf. Als Parameter werden übergeben: 1. Nummer des zu speichernden Profils **Anmerkung:** Die Methode ***setze_wert***, die in allen anderen Container-Klassen verwendet wird, ist für *Profil_Container* nicht erforderlich. Das liegt daran, daß Änderungen an einem *Profil* immer mehrere Werte umfassen. Die Klasse →*Profil_Aktion_FE* nimmt alle Änderungen auf einmal vor (vgl. Methode ***aender_profil*** der Klasse *Profil_Aktion_FE*).
Publishing_System	Abstrakte Klasse, die während der Problemanalyse das gesamte Publikationssystem PEP repräsentiert hat.
Sektion	Unterteilt eine Vorlage in mehrere voneinander unabhängige Bestandteile (Sektionen). **Attribute:** - **std_platzhalter** Liste von Referenzen auf Instanzen der Klasse →*Platzhalter*. In der aggregierten Klasse *Platzhalter* werden Standard-Werte gespeichert, die innerhalb der gesamten Sektion verwendet werden können. - **constraint** Referenz auf eine Instanz der Klasse →*Constraint*. - **kette** Referenz auf eine Instanz der Klasse →*SektionGlied*. - **naechste** Referenz auf eine Instanz der Klasse *Sektion*. Über dieses Attribut werden die einzelnen Sektionen einer Vorlage aneinander gereiht. **Methoden:** - ***naechste_sektion*** Liefert die *Sektion*, die im Attribut **naechste** gespeichert ist. - ***gib_wert*** Liefert Werte der Attribute der Klassen *Sektion*, *Platzhalter*, *Constraint* und *SektionGlied*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils

	eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. - ***setze_wert*** Verändert Werte der Attribute der Klassen *Sektion*, *Platzhalter*, *Constraint* und *SektionGlied*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu verändernden Inhalte →*PEPText* und int. 1. Name des zu verändernden Wertes. 2. Der zu setzende Wert. - ***gib_naechstes*** Liefert das nächste Objekt des Sektionsbaumes und den Typ des gewählten Objektes. Als Parameter werden übergeben: 1. Referenz auf eine Variable, in die der Typ des gewählten Objektes geschrieben werden soll.
SektionGlied	Klasse, die den eigentlichen Inhalt einer →*Sektion* durch eine verkettete Liste von *SektionGlied*-Objekten abbildet. Ein SektionGlied kann vom Typ „platzhalter“ oder „text“ sein. Der Typ „platzhalter“ gibt an, daß der Wert des Attributs **inhalt** einen Wert aus der Datenbank referenziert. Ist ein *SektionGlied* vom Typ „text“, so beinhaltet das Attribut **inhalt** den zur Generierung zu verwendenden Wert. **Attribute:** - **typ** Beinhaltet den Typ des *SektionGliedes*. - **inhalt** Beinhaltet entweder einfachen Text (zum Beispiel „Darsteller: “ oder den Namen eines Attributes der Klassen →*Beitrag*, →*Profil*, →*Werbung*, →*Mantel* oder →*Betreiber* (zum Beispiel *Beitrag*.**darsteller**.**rolle**). - **naechstes** Referenz auf eine Instanz der Klasse *SektionGlied*. Über dieses Attribut wird die verkettete Liste von *SektionGlied*-Objekten realisiert. - **constraint** Referenz auf eine Instanz der Klasse →*Constraint*. **Methoden:** - ***naechstes_glied*** Liefert das nächste Objekt des Sektionsbaumes und den Typ des gewählten Objektes. Als Parameter werden übergeben: 1. Referenz auf eine Variable, in die der Typ des gewählten Objektes geschrieben werden soll. - ***gib_typ*** Liefert den Wert des Attributes **typ**. - ***setze_typ*** Verändert den Wert des Attributes **typ**. Als Parameter werden übergeben: 1. Der zu setzende Typ

