Diplomarbeit

Realisierung von Webapplikationen auf Basis von Content Management Systemen

Implementierung eines Rahmenwerks am Beispiel von ASP.NET und CoreMedia SCI

Vorgelegt von:

Dennis Homann

Gärtnerstraße 92b, 20253 Hamburg

August 2004

Betreut durch:
Prof. Dr. Joachim W. Schmidt
Prof. Dr. Ralf Möller
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

TUHH
Technische Universität Hamburg-Harburg

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Eines der Hauptprobleme beim Betrieb klassischer Websites ist die Pflege ihrer Inhalte. Die ursprünglichen dateisystembasierten Websites mussten typischerweise manuell gepflegt werden, wobei wenig Unterstützung durch Werkzeuge geboten wurde. Aufgrund der Komplexität der Verweise zwischen Webseiten sowie der Verweise auf eingebettete Ressourcen wie z. B. Bilder lassen sich insbesondere große Mengen von Inhalten auf diese Weise nicht mit vertretbarem Aufwand aktuell und konsistent halten. Insbesondere ist es aufwendig, die Konsistenz der Verweise sicherzustellen, da die Beziehungen zwischen Seiten nicht explizit bekannt sind. Auch die gleichzeitige und verteilte Bearbeitung durch mehrere Benutzer und die Wiederverwendung von Inhalten auf mehreren Seiten bereiten Probleme.

Im Zuge der Verbreitung des WWW wurden daher neue Systeme entwickelt, welche die Pflege solcher großen Websites erlauben. Diese *Content Management Systeme* (CMS) lösen die meisten der oben genannten Probleme durch die Verwaltung der Inhalte in einem *content repository*. Ein *content repository* verwaltet Inhalte als versionierte Entitäten mit expliziten Verweisen und kann so die referentielle Integrität sicherstellen. Webseiten werden über eine Auslieferungskomponente dynamisch aus diesen Entitäten berechnet, wodurch auf diese Weise eine Entität konsistent auf unterschiedlichen Seiten dargestellt werden kann.

Probleme

Es ist zu unterscheiden zwischen seiten- und inhaltsorientierten CMS. Während seitenorientierte Systeme eine feste Bindung von Inhalten an Darstellungsinformationen vorsehen, verwalten inhaltsorientierte Systeme die Entitäten vollkommen unabhängig vom späteren Ausgabeformat, dem Seitenlayout und der Ausgabetechnologie. Letztere können als allgemeiner betrachtet werden, da sie Lösungen für eine umfassendere Menge von Problemen anbieten. Insbesondere die Trennung von Inhalten und Darstellung erleichtert die Wiederverwendung von Inhalten in verschiedensten Formen, ohne Mehrarbeit für den Anwendungsentwickler. Gleichzeitig bedeutet diese Freiheit aber auch mehr Verantwortung, da das Verknüpfen der Inhalte mit Darstellungsinformationen häufig der Anwendung überlassen bleibt, die dabei vom CMS unterschiedlich gut unterstützt wird. Dabei müssen unter anderem die folgenden Probleme gelöst werden:

Abbildung der Entitäten auf Anwendungsobjekte Wegen der Unabhängigkeit des Entitätsmodells im CMS von der Darstellung in einer Webapplikation werden die Inhalte typischerweise zunächst auf Anwendungsobjekte abgebildet. Anwendungsobjekte sind in diesem Fall Objekte, die eine Bedeutung in der Domäne der Anwendung haben. Je nach Art der Anwendung könnten dies

textliche Artikel, Bilder, Preislisten, Produkte, Personen oder Ähnliches sein.

Effizienz Eine wichtige Eigenschaft von Webanwendungen ist eine möglichst kurze Antwortzeit bei der Verarbeitung von Anfragen. Die Generierung der Ausgabe aus Inhalten im CMS darf daher keine langen Bearbeitungszeiten zur Folge haben. Effizientes Caching verhindert häufiges Laden und Transformieren der Entitäten.

Aktualität Änderungen an den Entitäten im CMS müssen die zeitnahe Aktualisierung der erzeugten Webseiten nach sich ziehen. Mit einer Änderung müssen somit alle Objekte und Seiten invalidiert werden, die von dieser Entität abhängen und möglicherweise im Cache liegen.

Personalisierung Webanwendungen sind üblicherweise interaktiv und personalisiert, d. h. die Ausgabe der Seite ist abhängig vom Kontext der Anfrage. Da dieser Kontext dynamisch ist, muss ein Kompromiss zwischen Caching und dynamischer Generierung gefunden werden.

Wiederverwendbarkeit von Softwarekomponenten Die Wiederverwendbarkeit von Softwarekomponenten ist auf drei Ebenen wünschenswert: Zunächst sollten Lösungen für die oben genannten Probleme in Form von Klassenbibliotheken zur Verfügung gestellt werden, um sie nicht für jede Anwendung neu implementieren zu müssen. Des weiteren sollte der Anwendungsentwickler in der Lage sein, Teile seiner Anwendungslogik wiederzuverwenden, um z. B. statt einer interaktiven Webanwendung einen Web Service auf Basis des gleichen Entitätsmodells und der gleichen Inhalte zur Verfügung zu stellen. Außerdem sollte es möglich sein, allgemeine Funktionen wie beispielsweise die Behandlung mehrsprachiger Seiten in mehreren Anwendungen verwenden zu können.

1.1 Ziel und Aufbau der Arbeit

Ziel dieser Arbeit ist es, ein Rahmenwerk zu implementieren, das einem Anwendungsentwickler Lösungen für die oben genannten Probleme bietet. Der Vorteil der „freien“ Implementierung soll erhalten bleiben, gleichzeitig sollen aber einfach zu verwendende Lösungsmuster entwickelt werden, die dem Programmierer die Implementierung der Details abnehmen.

Das Rahmenwerk liefert dazu verschiedene Bausteine, die auch unabhängig voneinander einsetzbar sind:

- eine Schnittstelle zu einem CMS, um auf Entitäten zuzugreifen,
- ein Cache-Modul, das automatisch Abhängigkeiten der Anwendungsobjekte von anderen Anwendungsobjekten oder Entitäten aus dem CMS verfolgt und bei Bedarf Einträge im Cache transitiv invalidiert,
- Personalisierungsfunktionen für Anwendungsobjekte sowie die Aufbereitung von XML-Inhalten aus dem CMS und die Verwaltung von Verweisen.
- Außerdem beinhaltet das Rahmenwerk eine Abstraktion für die Ausgabe von Anwendungsobjekten in unterschiedlichen Präsentations-Frameworks.

Im Rahmen dieser Arbeit wird das vorgestellte Rahmenwerk auf Basis der Microsoft .NET Plattform implementiert. Als Präsentations-Framework dient ASP.NET, welches über eine Adapterschicht an die generische Ausgabekomponente des Rahmenwerks angebunden wird. Für die Caching-Funktionalität existiert bereits eine geeignete Komponente, die wiederverwendet werden kann. Die Implementierung verwendet die „CoreMedia Smart Content Technology“ (SCT) als zugrunde liegendes CMS. Zu diesem Zweck wird eine Schnittstelle für den Zugriff auf CMS-Ressourcen aus .NET-Anwendungen entwickelt und implementiert. Diese kann nicht nur in Webapplikationen, sondern auch in anderen .NET-Anwendungen verwendet werden. Der schreibende Zugriff auf das Content Management System ist nicht Thema der Arbeit.

Es folgt zunächst ein Überblick über Motivation, Aufgaben und Funktionen von Content Management Systemen. Die verwendete Terminologie wird erläutert, und anhand zweier ausgewählter Systeme werden unterschiedliche konzeptuelle Ansätze für die Implementierung dargestellt. Nach einer Zusammenstellung stets wiederkehrender Anforderungen an CMS-basierte Webanwendungen wird ein Framework vorgestellt, das im Rahmen dieser Arbeit implementiert wurde. Der Abschnitt über die Implementierung des Rahmenwerks geht dabei auf ausgewählte Aspekte ein. Schließlich folgt eine Zusammenfassung und eine Bewertung der Ergebnisse.

1.2 Anmerkungen

In dieser Arbeit abgebildete Klassen- und Sequenzdiagramme folgen der Syntax der *Unified Modeling Language* (UML) [34] [18]. Eigenschaften eines Typs des *Common Type System* (CTS) [12] sind dabei in Klassendiagrammen als Operationen mit dem Stereotyp „property“ dargestellt.

Bei einem längeren informationstechnischen Text, in dem unter anderem Softwaresysteme beschrieben werden, lässt sich die Verwendung englischer Begriffe nicht leicht vermeiden. Da sich viele dieser Begriffe auf Typbezeichnungen des implementierten Rahmenwerks beziehen oder es sich um feststehende Ausdrücke handelt, habe ich vielfach auf eine Übersetzung ins Deutsche verzichtet. Englische Begriffe, Definitionen und insbesondere Typ- und Methodenbezeichner sind im Text *kursiv* gesetzt.

2 Content Management Systeme

Seit der Einführung des *world wide web* (WWW) [3] ist die Anzahl seiner Benutzer dramatisch angestiegen. Während zunächst nur einige wissenschaftliche Institute Informationen auf diesem Weg für andere zugänglich machten, gilt dies heute für nahezu jedes Unternehmen, viele öffentlichen Institutionen und Privatpersonen. Die Anzahl der Nutzer des WWW wächst ständig an. So nutzten in 2003 durchschnittlich 50% der EU-Bürger regelmäßig das Internet (2002: 40%), in Deutschland 54% (2002: 49%). Bei den Unternehmen ist die Nutzung des WWW in der EU mit >84% (2002: >77%) noch deutlich weiter verbreitet, in Deutschland sind es sogar >94% (2002: >82%) [35]. Je weiter die Zahl der regelmäßigen Benutzer des WWW wächst, desto mehr lohnt es sich für Unternehmen, Dienste und Produkte auf diesem Wege anzubieten. Das WWW ermöglicht eine kostengünstige, personalisierte Kundenansprache und Informationsbereitstellung, die auch zunehmend von öffentlichen Einrichtungen genutzt wird. So werden z.B. im Rahmen der Initiative „BundOnline 2005“[9] viele Dienste des Bundes über das WWW für den Bürger verfügbar gemacht.

Mit steigendem Informationsangebot erweist sich die Verwaltung solcher Webseiten als zunehmend schwierig. Konzeptuell gesehen ist das WWW ein Netzwerk von Webressourcen, die durch Verweise (Links) lose verbunden [2] und auf unterschiedliche Server verteilt sind. Es kann angenommen werden, dass dieses Netzwerk von Ressourcen aus Partitionen besteht, die jeweils von einer oder mehreren Personen verwaltet werden. Diese Partitionen werden im Folgenden als Domänen bezeichnet. Für den Benutzer stellt sich eine Menge von Webressourcen üblicherweise als eine HTML-Seite (*hypertext markup language*) in seinem Browser dar [37], die wiederum Verweise auf andere Seiten enthält. Eine Seite setzt sich somit aus folgenden Elementen zusammen, die unabhängig voneinander betrachtet werden können:

Inhalt Auf der Seite sichtbare Informationen wie Text, Bilder oder multimediale Inhalte.

Darstellung Anordnung der Inhalte, verwendete Farben, Schriftarten und -größen und Symbole zur Orientierung auf der Seite.

Struktur Verweise (Links), die entweder auf Positionen auf der gleichen Seite, auf Seiten auf dem gleichen Server (interne Links) oder auf Seiten auf einem anderen Server (externe Links) zeigen können.

Bei dieser Unterscheidung kann eine auf der Seite sichtbare Grafik je nach Funktion entweder als Inhalt oder als Darstellungselement gewertet werden. Ebenso werden Verweise für die Einbettung von Grafiken und multimedialen Inhalten in die Seite als Inhalt bzw. als Darstellung gewertet, während Links auf andere oder dieselbe Seite als Strukturelemente betrachtet werden. Abbildung 2.1 zeigt diesen Zusammenhang anhand von drei Seiten, die jeweils aus verschiedenen Webressourcen zusammengesetzt sind.

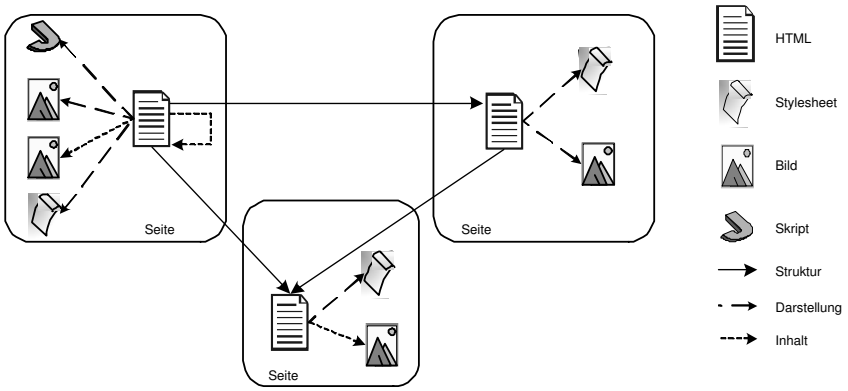

Abbildung 2.1: Zusammenhang zwischen Webressourcen und Seiten

In der ursprünglichen Form des WWW entspricht eine Webressource einer HTML-Datei im Dateisystem des Servers, die sowohl Inhalt als auch Darstellungsformat und Verweise auf andere Ressourcen enthält. Darüber hinaus entspricht der in Verweisen verwendete Name einer solchen Ressource dem Pfad der Datei im Dateisystem. Daraus ergeben sich insbesondere bei manueller Verwaltung der Dateien Probleme, die im Folgenden kurz vorgestellt werden [26].

Referentielle Integrität

Referentielle Integrität ist dann gewährleistet, wenn jede Ressource mindestens so lange existiert, wie es Verweise auf sie gibt. Wegen der dezentralen Verwaltung von Webressourcen kann im WWW keine referentielle Integrität garantiert werden. Verweise auf eine Ressource können unbemerkt angelegt und wieder gelöscht werden. Somit führt jedes Löschen einer Ressource potentiell zu ungültigen Verweisen in Ressourcen anderer Domänen. Innerhalb einer Domäne kann zumindest theoretisch referentielle Integrität für interne Links - also solche, die auf Seiten in der gleichen Domäne zeigen - gewährleistet werden. Dies setzt eine konsequente Verwaltung aller in einer Domäne vorhandenen internen Links voraus und ist damit bei manueller Pflege der Ressourcen kaum zu erreichen. Auch die so bestmöglich erreichbare Integrität kann auf Benutzerseite zu transienten Fehlern führen, da über das zustandslose HTTP-Protokoll Veränderungen an Ressourcen nicht mit der Darstellung auf dem Klienten des Benutzers synchronisiert werden können. Die Gültigkeit externer Links - also solcher, die in andere Domänen verweisen - kann im Allgemeinen nicht garantiert werden.

Durch die direkte Abbildung von Pfaden im Dateisystem auf Namen für Webressourcen kommt es beim Verschieben von Dateien oder ganzen Verzeichnissen nicht nur zu ungültigen Verweisen auf Ressourcen in der betroffenen Menge, sondern es müssen möglicherweise auch relative Verweise [2] innerhalb der verschobenen Ressourcen angepasst werden. Bei einer großen Menge von Dateien kann dies von Hand kaum geleistet werden.

Einheitlichkeit der Darstellung

Üblicherweise sollen Seiten innerhalb einer Domäne in ähnlicher Form dargestellt werden, d.h. das gleiche Farbschema, Layout, die gleiche Schriftart usw. verwenden. Dies ist nicht nur aus ergono-

mischen Gründen für den Benutzer angenehmer, weil er sich auf den Seiten schneller zurechtfindet, sondern ist häufig auch im Sinne des Betreibers, um eine Wiedererkennung der Identität des Unternehmens oder einer Marke zu erreichen. Üblicherweise findet man in verschiedenen Bereichen einer Domäne unterschiedliche Darstellungsformen vor, die dem Benutzer die Orientierung innerhalb der Domäne erleichtern soll. Änderungen an der Seitendarstellung kann daher bei dateibasierten Systemen das Bearbeiten von hunderten oder tausenden von Dateien erfordern. Selbst mit fortgeschrittenen Werkzeugen ist eine solche Änderung sehr aufwändig und fehleranfällig.

Seiten, die ungültige Verweise enthalten oder keine einheitliche Darstellung aufweisen, hinterlassen beim Benutzer den Eindruck mangelnder Qualität oder auch Zuverlässigkeit, der mit dem Betreiber der Seite assoziiert wird. Für kommerzielle Anbieter kann ein solcher Imageverlust am Ende einen Schaden bedeuten, obwohl der ursprüngliche Zweck der Webseite z.B. eine höhere Kundenzufriedenheit durch bessere Information oder mehr Umsatz war. Um sich dem Benutzer als professioneller Anbieter zu präsentieren, liegt es also in seinem Interesse, die oben genannten Probleme zu vermeiden. Die manuelle Verwaltung großer Webseiten mit vielen Inhalten erweist sich aber aus den beschriebenen Gründen als kostenintensiv. Es besteht daher ein Bedarf an Werkzeugen, die diese Probleme lösen.

Im Laufe der 90er Jahre wurden viele Lösungsansätze entwickelt, die sich im Wesentlichen in zwei Klassen unterscheiden lassen: Die eine Klasse versucht, die Verwaltung einer bestehenden Menge von dateibasierten Webseiten zu unterstützen. Zu dieser gehört beispielsweise *MOMSpider* [17]. *MOMSpider* traversiert automatisch die Verweise innerhalb einer Menge von Webseiten und liefert als Ergebnis eine Zusammenfassung über die Qualität der Verweise. Die Ergebnisse einer Domäne können mit anderen geteilt werden, um so domänenübergreifende ungültige Verweise zu identifizieren. Die andere Klasse von Systemen geht einen anderen Weg. Hier werden die Inhalte der Seiten in strukturierter Form in einer Datenbank gespeichert. Informationen über Verweise werden in der Datenbank explizit modelliert und sind so auch getrennt von den Inhalten verfügbar. Für WWW-Klienten wird aus den Datenbankinhalten eine Hypertextansicht erzeugt. Die Speicherung der Inhalte in Datenbanken erlaubt die Nutzung verfügbarer Datenbanktechnologien für die Suche, Indizierung und Integritätsprüfung. Ein Vertreter dieser Gruppe von Systemen ist *Hyper-G* [1]. Das Datenmodell von *Hyper-G* beinhaltet neben Dokumenten auch Cluster und sich überschneidende Sammlungen von Dokumenten, aus denen verschiedene Ansichten für WWW-, Gopher-, oder Hyper-G-Klienten generiert werden.

Heutige Content Management Systeme (CMS) zeichnen sich üblicherweise ebenfalls dadurch aus, dass Inhalte in strukturierter Form gehalten werden, und daraus unterschiedliche Ansichten für verschiedene Klienten erzeugt werden können. Darüber hinaus bieten sie einen formalisierten Produktionsprozess, sind mehrbenutzerfähig, ermöglichen Archivierung und Versionierung von Inhalten und vieles mehr. Ein Content Management System dieser Art ist die „CoreMedia Smart Content Technology“ (CoreMedia SCT), die als Grundlage für diese Arbeit dient.

Die folgenden Abschnitte beschreiben die Aufgaben von Content Management Systemen und wie sich heute verfügbare Systeme voneinander unterscheiden. Dazu werden die Modellierung der Inhalte, die funktionalen Komponenten eines solchen Systems, seine Integrierbarkeit mit anderen Systemen sowie die Art der Aufbereitung von Inhalten untersucht. Schließlich wird erläutert, wie

zwei ausgewählte Produkte diese Aufgaben erfüllen.

2.1 Definition des Begriffs

In dieser Arbeit werden die Begriffe Website und Webanwendung verwendet. Eine Webanwendung ist eine Website, wobei allerdings der interaktive Charakter betont wird. Webanwendungen haben typischerweise hohe dynamische Anteile und bieten dem Nutzer über die bloße Informationsbereitstellung hinaus weitere Dienste an. Sie erfordern außerdem üblicherweise ein Anmelden des Benutzers. Zu dieser Gruppe unter anderem Shop-Systeme und Web-Mail-Dienste. Im Folgenden werden Nutzer einer Website bzw. Webanwendung als Konsumenten bezeichnet, um Verwechslungen mit den Anwendern des CMS auf der Produktionsseite zu vermeiden.

Zum Begriff „Content Management System“ (CMS) gibt es viele unterschiedliche Definitionen, insbesondere im Hinblick auf den Funktionsumfang. Eine ebenso kurze wie allgemeine Formulierung lautet:

> „Ein Content Management System (CMS) ist eine Software zur Verwaltung des Inhalts einer Website.“[1]

Diese Beschreibung trifft auf eine große Menge sehr verschiedener Softwaresysteme zu. Um genauer zu beschreiben, welche Art von Systemen in dieser Arbeit gemeint ist, werden in diesem Abschnitt Gemeinsamkeiten und Unterschiede der als CMS bezeichneten Softwaresysteme erläutert. Häufig findet man auch die Begriffe „Web Content Management System“ (WCMS) und „Redaktionssystem“. In dieser Arbeit werden WCMS und CMS synonym verwendet. Sie sind geeignet, große Websites mit einer größeren Anzahl von Personen zu erstellen und zu pflegen, während Redaktionssysteme üblicherweise einen etwas reduzierten Funktionsumfang bieten und für den Einsatz in kleineren bis mittleren Projekten entworfen wurden.

Die Funktionen eines CMS lassen sich im Allgemeinen in zwei Kategorien aufteilen:

1. Verwaltung von strukturierten und semi-strukturierten Inhalten und
2. Auslieferung von Inhalten unter Einbeziehung von Darstellungsinformationen.

2.1.1 Verwaltung von Inhalten

Art der verwalteten Daten

Anspruchsvolle Webseiten bestehen nicht nur aus mit Auszeichnungen versehenem Text, sondern häufig auch aus Bildern, Video, Musikdateien, Flash-Animationen[2] und anderen Inhalten. Für eine automatisierte Pflege und Verwaltung ist eine möglichst gute Strukturierung der Daten erforderlich. Je strukturierter die Daten vorliegen, desto einfacher lassen sich automatisierte Arbeitsschritte implementieren. Andererseits sollen unstrukturierte Multimediadateien häufig nur im CMS abgelegt, aber nicht durch dieses bearbeitetwerden. Ein CMS sollte daher strukturierte, semi-strukturierte und unstrukturierte Daten verwalten können.

[1] Net-Lexikon: http://www.net-lexikon.de
[2] Macromedia Inc.: http://www.macromedia.com/software/flash

Automatische Pflege

Durch Festlegung einer bestimmten Struktur der Inhalte wird die automatische Pflege erst möglich. Das CMS erzwingt z. B. die Einhaltung bestimmter Bedingungen wie referentieller Integrität bei Verweisen zwischen Inhaltselementen. Auch Transformationen von Daten in andere Formate sind denkbar.