	- ***gib_inhalt*** Liefert den Wert des Attributes **inhalt**. - ***setze_inhalt*** Verändert den Wert des Attributes **inhalt**. Als Parameter werden übergeben: 1. Der zu setzende Inhalt
Sender	Klasse, die einen TV-Sender repräsentiert. **Attribute:** - **sortier_reihenfolge** Legt die Stellung des Senders für die Sortierung der *Sender*-Objekte bei der Anforderung aus der Datenbank fest. - **name** Name des Senders (zum Beispiel „ARD“ oder „TRTint“) **Methoden:** - ***gib_name*** Liefert den Wert des Attributes **name**. - ***gib_sortierreihenfolge*** Liefert den Wert des Attributes **sortier_reihenfolge**. - ***setze_name*** Verändert den Wert des Attributes **name**. Als Parameter werden übergeben: 1. Der zu setzende Name - ***setze_sortierreihenfolge*** Verändert den Wert des Attributes **sortier_reihenfolge**. Als Parameter werden übergeben: 1. Der zu setzende Wert
Server	Superklasse von →*Web_Server* und →*Mail_Server*.
Sicherheit	Eine in sich geschlossene Anforderung von Operationen, die sicherheitsrelevante Daten und Datenströme verschlüsseln und entschlüsseln sowie Authentifizierungsmechanismen zur Verfügung stellen. Verschlüsselt und entschlüsselt z.B. Inhalte von →*Profil*.
Sparte	Klasse zur groben Kategorisierung eines →*Beitrages*. Mögliche Werte sind z.B. „Spielfilm“, „Serie“, „Sport“. Die verfeinerte Kategorisierung wird durch die Klasse →*Charakter* vorgenommen. **Attribute:** - **name** Name der Sparte. **Methoden:** - ***gib_name*** Liefert den Wert des Attributes **name**. - ***setze_name*** Verändert den Wert des Attributes **name**. Als Parameter werden übergeben: 1. Der zu setzende Name.
Transaktion	Klasse zur Verwaltung und Wiederherstellung eines POET Transaktions-Kontextes. Die Klasse *Transaktion* ist eine Klasse ohne Methoden und stellt eine Struktur dar, die

	lediglich zwei Referenzen auf POET Transaktions-Objekte (PtTransaction) bereitstellt. Die beiden entsprechenden Attribute der Klasse *Transaktion* (**current** und **previous**) sind als „public“, zumindest aber als „friend“ für die Klasse →*Datenbank_Server*, zu definieren. Der Zugriff auf die POET Transaktionsobjekte erfolgt direkt von der Klasse *Datenbank_Sever* während des Starts, der Beendigung und dem Abbruch einer Transaktion. **Attribute:** - **current** Referenz auf eine Instanz der POET Klasse PtTransaction. Speichert die aktuelle Transaktion - **previous** Referenz auf eine Instanz der POET Klasse PtTransaction. Speichert die vorherige Transaktion **Anmerkung:** Diese Klasse ist während der Implementierung entstanden und somit nicht Bestandteil des Objektmodells.
TV-Programm	Abstrakte Klasse, die während der Problemanalyse die Gesamtheit aller →*Beiträge* repräsentiert hat.
Vorlage	Repräsentiert eine Vorlage die von →*HTML_Kneter* zur Generierung eines HTML-Dokumentes verwendet wird. **Attribute:** - **typ** Typ der HTML-Vorlage. Dient zur Unterscheidung verschiedener Vorlagen. - **sektionen** Referenz auf eine Instanz der Klasse →*Sektion*. **Methoden:** - ***naechste_sektion*** Liefert die Instanz der Klasse *Sektion*, die im Attribut **sektionen** gespeichert ist. - ***gib_wert*** Liefert Werte der Attribute der Klassen *Vorlage*, *Sektion*, *Platzhalter*, *Constraint* und *SektionGlied*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. - ***setze_wert*** Verändert Werte der Attribute der Klassen *Vorlage*, *Sektion*, *Platzhalter*, *Constraint* und *SektionGlied*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu verändernden Inhalte →*PEPText* und int. 1. Name des zu verändernden Wertes. 2. Der zu setzende Wert.
Web_Server	Server-Software zur Kommunikation des Systems mit einem

	→*Benutzer* über einen →*Browser*.
Werbe_Aktion_BE	Eine in sich geschlossene Anforderung von Operationen auf →*Werbung*. *Werbe_Aktion_BE* füllt eine Instanz der Klasse *Werbe_Container* mit *Werbung*-Objekten aus der Betriebs-Datenbank. Beauftragt →*Datenbank_Server* mit der Speicherung von Änderungen. Wird ausschließlich von einer Instanz der Klasse →*Werbe_Container* verwendet. **Attribute:** - **werbe_container** Referenz auf eine Instanz der Klasse *Werbe_Container* **Methoden:** - ***fuell_mich*** Veranlaßt das Laden von Instanzen der Klasse *Werbung* aus der Datenbank. Anschließend werden die geladenen *Werbung*-Objekte an den →*Werbe_Container* angehängt (Attribut *Werbe_Container*.**werbe_set**). Als Parameter werden übergeben: 1. Instanz der Klasse →*Anfrage_Daten* 2. Instanz der Klasse →*Betreiber_Container* 3. Instanz der Klasse fi *Profil_Container* 4. Flag, ob eine Änderung erfolgen soll (→*Werbe_Container*.***fuell_dich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Werbung*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Werbe_Container*.***gib_wert***, →*Werbe_Aktion_FE*.***gib_wert***, →*Werbung*.***gib_wert***) - ***speicher_werbung*** Veranlaßt das Speichern einer Instanz der Klasse →*Werbung* in die Datenbank. Als Parameter werden übergeben: 1. Nummer der zu speichernden Werbung
Werbe_Aktion_FE	Eine in sich geschlossene Anforderung von Operationen auf →*Werbung*. Liest und ändert Inhalte von Instanzen der Klasse *Werbung*, die von →*Werbe_Aktion_BE* an →*Profil_Container* angehängt wurden. **Attribute:** - **werbe_container** Referenz auf eine Instanz der Klasse *Werbe_Container* **Methoden:**

	- ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Werbung*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Der Name des zu liefernden Wertes 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird (→*Werbe_Container*.***gib_wert***, →*Werbung*.***gib_wert***, →*Werbe_Aktion_BE*.***gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Werbung*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Der Name des zu setzenden Wertes 2. Der zu setzende Wert (→*Werbe_Container*.***setze_wert***, →*Werbung*.***setze_wert***) - ***gib_werbe_kopie*** Liefert eine Instanz der Klasse →*Werbung*. Diese Instanz ist eine Kopie eines an *Werbe_Container* angehängten *Werbung*-Objektes. Als Parameter werden übergeben: 1. Nummer der gewünschten Werbung Diese Methode wird ausschließlich von der Klasse →*Eingabe_Server* verwendet, um bei einer Anforderung eines *Werbe_Containers* durch →*Eingabe*, eine Kopie der geforderten *Werbung* zu liefern.
Werbe_Container	Container-Klasse für →*Werbung*. Dient als Interface zwischen Werbe-Aktionen und *Werbung*. **Attribute:** - **werbe_set** Zeigerstruktur mit Instanzen der Klasse *Werbung*. **Methoden:** - ***fuell_dich*** Veranlaßt das Laden von Instanzen der Klasse *Werbung* aus der Datenbank. Anschließend werden die geladenen *Werbung*-Objekte in **werbe_set** gespeichert. Als Parameter werden übergeben: 1. Eine Instanz der Klasse →*Anfrage_Daten*. 2. Eine Instanz der Klasse →*Betreiber_Container*. 3. Eine Instanz der Klasse →*Profil_Container*. 4. Flag, ob eine Änderung erfolgen soll. (→*Werbe_Aktion_BE*.***fuell_mich***) - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Werbung*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu

	ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Der Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. 3. Nummer der gewünschten Werbung (→*Werbe_Aktion_FE.**gib_wert***, →*Werbung.**gib_wert***, →*Werbe_Aktion_BE.**gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Werbung*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Der Name des zu setzenden Wertes. 2. Der zu setzende Wert. 3. Nummer der gewünschten Werbung (→*Werbe_Aktion_FE.**setze_wert***, →*Werbung.**setze_wert***) - ***gib_werbe_kopie*** Liefert eine Instanz der Klasse *Werbung*. Diese Instanz ist eine Kopie eines an *Werbe_Container* angehängten *Werbung*-Objektes. Als Parameter werden übergeben: 1. Nummer der gewünschten Werbung Diese Methode wird ausschließlich von der Klasse →*Eingabe_Server* verwendet, um bei einer Anforderung eines *Werbe_Containers* durch →*Eingabe* eine Kopie der geforderten *Werbung* zu liefern. (→*Werbe_Aktion_FE.**gib_werbe_kopie***) - ***speicher_werbung*** Ruft die Methode ***speicher_werbung*** der Klasse →*Werbe_Aktion_BE* mit denselben Parametern auf. Als Parameter werden übergeben: 1. Nummer des zu speichernden Profils
Werbung	Textuelle und grafische Information eines Werbekunden bei einem →*Betreiber*, die in Abhängigkeit von →*Anfrage_Daten* und/oder eines →*Profil*s in ein →*Dokument* integriert wird. **Attribute:** - **buchungs_zeitraum** Zeitraum, für den der Werbeplatz gebucht wurde. - **schluessel_worte** Liste von Referenzen auf Instanzen der Klasse →*PEPText*. Schlüsselworte, die den Inhalt der Werbung beschreiben. Anhand der Schlüsselworte kann zum Beispiel personalisierte Werbung in Dokumente integriert werden. - **banner** Referenz auf eine Instanz der Klasse →*PEPBild*. Beinhaltet das Werbe-Banner des Werbungtreibenden. - **kunde** Referenz auf eine Instanz der Klasse *PEPText*.