Einfaches und dezentrales Erstellen von Inhalten

Ein CMS erlaubt es auch technisch weniger versierten Anwendern Inhalte zu erstellen und zu pflegen. Für das Erstellen der Inhalte sind keine technischen Kenntnisse erforderlich, wodurch die Publikation und Pflege von Inhalten auf der Website einer deutlich größeren Gruppe von Menschen in einer Organisation offen stehen. Die Zeit von der Idee bis zur Publikation wird verkürzt, da der Zwischenschritt der Bearbeitung durch einen Webadministrator mit HTML-Kenntnissen entfällt. Hinzu kommt eine höhere Effektivität, weil nicht jeder Beitrag zunächst von technischem Personal für die Webpublikation aufbereitet werden muss. Das CMS unterstützt somit die dezentrale Erstellung und Pflege einer Website.

Mehrbenutzerfähigkeit

Insbesondere im Zusammenhang mit dezentraler Wartung ist die Mehrbenutzerfähigkeit eines CMS wichtig. Das System muss entweder verhindern, dass mehrere Anwender gleichzeitig die gleichen Daten bearbeiten, oder entstehende Konflikte auflösen können. Es sind ausserdem verschiedene Berechtigungen für unterschiedliche Benutzergruppen erforderlich, so dass ein Anwender z. B. nur Inhalte in einem bestimmten Bereich bearbeiten kann.

Versionierung

Die im CMS abgelegten Daten sind kontinuierlichen Veränderungen unterworfen. Um Änderungen an Inhalten nachvollziehbar zu machen, legen CMS eine Versionshistorie der gespeicherten Inhalte an. Somit kann ein Anwender auf frühere Versionen eines Elements zugreifen oder nachverfolgen, welche Anwender zu welchen Zeitpunkten Änderungen vorgenommen haben.

Archivierung

Inhalte stellen üblicherweise wertvolle Informationen dar. Dieser Wert soll nicht verloren gehen, auch wenn Inhalte nicht mehr öffentlich auf der Website verfügbar sind. Daher verfügen CMS entweder über ein eigenes Archiv, aus dem gelöschte bzw. aktuell nicht mehr benötigte Inhalte jederzeit wieder hergestellt werden können, oder sie können an externe Archivierungssysteme angebunden werden.

Unterstützung des Produktionsprozesses

Das Erstellen von Inhalten ist ein Produktionsprozess. Am Ende dieses Prozesses steht der Inhalt als Produkt mit einem gewissen Wert für das Unternehmen. Obwohl manchmal Informationen *ad hoc* publiziert werden, hat dieser Produktionsprozess meist mehrere Stufen und lässt sich formal beschreiben. Das CMS verbessert die Qualität der erstellten Daten, indem es die Einhaltung des Prozesses erzwingt. Gleichzeitig befreit es den Anwender davon, die einzelnen Prozessschritte genau zu kennen.

2.1.2 Auslieferung von Inhalten

Die zweite Hauptaufgabe des CMS ist die Auslieferung der erstellten Inhalte in aufbereiteter Form an Konsumenten. Die Aufbereitung der Inhalte ist dabei der eigentlich interessante Vorgang. Sie wird durch mehrere Parameter bestimmt:

1. Der Konsument erwartet die Inhalte in einem bestimmten Format. Im Fall eines Menschen, der mit Hilfe eines Web Browsers Seiten ansieht, sind dies üblicherweise HTML oder PDF (*portable document format*). Konsumenten können Seiten aber über andere Endgeräte wie z.B. Mobiltelefone, *personal digital assistants* (PDA) oder andere abrufen. In diesen Fällen sind anders formatiertes HTML, WML oder andere Ausgabeformate erforderlich. Für automatisierte Prozesse, die Inhalte konsumieren und weiterverarbeiten, sind dagegen strukturiertere Formate wie XML (*extensible markup language*) oder anwendungsspezifische Binärformate besser geeignet.

2. Die Auslieferungskomponente eines CMS transformiert eine Menge von Inhaltselementen in ein für den Konsumenten geeignetes Format. Für jede Art von Inhaltselementen wird diese Transformation beschrieben. Eine Änderung der Transformationsbeschreibung wirkt sich somit auf die Darstellung aller Inhalte des entsprechenden Typs aus. Häufig ist die Transformation durch die Verknüpfung mit Vorlagen implementiert.

3. Je nach Art der Webanwendung ist eine Personalisierung der Ausgabe notwendig. Die Ausgabe wird dabei für jeden Konsumenten speziell angepasst, um z. B. individuelle Angebote anzuzeigen oder Teile auszublenden, für die der Empfänger keine Leseberechtigung hat. In anderen Fällen ist evtl. nur eine Unterscheidung zwischen angemeldeten und anonymen Konsumenten notwendig.

2.2 Modellierung der Inhalte

2.2.1 Interne Struktur

Es gibt verschiedene Ansätze für die Modellierung von Inhalten im CMS. Das Modell muss insbesondere drei Voraussetzungen erfüllen:

Aggregation

Semantisch zusammengehörige Elemente werden zu einem „Inhaltsobjekt“ aggregiert. Diese Aggregation zeichnet sich dadurch aus, dass die Elemente nicht außerhalb eines Inhaltsobjekts existieren können. Beim Löschen eines Inhaltsobjekts werden auch alle von diesem aggregierten Elemente gelöscht. Elemente eines Inhaltsobjekts haben üblicherweise einen festen Typ, um die Forderung nach strukturierten Inhalten zu erfüllen (siehe Abschnitt 2.1.1). Typische Elementtypen sind formatierte Texte, Bilder, einfache Zeichenketten und Kalenderdaten. Obwohl hier der Begriff Inhaltsobjekt verwendet wird, muss das Inhaltsmodell nicht unbedingt objektorientiert sein. In einem objektorientierten Modell gibt es definierte Inhaltstypen, die die Anzahl und Typen ihrer Elemente festlegen. Ein Inhaltsobjekt hat in diesen Modellen einen festen, unveränderbaren Typ. Jedes

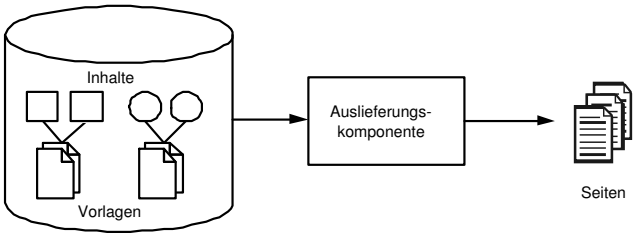

Abbildung 2.2: Schwache Trennung von Inhalt und Darstellung

Inhaltsobjekt im System gehört somit zu einer Klasse, deren Instanzen vom gleichen Typ sind. Weitere Merkmale objektorientierter Inhaltsmodelle sind die Subtypisierung, durch die bestehende Typen um Elemente erweitert werden können, und die Polymorphie.

Referenzen

Das Modell muss Verweise zwischen Inhaltsobjekten ermöglichen, um explizite Beziehungen zwischen ihnen erstellen zu können. Sowohl inhaltliche Verweise als auch strukturelle Verweise müssen im Modell abbildbar sein.

Hierarchische Ablage der Inhalte

Um die Organisation großer Mengen von Inhalten zu erleichtern sollten Inhaltsobjekte in hierarchischen Strukturen ablegbar sein. Anwender sind es sowohl aus dem Dateisystem als auch von den überlicherweise auf Websites verwendeten Navigationsstrukturen gewohnt, Informationen in Bäumen zu organisieren.

2.2.2 Sicht des Anwenders

Bei der Untersuchung verschiedener CMS fällt der Unterschied zwischen einem seitenorientierten und einem inhaltsorientierten Ansatz auf.

Seitenorientierter Ansatz

Bei diesem Ansatz arbeitet der Anwender des CMS zur Verwaltung seiner Inhalte nicht mit Inhaltsobjekten, sondern mit Seiten, die aus einem Inhaltsobjekt und Darstellungsinformationen bestehen. Inhaltsobjekte werden in diesen Modellen immer im Kontext einer bestimmten Darstellung verwaltet. Daraus folgt, dass schon bei der Erstellung eines Inhaltsobjekts eine Darstellungsform festgelegt werden muss, die jedoch nachträglich auch verändert werden kann. Inhaltsobjekte sollten jedoch wiederverwendet werden können, so dass mehrere Seiten den gleichen Inhalt unterschiedlich darstellen. Diese Art der Modellierung ist nur eine „schwache Trennung“ von Inhalt und Darstellung (siehe Abbildung 2.2). „Typo3“[3] und „Microsoft Content Management Server 2002“ (siehe 2.5.1) gehören zu den Systemen, die diesen Ansatz verfolgen.

Inhaltsbasierter Ansatz

Dieser Ansatz kennt im Rahmen der Inhaltsverwaltung keine Verknüpfung mit bestimmten Darstellungsformen. Darstellungsinformationen werden erst zum Zeitpunkt der Auslieferung mit den

[3] Typo3: http://www.typo3.com

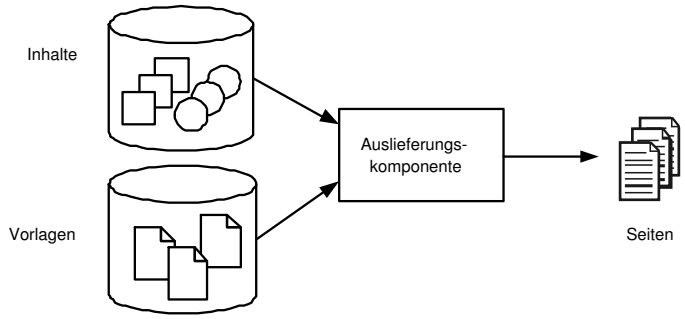

Abbildung 2.3: Strikte Trennung von Inhalt und Darstellung

Inhalten verknüpft. Daher besteht in diesen Fällen häufig weniger Flexibilität auf Seiten des CMS-Anwenders, Einfluss auf die endgültige Darstellung seiner Inhalte zu nehmen. Wenn dies dennoch gewünscht wird, müssen durch den Anwender veränderbare Darstellungsinformationen als Inhalte modelliert und bei der Auslieferung entsprechend berücksichtigt werden. Der eingeschränkte Einfluss des Inhalterstellers auf die endgültige Darstellung ist vorteilhaft, da sich der Ersteller auf die Inhalte selbst konzentrieren muss und außerdem eine einheitlichere Darstellung erreicht wird. Diese „strikte Trennung“ von Inhalt und Darstellung (siehe Abbildung 2.3) findet z. B. bei der CoreMedia SCT Anwendung.

2.3 Funktionale Komponenten

2.3.1 Content Repository

Das sogenannte *content repository* ist der Teil des CMS, in dem die Inhalte abgelegt sind. Es abstrahiert von einer darunter liegenden Persistenzschicht, in der die Daten dauerhaft gespeichert werden. Dabei bietet es üblicherweise eine hierarchische Sicht auf die abgelegten Objekte. Zusätzlich stellt es spezielle Dienste für die Verwaltung der Inhaltsobjekte zur Verfügung, die im Folgenden beschrieben sind.

Versionskontrolle

Ein *content repository* bietet die Möglichkeit, Änderungen an Inhaltsobjekten nachzuvollziehen. Dazu wird der Zustand eines Inhaltsobjekts versioniert. Das System bietet dem Bearbeiter Funktionen, um mehrere Bearbeitungsschritte an einem Inhaltsobjekt zu einer „Änderung“ zusammenzufassen. Durch eine solche Änderung wird eine neue Version erstellt. Eine Version beinhaltet den Zustand des Objekts zu einem bestimmten Zeitpunkt und zusätzlich Informationen über den Anwender, der die Version angelegt hat, sowie den Änderungszeitpunkt. Auf diese Weise können Änderungen über die Lebensdauer eines Inhaltsobjekts nachvollzogen werden. Versionskontrolle ist im Bereich der *Software Configuration Management* (SCM) Systeme weit verbreitet [7]. Die Konzepte sind von dort auf CMS übertragbar, wenn auch häufig nur der eingeschränkte Fall sequentieller Versionierung in CMS verfügbar ist. Hierbei hat jede Version höchstens eine Nachfolgerversion. Eine Änderung fügt eine neue Version zum Ende der Sequenz hinzu.

Synchronisation und Konfliktauflösung

Einer der Vorteile von Content Mangement Systemen ist die dezentrale Erstellung und Pflege von Inhalten. Anwender können dabei unabhängig voneinander an Inhaltsobjekten arbeiten. Es ist die Aufgabe des *content repository*, Konflikte bei der gleichzeitigen Bearbeitung eines Inhaltsobjekts durch mehrere Anwender zu vermeiden oder aufzulösen. Hierbei sind verschiedene Alternativen denkbar: Eine Möglichkeit ist die Verwendung von Sperren, die die Bearbeitung eines Objekts verhindern, solange ein anderer Anwender seine Arbeit an diesem Objekt noch nicht abgeschlossen hat. Eine anderere Alternative erlaubt die gleichzeitige Bearbeitung. Wenn ein Anwender eine Änderung abschließt und eine zwischenzeitliche Änderung durch einen anderen Bearbeiter festgestellt wird, muss er den Konflikt zunächst auflösen, um den Vorgang erfolgreich zu beenden.

Suche

Je mehr Inhalte im *content repository* verwaltet werden, desto wichtiger ist die effiziente Suche nach Inhaltsobjekten. Zu unterscheiden sind hier die strukturierte Suche anhand von Prädikaten auf den Elementen und die allgemeine Volltextsuche über alle Inhalte. Die strukturierte Suche bietet sich an, wenn das System Inhaltstypen mit vorgegebenen Elementen kennt.

Metadatenverzeichnis

Das *content repository* verwaltet nicht nur die Inhalte, sondern auch entsprechende Metadaten. Dazu gehören

- Namen der definierten Inhaltstypen,
- ggf. Typhierarchie für Inhalte,
- Struktur der Inhaltstypen, d. h. aus welchen Elementen besteht ein Objekt eines gegebenen Typs und welche Eigenschaften sind diesen Elementen zugeordnet,
- Informationen über ausgeführte Operationen auf Inhaltsobjekten wie Name des Bearbeiters, Zeitpunkt der Operation und Ähnliches.

Mit Hilfe eines solchen Metadatenverzeichnisses kann ein Klient ein unbekanntes *content repository* erforschen. Es ist auch Voraussetzung für generische Klienten, die auf einem beliebigen *Repository* arbeiten.

2.3.2 Benutzerverwaltung und Rechtevergabe

Einer der Gründe für die Einführung eines CMS ist die dezentrale Erstellung und Pflege von Inhalten durch mehrere Anwender. Um nicht jedem Zugriff das komplette System zu geben, ist eine Benutzerverwaltung und ein flexibles Rechtesystem unerlässlich. Benutzer sollten in Gruppen organisiert werden können, um die Pflege der Zugriffsrechte zu vereinfachen. Zugriffsrechte in hierarchischen Strukturen sind ein im Bereich der Dateisysteme lange bekanntes Problem, für das es verschiedene Lösungswege gibt. So besteht einerseits die Möglichkeit, für die Objekte im *Repository* Listen mit den jeweiligen Zugriffsrechte für bestimmte Benutzer oder Gruppen zu verwalten (sogenannte *access control lists*). Alternativ erhält jeder Benutzer bzw. jede Gruppe eine Liste von Berechtigungen (*capabilities*) auf Objekten. Die beiden Verfahren sind auch kombinierbar [38].

2.3.3 Prozessunterstützung

Unternehmen, die CMS für die Produktion und Pflege ihrer Inhalte einsetzen, haben individuelle Produktions- und Qualitätssicherungsprozesse, die durch die eingesetzten Softwaresysteme möglichst effizient unterstützt werden müssen. Das Ziel sollte dabei sein, den Anwendern die Teilnahme an diesen Prozessen zu erleichtern und ihnen die gesamte Prozessverwaltung abzunehmen. Um ein CMS in ein solches Umfeld zu integrieren, gibt es zwei Möglichkeiten:

- Das CMS bietet eine eigene Komponente für die Vorgangssteuerung. Die Nutzung dieser integrierten Komponente ist sinnvoll, wenn das CMS die wesentliche Softwarekomponente der relevanten Prozesse darstellt oder noch kein Vorgangssteuerungssystem eingesetzt wurde.
- Das CMS bietet die Möglichkeit, mit einem externen Vorgangssteuerungssystem zu kommunizieren und kann so in vorhandene Prozesse eingebunden werden. Wenn bereits ein solches System im Einsatz ist oder die Vorgänge sich über mehrere unterschiedliche Softwaresysteme erstrecken, ist dies möglicherweise der einzige Weg für eine erfolgreiche Integration. Die Bemühungen der Industrie um standardisierte Schnittstellen zwischen den Komponenten für die Vorgangssteuerung werden von der *Workflow Management Coalition* (WfMC)[4] koordiniert.

2.3.4 Auslieferung

Die Aufgabe der Auslieferungskomponente eines CMS ist die Erzeugung von Objekten aus Inhalten im *content repository* und die Übertragung dieser Objekte an einen Konsumenten. Im Allgemeinen wird dabei keine Aussage über die Art der erzeugten Objekte, die Kommunikation mit dem Konsumenten oder die Art des Konsumenten selbst gemacht. Die möglichen Szenarien sind unter anderem:

- Auslieferung von HTML-Seiten über HTTP (*hypertext transfer protocol*) [16] an einen Web Browser als Antwort auf eine Anfrage
- Auslieferung von XML-Dokumenten über HTTP an eine entfernte Anwendung als Antwort auf eine Anfrage
- Senden einer SOAP-Nachricht (*simple object access protocol*) [14] über HTTP als Antwort auf einen Web Service Aufruf [13]
- Schreiben einer Binärdatei in das lokale Dateisystem als Ergebnis einer Änderung im *content repository*

Das letzte Beispiel unterscheidet sich von den anderen insofern, als die Auslieferung nicht als Antwort auf eine Anfrage eines Konsumenten erfolgt, sondern durch eine Änderung im System ausgelöst wird. Diese Arbeit beschränkt sich auf die vom Konsumenten ausgelöste Auslieferung. Um die Beschreibung zu vereinfachen, wird außerdem im Folgenden ohne Einschränkung der Allgemeinheit vom ersten Szenario, der Auslieferung von HTML-Seiten an einen Web Browser, ausgegangen.

[4] Workflow Management Coalition: http://www.wfmc.org

Es folgt eine kurze Übersicht über die verschiedenen Funktionen, die eine Auslieferungskomponente abdecken muss.

Aufbereitung der Inhalte

Die Aufbereitung der Inhalte bei der Auslieferung kann als Transformation eines Graphen betrachtet werden. Der Ausgangsgraph besteht dabei aus den im *content repository* enthaltenen Inhaltsobjekten, wobei Verweise zwischen Objekten und hierarchische Abhängigkeiten die Kanten im Graphen darstellen. Der Konsument sieht dagegen einen Graphen, dessen Knoten Seiten bzw. andere Webressourcen sind und dessen Kanten durch Verweise (*uniform resource locators*, URLs) repräsentiert werden. Als Antwort auf eine Anfrage wird ein Knoten an den Konsumenten zurückgegeben, der Verweise enthalten kann. Diese Transformation wird vom CMS auf ganz unterschiedliche Weise definiert. Ein häufiges Verfahren ist die Verwendung von Vorlagen, die in deklarativer Form das Ausgabeformat beschreiben. Beim Binden einer Vorlage an ein Inhaltsobjekt werden Platzhalter in der Vorlage durch Inhalte ersetzt. Bei diesem Verfahren stellen die Vorlagen die Darstellungsinformationen dar, die durch das Binden mit Inhaltsobjekten zusammengeführt werden.

Personalisierung

Ein wesentliches Merkmal moderner Webanwendungen ist die Personalisierung der Ausgabe. Das Ziel ist es dabei, die Ausgabe nach festgelegten Regeln so anzupassen, dass sie die Identität des Konsumenten berücksichtigt. Anwendungsbeispiele sind eine persönliche Ansprache innerhalb der Webanwendung, Herausfiltern von Inhalten, die der Konsument nicht lesen darf oder das Hervorheben von Informationen, die für den Konsumenten von besonderem Interesse sind.

Eine Voraussetzung für die Personalisierung ist die Feststellung der Identität des Konsumenten bei einer Anfrage. In einigen Fällen reicht es schon aus, mehrere Anfragen einem einzelnen anonymen Konsumenten zuordnen zu können. Meistens ist jedoch bei personalisierten Websites eine Authentisierung erforderlich, um die (bekannte) Identität des Konsumenten sicherzustellen. Dies ist insbesondere bei Webanwendungen der Fall, die Geschäftsprozesse mit rechtlicher Bindung wie z. B. Einkäufe abbilden.

Caching

Ein Webbenutzer erwartet möglichst kurze Ladezeiten für eine Seite. Eine Website muss aber nicht nur für einen Konsumenten eine Antwort schnell ausliefern, sondern eine große Anzahl gleichzeitiger Anforderungen bearbeiten, da sonst die Ladezeiten für den einzelnen Konsumenten sehr lang werden und so die wahrgenommene Qualität abnimmt. Die Schwierigkeit liegt dabei in der Kommunikation der Auslieferungskomponente mit dem *content repository*, insbesondere wenn diese Kommunikation über Prozess- oder sogar Rechnergrenzen hinweg abläuft, da dann die Latenz stark zunimmt. Der Effekt verstärkt sich noch, wenn das Repository dabei auf die Persistenzschicht zugreifen muss. Das übliche Verfahren der Zwischenspeicherung häufig verwendeter Objekte (*Caching*), die sich nur selten ändern, bietet sich daher auch hier an. Diese Technik wird im Bereich der Rechnerarchitekturen und der Betriebssysteme schon sehr lange verwendet [25] [39]. *Caching* ist immer dann sinnvoll, wenn Zugriffe auf einen langsamen Datenspeicher oder über ein Kommunikationsmedium mit großer Latenz oder geringer Bandbreite vermieden werden sollen. CMS

verwenden diese Technik z. B. im *content repository*, in der Auslieferungskomponente oder auf dem Webserver.

Verweise

Die an den Konsumenten ausgelieferte Webressource enthält im Allgemeinen Verweise in Form von URLs auf andere Webressourcen, die erst durch die Transformation in der Auslieferungskomponente aus Inhaltsobjekten entstehen. Eine Webressource existiert insofern nur virtuell, als sie erst zum Zeitpunkt der Anfrage durch Verknüpfung von Inhaltsobjekten mit Darstellungsinformationen erstellt wird.

Um eine Anfrage beantworten zu können, muss jede Auslieferungskomponente daher folgende Schritte ausführen, wobei an geeigneten Stellen Caches verwendet werden:

1. Authentisieren des Konsumenten anhand der Anfrage (optional)
2. Interpretieren der angefragten URL und Bestimmung der auszuführenden Transformation sowie der einzubeziehenden Inhaltsobjekte
3. Ausführen der Transformation auf den Inhaltsobjekten, also beispielsweise Binden einer Vorlage gegen Inhaltsobjekte, dabei:
   - Personalisierung der Ausgabe
   - Konvertieren von Verweisen auf andere Inhaltsobjekte in Verweise auf andere virtuelle Webressourcen
4. Auslieferung des Ergebnisses an den Konsumenten

Je nach Architektur und unterstützten Konzepten werden diese Schritte von verschiedenen CMS unterschiedlich implementiert. Beispiele werden in Abschnitt 2.5 vorgestellt.