	Beinhaltet den Namen des Werbungtreibenden. **Methoden:** - ***gib_wert*** Liefert Inhalte von Attributen der Klasse *Werbung*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Der Name des zu liefernden Wertes. 2. Referenz auf eine Variable, in die der zu liefernde Wert geschrieben wird. (→*Werbe_Aktion_FE.**gib_wert***, →*Werbe_Container.**gib_wert***, →*Werbe_Aktion_BE.**gib_wert***) - ***setze_wert*** Verändert Inhalte von Attributen der Klasse *Werbung*. Für diese Methode gibt es mindestens 2 Implementationen; jeweils eine Implementation für die zu ermittelnden Inhalte →*PEPText* und int. Als Parameter werden übergeben: 1. Der Name des zu setzenden Wertes. 2. Der zu setzende Wert. (→*Werbe_Aktion_FE.**setze_wert***, →*Werbe_Container.**setze_wert***) **Anmerkung:** Das Lesen und Schreiben der Attribute aggregierter Klassen geschieht über die jeweiligen Methoden der aggregierten Klasse.
Zahlung	Eine in sich geschlossene Anforderung von Operationen, die einen Zahlungsvorgang eines →*Benutzers*, für die Freischaltung eines eine *Zahlung* betreffenden →*Profil*s, ermöglichen. **Methoden:** - ***zahlung_anmelden*** Meldet einen Zahlungsvorgang beim →*Abrechnung_Gateway* an. - ***zahlung_vornehmen*** Gibt dem →*Abrechnungsanbieter* über *Abrechnung_Gateway* den Auftrag, die angemeldete Zahlung durchzuführen. **Anmerkung:** Eine genaue Beschreibung der Klasse *Zahlung* ist erst dann möglich, wenn eine Entscheidung für einen bestimmten Abrechnungsanbieter getroffen worden ist.

Tabelle 9: Data-Dictionary

9.8 PEP Mini Transfer Protokoll

Das im Rahmen dieser Diplomarbeit entwickelte PEP Mini Transfer Protokoll (PMTP) arbeitet paketorientiert. Die Größe der Pakete ist dabei variabel. Lediglich der Kopf, eines Paketes, der die Anzahl der innerhalb eines Pakets übertragenen Datenbytes und einen Befehl zur Verarbeitung der Daten enthält, hat eine fest Länge von 4 Byte. Abbildung 56 zeigt die Struktur eines PMTP Paketes. Das Feld für die Angabe der Anzahl der Datenbytes umfaßt dabei insgesamt zwei Byte und bildet den Beginn des Kopfes. Die Größe dieses Feldes (2 Byte) beschränkt die Anzahl der Datenbytes auf maximal 65535 Byte. Der Befehl für die Verarbeitung der übertragenen Daten umfaßt ein Byte, womit ein Befehlsumfang von insgesamt 256 verschiedenen Kommandos realisiert werden kann.

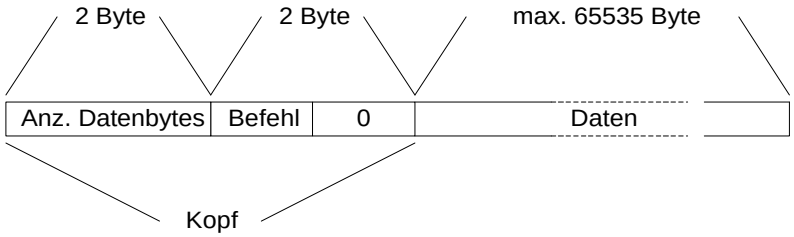


Abbildung 56: PMTP Paket

Von diesen 256 möglichen Befehlen werden zur Zeit 12 Befehle, die in Tabelle 10 erläutert sind, zur Kommunikation zur Verfügung gestellt.