2.4 Integration mit Drittsystemen

Ein wichtiges Merkmal bei der Beurteilung eines CMS ist die Offenheit der Architektur. Es sollte möglich sein, das System um eigene Komponenten oder Komponenten anderer Hersteller zu erweitern. Hierzu sind offene Programmierschnittstellen und die Verwendung standardisierter Ein- und Ausgabeformate erforderlich. So sollte es Programmierern ermöglicht werden, in einer verbreiteten Programmiersprache eine Softwarekomponente zu entwickeln, die Operationen auf Objekten im *content repository* ausführt oder Daten importiert bzw. exportiert. Eine offene Architektur erleichtert auch die Anbindung externer Systeme wie Abrechnungs-, Personalisierungs- oder Shop-Systemen.

2.5 Architektur und Inhaltsmodellierung ausgewählter Systeme

Dieser Abschnitt zeigt anhand zweier Produkte wie die im Abschnitt 2.1 vorgestellte Funktionalität in verschiedenen verfügbaren Content Management Systemen implementiert ist. Hierbei werden die unterschiedlichen Ansätze der Systeme deutlich.

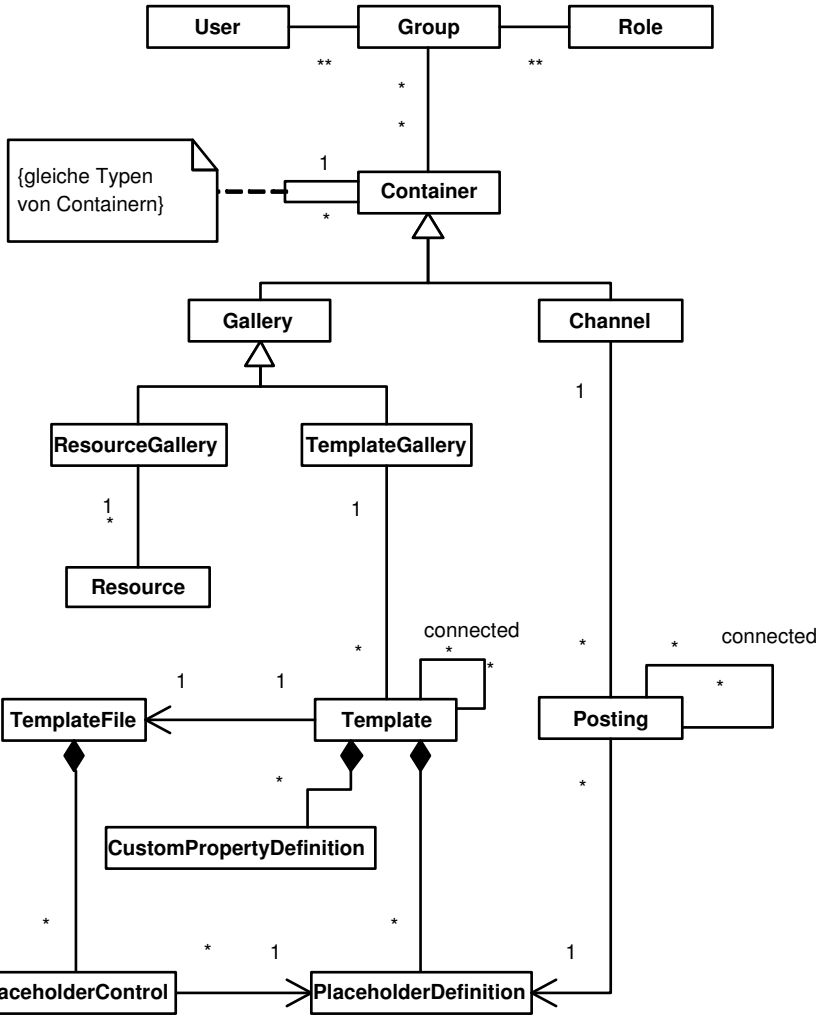

Abbildung 2.4: MS Content Management Server 2002: Daten- und Vorlagenmodell

2.5.1 Microsoft Content Management Server 2002

„Microsoft Content Mangement Server 2002“ (MCMS) verfolgt bei der Modellierung von Daten und Darstellungsinformationen einen seitenorientierten Ansatz. Zwar werden Vorlagen und Inhalte getrennt verwaltet, sie sind jedoch eng aneinander gebunden, so dass man nicht von einer strikten Trennung sprechen kann. Das gemeinsame Modell der Vorlagen und Inhalte ist in Abbildung 2.4 dargestellt.

Aus dem seitenorientierten Ansatz folgt, dass Vorlagen definiert werden müssen, bevor Inhalte im System angelegt werden können. Um dennoch die Inhalte von der Darstellung zu entkoppeln, unterscheidet MCMS zwischen Vorlagen und Vorlagendateien. Es gibt in MCMS einen Baum von *TemplateGalleries*, in dem Vorlagen (*Template*) abgelegt sind. Eine Vorlage enhält eine Reihe von Platzhalterdefinitionen (*PlaceholderDefinition*), in die später bei der Auslieferung Inhalte einfügt werden. Um zu beschreiben wie eine Vorlage dargestellt wird, referenziert sie eine Vorlagendatei

(*TemplateFile*). Diese ist eine ASP.NET-Seite (siehe Seiten 36ff), die die statischen Darstellungsinformationen enthält und einige oder alle Platzhalter des Templates durch *PlaceholderControls* visualisiert. Dazu verweist ein *PlaceholderControl* auf die darzustellende Platzhalterdefinition. Zur Personalisierung und für das Caching von Seiten können alle bekannten ASP.NET-Techniken verwendet werden.

Das System kennt Platzhalterdefinitionen für HTML- und XML-Dokumente sowie für Bild-Dateien. Jede dieser Definitionen ist durch eine Klasse implementiert. Weiter anwendungsspezifische Platzhalterdefintionen können auf Basis der XML-Platzhalterdefinition oder auf Basis einer generischen Definition für Zeichenketten aufgebaut werden. Ein „Platzhalter“ besteht aus einer Platzhalterdefinitionsklasse und allen Subklassen von *PlaceholderControl* im System, die mit dieser Definition kompatibel sind.

Inhaltsobjekte können nur in Form einer Seite (*Posting*) erstellt werden und sind analog zu Vorlagen und *TemplateGalleries* in einer Hierarchie von *Channels* angeordnet. Die Anordnung der *Channels* erlaubt eine logische Gliederung der Inhalte. Beim Erstellen einer Seite wird diese an eine Vorlage gebunden. Dadurch wird sowohl die Darstellung der Seite als auch über die Platzhalterdefinitionen die Struktur der veränderlichen Inhalte vorgegeben. Konzeptuell werden die Inhalte getrennt von der Seite gespeichert und sind nur an die Platzhalterdefinition gebunden, obwohl auf sie nur über eine Seite zugegriffen werden kann.

Diese Trennung erlaubt allerdings eine Wiederverwendung von Inhalten in anderen Seiten. Zu diesem Zweck wird das Konzept „verbundener“ Vorlagen eingeführt. Vorlagen heissen „verbunden“, wenn sie dieselbe Menge von Platzhalterdefinitionen verwenden. Seiten, die auf verbundenen Vorlagen beruhen, können dieselben Inhalte in verschiedenen Formen darstellen und werden dann auch „verbunden“ genannt.

Neben *TemplateGalleries* gibt es auch *ResourceGalleries*, die beliebige binäre Objekte (*Resources*) enthalten. Diese können z.B. für die Darstellung verwendet werden. *Galleries* und *Channels* sind spezielle Arten des Typs *Container*, der einige allgemeine Eigenschaften besitzt: einerseits die bereits erwähnte Möglichkeit zum rekursiven Schachteln von Containern gleichen Typs, andererseits können einem Container Gruppen von Benutzern zugeordnet werden. Eine Gruppe gehört zu einer von acht fest definierten Rollen (*subscriber*, *author*, *editor*, *moderator*, *resource manager*, *template designer*, *channel manager*, *site manager*), die jeweils bestimmte Rechte auf Objekten in MCMS haben. Zusätzlich zu Benutzern können Gruppen auch ganze Windows 2000-Benutzergruppen enthalten, um so die Benutzerverwaltung zu vereinfachen.

Die Publikation von Inhalten besteht aus bis zu 3 Schritten:

1. Ein **Autor** (*author*) erstellt eine neue Seite auf Basis einer Vorlage.

2. Ein **Redakteur** (*editor*) korrigiert die Seite und gibt sie zur Publikation frei oder lehnt sie ab (optional). Die Freigabe durch einen Redakteur erstellt eine neue Version der Seite. Seiten sind linear versioniert.

3. Ein **Moderator** (*moderator*) gibt die Seite zur Publikation frei (optional). Nach der Freigabe ist die Seite für Benutzer mit der Rolle *subscriber* sichtbar.

Die letzten beiden Schritte können entfallen. Anwendungen können vor und nach jedem Schritt dieses Vorgangs anwendungsspezifische Aktionen ausführen.

Das Erstellen, Bearbeiten und Freigeben von Seiten erfolgt in MCMS ausschliesslich über eine Web-Schnittstelle. Das Bearbeiten von Platzhalterinhalten in Seiten wird durch die *Placeholder-Controls* realisiert, die verschiedene Ansichten unterstützen. Während sie in der normalen Ansicht nur ihren Inhalt darstellen, erlauben sie in einer anderen Ansicht auch das Bearbeiten der Inhalte. Vorlagendateien beinhalten ein spezielles *WebAuthorControl*, um zwischen den unterschiedlichen Ansichten zu wechseln. Dieses ist nur für Benutzer mit entsprechenden Rechten sichtbar.

Externe Anwendungen, die nicht im Kontext der *Windows Internet Information Services* (IIS) laufen, können auf MCMS über die sogenannte *Publishing API* zugreifen und am Publikationsprozess teilnehmen oder Objekte erstellen und bearbeiten.

2.5.2 CoreMedia Smart Content Technology 4.2

Die „CoreMedia Smart Content Technology 4.2“ (SCT) verfolgt bei der Modellierung der Inhalte den inhaltsorientierten Ansatz (siehe Abschnitt 2.2.2). Die Inhaltsobjekte werden als „Dokumente“ dargestellt, die in einer Ordnerhierarchie ähnlich einem Dateisystem abgelegt werden. Ordner und Dokumente werden dabei als Ressourcen bezeichnet.

Dokumente haben einen festen Dokumenttyp. Die im System verfügbaren Dokumenttypen werden bei der Einführung des CMS anhand der Bedürfnisse der konkreten Anwendung entworfen. Der Entwurf orientiert sich dabei an der Anwendungsdomäne und nicht an der Struktur der später zu generierenden Seiten. Dies können im Fall einer Zeitung Artikel und Leserbriefe sein, bei einem Fernsehsender dagegen Sendungen, Filmrezensionen und Gewinnspiele. Das Typsystem unterstützt sowohl Vererbung als auch abstrakte Dokumenttypen, die nicht instanziiert werden können.

Ein Dokumenttyp definiert eine feste Menge von Eigenschaften, die jedes Dokument dieses Typs besitzt. Subtypen erben alle Eigenschaften ihrer Supertypen. Es gibt eine feste Menge von Eigenschaftstypen, aus denen beim Entwurf der Dokumenttypen ausgewählt werden kann. Diese sind

- Zeichenketten begrenzter Länge (*StringProperty*),
- ganze Zahlen (*IntegerProperty*),
- Datums- und Zeitwerte (*DateProperty*),
- XML-Dokumente mit Angabe eines XML-Schemas [41] (*XmlProperty*),
- Listen von Verweisen auf Dokumente eines bestimmten Dokumenttyps (*LinkListProperty*) und
- Binärdaten mit Angabe eines MIME-Typs [20] (*BlobProperty*).

Es werden somit sowohl strukturierte als auch unstrukturierte Daten und Verweise auf andere Inhaltsobjekte unterstützt. Dokumente sind linear versioniert, d. h. jedes Dokument hat mindestens eine Version und jede Version hat höchstens eine Nachfolgeversion. Das Metadatenmodell der *CoreMedia SCT* ist in Abbildung 2.5 dargestellt. Durch die Definition der Dokumenttypen oder

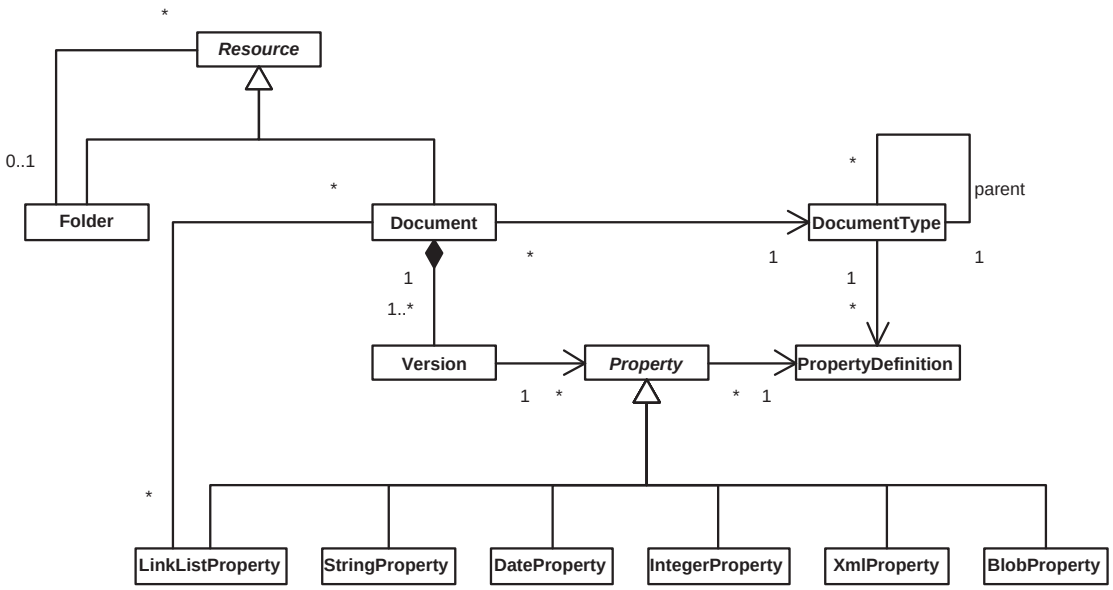

Abbildung 2.5: Metadatenmodell der CoreMedia SCT

das Anlegen von Dokumenten wird keine Entscheidung bezüglich der späteren Darstellung bei der Auslieferung getroffen. Die SCT bietet somit eine strikte Trennung von Inhalt und Darstellung.

Die SCT hat eine offene Architektur, die sich dadurch auszeichnet, dass verschiedene Komponenten über öffentliche Schnittstellen kommunizieren. Die Kommunikationsplattform ist die *Common Object Request Broker Architecture* (CORBA) und das Protokoll für diese Kommunikation das *Internet Inter-ORB Protocol* (IIOP) [33]. Alle Komponenten sind in der Programmiersprache Java [23] implementiert. Die folgenden Absätze beschreiben kurz die einzelnen Komponenten des Systems und wie sie die im vorigen Abschnitt beschriebenen Anforderungen umsetzen.

Die zentrale Komponente ist der *content server*, der das *content repository* implementiert. Sowohl der Zustand aller Dokumente als auch die vorhandenen Benutzer und Benutzergruppen, Mitgliedschaften und Zugriffsrechte sowie die Metadaten über die vorhandenen Dokumenttypen werden von ihm verwaltet. Alle Daten sind in einer relationalen Datenbank persistent gespeichert.

Um Konflikte durch gleichzeitige Bearbeitung eines Dokuments durch mehrere Benutzer zu vermeiden, muss ein Dokument zur Bearbeitung „ausgeliehen“ werden, was nur für einen Benutzer zur Zeit möglich ist. Beim „Zurückgeben“ wird eine neue Version des Dokuments angelegt, die die neuen Eigenschaftswerte beinhaltet. Eine einmal angelegte Version ist damit unveränderbar. Um die Trennung von Produktions- und Livesystem zu unterstützen, werden Dokumente auf den *live server* „publiziert“. Bei diesem Publikationsvorgang, der von einem Benutzer ausgelöst wird, stellt der *content server* sicher, dass alle von den zu publizierenden Dokumentversionen über Verweise erreichbaren anderen Dokumente ebenfalls eine publizierte Version besitzen, so dass es auf dem *live server* keine ungültigen Verweise gibt. Eine Dokumentversion muss außerdem zur Publikation „freigegeben“ sein, bevor sie publiziert werden kann. Die Zustände einer Dokumentversion zeigt das Zustandsdiagramm in Abbildung 2.6.

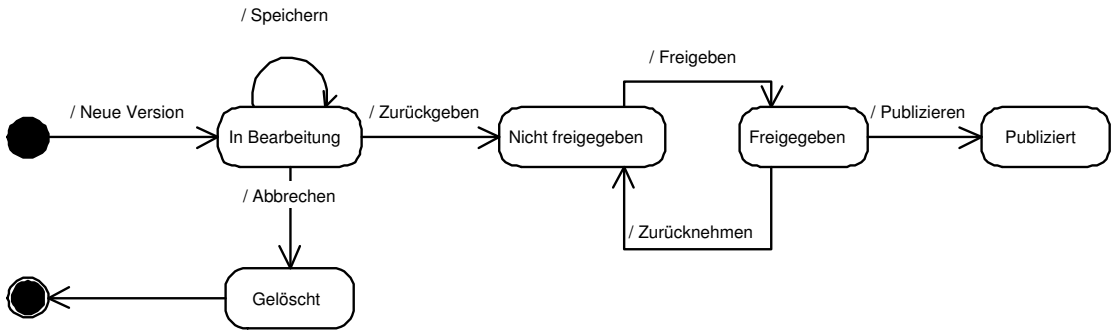

Abbildung 2.6: Zustandsübergänge einer Dokumentversion

Benutzer werden in Gruppen organisiert, wobei auch ein externer Verzeichnisdienst über das *Lightweight Directory Access Protocol* (LDAP) [43] angebunden werden kann. Gruppen haben verschiedene Rechte auf Dokumenten oder Ordnern (lesen, bearbeiten, löschen, freigeben, publizieren, Rechte vergeben), die auf bestimmte Dokumenttypen eingeschränkt werden können.

Des weiteren erlaubt der *content server* die strukturierte Suche nach Dokumenten anhand eines Suchausdrucks. Auch das Empfangen von Benachrichtigungen über Änderungen an Dokumenten und Ordnern ist möglich.

Neben dem *content server* gibt es einen *workflow server*, der komplexe Vorgänge abbilden und steuern kann. Ein Vorgang wird über ein XML-Dokument definiert, wobei beliebige Aktionen in Form von Java-Klassen implementiert und vom *workflow server* ausgeführt werden können.

Das wichtigste Instrument für die Erstellung und Bearbeitung von Inhalten ist der *content editor*, eine Anwendung, die sich als Klient mit dem *content* und dem *workflow server* verbindet und dem Benutzer eine grafische Benutzeroberfläche bietet. Der *content editor* ist flexibel anpassbar, um das Bearbeiten spezieller Dokumenttypen zu erleichtern.

Die SCT beinhaltet zwei Auslieferungskomponenten. Ein sogenannter *proactive delivery server* liefert, ausgelöst durch Änderungen an Inhalten, Objekte an Konsumenten aus und wird hier nicht betrachtet. Ein *active delivery server* (ADS) dagegen generiert nur Ausgaben als Anworten auf Konsumentenanfragen. Der ADS läuft als J2EE Webapplikation [8] in einem Anwendungsserver. Die drei wesentlichen Konzepte des ADS sind:

ResourceUri Eine *ResourceUri* ist eine URL, die den Bezeichner einer Ressource im *content repository*, den Namen einer Ansicht sowie beliebige Name/Wert-Paare als Parameter enthält.

TemplateFinder Ein *TemplateFinder* implementiert eine Strategie, um Anhand einer Ressource und dem Namen einer Ansicht eine Vorlage zu finden. Die Standardimplementierung, der *ViewDispatcher*, bildet dazu den Methodenaufruf auf einem Objekt in einer objektorientierten Programmiersprache nach. Sie sucht anhand der Vererbungshierarchie der Dokumenttypen nach einer möglichst speziellen Vorlage mit dem angegebenen Namen für den vorliegenden Ressourcetyp.

Vorlagen Eine Vorlage ist eine *JavaServer Pages* Datei (JSP) [36], in deren Kontext auch eine

Ressource und Parameter in Form von Name/Wert-Paaren bekannt sind. Neben statischen Elementen enthält die Vorlage auch die Werte von Eigenschaften der darzustellenden Ressource oder abgeleitete Werte. Aus der Vorlage heraus können weitere *ResourceUris* erstellt und über diese anderere Vorlagen als „Fragmente“ eingebunden werden.

Wie bereits beschrieben ist effizientes *caching* wichtig, um einen hohen Durchsatz einer Website zu gewährleisten. Der ADS unterstützt dies, indem der Entwickler Vorlagen als „cacheable“ markieren kann. Die Ausgabe einer solchen Vorlage wird dann für Seiten und Fragmente mit der *ResourceUri* als Schlüssel im Dateisystem abgelegt, um bei wiederholten Zugriffen schnell ausgeliefert werden zu können. Bei der Ausführung der JSP-Dateien verfolgt der ADS alle Abhängigkeiten zu existierenden Ressourcen. Bei entsprechenden Änderungen im *content repository* werden die abhängigen Seiten und Fragmente im Dateisystem transitiv als nicht mehr gültig markiert, um sie beim nächsten Zugriff neu zu berechnen.

Abschließend betrachten wir die Möglichkeit zur Integration mit anderen Systeme und die Erweiterbarkeit des Systems. Um die Anbindung externer Systeme zu vereinfachen, beinhaltet die SCT eine Klassenbibliothek, die Java-Entwicklern einfachen Zugriff auf das *content repository* und die Möglichkeit zum Reagieren auf Benachrichtigungen des *content servers* bietet. Die Klassenbibliothek implementiert dazu einen einfachen Ressource-Cache, um die CORBA-Kommunikation zu reduzieren. Dem Entwickler wird dadurch ein wesentlicher Teil des Programmieraufwands abgenommen, der bei der direkten Kommunikation mit den anderen Komponenten über IIOP anfiele.

3 Realisierung von Webapplikationen mit Content Management Systemen

3.1 Probleme

Dieser Abschnitt beschreibt einige der Probleme, die für eine bestimmten Klasse von Webanwendungen zu lösen sind. Diese Art von Anwendungen zeichnet sich insbesondere durch die folgenden Eigenschaften aus:

- Die dargestellten Inhalte sind strukturiert und stammen aus einer darstellungsunabhängigen Quelle. Im Folgenden wird von einem Content Management System mit strikter Trennung von Inhalt und Darstellung als Quelle ausgegangen. Die Anwendung ist dafür verantwortlich, die darzustellenden Inhalte auszuwählen und die Darstellungsinformationen hinzuzufügen. CMS mit schwacher oder überhaupt keiner Trennung von Inhalt und Darstellung bieten der Anwendung üblicherweise weniger Flexibilität bei der Auswahl und Darstellung der Inhalte. In diesen Fällen sind die Problemstellungen für die Anwendung anders gelagert und sind daher nicht Gegenstand der Betrachtung.
- Inhalte werden an unterschiedlichen Stellen der Anwendung in verschiedenen Formen wiederverwendet.
- Der überwiegende Anteil aller Zugriffe auf die Inhalte ist lesend. Schreibzugriffe sind zwar möglich, stellen aber eine Ausnahme bezogen auf die Anzahl der Lesezugriffe dar. Inhalte werden demnach typischerweise über andere Wege erstellt und bearbeitet.
- Die Anzahl der Zugriffe, d. h. die Anzahl der Anfragen an die Webanwendung ist sehr hoch. Um eine große Anzahl von Anfragen in angemessener Zeit bearbeiten zu können, muss die Antwortzeit für eine Anfrage so kurz wie möglich gehalten werden.

Für die nachfolgende Beschreibung der Herausforderungen bietet sich ein einfaches Beispiel an. Es orientiert sich an den Möglichekeiten der in Abschnitt 2.5.2 vorgestellten CoreMedia SCT. Der Inhalt eines beispielhaften *content repository* ist in Abbildung 3.1 dargestellt. Es basiert auf den sechs in Tabelle 3.1 aufgeführten Dokumenttypen.

Der Ordner „Site“ enthält einen Ordner „Navigation“, in dem Dokumente vom Typ *Page* abgelegt sind. Die Verweisstruktur dieser Dokumente untereinander bildet die Navigationsstruktur des Beispiels. Die Ablage der Dokumente im Repository sagt nichts über die Navigation aus, sondern erleichtert den Bearbeitern die Verwaltung. Die eigentlichen Inhalte sind dagegen im Ordner „Inhalte“ abgelegt. Diese Dokumenttypen und Annahmen über die Ablage der Dokumente sind

Dokumenttyp	**Supertyp**	**Beschreibung/Eigenschaften**
Page		Repräsentiert einen Navigationsknoten, in dessen Kontext Inhalte angezeigt werden können. Name (Zeichenkette) Nachfolgerseiten (Verweis auf *Page*-Dokumente) Inhalte (Verweis auf *Content*-Dokumente)
Image		Enthält ein Foto oder eine Abbildung. Beschreibung (Zeichenkette) Bilddaten (Blob)
Content		Abstrakter Supertyp für alle Inhaltsdokumenttypen. Kurzbeschreibung (Zeichenkette) Thumbnail (Blob)
Article	Content	Enthält einen Artikel, der aus einem Fließtext besteht und Verweise auf andere Inhalte sowie auf zugehörige Bilddokumente besitzt. Überschrift (Zeichenkette) Text (XML) Bilder (Verweise auf *Image*-Dokumente) Querverweise (Verweise auf *Content*-Dokumente)
Results	Content	Enthält strukturierte Sportergebnisse in Form eines XML-Dokuments. Tabelle (XML)
ContentList	Content	Aggregiert andere Inhalte. Inhalte (Verweise auf *Content*-Dokumente)

Tabelle 3.1: Dokumenttypen einer Beispielanwendung

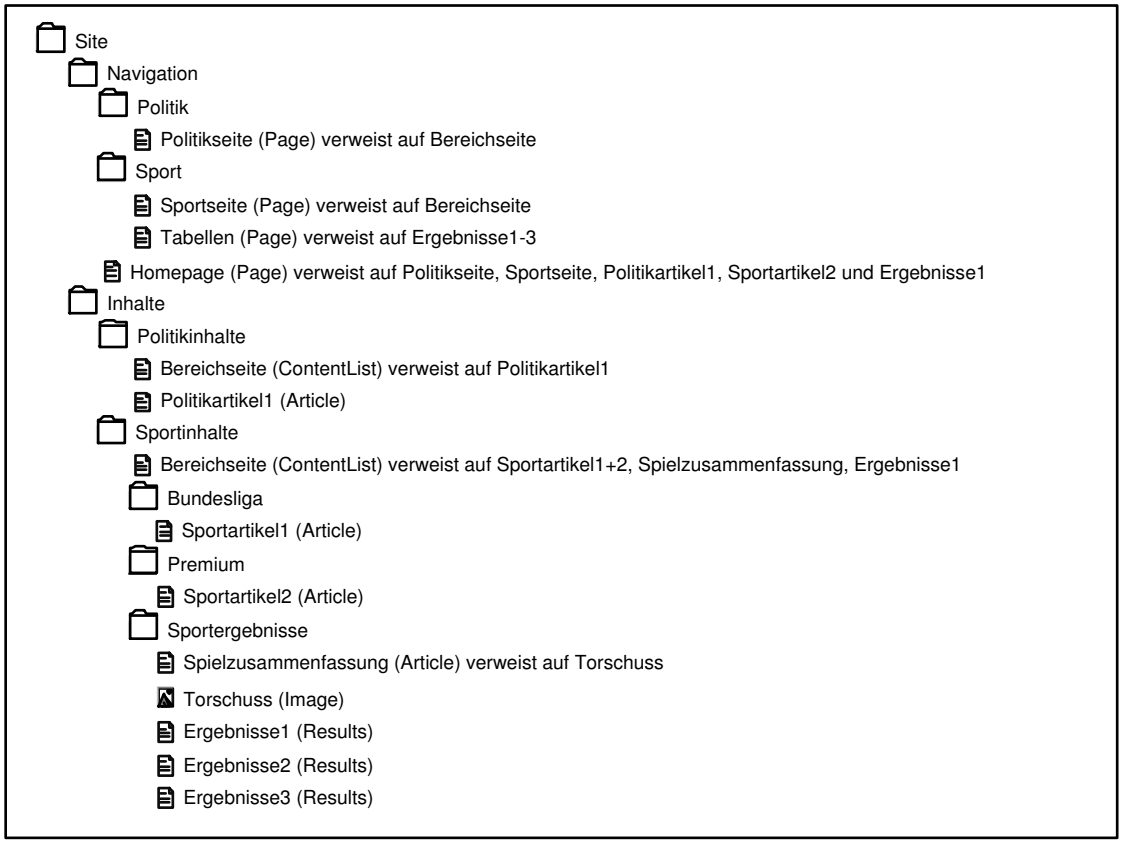

Abbildung 3.1: Inhalt eines beispielhaften *content repository*

spezifisch für dieses Beispiel und können in anderen Anwendungen ganz anders gewählt werden. Das Beispiel wird dadurch nicht in seiner Allgemeinheit eingeschränkt. Die folgenden Abschnitte beschreiben Anforderungen, die auf viele Webanwendungen zutreffen, und beziehen sich zur Verdeutlichung auf das vorgestellte Beispiel.

3.1.1 Abbildung von Inhaltsobjekten auf Anwendungsobjekte

Für die Darstellung in einer Webanwendung müssen die Inhalte im *content repository* interpretiert werden. So ist zum Beispiel die Information, dass *Page*-Dokumente die Navigationsstruktur repräsentieren, anwendungsspezifisch und kann den Dokumenten selbst nicht angesehen werden. Darüber hinaus können Verweise zwischen Dokumenten sowohl als strukturelle als auch als inhaltliche Verweise interpretiert werden. So gehören die Verweise in *ContentList*-Dokumenten zur ersten Gruppe, da sie Inhalte nur gruppieren, während die Zuordnung von Bildern zu einem Artikel einen inhaltlichen Zusammenhang darstellt, der sich auch in der Anwendung widerspiegeln wird.

Für den Entwickler der Anwendung stellt sich daher zunächst das Problem, die Daten aus dem CMS in eine für die Anwendung natürliche Form zu transformieren. Aufbauend auf dieser Form kann dann die Präsentationsschicht der Anwendung entwickelt werden. Für das oben eingeführte

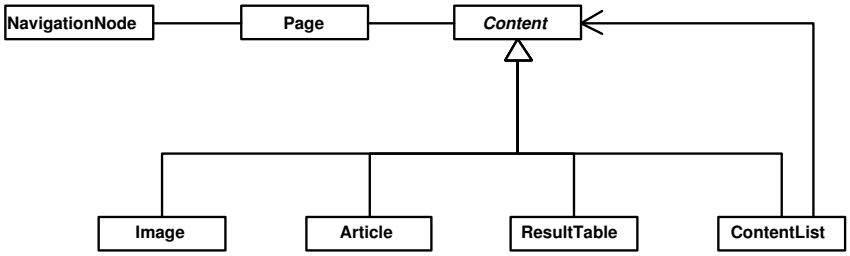

Abbildung 3.2: Ausgewählte Klassen der Anwendungslogik für das Beispiel

Beispiel soll hier angenommen werden, dass eine Seite der Anwendung immer den gleichen Aufbau hat, der für viele Websites typisch ist. Auf der linken Seite befindet sich ein Bereich für einen Navigationsbaum, oben ist der Pfad von der Wurzel dieser Navigation zum aktuellen Knoten zu lesen. In der Mitte der Seite befindet sich der eingentliche Inhalt. Ein Inhalt kann ein Artikel mit Bildern sein oder eine Tabelle mit Sportergebnissen. Die Verwendung des *Composite*-Entwurfsmusters [21] erlaubt auch die Komposition mehrerer Inhalte zu einem neuen Inhaltsobjekt. Dieses Entwurfsmuster findet sich sowohl im Dokumenttypmodell des CMS als auch in der objektorientierten Modellierung der Anwendung wieder. Abbildung 3.2 zeigt das beschriebene Objektmodell. Die Abbildung von Dokumenten des CMS auf Objekte dieser Klassen muss als Teil der Anwendungslogik implementiert werden.

Das Objektmodell dieser Beispielanwendung dupliziert im Wesentlichen die Dokumenttypen im CMS. Ohne dieses Objektmodell könnte ein Anwendungsprogrammierer die CMS-Inhalte nur über generische Zugriffsmethoden (Dokumente, Ordner, Eigenschaften) auslesen. Es ist im Allgemeinen nicht notwendig, dass das Objektmodell dem Inhaltsmodell direkt entspricht. Falls die Inhalte aus anderen Datenquellen wie Dateien oder aus CMS ohne Subtypisierung stammen, ist die Entsprechung von Dokumenttypen und Anwendungsklassen möglicherweise weniger eindeutig.

3.1.2 Effiziente Auslieferung der Ausgabe

Die im vorigen Abschnitt beschriebene Transformation, also die Erstellung anwendungsspezifischer Objekte aus Inhalten im Content Management System, ist im Allgemeinen deutlich aufwändiger als die Generierung einer Seite auf Basis dieser Objekte. Bei getrennten Komponenten für die Inhaltsverwaltung und die Auslieferung ist für den Zugriff auf das *content repository* eine Form von entferntem Methodenaufruf (RPC), zumindest aber Datei- oder Netzwerkzugriff notwendig. Im Fall der CoreMedia SCT kommunizieren die Komponenten über IIOP. Durch diese notwendige Kommunikation mit der Inhaltsverwaltung entstehen Latenzen, die im Vergleich zur Generierung der Seite sehr lang sind. Darüber hinaus können weitere Verarbeitungsschritte wie z. B. XSL-Transformationen oder die dynamische Generierung von Abbildungen aus Inhalten im Vergleich zur eigentlichen Ausgabe der Antwort viel Zeit beanspruchen.

Die Generierung der Ausgabeseite wird durch die Ersetzung von Platzhaltern in Vorlagen erreicht. Das Auswerten dieser Vorlagen übernimmt ein Präsentations-Framework. In der vorliegenden Implementierung ist dies Microsoft ASP.NET, das in einem der folgenden Kapitel genauer

beschrieben wird.

3.1.3 Aktualität der Seiten

Jede Änderung der Ressourcen im Repository muss so bald wie möglich eine entsprechend geänderte Ausgabe der ausgelieferten Seiten zur Folge haben. Idealerweise spiegelt bereits die erste Anfrage nach einer Änderung die neuen Inhalte wider. Insbesondere sollten Änderungen keine inkonsistenten Zustände wie ungültige Verweise auf einer generierten Seite bewirken.

Bezogen auf das vorgestellte Beispiel folgt daher unter anderem:

- Änderungen an dem Dokument „Ergebnisse1“ wirken sich sofort auf die Homepage, die Sport-Bereichsseite und die Detailseite des Dokuments aus, da an diesen Stellen die Dokumentinhalte angezeigt werden.

- Eine Änderung der Eigenschaft „Name“ eines *Page*-Dokuments verändert die Ausgabe aller Seiten, da dieser Name als Knoten im Navigationsbaum jeder Seite auftaucht.

Es folgt also, dass eine einzelne Änderung im Repository potentiell viele Seitenänderungen nach sich ziehen kann.

3.1.4 Personalisierung der Ausgabe

Oft soll eine Website personalisierbar sein. Darunter ist zu verstehen, dass die Ausgabe der Webanwendung an den jeweiligen Benutzer angepasst wird. Eine Voraussetzung hierfür ist die Identifizierbarkeit des Benutzers. Dazu gibt es verschiedene Möglichkeiten mit unterschiedlichen Vor- und Nachteilen. Weit verbreitet ist die Nutzung sogenannter *cookies*, die es dem Server ermöglichen, Informationen auf dem Klienten abzulegen, der diese mit jeder folgenden Anfrage wieder an den Server schickt [28].

Grundsätzlich kann eine Ausgabe sowohl für anonyme als auch für wohlbekannte Benutzer personalisiert werden. Um sicherzustellen, dass es sich um wohlbekannte Benutzer handelt, sind Authentisierungsverfahren notwendig. Je nach Anwendung sollten weitere Technologien wie z. B. SSL/TLS [10] eingesetzt werden, um die Identität des Klienten sicherzustellen. Auf die hiermit verbundenen Probleme wird in dieser Arbeit nicht eingegangen. Im Folgenden gehen wir davon aus, dass eine Anfrage einem bestimmten anonymen oder authentisierten Benutzer zugeordnet werden kann.

Die Personalisierung ist eine Voraussetzung für interaktive Anwendungen, da der Zustand der Interaktion zwischen zwei Anfragen erhalten bleiben und dafür beide Anfragen dem selben Benutzer zugeordnet werden müssen. Doch auch nicht-interaktive Webanwendungen, die nur Inhalte ausliefern, können von Personalisierung profitieren, indem sie die Ausgabe für verschiedene Benutzergruppen anpassen.

Um die Beispielanwendung um Personalisierung zu erweitern wird angenommen, dass Benutzer Guthaben auf ein virtuelles Konto einzahlen und auf diese Weise zusätzliche kostenpflichtige Inhalte einsehen können:

- Ressourcen unterhalb des Ordners „Premium“ sind nur für authentisierte Benutzer mit ausreichendem Guthaben sichtbar. Sowohl die Homepage als auch die Sport-Bereichsseite haben somit für verschiedene Benutzer unterschiedliches Aussehen.
- Neben der Position in der Navigationshierarchie wird auch das aktuelle Restguthaben für den angemeldeten Benutzer auf jeder Seite angezeigt.

Entsprechend des vorhergehenden Abschnitts folgt damit, dass eine Verschiebung von „Sportartikel1“ in den Ordner „Premium“diesen Artikel sofort vor allen nicht authentisierten Benutzer verbirgt.

3.1.5 Wiederverwendung von Softwarekomponenten

Dieser Abschnitt untersucht die verschiedenen Softwarekomponenten, aus denen eine Webanwendung besteht, im Hinblick auf ihre Wiederverwendbarkeit in anderen Anwendungen.

Für die Präsentationsschicht der Anwendung, also die Schicht, die von der direkten Kommunikation mit dem Klienten und der Generierung der Ausgabe abstrahiert, gibt es eine Reihe von Präsentations-Frameworks. Die Auswahl des jeweiligen Frameworks bestimmt, auf welche Art Seiten definiert werden. Einige dieser Frameworks sind speziell für interaktive Anwendungen geeignet, da sie den Anwendungsentwickler hierbei besonders unterstüzen.

Unterhalb der Präsentationsschicht liegt die Schicht, die die bisher beschriebenen Probleme löst und die zwar spezifische Anwendungslogik implementiert, aber dafür Funktionalität nutzt, die in den meisten Webanwendungen benötigt wird. Dazu gehören das Erstellen und Auflösen von Verweisen, die Transformation von XML in HTML, die Ausgabe von binären Daten und andere. Diese gemeinsamen Anteile sollten getrennt von der einzelnen Anwendung als Bibliothek zur Verfügung stehen, um so wiederverwendet werden zu können. Darüber hinaus verwenden Anwendungen üblicherweise weiter Unterstützungsbibliotheken, auf die hier nicht weiter eingegangen werden soll.

Andererseits kann es auch hilfreich sein, die Anwendungslogik selbst wiederzuverwenden, nämlich dann, wenn mehrere Anwendungen auf dem gleichen Dokumentenmodell laufen sollen. Ein denkbares Szenario ist neben der interaktiven Webanwendung eine Implementierung von Web Services. Beide Anwendungen könnten sich die Implementation der Abbildung von CMS-Dokumenten auf Anwendungsobjekte teilen.

Zwischen allgemeiner Funktionalität und anwendungsspezifischer Implementierung liegen die „Anwendungsbausteine“. Solche Bausteine sind wiederverwendbare Teile einer Anwendung wie z. B. die Abbildung der *Page*-Dokumente auf Navigationsknoten. Sie setzen bestimmte Dokumenttypen voraus, sind aber in vielen verschiedenen Anwendungen einsetzbar.

3.2 Analyse der Anforderungen

Dieser Abschnitt definiert die Anforderungen an das in dieser Arbeit zu implementierende Rahmenwerk für die Unterstützung von Webapplikationen, die auf inhaltsorientierten Content Management Systemen basieren. Die Anforderungen leiten sich aus den im letzten Abschnitt vorgestellten allgemeinen Problemstellungen dieser Klasse von Anwendungen ab. Eine durch Anwendungsfälle

getriebene Analyse — wie sie für die Entwicklung konkreter Anwendungen üblich ist — ist dagegen im Fall eines allgemeinen Rahmenwerks ungeeignet. Der hier gewählte Ansatz der Identifikation gemeinsamer Anforderungen anhand existierender Anwendungen für die Entwicklung eines Rahmenwerks wird von Mattsson als „development process based on domain analysis“ bezeichet [29].

Obwohl die zu lösenden Probleme allgemeiner Natur sind, muss die Entwicklung der Anforderungen bereits die Eigenschaften des verwendeten Content Management Systems sowie des ausgewählten Präsentations-Frameworks berücksichtigen.

3.2.1 Zugriff auf Inhalte

In dieser Arbeit geht es um Webapplikationen, deren Hauptaufgabe es ist, personalisierte Inhalte effizient auszuliefern, wobei die wesentliche Inhaltsquelle das *content repository* eines inhaltsorientierten CMS nach Abschnitt 2.2.2 ist. Die Inhaltsobjekte sind in einer Hierarchie abgelegt und enthalten sowohl strukturierte als auch semi-strukturierte und unstrukturierte Elemente.

Das CMS ist jedoch nicht notwendigerweise die einzige Datenquelle. Als sekundäre Quellen sind Dateien, Netzwerkressourcen, relationale Datenbanksysteme oder andere denkbar. Allen Datenquellen gemeinsam ist eine mehr oder weniger große Latenz beim lesenden Zugriff.

Die Implementierung des Frameworks in dieser Arbeit verwendet die CoreMedia SCT als Datenquelle. Die Architektur und die unterschiedlichen Komponenten dieses CMS sind im Abschnitt 2.5.2 beschrieben. Des weiteren ist die Zielplattform das Microsoft .NET Framework mit ASP.NET als Präsentationsschicht. Obwohl die Architektur der SCT grundsätzlich offen ist und die Komponenten über das sprachunabhängige Protokoll IIOP kommunizieren, gibt es in der aktuellen Version nur Java-Bibliotheken für den Zugriff auf Inhalte im CMS.

Es muss daher eine Programmierschnittstelle (*application programming interface*, API) für das .NET Framework entworfen und implementiert werden, die für den Datenzugriff verwendet werden kann. Im Folgenden werden die Anforderungen an diese Programmierschnittstelle und ihre Implementierung genauer ausgeführt.

Inhalte

Die Inhalte aller Dokumente und Ordner müssen über die Programmierschnittstelle lesbar sein. Für Dokumente bedeutet dies auch den Zugriff auf alle verfügbaren Versionen. Der lesende Zugriff ist nur eingeschränkt durch die Rechte des Benutzers, dessen Anmeldedaten beim Verbindungsaufbau mit dem *content server* angegeben werden.

Alle Eigenschaftstypen, also ganzzahlige Werte, Zeichenkette, Datumswerte, XML-Dokumente, Verweislisten und binäre Objekte sollen auf einfache Art und Weise abrufbar sein. Ebenso sollte es eine einfache Möglichkeit zur Behandlung von Ressourcenmengen geben, wie sie z. B. in Verweislisten oder als Inhalte von Ordnern auftreten.

Obwohl in einer praktischen Anwendung durchaus relevant, wird der schreibende Zugriff auf das *content repository* in dieser Arbeit nicht betrachtet. Der Entwurf oder die Implementierung sollte allerdings eine entsprechende spätere Erweiterung berücksichtigen oder zumindest dieser nicht im Wege stehen.

Metadaten

Alle verfügbaren Metadaten von Ressourcen müssen über die API abrufbar sein. Dazu gehören unter anderem der Typ eines Dokuments und Informationen über die durch diesen Typ definierten Eigenschaften, wie die maximale Länge einer Zeichenkette oder der erlaubte MIME-Typ einer binären Eigenschaft. Über die statischen Metadaten des Repository hinaus ist auch der Zugriff auf die dynamischen Metadaten der Ressourcen wie Freigabezeitpunkt, Publikationsstatus usw. erforderlich.

Die Programmierschnittstelle muss Funktionen bieten, um alle dem Repository bekannten Dokumenttypen zu enumerieren. Sie ermöglicht so auch generische Anwendungen, die keine Annahmen über bestimmte Dokumenttypen machen.

Gruppen, Benutzer und Rechte

Obwohl diese nicht direkt Inhalte sind, muss die Programmierschnittstelle Möglichkeiten bieten, die Gruppen und Benutzer des Systems aufzulisten sowie Mitgliedschaften von Benutzern und Gruppen festzustellen. Die CoreMedia SCT verwaltet Gruppen und Benutzer in Domänen, so dass auch diese Informationen abrufbar sein müssen.