Befehl	**Wert**	**Beschreibung**
ADR	97	Das Paket enthält eine Mail-Adresse. Dieser Befehl wird bei der regelmäßigen Generierung der personalisierten Zeitschriften verwendet. Er markiert die Übermittlung der E-Mail-Adresse, an die die nachfolgenden Daten übermittelt werden sollen. Voraussetzung für das Senden dieses Befehls ist, daß sich der Client im Display-Modus befindet (→DSPL) und das einmalige vorherige Senden des Befehls →MAIL vom Client an den Server. Senderichtung: Server → Client
CLSK	99	Schließe Socket-Verbindung Wird gesendet, wenn die Verbindung zum Server bzw. zum Client geschlossen werden soll. Dieser Befehl kann zu jedem Zeitpunkt der Client-Server Kommunikation gesendet werden. Dieser Befehl hat die sofortige Schließung der Socket-Verbindung zur Folge. Senderichtung: Server → Client, Client → Server
DSPL	100	Stelle die Daten der folgenden Pakete dar. Fordert den Client auf, die nachfolgenden Pakete zur Übertragung an den entsprechenden Server zu verwenden. Der Client befindet sich nach diesem Befehl im Display-Modus. Durch den Befehl →IGN wird der Display-Modus verlassen. Senderichtung: Server → Client

EOT	101	Ende der Übertragung. Wird gesendet, wenn die Übertragung der Daten beendet ist. Dieser Befehl veranlaßt einerseits den Server mit der Generierung zu beginnen. Andererseits teilt das Senden dieses Befehls dem Client mit, daß die Übertragung des/der generierten Dokumentes/Dokumente abgeschlossen ist. Nach dem Senden dieses Befehls sendet der Server den Befehl →CLSK um die Socket-Verbindung zu schließen. Nach der Übertragung dieses Befehls vom Client an den Server wird üblicherweise vom Server der Befehl →DSPL gesendet, um den Client in den Display-Modus zu bringen. Senderichtung: Server → Client, Client → Server
GEN	103	Generiere mit allen folgenden Paketen. Fordert den Server auf, mit den folgenden Paketen das angeforderte Dokument zu generieren. Der Server befindet sich nach diesem Befehl im Generierungs-Modus. Durch den Befehl →IGN wird der Generierungs-Modus verlassen. Durch das erneute Senden des Befehls GEN wird der Server wieder in den Generierungs-Modus gebracht. Senderichtung: Client → Server
IGN	105	Ignoriere die folgenden Pakete. Beendet Display- bzw. Generierungs-Modus. (→DSPL, →GEN) Senderichtung: Server → Client, Client → Server
MAIL	109	Generiere alle Abonnenten-Dokumente für Mail-Versand. Veranlaßt den Server, alle personalisierten Zeitschriften zu generieren. Der Server erwartet nach dem Senden dieses Befehls das Senden des Befehls →NXT um mit der Generierung des ersten Dokumentes zu beginnen. Senderichtung: Client → Server
NXT	110	Übertrage nächstes Dokument. Veranlaßt den Server, das nächste Dokument zu übertragen. Voraussetzung für das Senden dieses Befehls ist die vorherige einmalige Übertragung des Befehls →MAIL. Senderichtung: Server → Client, Client → Server
TM	116	Übertragung von Daten Fordert einerseits den Server auf, die Daten dieses Paketes ein Dokument zu generieren, wenn der Server im Generierungs-Modus ist. Voraussetzung für das Senden dieses Befehls vom Client ist die vorherige einmalige Übertragung des Befehls →GEN. Andererseits wird der Client aufgefordert, die Daten dieses Paketes an den entsprechenden Server zu übermitteln, wenn der Client im Display-Modus ist. Voraussetzung für das Senden dieses Befehls vom Server ist die vorherige einmalige Übertragung des Befehls →DSPL. Senderichtung: Server → Client, Client → Server
EOD	120	Ende des aktuellen Dokumentes. Teilt dem Client mit, daß die Übertragung des aktuellen Dokumentes