Damit Anwendungen, die die API benutzen, weitere Dienste wie Personalisierung anbieten können, sind nicht nur die Informationen über Benutzer und Gruppen, sondern auch über die Berechtigungen von Gruppen erforderlich. Die SCT verwaltet für jede Gruppe eine Liste von Berechtigungen auf bestimmten Ressourcen, die über die API auslesbar sein müssen.

Ereignisse

Die Architektur der SCT ist ereignisbasiert. Veränderungen an Inhalten, Gruppen, Benutzern, Mitgliedschaften oder Berechtigungen werden den verbundenen Klienten über asynchrone Ereignisse mitgeteilt. Neben der möglichen internen Verwendung dieser Ereignisse sollten diese in aufbereiteter Form auch dem Nutzer der API zur Verfügung gestellt werden.

Anwendungen sollen sich als Empfänger bestimmter Ereignisse registrieren können. Dies erlaubt die Implementierung ereignisgesteuerter Anwendungen, die ausgelöst durch Veränderungen an Inhalten Aktionen ausführen.

Sitzungsverwaltung

Für die Anmeldung an einem CoreMedia *content server* müssen Anmeldedaten eines Benutzers übergeben werden. Hierzu gehören der Name der Domäne, des Benutzers sowie ein aus dem Passwort abgeleiteter Hashwert. Bei erfolgreicher Anmeldung wird eine neue Sitzung erstellt, die Zugriff auf das *content repository* entsprechend den Rechten des angemeldeten Benutzers hat.

Jede Sitzung belegt auf dem Server Ressourcen, so dass die Leistung des Servers bei vielen gleichzeitigen Sitzungen abnehmen kann. So müssen alle Änderungen am Repository als Ereignisse an alle Sitzungen geschickt werden. Ein *content server* unterstützt daher auch sogenannte leichtgewichtige Sitzungen, wobei die Kommunikation mehrerer dieser leichtgewichtigen Sitzungen über

eine normale Sitzung abgewickelt werden kann. Leichtgewichtige Sitzungen belegen somit nur sehr wenig zusätzliche Ressourcen auf dem Server. Sie werden in dieser Arbeit nicht implementiert.

Da ein *content server* abhängig von der Lizenz nur eine begrenzte Anzahl gleichzeitiger Benutzer zulässt, ist die Sitzung nach ihrer Verwendung zu schließen.

Effizienz

Die Kommunikation mit dem *content server* erfordert entfernte Methodenaufrufe über das IIOP-Protokoll. Jeder dieser Aufrufe benötigt eine gewisse Zeit, die sich aus der Serialisierung der Parameter, der Latenz und Bandbreite der Verbindung, der Verarbeitungsdauer auf dem Server sowie der Deserialisierung der Rückgabewerte zusammensetzt. Auch wenn die Verarbeitungsdauer auf dem Server kurz ist, ergibt sich daher eine gewisse minimale Gesamtdauer des Aufrufs, die nur durch die Kommuniation bedingt ist.

Um unnötige und aufwendige entfernte Methodenaufrufe zu umgehen, sind Informationen, die vom Server gelesen wurden, in einem Cache zwischenzuspeichern. Diese Zwischenspeicherung ist für den Benutzer der Programmierschnittstelle transparent.

Da andere Klienten des *content server* gleichzeitig auf dem Server arbeiten, können im Cache zwischengespeicherte Informationen ungültig werden. Eine Änderung kann immer nur verzögert an andere Klienten übermittelt werden. Die Implementierung der API muss jedoch Folgendes sicherstellen:

1. Die Verzögerung zwischen einer Veränderung im Repository und dem Zeitpunkt, an dem ein erneutes Lesen im Cache abgelegter Informationen den veränderten Wert zurückliefert, sollte möglichst kurz sein.

2. Änderungen müssen für den Klienten in der gleichen Reihenfolge sichbar werden wie sie auf dem Server ausgeführt wurden.

Verwendung

Die zu entwerfende Programmierschnittstelle soll in erster Linie als Datenzugriffsschicht für die Inhalte einer Webapplikation dienen. Es ist jedoch wünschenswert, diese Schnittstelle auch in anderen Anwendungen verwenden zu können. Die Funktionalität selbst ist web-unabhängig, so dass dies grundsätzlich möglich ist.

Insbesondere bei einer späteren Erweiterung um den Schreibzugriff auf Inhalte ist die Implementierung von alleinstehenden Diensten oder Anwendungen denkbar, die Änderungen am Repository vornehmen oder auf Ereignisse reagieren. Beim Entwurf sollte daher vorgesehen werden, die API in einer möglichst großen Klasse von Anwendungen nutzen zu können.

Anpassung an die Zielplattform

Dieser Punkt ist für eine gute Programmierschnittstelle selbstverständlich, soll hier aber dennoch erwähnt werden. Um die Verwendung der API zu erleichtern, sollten die Bezeichnungen der Typen, Methoden und Parameter den für die Zielplattform üblichen Richtlinien entsprechen. Auch weit

verbreitete Verwendungsmuster sollten so weit wie möglich Anwendung finden, so dass dem Benutzer der Schnittstelle durch die Einhaltung bestimmter Konventionen die Einarbeitung erleichtert wird. Für Klassenbibliothken, die auf dem *.NET Framework* basieren, sind die empfohlenen Richtlinien in [31] beschrieben.

Die erstellten Klassenbibliotheken sollten weiterhin möglichst CLS-kompatibel sein, da sie damit von jeder CLS-kompatiblen Sprache verwendet werden können. Um dies zu erreichen, darf eine Klassenbibliothek nur CLS-kompatible öffentliche Typen besitzen. Öffentliche Typen dürfen also ihren Signaturen keine erweiterten Eigenschaften einer Programmiersprache benutzen, die nicht in der *common language specification* spezifiziert sind [12].

3.2.2 Effizienz der Auslieferung

Die Seiten einer Webanwendung bestehen aus statischen und dynamischen Anteilen (Fragmenten). Die Aufgabe des eingesetzten Präsentations-Framework ist es, eine Anfrage entgegenzunehmen, den dynamischen Anteil zu berechnen, diesen mit dem statischen Anteil zu kombinieren und das Ergebnis an den Klienten zurückzugeben. Der statische Teil der Seite und die Platzhalter für dynamische Anteile werden üblicherweise über Vorlagen definiert.

Um eine möglichst vollständige Aufgabentrennung zwischen der Entwicklung des Layouts und Entwicklung der Anwendungslogik zu erreichen, ist es sinnvoll, jegliche Generierung dynamischer Inhalte von der Präsentationsschicht an Objekte der Anwendungsschicht zu delegieren. Im Folgenden wird daher angenommen, dass die Anwendungsschicht für eine Seite eine Reihe von Objekten erstellt, deren Eigenschaften gerade die dynamischen Anteile der auszugebenden Seite enthalten bzw. für die Ausgabe nur noch triviale Weiterverarbeitungsschritte auf diesen Eigenschaftswerten notwendig sind. Diese Objekte werden an die Präsentationsschicht zurückgegeben und ihre Eigenschaftswerte in die Vorlagen eingesetzt. Im Folgenden werden sie als *Präsentationsobjekte* bezeichnet.

Dynamische Anteile sind Teile der Seite, die sich ohne Eingriff in das System, also ohne Änderung an der Konfiguration oder an Vorlagen, über die Zeit verändern können. Wie bereits in einem früheren Kapitel beschrieben werden dynamische Fragmente in den hier betrachteten Anwendungen aus Inhalten erstellt, die im *content repository* vorliegen. Je nach Anwendung und Inhaltsmodell kann diese Abbildung beliebig komplex und somit aufwändig zu berechnen sein. Selbst wenn also der Aufwand für die Kommunikation mit dem *content server* durch den Einsatz eines Caches reduziert wird (siehe vorhergehender Abschnitt), ist die Dauer für die Transformation in das Zielfragment möglicherweise noch immer beträchtlich. Es sollte daher auch für diese Berechnung die Möglichkeit des Zwischenspeicherns geben, wobei sich wie gehabt Änderungen an Ressourcen im Repository direkt auf die aus ihnen generierten dynamischen Fragmente auswirken. Zu diesem Zweck ist eine Verwaltung der Abhängigkeiten zwischen Ressourcen und erstellten Fragmenten notwendig. Da dies ein allgemeines Problem jeder Anwendung ist, muss das zu entwerfende Framework eine Unterstützung für diese Art der Generierung dynamischer Fragmente anbieten.

Es folgen weitere Anforderungen an das zu implementierende Framework, das diese Unterstützungsfunktion übernimmt.

- Die Freiheit des Anwendungsprogrammierers soll möglichst wenig eingeschränkt werden, d. h.

die Verwendung eines solchen Zwischenspeichers für dynamische Fragmente soll weitestgehend transparent sein.

- Wiederverwendbare Zwischenergebnisse müssen unterstützt werden. Denkbar sind Abbildungen in mehreren Schritten, wobei sich verschiedene Fragmente Zwischenergebnisse teilen und so den Gesamtaufwand für die Berechnung mehrerer unterschiedlicher Fragmente reduzieren können. Die Granularität der Zwischenergebnisse, also welche Objekte als Zwischenergebnis im Cache vorgehalten werden und welche einfach ein Teil des Gesamtergebnisses sind, soll vom Anwendungsprogrammierer von Fall zu Fall entschieden werden, da dieser seine Anwendung am besten kennt.

- Personalisierung von Fragmenten muss unterstützt werden. Personalisierte Fragmente sind häufig nur Variationen nicht personalisierter Fragmente und können daher ebenfalls vom Caching profitieren.

- Transitive Abhängigkeiten werden automatisch festgestellt und verwaltet. Ebenso erfolgen transitive Invalidierungen automatisch.

- Zusätzlich zu Abhängigkeiten von Ressourcen im CMS sollen anwendungsspezifische Abhängigkeiten möglich sein. Das System soll dahingehend erweiterbar sein, dass auch andere Komponenten als das CMS eine Invalidierung bestimmter Werte im Cache auslösen können.

3.2.3 Applikationslogik

Die Applikationsschicht implementiert neben den Geschäftsprozessen der Anwendung auch die Abbildung der Inhalte des CMS und anderer Datenquellen auf Präsentationsobjekte. Im Rahmen dieser Abbildung immer wiederkehrende Operationen sollten durch das Framework in besonderer Weise unterstützt werden. Zu dieser Art von Operationen gehören die Behandlung von Verweisen sowie die Verarbeitung von XML wie sie in diesem Abschnitt beschrieben werden.

Der Begriff „Verweis“ hat hier zwei Bedeutungen: Er bezeichnet einerseits eine Referenz zwischen zwei Inhaltsobjekten des CMS und andererseits eine Referenz zwischen zwei Webressourcen. In einer Webanwendung sind aufgrund der Abbildung von Inhaltsobjekten auf Präsentationsobjekte und deren Ausgabe als Webressource auch die Referenzen zwischen Inhaltsobjekten in geeigneter Form auf Verweise zwischen Webressourcen zu transformieren.

Ein Präsentationsobjekt leitet sich im Allgemeinen aus mehreren Inhaltsobjekten ab, wobei beliebige Verarbeitungsschritte ausgeführt werden können. Eine Umkehrabbildung von einem Präsentationsobjekt auf ein bestimmtes oder eine Menge von Inhaltsobjekten ist daher üblicherweise nicht möglich.

Eine Webseite besteht — wie in Kapitel 2 beschrieben — aus mehreren Webressourcen, die jeweils durch eine URL identifiziert werden und Verweise auf andere Webressourcen in Form von URLs enthalten. Des weiteren enthält eine Webressource dynamische Fragmente, deren Inhalt durch Präsentationsobjekte beschrieben ist. Im Fall einer Website, deren Hauptzweck die Präsentation von Inhalten aus einem CMS ist, kann angenommen werden, dass diese dynamischen Fragmente die wesentlichen Anteile einer Webressource ausmachen. Daraus folgt, dass zur Bearbeitung der

Anfrage eines Klienten diese auf die Präsentationsobjekte und entsprechende Vorlagen abgebildet werden muss. Dazu legen wir fest, dass eine URL auf genau ein Präsentationsobjekt abgebildet wird, das wiederum beliebige andere Präsentationsobjekte referenzieren kann. Mit anderen Worten bezeichnet eine URL in der Webanwendung einen Knoten eines gerichteten Graphen von Präsentationsobjekten, dessen Kanten Objektreferenzen sind und der im Folgenden als „Komponentengraph“ bezeichnet wird. Eine Komponente ist ein Präsentationsobjekt und der Graph eine Repräsentation aller dynamischer Anteile der Webressource, die als Antwort auf die durch die URL spezifizierte Anfrage ausgeliefert wird. Für die Generierung der Ausgabe ist nur noch die Verknüpfung des Komponentengraphen mit Vorlagen erforderlich, was im späteren Abschnitt über Präsentation beschrieben wird.

Verweise kommen im *content repository* in Form von Verweislisten als Eigenschaft von Dokumenten oder als interne bzw. externe Links innerhalb von Fließtexten vor. Interne Verweise sind Referenzen auf andere Ressourcen im Repository und schließen somit auch die Einträge in Verweislisten ein. Externe Verweise werden durch URLs repräsentiert, die Webressourcen außerhalb der eigenen Domäne referenzieren. Interne Verweise haben zunächst außerhalb des Systems keine Bedeutung und auch keine allgemeine Repräsentation in Form einer URL. Da Verweise aber auch als URLs in die Ausgabe eingefügt werden müssen — sei es als Verweis auf einen anderen Artikel oder als Verweis auf ein Bild, das auf der Seite angezeigt werden soll — müssen interne Verweise in Verweise auf Präsentationsobjekte transformiert werden.

URLs werden zu gerichteten Graphen von Präsentationsobjekten aufgelöst, die wiederum aus Inhaltsobjekten des CMS oder anderen Datenquellen berechnet werden. Verweise zwischen Inhaltsobjekten, die für die Anwendung von Relevanz sind, werden wiederum in Verweise auf Präsentationsobjekte transformiert und in die Ausgabe eingefügt. Dieser Prozess muss durch das zu entwickelnde Framework unterstützt werden. Externe Verweise, d. h. Verweise auf externe Webseiten sind von einer Transformation üblicherweise nicht betroffen, da sie nicht von der zu erstellenden Webanwendung ausgewertet werden. Dennoch soll das Framework interne und externe Verweise konsistent und gleichförmig behandeln.

Die Notwendigkeit für die Verarbeitung von XML-Dokumenten ergibt sich aus der Tatsache, dass viele Inhalte als XML vorliegen, so z. B. auch längere Fließtexte in Dokumenten der CoreMedia SCT. Sie enthalten interne Verweise, die in oben beschriebener Weise behandelt werden müssen. Dabei soll das Framework auf die für die Zielplattform üblichen Verfahren zur XML-Verarbeitung zurückgreifen.

3.2.4 Personalisierung

Die dynamischen Fragmente einer Seite lassen sich in zwei Gruppen unterteilen:

1. *Nicht personalisierte* Fragmente bleiben typischerweise für sehr viele aufeinanderfolgende Anfragen an die Website gleich, obwohl sie grundsätzlich dynamisch sind. Dies können zum einen wiederholte Anfragen an die gleiche Webressource sein, häufig wird aber auch dasselbe Fragment auf verschiedenen Seiten in genau der gleichen Form verwendet. Ein Beispiel für ein solches Fragment ist eine nicht personalisierte Navigationshierarchie. Diese kann sich zwar

über die Zeit verändern, ist aber auf jeder Seite sichtbar und bleibt über viele Anfragen hinweg konstant.

2. *Personalisierte* Fragmente unterscheiden sich häufig schon bei zwei aufeinanderfolgenden Anfragen, da diese im Allgemeinen von zwei verschiedenen Benutzern gestellt werden. Obwohl für einen einzelnen Benutzer das Fragment möglicherweise immer gleich aussieht, ändert es sich aus Sicht des Servers abhängig vom Benutzer. Beispiele für personalisierte Fragmente sind eine persönliche Anrede oder ein angezeigter Warenkorb. Es ist zu beachten, dass im Folgenden alle Fragemente als personalisiert bezeichnet werden, für die eine Berechnung bei jeder Anfrage notwendig ist.

Das Framework muss sowohl personalisierte als auch nicht personalisierte Fragmente unterstützen. Die Art der Personalisierung selbst wird durch die Anwendung implementiert. Da nicht personalisierte Fragmente typischerweise direkt in einem Cache zwischengespeichert werden können, sollte sich ein Verzicht auf Personalisierung der Fragmente auch in Form eines erhöhten Durchsatzes auswirken. Die dynamischen Fragmente werden im Rahmen des Frameworks durch Präsentationsobjekte repräsentiert, so dass die Personalisierung von Fragmenten auf Ebene der Präsentationsobjekte gelöst werden muss. Typische Formen der Personalisierung sind die Ausblendung von Informationen aufgrund fehlender Rechte des Benutzers, unterschiedliche Darstellungen abhängig vom Benutzerprofil oder die Einblendung dynamischer Informationen wie z. B. ständig aktualisierter Aktienkurse oder Messdaten.

Im Zusammenhang mit Informationen aus einem *content repository* ist für die Personalisierung insbesondere die Auswertung von Rechten des aktuellen Benutzers in Bezug auf die Inhaltsobjekte von Bedeutung. Das Framework sollte daher für die Auswertung dieser Rechte und die daraus resultierende eingeschränkte Darstellung von Informationen unterstützende Funktionen bereitstellen. Aus dieser Anforderung ergibt sich, dass für Anwendungen, die diese Funktionen nutzen, Möglichkeiten zur Authentisierung und Autorisierung erforderlich sind. Es sollte einerseits leicht möglich sein, einem Benutzer der Webanwendung im *content repository* definierte Rechte zuzuordnen. Andererseits sollte das Framework auch anwendungsspezifische Authentisierungs- und Autorisierungsfunktionen unterstützen.

3.2.5 Präsentationsschicht

Die Aufgabe der Präsentationsschicht ist die Zusammenführung dynamischer und statischer Fragmente und die Ausgabe einer Webressource an den Klienten als Antwort auf eine Anfrage. Obwohl der typische Fall die Ausgabe einer HTML-Seite ist, gehört dazu auch die Auslieferung von Binärdaten oder anderen Formaten.

Es gibt eine ganze Reihe von Frameworks für Webanwendungen, die diese Aufgabe mit unterschiedlichen Schwerpunkten erfüllen. So implementiert zum Beispiel „Apache Struts“[1] aufbauend auf der „Java Server Pages“ API von Sun [36] ein seitenorientiertes Model-View-Controller Paradigma. Andere Ansätze wie Microsofts „ASP.NET“ oder Suns „Java Server Faces“ [30] definieren

[1] Apache Struts: http://struts.apache.org

eine Komponentenarchitektur, wobei Seiten aus wiederverwendbaren Komponenten zusammengesetzt werden, die die Implementierung interaktiver Webanwendungen vereinfachen. Eine Übersicht über verschiedene Frameworks für die Präsentationsschicht von Webanwendungen bietet [22].

Jede Anwendung und ihr jeweiliges Umfeld erfordern eine geeignete Wahl eines Frameworks für die Präsentationsschicht. Das in dieser Arbeit implementierte Rahmenwerk für die Erstellung von Webanwendungen mit Content Management Systemen sollte daher möglichst wenig Annahmen über die verwendete Präsentationsschicht machen und sich in verschiedene Umgebungen integrieren lassen. Dazu wird eine allgemeine Schnittstelle beschrieben, über die das Rahmenwerk mit der Präsentationsschicht kommuniziert. Ein Adapter implementiert diese Schnittstelle für das jeweils verwendete Präsentations-Framework. Durch eine solche Architektur werden zwei Dinge erreicht: Einerseits ist das Rahmenwerk unabhängig von der verwendeten Präsentationstechnologie. Gleichzeitig kann der Adapter die Anpassung so implementieren, dass die Integration dem Programmiermodell des verwendeten Präsentations-Framework möglichst gut entspricht. Für die Implementierung in dieser Arbeit wurde ASP.NET von Microsoft als Präsentationstechnologie ausgewählt, da es der *de facto* Standard für .NET-basierte Webanwendungen ist und gute Entwicklungswerkzeuge vorhanden sind. Ein Adapter für die Verwendung des implementierten Rahmenwerks mit ASP.NET ist daher erforderlich. ASP.NET wird im folgenden Abschnitt beschrieben.

Die genannte Schnittstelle hat im Wesentlichen die Aufgabe, Präsentationsobjekte an die Präsentationsschicht durchzureichen, wobei üblicherweise eine Verknüpfung mit statischen Inhalten in Form von Vorlagen stattfindet. Die Definition der Vorlagen und wie diese Verknüpfung implementiert wird, sind abhängig von der verwendeten Technologie. Es ist aber zu beachten, dass auf dieser Ebene keine direkte Abhängigkeit von den Inhalten oder dem Inhaltsmodell im CMS mehr besteht. Für die Definition der Vorlagen muss daher das Inhaltsmodell der Ressourcen im CMS nicht bekannt sein, sondern nur die Typen der bereitgestellten Präsentationsobjekte. Darstellungs- und Anwendungslogik können somit unabhängig voneinander bearbeitet werden, was eine Evolution der Anwendung und die Aufgabenteilung während des Entwicklungs- und Wartungsprozesses erleichtert.

3.2.6 Microsoft ASP.NET 1.1

„Active Server Pages .NET“ (ASP.NET)[2] ist ein Rahmenwerk für die Entwicklung und Ausführung von Webanwendungen auf Basis des „.NET Framework“ von Microsoft. ASP.NET bietet eine Umgebung, um mit Hilfe des .NET Frameworks sowohl interaktive Webanwendungen als auch Web Services [13] in einer beliebigen .NET-kompatiblen Programmiersprache zu entwickeln und auszuführen. Die ASP.NET-Laufzeitumgebung, die dies ermöglicht, wird dazu an einen Webserver angebunden. Typischerweise ist dies der Webdienst der Windows „Internet Information Services“, aber auch die Einbindung in andere Webserver ist denkbar.

In diesem Abschnitt werden zunächst das zugrunde liegende .NET Framework und anschließend verschiedene Aspekte der ASP.NET Laufzeitumgebung beschrieben. Dazu gehören die Architektur insgesamt, allgemeine Dienste für ASP.NET-Anwendungen sowie das Programmiermodell für interaktive Webanwendungen. Das Modell für Web Services unterscheidet sich davon nur wenig und

[2] Microsoft Active Server Pages .NET: http://www.asp.net

wird nicht näher beschrieben.

Das .NET Framework und ASP.NET stellen die Grundlage für die Implementierung des in dieser Arbeit vorgestellten Rahmenwerks dar (siehe Kapitel 3.4).

Microsoft .NET Framework 1.1

Das .NET Framework besteht aus zwei wesentlichen Komponenten:

- der „Common Language Runtime“ (CLR) und
- der .NET Framework Klassenbibliothek.

Die CLR ist die Implementierung einer virtuellen Maschine, die verschiedene Dienste für die Ausführung sogenannten „verwalteten Codes“ (*managed code*) zur Verfügung stellt. Zu diesen Diensten gehören z.B. Speicherverwaltung, Verwaltung von *Threads*, die Ausführung und Sicherheitsprüfung sowie das Kompilieren von Code.

Eines der Ziele der CLR ist die Sprachunabhängigkeit, d. h. die Ausführung von Code, der in einer beliebigen kompatiblen Programmiersprache entwickelt wurde. Dazu gehört auch die Interoperabilität zwischen in verschiedenen Sprachen implementierten Modulen. Um dieses Ziel zu erreichen, wurde die *common language infrastructure* (CLI) definiert und von der ECMA [12] sowie der ISO standardisiert. Die CLI definiert die Regeln für Compiler, Sprachen und Werkzeuge, die für die Interoperabilität mit anderen CLI-kompatiblen Systemen einzuhalten sind. In diesem Sinne ist die CLR eine Implementierung der CLI. Der CLI-Standard ermöglicht auch die Entwickler anderer, kompatibler Implementierungen wie der frei verfügbaren *Mono*-Plattform[3]. Die CLI-Spezifikation umfasst mehrere Bestandteile:

Common Language Specification (CLS) Die CLS definiert eine Menge von Regeln, die öffentliche Typen eines Moduls einhalten müssen, um in einer beliebigen CLI-Implementierung verwendet werden zu können.

Common Type System (CTS) Das CTS definiert die Art der Typen, die in einer CLI-Implementierung verwendet werden. Dazu gehören verschiedene Referenztypen sowie Werttypen. Grundsätzlich unterstützt das CTS sowohl objektorientierte als auch funktionale und prozedurale Programmiersprachen.

Metadaten Dieser Abschnitt beschreibt die Verwendung von Metadaten in der CLI. Alle CLI-kompatiblen Kompilate beschreiben die enthaltenen Typen in Form standardisierter Metadaten. Eine CLI-kompatible Sprache muss eine Syntax für die Annotation von Typen, Methoden und Feldern durch Metadaten bereitstellen.

Virtual Execution System (VES) Das VES definiert die Ausführungsumgebung für verwalteten Code. Diese unterstützt eingebaute Datentypen und definiert das Modell einer virtuellen Maschine, den Kontrollfluss und die Ausnahmebehandlung. Sie ist die Basis für die Ausführung der *common intermediate language*.

[3] Mono Project: http://www.go-mono.com

Common Intermediate Language (CIL) Die CLI definiert die Instruktionen, die auf dem VES ausgeführt werden können.

Profile und Basisbibliotheken Der CLI-Standard definiert eine Reihe von Basisbibliotheken: die *Runtime Infrastructure Library, Base Class Library, Network Library, XML Library, Extended Numerics Library* und die *Extended Array Library*. Ein Profil ist eine Menge von Basisbibliotheken. Der Standard definiert die Profile „Kernel“ (die minimale Konfiguration) und „Compact“ (für Zielplattformen mit eingeschränkten Ressourcen).

Mit der .NET Framework Klassenbibliothek liefert Microsoft darüber hinaus Klassen, die die Entwicklung von Windows-, Konsolen-, ASP.NET- und Web Services-Anwendungen erleichtern.

Die Entwicklungsumgebung von Microsoft unterstützt vier CLI-kompatible Sprachen: C#, Visual Basic.NET, J# sowie MC++ (*managed* C++). Insgesamt gibt es allerdings über 20 Sprachen mit Erweiterungen für CLI-Kompatibilität. Dazu gehören Sprachen wie Ada, Cobol, Eiffel, Fortran, Pascal, Perl, Python, Scheme oder Smalltalk [24]. Die häufig bevorzugte Sprache ist C#, die sowohl von der ECMA und als auch der ISO standardisiert ist [11]. C# ist — soweit nicht anders angegeben — die Grundlage für die Implementierung des in dieser Arbeit vorgestellten Rahmenwerks.

Architektur

Die folgende Beschreibung bezieht sich auf ASP.NET 1.1 in Verbindung mit IIS 5.1. Der Webserver leitet alle Anfragen mit dafür registrierten URL-Endungen an ein ASP.NET ISAPI-Modul weiter, das mit einem externen Prozess kommuniziert, der alle Anfragen an ASP.NET Webanwendungen und Web Services behandelt. Innerhalb dieses Prozesses sind verschiedene Anwendungen und Dienste durch die Verwendung mehrerer *AppDomain*s voneinander isoliert.

Jede Anfrage durchläuft eine Pipeline, in der Applikationen (*HttpApplication*), Module (*IHttpModule*), Handler (*IHttpHandler*) und HandlerFactories (*IHttpHandlerFactory*) zusammenwirken, um die Antwort zu generieren und über den Webserver an den Klienten zurückzuschicken. Jeder Schritt der Pipeline kann durch eine konkrete Anwendung erweitert werden, um zusätzliche Funktionalität zu implementieren. Das Zusammenspiel der verschiedenen beteiligten Objekte wird im Folgenden erläutert, da es die Basis für die Integration des zu implementierenden Frameworks in eine ASP.NET-Umgebung darstellt. Detaillierte Informationen zur HTTP-Pipeline von ASP.NET liefert [15].

Zunächst prüft die Klasse *HttpRuntime*, welche Anwendung Empfänger der Anfrage ist und erstellt ggf. ein Applikationsobjekt vom Typ *HttpApplication*. Applikationsobjekte können für nachfolgende Anfragen wiederverwendet werden, behandeln jedoch nur eine Anfrage zur Zeit. Im Fall mehrerer gleichzeitiger Anfragen werden neue Objekte erzeugt und diese in einem Pool verwaltet. Eine Konfigurationsdatei *web.config* definiert die von einer Applikation verwendeten Module, *Handler* und *HandlerFactories*. Jede Applikationsinstanz erhält eigene Instanzen dieser Klassen, so dass Nebenläufigkeitsprobleme bei der Anfragebehandlung vermieden werden.

Module können auf jeden Schritt der Pipeline Einfluss nehmen, die Behandlung abbrechen oder umleiten. Handler werden in einem Schritt der Pipeline ausgeführt und dienen üblicherweise der

Ereignis	**Wird ausgelöst**
BeginRequest	vor der Anfragebehandlung
AuthenticateRequest	nach der Authentisierung des Benutzers
AuthorizeRequest	nach der Autorisierung des Benutzers
ResolveRequestCache	nach Laden einer Antwort aus ASP.NET-Cache
Laden des Handlers über die HandlerFactory	
AcquireRequestState	nach Laden des Sitzungszustands (optional)
PreRequestHandlerExecute	vor der Ausführung des Handlers
Ausführen des Handlers	
PostRequestHandlerExecute	nach der Ausführung des Handlers
ReleaseRequestState	nach Speichern des Sitzungszustands (optional)
UpdateRequestCache	nach dem Aktualisieren des ASP.NET-Cache
EndRequest	nach der Behandlung der Anfrage
PreSendRequestHeaders	vor dem Senden der gepufferten HTTP-Header
PreSendRequestContent	vor dem Senden der gepufferten Antwort

Tabelle 3.2: HTTP-Pipeline und von *HttpApplication* ausgelöste Ereignisse

Erzeugung einer Antwort auf die Anfrage. *HttpHandlerFactories* dienen der Erzeugung von *HttpHandler*-Instanzen. Die Kommunikation zwischen Applikationsobjekt und Modulen wird über .NET-Ereignisse realisiert. In jedem Schritt der Pipeline löst die Applikation ein entprechendes Ereignis aus. Während der Initialisierungsphase abonniert ein Modul die für seine Aufgabe relevanten Ereignisse. Wenn eine Webanwendung eine Datei *global.asax* enthält, wird diese von der Klasse *HttpRuntime* als Subklasse von *HttpApplication* kompiliert und als Applikationsobjekt zur Behandlung der Anfrage instanziiert. In *global.asax* definiert die Applikation Ereignisbehandlungsmethoden, um auf Applikationsereignisse (für die Pipeline-Schritte) und auf mögliche Ereignisse eingebundener Module zu reagieren. Auf diese Weise ist eine lose bidirektionale Kommunikation zwischen Applikation und Modulen möglich. ASP.NET enthält eine Reihe von Modulen für Authentisierung, Autorisierung, Caching und die Verwaltung des Sitzungszustands.

Tabelle 3.2 zeigt die Schritte der HTTP-Pipeline und die durch die Applikation ausgelösten Ereignisse. Der Sitzungszustand wird nur dann geladen, wenn der auszuführende Handler ihn benötigt. Die Handler-Klasse implementiert dann die Schnittstelle *IHttpSessionState*.

Handler werden über eine HandlerFactory geladen. Diese sind für einen regulären Ausdruck und eine Menge von HTTP-Methoden in der Konfigurationsdatei *web.config* registriert. Eine HandlerFactory instanziiert eine Handler-Klasse für eine Anfrage und gibt an, ob diese Instanz für nachfolgende Anfragen wiederverwendet werden kann. ASP.NET implementiert eine Reihe von Handler- und HandlerFactory-Klassen, z.B. für Web Services, Seiten (siehe Abschnitt 3.2.6) und andere. Während der Bearbeitung einer Anfrage ist der gesamte Kontext in allen Pipeline-Schritten

über die statische Eigenschaft *Current* der Klasse *HttpContext* erreichbar.

Authentisierung

ASP.NET implementiert drei Module für die Authentisierung von Benutzern. In der Anwendungskonfiguration wird entweder keins oder eines dieser drei Module konfiguriert. Alle Authentisierungsmodule setzen im Kontext der Anfrage ein Objekt, das die Schnittstelle *IPrincipal* implementiert. Dieses Objekt beschreibt die Identität und die Rollenzugehörigkeiten des Benutzers, der die Anfrage stellt. Module und Handler können auf dieses Objekt nach dem *AuthenticateRequest*-Ereignis über die Eigenschaft *HttpContext.User* zugreifen, um z. B. eine Autorisierung vorzunehmen.

FormsAuthentication Dieses Modul leitet zunächst die Anfrage auf eine Anmeldeseite um, wenn eine Autorisierung fehlschlägt. Beim Abschicken eines Formulars mit Benutzername und Passwort leitet die Anwendung diese an das *FormsAuthentication* Modul weiter. Das Modul liest die bekannten Benutzer und ihre Passwörter und Rollen aus einer XML-Datei. Bei erfolgreicher Authentisierung wird ein Cookie [28] mit den Anmeldeinformationen an den Klienten zurückgeschickt, um die Anmeldung nicht mehrfach durchführen zu müssen. Dieses Modul unterstützt eine Reihe von weiteren Funktionen wie das Erfordern einer SSL/TLS-Verbindung [10], Authentisierung ohne Cookies und die Verwendung von Hash-Werten statt der Passwörter in der Konfigurationsdatei. Andere Authentisierungsmodule können die Funktionalität von *FormsAuthentication* nutzen, ohne dass zwingend die Benutzerinformationen aus einer XML-Datei gelesen werden müssen (siehe Abschnitt 3.4.3 über die Implementierung eines Authentisierungsmoduls für den CoreMedia *content server*).

WindowsAuthentication Das *WindowsAuthentication* Modul nutzt Authentisierungsfunktion des IIS. Dieser unterstützt sowohl Basic und (erweiterte) Digest HTTP-Authentisierung [19] als auch die integrierte Windows-Authentisierung über das proprietäre Protokoll NTLM oder Kerberos V5 [27] und Klientenzertifikate in Verbindung mit SSL/TLS [10]. Mehr Informationen zu den Authentisierungsfunktionen in IIS liefert [32].

PassportAuthentication Dieses Modul unterstützt die Authentisierung des Benutzers über den Passport-Dienst[4] von Microsoft.

Sitzungszustand

Das Modul *SessionStateModule* implementiert die Verwaltung des Sitzungszustands für Benutzer. Die Sitzungszustands kann entweder deaktiviert oder nach einer von drei möglichen Arten verwaltet werden:

In process Der Sitzungszustand wird im Speicher des Prozesses gehalten, der die Anfragen bearbeitet. Bei einem Neustart der Webanwendung gehen alle Zustandsinformationen verloren.

StateServer In diesem Modus wird der Sitzungszustand im Speicher eines externen Prozesses, der auf dem gleichen oder einem anderen Rechner läuft, verwaltet. Dadurch bleiben die Zustandsinformationen bei einem Neustart der Webanwendung erhalten. Auch die Verteilung

[4] Microsoft .NET Passport: http://www.passport.com

der Anfragen auf mehrere Prozesse oder Server ist so möglich, da alle Prozesse den Sitzungszustand vom *StateServer* beziehen können.

SqlServer Dieser Modus ist ähnelt dem *StateServer* Modus. Er skaliert allerdings besser, da der Sitzungszustand in einer relationalen Datenbank gehalten wird.

Seiten

Interaktive Webseiten (*web forms*) werden in ASP.NET typischerweise in Form von Dateien mit der Endung „.aspx“ erstellt. Für diese Dateiendung ist die *PageHandlerFactory* registriert, die eine aspx-Datei in eine Subklasse von *Page* kompiliert und ein Objekt dieser Klasse instanziiert. *Page* implementiert dabei *IHttpHandler* und ist für die Generierung der Ausgabe verantwortlich. *Page* ist ein sehr komplexer Handler und erfüllt die folgenden Aufgaben:

- Ein *web form* wird üblicherweise durch zwei getrennte Dateien implementiert, um eine Trennung des Seitenlayouts und der Handler-Logik zu erreichen.
- *Web forms* erleichtern die Implementierung interaktiver Webanwendungen. Dazu bieten sie dem Entwickler ein Programmiermodell, das dem der Erstellung von grafischen Benutzeroberflächen sehr ähnlich ist. In diesem Modell bestehen *web forms* aus Steuerelementen, die bei Benutzeraktionen Ereignisse auslösen. Das Komponenten- und Ereignismodell von *web forms* wird unten erläutert.

Die *PageHandlerFactory* kompiliert ein *web form* aus zwei Dateien: einer Datei mit der Endung „.aspx“ und einer Klasse, die von *Page* erbt und in einer beliebigen CLS-kompatiblen Sprache implementiert ist (*code-behind* Datei). Die aspx-Datei ist eine Vorlage, die in deklarativer Form das Aussehen der zu generierenden Seite beschreibt. Sie ist im wesentlichen eine HTML-Datei, die zusätzlich ASP.NET-Steuerelemente (siehe unten) und Code in einer .NET-kompatiblen Sprache enthalten kann. Am Anfang der Datei steht darüber hinaus eine Deklaration mit Attributen der Seite – wie z. B. Cache-Verhalten, die Verwendung des Sitzungszustands des Benutzers und die verwendete .NET-kompatible Sprache – sowie dem Namen der *code-behind* Datei. Beim Kompilieren wird aus der aspx-Datei eine Subklasse der in der *code-behind* Datei implementierten Klasse erstellt. Die *code-behind* Datei enthält dabei die Logik für die Initialisierung der Steuerelemente und die Reaktion auf Ereignisse, die diese auslösen. Alle Steuerelement und andere Variablen, die in beiden Dateien verwendet werden, müssen als geschützte oder öffentliche Felder oder Eigenschaften in der *code-behind* Klasse definiert sein. Die ASP.NET-Laufzeitumgebung erkennt Änderungen an ASPX Dateien und kann diese neu kompilieren, ohne die Anwendung neu zu starten.

Komponentenmodell

Eine ASP.NET-Seite besteht aus einem Baum von Objekten, die von der Klasse *System.Web.UI.Control* erben (im Folgenden Steuerelemente oder „Controls“ genannt). Die Wurzel dieses Baumes ist das *Page*-Objekt, das ebenfalls von *Control* erbt. Controls haben einige allgemeine Eigenschaften wie einen Bezeichner und Verweise auf die Kindsteuerelemente im Baum. Die ASP.NET-Klassenbibliothek enthält eine ganze Reihe von Steuerelementen, die eine der Klassen in Abbildung 3.3 erweitern.

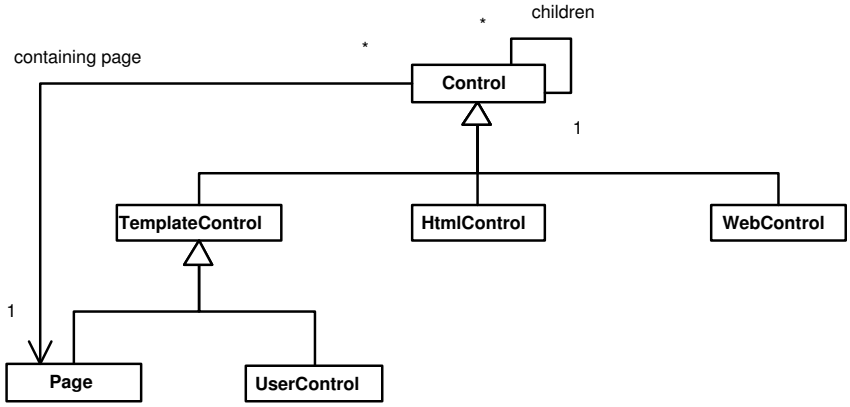

Abbildung 3.3: Diagramm der Basisklassen für ASP.NET-Steuerelemente

Die Klassen in diesem Klassendiagramm dienen als Superklassen für Steuerelemente mit bestimmten Funktionen:

Control *Control* ist die Superklasse für alle ASP.NET-Serversteuerelemente. Es definiert Methoden und Ereignisse für den Ausführungszyklus des Steuerelements (siehe Abschnitt über das „Ereignismodell“). Es definiert allgemeine Eigenschaften wie den eindeutigen Bezeichner eines Steuerelements auf der Seite und Eigenschaften, um im Baum auf den Elternknoten zuzugreifen und die Liste der Kindknoten zu manipulieren.

WebControl *WebControl* ist die Superklasse für alle Serversteuerelemente, die eine Benutzeroberfläche darstellen. Sie definiert darstellungsspezifische Eigenschaften wie Vordergrund- und Hintergrundfarbe, Schriftarten und Randeinstellungen.

HtmlControl Subklassen von *HtmlControl* entsprechen HTML-Elementen in HTML-Seiten.

TemplateControl *TemplateControl* stellt allgemeine Funktionen für *Page* und *UserControl* bereit, um z.B. Benutzersteuerelemente aus „.ascx“-Dateien zu laden (siehe unten).

Page Diese Klasse ist die Superklasse aller ASP.NET-Seiten und definiert Methoden und Ereignisse für den Ausführungszyklus einer Seite.

UserControl *UserControl* ist die Superklasse für Benutzersteuerlemente, die in einer „.ascx“-Datei definiert werden.

Neben der großen Zahl von Serversteuerelementen, die in der ASP.NET-Klassenbibliothek enthalten sind, gibt es verschiedene Wege, neue Steuerelemente zu erstellen. Die einfachste Möglichkeit ist die Erstellung eines „Benutzersteuerelements“, das analog zu aspx-Seiten in einer Datei mit der Endung „.ascx“ und einer *code-behind* Datei definiert wird. Die ASP.NET Laufzeitumgebung kompiliert aus diesen beiden Dateien eine Steuerelementklasse, die *UserControl* erweitert. Subklassen von *TemplateControl* – also *Page* und *UserControl* – können diese Steuerelemente über den Pfad der „.ascx“-Datei laden und einbinden. Aufgrund des Erstellungsprozesses eignet sich diese Art

von Steuerelement besonders für anwendungsspezifische Elemente der Benutzeroberfläche, die innerhalb der Anwendung wiederverwendet werden sollen.

Die anderen beiden Arten der Erstellung von Steuerelementen erlauben eine einfache Wiederverwendung in mehreren Webanwendungen und die Verteilung der Steuerelemente in Form von Klassenbibliotheken. Hierbei implementiert der Entwickler direkt Subklassen von *Control* und kompiliert diese als Sammlung von Steuerelementen in eine *assembly*. Zu unterscheiden sind dabei der Ansatz der Komposition („zusammengesetzte Steuerelemente“) und der Vererbung. Zusammengesetzte Steuerelemente bieten sich an, um komplexe Controls aus einfacheren zu erstellen. Dabei können die enthaltenen Steuerelemente untereinander interagieren und das zusammengesetzte Control speziellere Ereignisse nach aussen sichtbar machen. Die Eigenschaften der inneren Steuerelemente sind von aussen nicht sichtbar. Dazu werden die Ereignisse der Komponenten gefiltert, in neue Ereignisse transformiert oder aggregiert, bevor sie vom zusammengesetzten Steuerelement ausgelöst werden. Empfänger dieser Ereignisse sehen nicht den ursprünglichen Absender, sondern nur das zusammengesetzte Element.

Wenn dagegen die Funktionalität eines vorhandenen Controls erhalten bleiben und nur erweitert werden soll, bietet sich Vererbung an. Eigenschaften, Methoden und Ereignisse des vorhandenen Controls können dabei überschrieben oder erweitert werden.

Steuerelemente, die über Komposition oder Vererbung erstellt wurden, können im Gegensatz zu Benutzersteuerelementen in der graphischen Entwicklungsumgebung Visual Studio .NET verwendet werden.

Ereignismodell

Die meisten Serversteuerelemente stellen haben eine eigene Benutzeroberfläche, die auf der Seite dargestellt wird. Im ASP.NET-Programmiermodell interagiert der Benutzer mit diesen Steuerelementen, die daraufhin entsprechende Ereignisse auslösen. Registrierte Ereignisbehandlungsmethoden werten diese aus führen daraufhin Aktionen auf dem Server aus. Jede Interaktion, die auf dem Server eine Aktion auslösen soll, ist eine eigene Anfrage. Die Steuerelemente auf der Seite behalten dabei über mehrere Benutzeraktionen, das heisst über mehrere Anfragen, hinweg ihren Zustand.

Es ist die Aufgabe der ASP.NET-Umgebung, den Zustand der Serversteuerelemente über mehrere HTTP-Anfragen hinweg zu halten, wobei das HTTP-Protokoll selbst zustandslos ist. Zwar unterstützen die meisten Frameworks für Webanwendungen das anfrageübergreifende Speichern des Anwendungszustands auf dem Server (in einer *Session*) oder auf dem Klienten (in einem *cookie* oder einem versteckten Eingabefeld), aber üblicherweise ist der Anwendungsprogrammierer für die Verwaltung dieses Zustands verantwortlich. Die ASP.NET-Laufzeitumgebung vereinfacht die Verwaltung, indem der Programmierer interaktive Webanwendungen ebenso ereignisbasiert programmieren kann wie herkömmliche Applikationen mit grafischer Benutzeroberfläche. Die Serversteuerelemente kapseln ihren eigenen Zustand und die Laufzeitumgebung stellt sicher, dass dieser Zustand zu Beginn jeder Anfragebearbeitung wiederhergestellt wird. Das Ergebnis ist das Modell eines Baums von Steuerelementen, die ihren Zustand durch Benutzeraktionen ändern und daraufhin Ereignisse auslösen.