		beendet ist. Der Client kann nun mit →NXT das nächste Dokument anfordern. oder mit →CLSK die Socket-Verbindung schließen. Senderichtung: Server → Client
SEC	130	Sichere Übertragung durch Verschlüsselung der Daten. Die in den nachfolgend übertragenen Paketen enthaltenen Daten sind verschlüsselt. Die sichere Übertragung kann durch den Befehl →ESEC aufgehoben werden. Senderichtung: Server → Client, Client → Server
ESEC	131	Ende der sicheren Übertragung. Voraussetzung für das Senden dieses Befehls vom Server ist die vorherige einmalige Übertragung des Befehls →SEC. Senderichtung: Server → Client, Client → Server

Tabelle 10: PMTP Befehle

9.9 Software

	Visio	Object Domain	WithClass	Rational Rose	Together/Professional	Paradigm Plus	OODesign[3]
Version	**4.5**	**1.19**	**2.5**	**3.0**	**2.5**	**3.5**	**2.3.0.3**
Hersteller	Visio Corporation	Object Domain Systems	Microgold	Rational Software Corporation	Object International Software GmbH	Platinum Technology, Inc	Prof. Kim
Methodologien							
UML	ja	nein	nein	ja	ja	ja	nein
Booch	ja	ja	ja	ja	nein	ja	nein
OMT	ja	ja	ja	ja	ja	ja	ja
Use Case	ja	nein	nein	ja	ja	ja	nein
Code-Generierung							
C++	nein	ja[1]	ja	ja	ja	ja	ja
Java	nein	ja[1]	ja	ja	ja	ja	ja
Vollständigkeit[2]	nein	nein	ja	ja	ja	ja	
Export	OLE	Bitmap	nein	OLE	Bitmap	OLE	
Bedienbarkeit	++	+	-	+	--	+	
Preis pro Lizenz	DM 349,-	DM 168,30	DM 331,50	DM 9.974,-	DM 4.950,-	DM 3.900,-	Freeware

[1] Über Scriptsprache TCL programmierbar [2] Vollständige Umsetzung der OMT-Notation [3] Nur eingeschränkte Angaben möglich, da die Software nicht installierbar war. Informationen aus [URL-OOC97]

Tabelle 11: OO Case-Tools

9.10 Beispiel für den Inhalt eines Fernsehbeitrages

FELDNAME	BESCHREIBUNG	BEISPIEL
SenderName		ARD
Datum		07.07.97
Anfangszeit		20.00
Endezeit		21.45
Dauer		105 min
VPS	VPS-Zeit	20.00
ShowView	ShowView-Nr	123-456-78
Sparte	Sparte des Filmes: Spielfilm, Sport, Politik, etc	SPI (=Spielfilm)
Titel	Sendungstitel	12 Uhr mittags
Untertitel	Untertitel z.B. bei Talkshows	
Originaltitel	bei fremdsprachlichen Originalen	High noon
Genre	Genre-Bezeichnung: Western, Krimi, etc.	Western
Land		USA
Jahr	Drehjahr	1970
Drehbuch		
Regie		
Rolle 1	Name der wichtigsten Rolle	Marlowe
Darsteller 1	Name des Schauspielers der wichtigsten Rolle	H. Bogart
Rolle 2		
Darsteller 2		
Rolle 3		
Darsteller 3		
Weitere Darsteller		
Interpret		
Gaststar		
Produzent		
Moderator		
Details	DetailText - kurze Sendungszusammenfassung	Der arbeitslose Detektiv M. brauchte Geld. Er...
Stereo	Stereo Ja/Nein	Nein
SW	Schwarz-Weiß Ja/Nein	Ja
Untertitel für Hörgesch.		Nein
Zweikanalton	ZKT Ja/Nein	Nein
Wiederholung	WH JA/Nein	Nein
Variabel	Sonstiges	Start dieser Sendung kann sich wegen der Tennis-WM verzögern

Tabelle 12: Beispiel für den Inhalt eines Fernsehbeitrages

9.11 Erklärung der Verfasser

Hiermit erklären wir, daß wir die Diplomarbeit selbständig verfaßt und nur die angegeben Quellen und Hilfsmittel verwendet haben.

Hamburg den, 29.12.1998

Jan Krogmann Nils Münch