Um dies zu erreichen kommunizieren die *Page* und die enthaltenen Steuerelemente über ein festes Protokoll von Methodenaufrufen und Ereignissen, die bei jeder Anfrage in einer bestimmten

Reihenfolge ausgelöst werden. Zwei verschiedene Rollen sind zu unterscheiden:

- **Entwickler von Steuerelementen** müssen auf diese Methodenaufrufe dem Protokoll entsprechend reagieren und zu geeigneten Zeitpunkten Ereignisse auslösen.
- Diese Ereignisse werden vom **Entwickler der Seiten** behandelt. Dabei können die Zustände anderer Steuerelemente verändert oder andere serverseitige Aktionen ausgelöst werden.

Ausführungsschritte

Die folgenden Schritte werden bei der Bearbeitung einer Anfrage ausgeführt, um das beschriebene Modell zu implementieren:

1. Zunächst wird der Konstruktor der *Page*-Subklasse ausgeführt. Die in der aspx-Datei aufgeführten Steuerelemente werden instanziiert und ihre Eigenschaften mit den Attributwerten aus der aspx-Datei initialisiert. Die Methode *DeterminePostBackMode* stellt fest, ob es sich um eine *PostBack*-Anfrage handelt, also die Bearbeitung einer Aktion auf einem Steuerelement. Daraufhin wird die Methode *OnInit* aufgerufen, die eine Seite überschreiben kann, um weitere Initialisierungsschritte durchzuführen. Die Schritte 2 bis 4 werden nur bei *PostBack*-Anfragen ausgeführt.
2. Die Methode *LoadPageStateFromPersistenceMedium* lädt den Zustand der Seite nach der letzten Anfrage in ein *ViewState*-Objekt. Dies umfasst den Zustand aller Serversteuerelemente auf der Seite. Die Standardimplementierung dekodiert den Zustand aus einer Base64-kodierten Zeichenkette, die als Wert eines versteckten Eingabefelds in die Seite eingefügt wurde. Eine Subklasse kann diese Methode überschreiben, um den Seitenzustand von einem serverseitigen Speicherort wie z. B. dem Sitzungsobjekt oder einer Datenbank zu laden.
3. Die Methode *LoadViewState* wird auf der Seite und rekursiv auf allen enthaltenen Controls aufgerufen, um ihren Zustand anhand der Informationen im *ViewState*-Objekt wiederherzustellen. Danach haben alle Steuerelemente wieder den gleichen Zustand wie nach der Bearbeitung der letzten Anfrage, vor der Auslösung der *PostBack*-Anfrage durch eine Benutzeraktion.
4. Im nächsten Schritt überträgt die *Page* die Zustandsänderungen, die auf dem Klienten stattgefunden haben, an die entsprechenden Controls. Dies können z. B. Eingaben in ein Texteingabefeld oder die Auswahl in einem Listenfeld sein. Dazu ruft die Seite für alle Steuerelemente, die die Schnittstelle *IPostBackDataEventHandler* implementieren, die Methode *LoadPostData* auf. Das Control ändert daraufhin seinen internen Zustand und gibt zurück, ob diese Zustandsänderung ein Ereignis auf dem Server auslösen soll.
5. Als nächstes wird *OnLoad* aufgerufen, die üblicherweise von Subklassen überschrieben wird, um weitere Initialisierungsschritte auszuführen. Zu diesem Zeitpunkt sind bereits alle Änderungen des Benutzers an den Controls übernommen.
6. Bei *PostBack*-Anfragen ruft die Seite für alle Steuerelemente, die in Schritt 4 eine Zustandsänderung angemeldet haben, die Methode *RaisePostDataChangedEvent* auf. Controls,

die diese Methode implementieren, lösen darin Ereignisse aus, um eine Änderung ihres Zustands mitzuteilen. Dies könnte z. B. das *TextChanged*-Ereignis eines Texteingabefelds oder das *SelectionChanged*-Ereignis einer Auswahlliste sein. Zu diesem Zeitpunkt werden die vom Entwickler der Seite registrierten Ereignisbehandlungsmethoden ausgeführt.

7. Ebenfalls nur bei *PostBack*-Anfragen wird *RaisePostBackEvent* auf den Controls aufgerufen, die die Schnittstelle *IPostBackEventHandler* implementieren und die die *PostBack*-Anfrage verursacht haben, also z. B. eine Schaltfläche, die daraufhin ihr *Click*-Ereignis auslöst.

8. In *OnPreRender* haben Subklassen von *Page* eine letzte Möglichkeit Änderungen vorzunehmen, bevor die Seite ausgegeben wird.

9. *SaveViewState* wird rekursiv für jedes Control ausgeführt. Ein Steuerelement gibt hier alle Zustandsinformationen zurück, die es bei einer nachfolgenden *PostBack*-Anfrage in Schritt 3 zur Wiederherstellung seines internen Zustands benötigt.

10. Entwickler von Seiten können *SavePageStateToPersistanceMedium* überschreiben, um alle *ViewState*-Informationen der Controls anstatt in einem versteckten Eingabefeld auf der ausgegebenen Seite an einem anderen Ort zu speichern (siehe Schritt 2). Dies ist z. B. sinnvoll, wenn dieses Feld viele Zustandsinformationen enthält und der Klient oder die Bandbreite eingeschränkt ist.

11. Die Methode *Render* wird auf der Seite und rekursiv auf allen Steuerelementen aufgerufen, um die Ausgabe zu generieren.

12. *OnUnload* kann überschrieben werden, um nach der Ausführung der Seite Serverressourcen wie Dateien oder Datenbankverbindungen freizugeben.

3.3 Entwurf

Das Ziel dieser Arbeit ist die Implementierung eines Rahmenwerks für die Implementierung von Webapplikationen auf Basis eines Content Management Systems. Im Folgenden wird für ein Rahmenwerk auch der im Englischen übliche Begriff „Framework“ verwendet. Es gibt unterschiedliche Definitionen für den Begriff „Framework“. Eine für den vorliegenden Fall passende liefert [29]:

> A (generative) architecture designed for maximum reuse, represented as a collective set of abstract and concrete classes; encapsulated potential behaviour for subclassed specializations.

Wesentlich dabei ist es, dass das Rahmenwerk darauf ausgerichtet ist, die Erstellung neuer Anwendungen zu erleichtern, wobei ausgenutzt wird, dass die zu erstellenden Anwendungen bestimmte Gemeinsamkeiten besitzen, deren Implementierung durch das Framework unterstützt wird. Im Hinblick auf die Wiederverwendbarkeit des Rahmenwerks unterscheidet man zwischen *blackbox* und *whitebox frameworks*. *Blackbox frameworks* ermöglichen ihre Wiederverwendung durch die Bereitstellung konkreter Klassen mit spezifizierten Schnittstellen. Eine Kenntnis der Implementierung

dieser Klassen ist nicht notwendig, was die Verwendung eines *blackbox* Rahmenwerks im Allgemeinen leicht macht. Dagegen beruht die Verwendung eines *whitebox* Rahmenwerks ausschließlich auf der Spezialisierung vorgegebener Komponenten, die selbst nur eine Teilfunktionalität implementieren. Die Spezialisierung durch Vererbung setzt allerdings ein Verständnis der Implementierung der Komponenten des Rahmenwerks voraus, wodurch die Verwendung häufig erschwert wird. Der Vorteil ist die Flexibilität, die dieser Ansatz bietet, da eine Spezialisierung das Verhalten einer Framework-Komponente besser an die Bedürfnisse der Anwendung anpassen kann. Um beiden Anforderungen – einfache Wiederverwendbarkeit und Flexibilität – zu begegnen, bieten Rahmenwerke oft einen *whitebox* Anteil mit abstrakten Klassen und konkrete Spezialisierungen dieser Klassen für die *blackbox* Verwendung an [29] [42].

Diese gemischte Form wird auch als *greybox framework* bezeichnet und ist das Ziel des Entwurfs in diesem Abschnitt. Weiterhin ist zu berücksichtigen, dass das zu implementierende Rahmenwerk in verschiedenen Kontexten einsetzbar sein soll, insbesondere in Verbindung mit verschiedenen Implementierungen der Präsentationsschicht.

3.3.1 Voraussetzungen

Dem Entwurf liegt eine Reihe von Voraussetzungen zugrunde. Zum einen ist die Zielplattform, das Microsoft .NET Framework 1.1, und das bereits in 3.2.6 beschriebene ASP.NET 1.1 als Präsentationsschicht zu berücksichtigen, zum anderen stützt sich der Entwurf auf die Verwendung der CoreMedia SCT 4.2 als Content Management System. Die Wahl des CMS hat insofern Einfluss auf den Entwurf, als dadurch die Form der Kommunikation mit dem CMS und das zugrundeliegene Typmodell für die Inhalte vorgegeben sind. Der beschriebene Entwurf orientiert sich weiterhin an Erfahrungen, die bei CoreMedia im Bereich der Entwicklung von Webanwendungen in den letzten Jahren gewonnen werden konnten sowie teilweise an Komponenten, die dort bereits prototypisch für die Java 2 Plattform entwickelt wurden.

Eine solche vorhandene Komponente ist ein flexibler Objekt-Cache, der im Rahmen dieser Arbeit für die Microsoft CLR portiert und für die Implementierung des Rahmenwerks verwendet wird. Die Cache-Komponente ist sehr flexibel und hat als wichtiger Teil des Rahmenwerks auch Auswirkungen auf den Entwurf. Die gebotene Funktionalität wird daher in 3.3.3 beschrieben, wobei auf eine Erläuterung der Implementierung oder der internen Funktionsweise dieser Komponente ausdrücklich verzichtet wird.

3.3.2 Architektur

Entsprechend der in Abschnitt 3.2 beschriebenen Anforderungen ergeben sich für das Rahmenwerk eine Reihe unterschiedlicher Teilfunktionalitäten, die in bestimmten Anwendungsszenarien auch unabhängig voneinander eingesetzt werden können. Das Framework soll den „Rahmen“ für die Anwendungslogik bilden, dem Anwendungsprogrammierer also Gestaltungsfreiraum bieten und die Implementierung der Anwendung unterstützen, nicht einschränken.

Die identifizierten Teilfunktionen sind in Abbildung 3.4 dargestellt: Zugriff auf das *content repository*, Cache, XML, Verweise, Authentisierung, Personalisierung und die Adapterschicht für

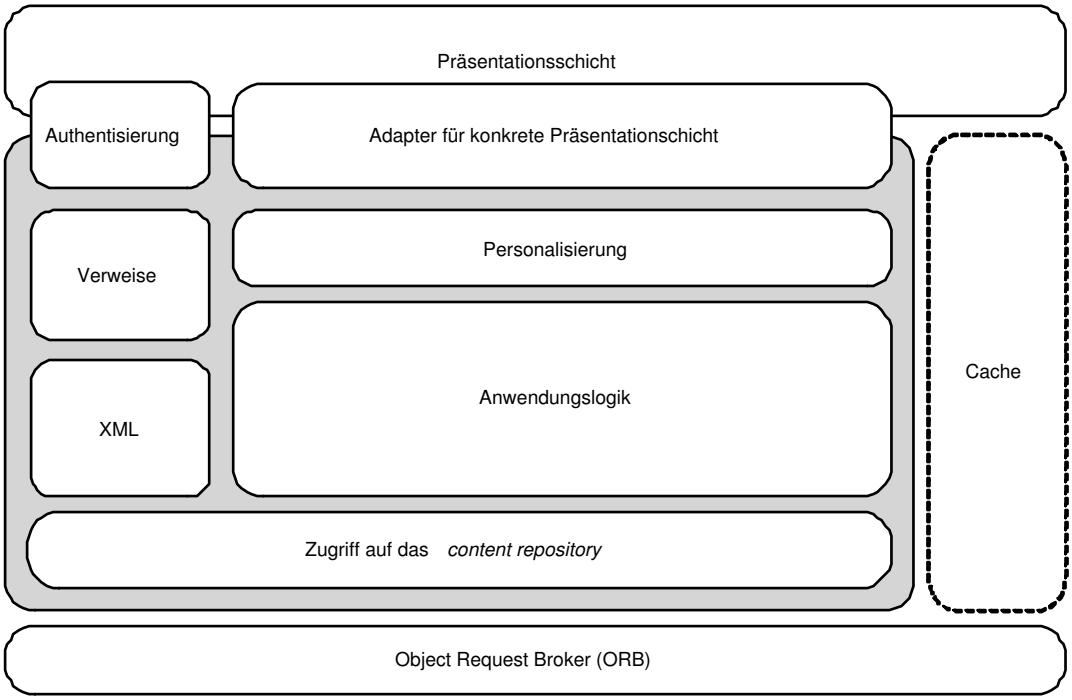

Abbildung 3.4: Teilfunktionen des Rahmenwerks

eine konkrete Präsentationsschicht. Die in dieser Arbeit zu implementierenden Teile sind grau hinterlegt. Die Präsentationsschicht (oben in der Abbildung) sowie ein *object request broker* (ORB, unten) für die Kommunikation mit dem *content repository* über IIOP werden vom Rahmenwerk verwendet und sind daher beim Entwurf zu berücksichtigen. Die Komponente „Cache“ stellt – wie bereits erwähnt – die Basis für eine effiziente Implementierung dar.

In einer vollständigen Webanwendung auf Basis des Rahmenwerks werden alle Teilfunktionen verwendet. Andere Anwendungen dagegen verzichten möglicherweise auf mehrere Teile. So könnte z. B. ein Dienst nur auf Basis der Zugriffsschicht für Inhalte und der Komponenten Cache und XML implementiert werden, um regelmäßige Aktionen auf Inhalten durchzuführen. Eine solche Applikation kommt ohne die Komponenten Verweise, Authentisierung, Personalisierung und eine Präsentationsschicht aus. Beim Entwurf ist daher auf eine Trennung der Funktionalitäten und eine Reduktion der Abhängigkeiten zu achten.

Bei der Verarbeitung einer Anfrage für eine bestimmte URL im Rahmen einer Webanwendung wird diese bei Bedarf zunächst authentisiert, um den Absender der Anfrage zu identifizieren. Dies ist für den späteren Personalisierungsschritt von Bedeutung. Die URL wird dann von der Komponente „Verweise“ aufgelöst. Anhand des Resultats der Auflösung erzeugt die Anwendungslogik unter Zuhilfenahme der Komponenten Cache, XML und CMS-Zugriff ein Präsentationsobjekt (siehe 3.2.3). Die XML-Komponente ist dabei insbesondere für die einfache Verarbeitung von Verweisen in XML-Infosets [4] verantwortlich, die aus dem CMS gelesen werden. Die Cache-Komponente bietet die Basis für die effiziente Erstellung von Präsentationsobjekten aus CMS-Inhalten. Das

Präsentationsobjekt wird schließlich personalisiert, d. h. für den Absender der Anfrage aufbereitet, und an die Präsentationsschicht weitergeleitet, die üblicherweise durch Verwendung von Vorlagen aus dem Präsentationsobjekt das Ausgabeformat wie z. B. HTML erzeugt.

Das Ziel ist es, ausreichend Infrastruktur für die Implementierung CMS-orientierter Anwendungen zu bieten, gleichzeitig aber die Anwendungslogik in den Vordergrund zu stellen. Dazu stellt das Rahmenwerk eine Reihe von Schnittstellen und Verwendungsmustern zur Verfügung, die eine Anwendung implementieren muss, um die Teilfunktionen zu nutzen, so z. B. Schnittstellen für Verweisobjekte, Personalisierungsklassen, Cache-Schlüsselobjekte und andere. Die Standardfunktionalität wird durch vorhandene Klassen geboten, die jedoch durch die Anwendung konfigurierbar und bei Bedarf durch Spezialisierung leicht anpassbar sein sollen. Im Folgenden werden zunächst die Cache-Komponente als Basis für den Entwurf und eine effiziente Implementierung vorgestellt und die genannten Teilfunktionen mit ihren Schnittstellen sowie Kollaborationen untereinander beschrieben.

3.3.3 Cache-Komponente

Die Aufgabe eines Caches ist das Vorhalten häufig verwendeter Werte in einem schnellen Speicher. Caches werden dann verwendet, wenn das initiale Erstellen oder Laden von Werten wesentlich länger dauert als das Laden aus dem schnellen Cache-Speicher und die Werte ausreichend häufig geladen werden müssen. In diesem Fall kann der Vorteil des schnelleren Cache-Speichers ausgenutzt werden. Die Werte selbst leiten sich aus einem externen Zustand ab, also beispielsweise aus dem Zustand eines externen *content repository* oder Dateisystem.

Caches besitzen eine Reihe von Merkmalen, die trotz unterschiedlicher Implementierungen und Anwendungsfelder immer wieder vorzufinden sind. So ist Cache-Speicher in seiner Größe beschränkt. Es muss daher eine Ersetzungsstrategie geben, nach der im Cache liegende Objekte durch neu zu ladende Werte verdrängt werden. Häufig angewandte Strategien sind die Ersetzung von *least recently used* (LRU) oder *least frequently used* (LFU) Werten. Das Ziel aller Ersetzungsstrategien ist es, die Menge der gerade benötigten Werte (*working set*) im Speicher zu halten, wobei die Annahme getroffen wird, dass diese Menge über einen betrachteten Zeitraum relativ konstant bleibt. Der Cache verliert seine leistungssteigernde Wirkung, wenn die Größe des *working set* die Größe des Cache-Speichers übersteigt und somit Ersetzungen häufig stattfinden. Für die Identifikation von Werten ist ein eindeutiger Schlüssel erforderlich, anhand dessen ein Wert aus dem externen Zustand erstellt und im Cache wiedergefunden wird. Schließlich bedarf es eines Benachrichtigungsmechanismus, um einen im Cache abgelegten Wert als ungültig zu markieren („invalidieren“), falls sich der Zustand, aus dem er berechnet wurde, ändert.

Die in dieser Arbeit verwendete Cache-Komponente ist sehr flexibel und besitzt einige Besonderheiten, auf die im Folgenden näher eingegangen wird. Sie dient dazu, aufwendig zu berechnende Objekte zwischenzuspeichern und die Kommunikation mit anderen System – z. B. über eine Netzwerkverbindung mit geringer Bandbreite und großer Latenz – zu reduzieren. Die wesentlichen Elemente der Komponente sind die Klasse *Cache* und die abstrakte Superklasse für alle Schlüsselobjekte, *CacheKey*. Die relevanten Teile der öffentlichen Schnittstellen beider Klassen zeigt Abbildung 3.5. Die Cache-Komponente wurde für diese Arbeit von der Java- auf die .NET-Plattform

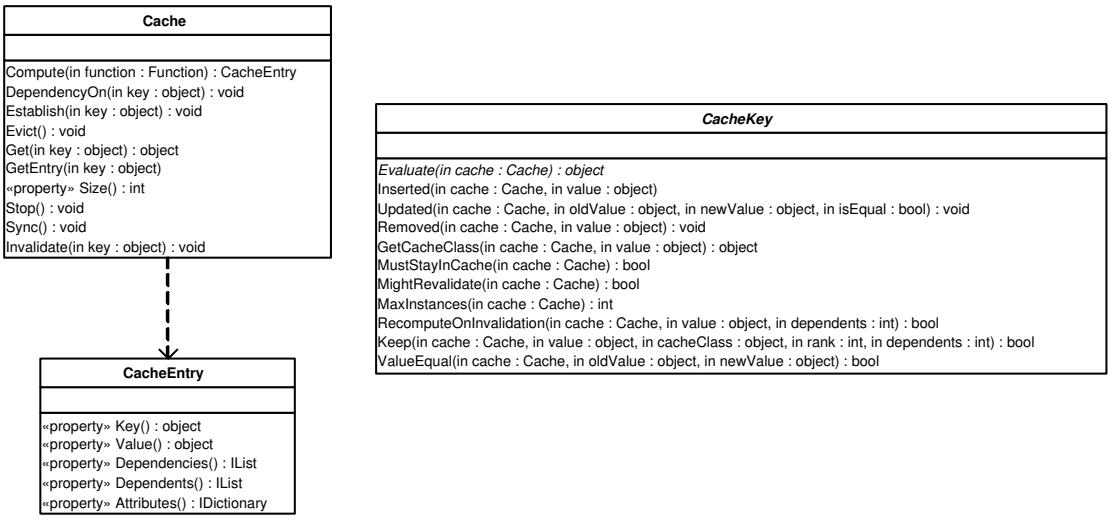

Abbildung 3.5: Relevante Klassen und Operationen der *Cache*-Komponente

portiert.

Eine Subklasse von *CacheKey* definiert eine Schlüsselklasse, beinhaltet die Berechnungsvorschrift für Werte dieser Klasse von Schlüsseln und implementiert indirekt die Ersetzungsstrategie für Werte. Darüber hinaus besitzt *CacheKey*-Operationen, die überschrieben werden können, um auf Ereignisse, die den zugehörigen Wert betreffen, zu reagieren.

Um die Identität eines Schlüsselobjekts zu definieren, muss eine Subklasse von *CacheKey* zunächst die Methoden *Equals* und *GetHashCode* überschreiben. Der zu einem Schlüssel gehörige Wert wird durch die Operation *Cache.Get(CacheKey)* berechnet und in den Cache eingefügt. Für die Berechnung selbst wird die Operation *Evaluate* des Schlüssels ausgeführt, deren Ergebnis der Wert zu diesem Schlüssel ist. Der Typ des Wertes ist dabei beliebig. Ein nachfolgender Aufruf von *Cache.Get* mit einem identischen Schlüsselobjekt liefert ohne erneute Auswertung den gleichen Wert zurück, solange er nicht aus dem Cache entfernt wurde. Das Einfügen des Wertes wird dem Schlüsselobjekt über die Operation *CacheKey.Inserted* mitgeteilt.

Eine Besonderheit dieser Komponente ist das automatische Verfolgen von Abhängigkeiten zwischen Werten im Cache. Bei einem Aufruf von *Cache.Get*($k1$) während der Berechnung des Wertes für den Schlüssel $k2$, wird für den Wert von $k2$ implizit eine Abhängigkeit vom Wert von $k1$ festgestellt. Es entsteht so während der Berechnung von Werten ein gerichteter azyklischer Graph, der von der Cache-Komponente verwaltet wird. Im Fall einer zyklischen Abhängigkeit wird eine Ausnahme ausgelöst. Werte können auch von externen Zuständen abhängen. Diese externen Zustände werden durch Objekte identifiziert, die als *roots* bezeichnet werden. Diese Bezeichnung ist etwas irreführend, da *roots* im Abhängigkeitsgraphen nur als Blätter vorkommen. Der Typ einer solchen „Wurzel“ ist beliebig. Eine Abhängigkeit von einer Wurzel wird der Cache-Komponente explizit durch Aufruf von *Cache.DependencyOn(object)* angezeigt. Ein beispielhafter Abhängigkeitsgraph ist in Abbildung 3.6 dargestellt.

Wurzeln ermöglichen die Invalidierung von Werten im Cache. Die Operation *Cache.Invalidate(object)*

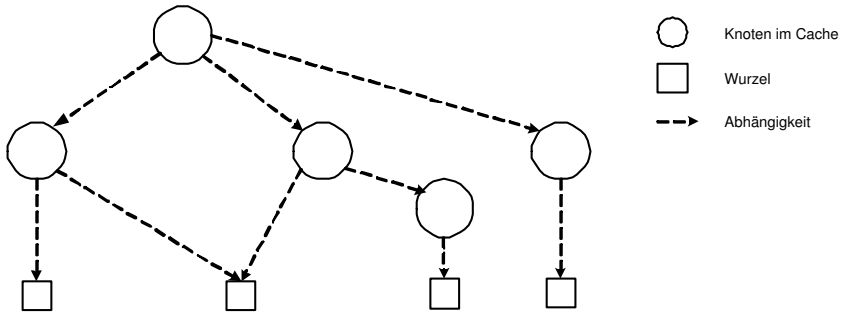

Abbildung 3.6: Beispiel für einen Abhängigkeitsgraphen

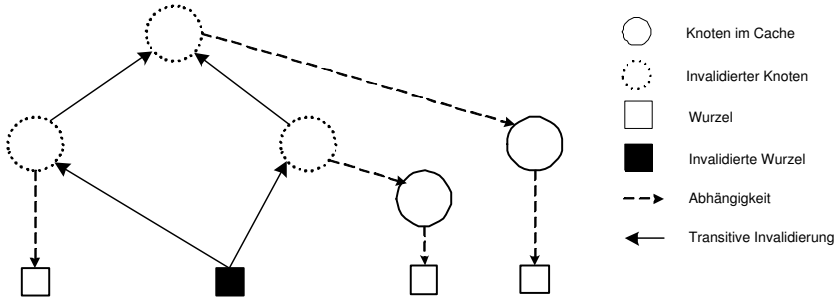

Abbildung 3.7: Invalidierung einer Wurzel und der abhängigen Werte

invalidiert die als Argument übergebene Wurzel und rekursiv alle Werte, die transitiv von ihr abhängen. Die erneute Abfrage eines invalidierten Wertes mittels *Cache.Get* führt dann zu einer Neuberechnung. Abbildung 3.7 zeigt die Invalidierung einer Wurzel im Abhängigkeitsgraphen anhand des vorigen Beispiels. Als eine Optimierung des Invalidierungsprozesses besteht die Möglichkeit, *CacheKey.RecomputeOnInvalidation* zu überschreiben und bei Bedarf *true* zurückzugeben. In diesem Fall wird ein zu dem Schlüsselobjekt gehörender invalidierter Wert sofort neu berechnet und der neue Wert mit dem alten verglichen. Sind beide identisch, wird die Invalidierung an dem aktuellen Knoten im Abhängigkeitsgraphen abgebrochen, da sich abhängige Werte durch eine Invalidierung entlang dieser Kante nicht ändern würden. Eine Invalidierung eines Wertes wird dem Schlüsselobjekt durch Aufruf der Operation *Invalidated* angezeigt. Danach folgt entweder ein Aufruf von *CacheKey.Removed* oder – im Fall von sofortiger Neuberechnung und Änderung des Wertes – *CacheKey.Updated*.

Die Größe des Caches ist nur durch den Hauptspeicher begrenzt, der dem Anwendungsprozess zur Verfügung steht. Damit der belegte Speicherplatz nicht unnötig ansteigt und die Werte im Cache immer wieder auf das *working set* reduziert werden, führt die Cache-Komponente in regelmäßigen Abständen eine Bereinigung durch, bei der nicht mehr benötigte Werte aus dem Speicher entfernt werden. Die periodische Bereinigung entspricht der Ersetzung von Werten in anderen Cache-Implementierungen, nur dass in diesem Fall nicht beim Einfügen ein neuer Wert einen alten ersetzt, sondern kontinuierlich neue Werte eingefügt und in regelmäßigen Abständen

überzählige Werte entfernt werden. Die Entscheidung, ob ein bestimmter Wert entfernt werden soll, wird ebenfalls durch die Schlüsselklasse implementiert. Für jeden abgelegten Wert, ruft die Cache-Komponente *CacheKey.Keep* auf dem zugehörigen Schlüsselobjekt auf. Nur falls *keep false* zurückgibt, wird der Wert entfernt. Die Standardimplementierung liefert genau dann *false* zurück, wenn der Rang des Werts den Rückgabewert von *CacheKey.MaxInstances* überschreitet. Der Rang eines Werts ist die Position an der nach LRU sortierten Liste aller Werte einer Wertklasse (die durch *CacheKey.GetCacheClass* bestimmt wird). Durch Überschreiben von *CacheKey.MaxInstances* ist eine LRU-Ersetzungsstrategie leicht zu realisieren, aber auch komplexere Entscheidungen sind möglich. Wenn ein Wert aus dem Cache entfernt wird, bleibt der Abhängigkeitsgraph unverändert. Die Entfernung eines Wertes aufgrund der Bereinigung wird dem Schlüsselobjekt durch einen Aufruf von *CacheKey.Evicted* mitgeteilt.

Schlüsselobjekte können bei der Ausführung von *Evaluate* nicht nur den Wert selbst zurückgeben, sondern zusätzlich auch den Wert eines Attributs für diesen Cache-Wert setzen. An jedem Knoten im Abhängigkeitsgraphen führt der Cache die Attribute aller abhängigen Werte mit denen des aktuellen Knotens zusammen. Die Attribute werden von der Cache-Komponente selbst nicht ausgewertet, können aber von der Anwendung abgefragt und somit für anwendungsspezifische Aufgaben verwendet werden.

Mehrere Cache-Instanzen können zusammenarbeiten. Dazu wird ein Cache bei einem anderen dem gleichnamigen Entwurfsmuster entsprechend als *Observer* angemeldet [21]. Werte können somit Cache-übergreifende Abhängigkeiten besitzen, wobei die Invalidierung über eine Benachrichtigung an den *Observer* sichergestellt wird.

3.3.4 Zugriff auf das Content Management System

Die in Abbildung 3.4 unten abgebildete Schicht dient dem Zugriff auf Ressourcen und andere im CMS gespeicherte Objekte wie Benutzer, Gruppen oder Berechtigungen. Diese Schicht implementiert sowohl die Kommunikation mit dem CMS als auch die Sitzungsverwaltung für einen angemeldeten Benutzer. Eine Sitzung (*Session*) ist immer einem Benutzer zugeordnet, in dessen Namen Aktionen auf Ressourcen im CMS ausgeführt werden. Bei ausschließlich lesenden Anwendungen wie den hier betrachteten beschränken sich diese Aktionen auf das Lesen von Ressourcen. Das CMS stellt sicher, dass ein Benutzer nur die Ressourcen lesen kann, für die er eine entsprechende Berechtigung besitzt. Die Zugriffschicht muss daher Berechtigungen zwar nicht selbst prüfen, aber die entsprechenden Informationen zugänglich machen, die darüberliegende Schichten benötigen, um solche Berechtigungsprüfungen durchzuführen. Abschnitt 3.4.4 geht auf diesen Punkt genauer ein.

Die Kommunikation mit dem CMS findet üblicherweise über eine Netzwerkverbindung mit einem entfernten *content server* statt, so dass die Kommunikationsgeschwindigkeit der Anwendung mit dem CMS durch die Latenz und Bandbreite der Verbindung beschränkt wird. Die Latenz der Verbindung ist bei der SCT besonders problematisch, da die zur Verfügung gestellten Schnittstellen für viele Aktionen den Austausch mehrerer Nachrichten erfordern, deren Latenzen sich addieren. Für die Implementierung effizienter Anwendungen ist es somit unerlässlich, einen großen Teil der CMS-Daten auf dem Klienten in einem Cache zwischenzuspeichern. Um dies zu unterstützen,

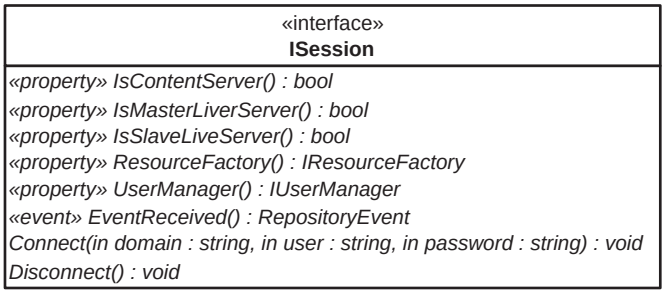

Abbildung 3.8: Schnittstelle eines Sitzungsobjekts

teilt das CMS jeder Sitzung alle Ereignisse mit, die den internen Zustand des *content repository* verändern. Die empfangenen Ereignisse können genutzt werden, um entsprechende Cache-Einträge zu invalidieren. Darüber hinaus ist es eine Anforderung an die Zugriffschicht, die Änderungen am Repository auch der Anwendung zur Verfügung zu stellen.

Abbildung 3.8 zeigt die Sitzung als Einstiegspunkt für den Zugriff einer Anwendung auf Inhalte des CMS. Sie wird zunächst unter Angabe des Benutzernames, einer Domäne und eine Passworts geöffnet. Ein Sitzungsobjekt besitzt Operationen, um Referenzen auf die *IResourceFactory*, den *IUserManager* und den *IRuleManager* zu erhalten. Jede dieser Schnittstellen ermöglicht den lesenden Zugriff auf die entsprechenden Aspekte des Repository. Für die Benachrichtigung einer Anwendung über Änderungen am Repository verfügt die *ISession* über ein Ereignis, für das Objekte als Empfänger registriert werden können. Eine Sitzung muss explizit geschlossen werden, um die belegten Ressourcen auf dem CMS Server freizugeben.

Der wichtigste Aspekt des Zugriffs auf das CMS ist das Auslesen von Inhalten und ihrer Metadaten. Dazu dient die *IResourceFactory*-Schnittstelle. Die *IResourceFactory*-Schnittstelle bietet viele überladene Operationen, um Referenzen auf einzelne Ressourcen zu erhalten. Diese wiederum implementieren die Schnittstellen *IFolder* und *IDocument*. Um auch auf ältere Dokumentversionen zugreifen zu können, besitzt *IResourceFactory* Operationen mit Angabe einer bestimmten Versionsnummer. Alle Ressourcen erlauben die Abfrage ihrer Metadaten wie Erstellungsdatum, letzter Bearbeiter oder ob die Ressource publiziert ist. Über die *Parent*-Eigenschaft ist der Ordner erreichbar, der die aktuelle Ressource enthält. Über die *IFolder*-Schnittstelle sind die in einem Ordner enthaltenen Ressourcen als *ResourceCollection* abrufbar. Eine *ResourceCollection* kann nach beliebigen Kriterien gefiltert werden, wobei häufig verwendete Filter als Operationen der *ResourceCollection*-Schnittstelle zur Verfügung stehen. Ein *IDocument* bietet über die *IResource*-Operationen hinaus die Möglichkeit zur Abfrage der Eigenschaften und des Typs sowie der Versionshistorie eines Dokuments und eine Operation, um eine andere Version des Dokuments zu erhalten. Die erwähnten Schnittstellen sind in Abbildung 3.9 dargestellt.

Die *IResourceFactory* besitzt eine Eigenschaft *IContentModel*, die eine Referenz auf eine Beschreibung der im Repository existierenden Dokumenttypen und ihrer Eigenschaften liefert. Diese Beschreibung ermöglicht die Implementierung generischer Anwendungen, die mit beliebigen Dokumenttypen arbeiten könen.

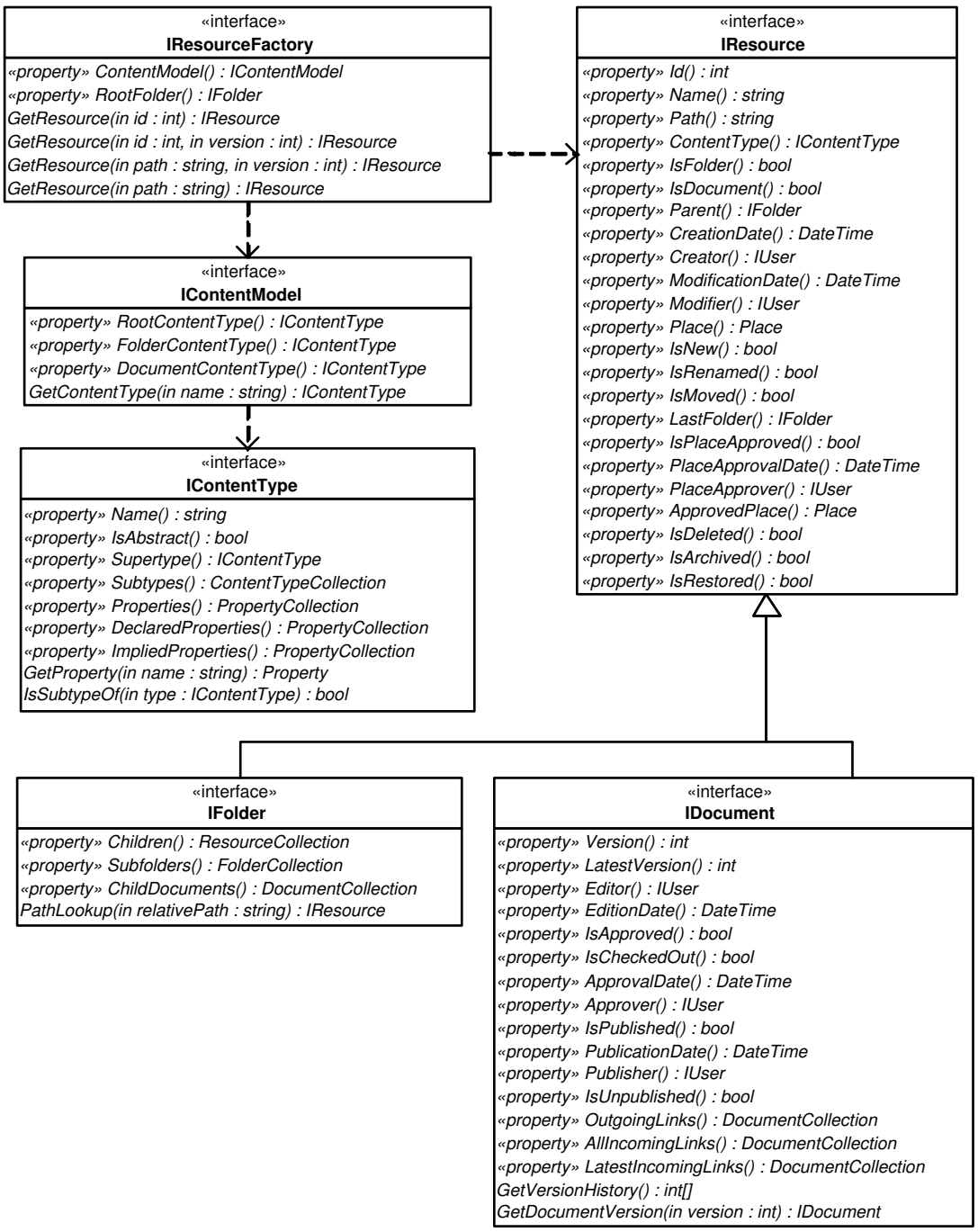

Abbildung 3.9: Schnittstellen für das Auslesen von Inhalten

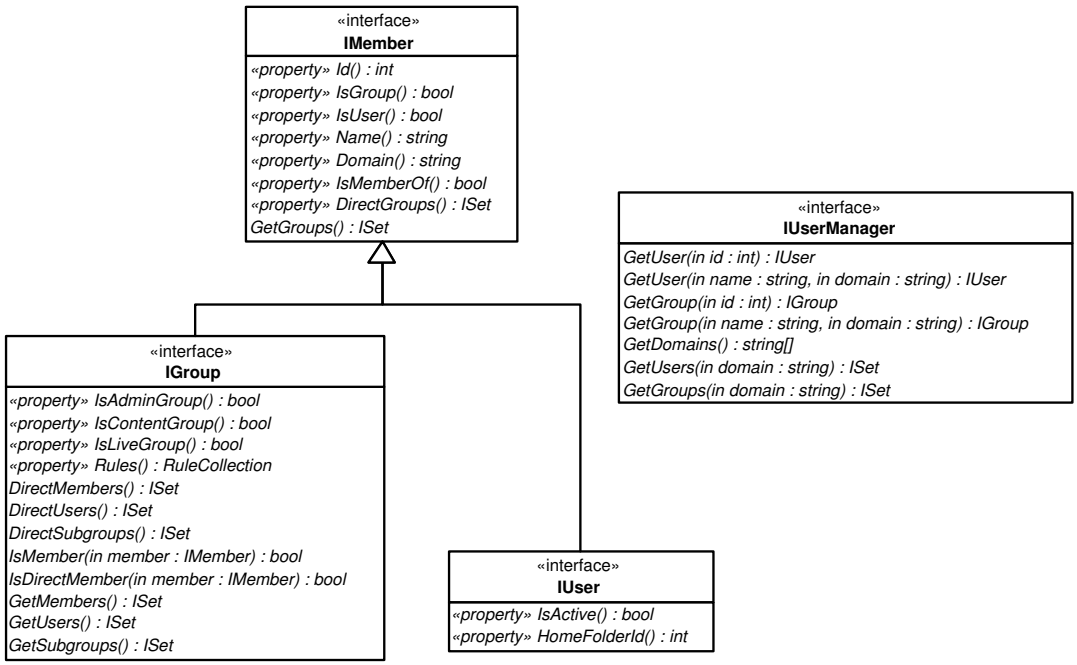

Abbildung 3.10: Schnittstellen für das Auslesen von Benutzer- und Gruppeninformationen

Wie oben beschrieben ist es eine wichtige Funktion der Zugriffsschicht, die Inhalte zwischenzuspeichern und bei einer Änderung im Repository diese zwischengespeicherten Inhalte zu invalidieren, um der Anwendung immer einen aktuellen Stand zu präsentieren. Die Details der Ereignisbehandlung sowie der verwendeten Cache-Schlüssel und der Invalidierung von Cache-Einträgen sind im Abschnitt über Implementierung zu finden.

Weitere Aspekte des CMS-Zugriffs sind das Auslesen von Informationen über Benutzer, Gruppen, ihre Beziehungen untereinander und über Berechtigungen. Die Schnittstelle *IUserManager* bietet die notwendigen Operationen, um alle dem System bekannten Domänen sowie alle bekannten Benutzer und Gruppen einer Domäne auszulesen. Ausserdem gibt es Operationen zum Finden aller direkten oder transitiven Mitglieder (*IMember*) einer Gruppe bzw. aller direkten oder transitiven Gruppen, die ein Mitglied enthalten. Die CoreMedia SCT implementiert Berechtigungen mit Hilfe von Regeln für Gruppen. Die Interpretation von Regeln wird in Abschnitt 3.4.4 erläutert. Eine Übersicht über die Schnittstellen für den Zugriff auf Benutzer, Gruppen, Domänen und Regeln liefert Abbildung 3.10.

3.3.5 Integration des Rahmenwerks in eine Webanwendung

Für die Integration des Rahmenwerks in eine Webanwendung ist es notwendig, dass alle Anfragen abgefangen, ausgewertet und gegebenenfalls umgeleitet werden. Die Auswertung betrifft dabei die Authentisierung, das Umformen einer URL in ein Verweisobjekt, das Auflösen des Verweises in ein Präsentationsobjekt (siehe Abschnitt 3.3.7) und die Verknüpfung mit den entsprechenden Vorlagen.

In ASP.NET ist ein *IHttpModule* ein geeigneter Punkt für eine solche Erweiterung einer Webanwendung. Alle Anfragen werden von einer Kette von *IHttpModule*-Instanzen verarbeitet, wobei die zusätzliche Funktionalität in Form der Modulklasse gekapselt bleibt und leicht in anderen Anwendungen wiederverwendet werden kann. Der Entwurf geht daher im Folgenden von einem *ContentModule* aus, das die *IHttpModule*-Schnittstelle implementiert. Eine ASP.NET-Anwendung kann das Framework nutzen, indem es das *ContentModule* verwendet, das dann auch die Verwaltung der CMS Sitzung übernimmt. Der Abschnitt 3.4.2 enthält Einzelheiten zur Implementierung dieses Moduls.

3.3.6 Authentisierung

Wie bereits erwähnt überprüft der CMS Server die Rechte des Benutzers beim Zugriff auf Inhalte. Dies ist ausreichend für Anwendungen, die sich direkt mit dem CMS verbinden und Inhalte auslesen, weil die Sitzung mit den entsprechenden Benutzerinformationen geöffnet wird. Im Fall einer personalisierten Webanwendung ist es jedoch wünschenswert, dass die Anfragen an das CMS für alle Benutzer der Webanwendung über eine einzige Sitzung bearbeitet werden. Dies ist schon deshalb notwendig, weil das Öffnen einer eigenen Sitzung für jeden Benutzer zu viele Serverressourcen in Anspruch nähme. Es ergibt sich somit für diesen Anwendungsfall, dass jede Webanwendung eine globale Sitzung öffnet, wobei für die Anmeldung ein Benutzer verwendet wird, der alle Ressourcen lesen darf.

Dennoch dürfen häufig nicht alle Benutzer einer personalisierten Webanwendung alle Inhalte lesen. So ist zunächst zwischen authentisierten (an der Webanwendung angemeldeten) und nicht-authentisierten (anonymen) Benutzern zu unterscheiden. Authentisierte Benutzer lassen sich wiederum Gruppen zuordnen, die bestimmte Rechte besitzen. Anhand dieser Rechte kann die Anwendung bestimmen, welche Inhalte für den Benutzer sichtbar sind. Eine Möglichkeit, die Rechte eines authentisierten Benutzers zu bestimmen, ist die Auswertung entsprechender Regeln des *content repository*. Aber auch beliebige andere Verfahren sind denkbar. Voraussetzung für jede Form der Zugriffskontrolle ist aber die Authentisierung.

Die Authentisierung von Anfragen in Webanwendungen ist naturgemäß technologieabhängig. Wir betrachten hier die Situation für ASP.NET, wie sie in Abschnitt 3.2.6 beschrieben ist. Danach kann eine Webanwendung eine von drei Authentisierungsarten wählen: *FormsAuthentication*, *WindowsAuthentication* und *PassportAuthentication*. Jedes Authentisierungsmodul setzt die *User*-Eigenschaft des aktuellen *HttpContext* auf ein entsprechendes *Principal*, das die Identität des Benutzers beschreibt und das eine Methode *IsInRole(string)* besitzt, um die Zugehörigkeit zu einer bestimmten Gruppe zu prüfen. Ohne auf die Besonderheiten des CoreMedia SCT Rechtesystems einzugehen (siehe Abschnitt 3.4.4), kann festgestellt werden, dass ein solcher einfacher Test auf Gruppenzugehörigkeit nicht ausreicht. Statt dessen ist es notwendig, für ein *Principal* die gesamte Hierarchie aller Gruppen abzufragen, deren Mitglied der Benutzer ist.

Daraus ergibt sich für das Rahmenwerk die Notwendigkeit, das *principal* bei Bedarf entsprechend zu erweitern. Um dieser Anforderung nachzukommen, wird eine Schnittstelle *IAuthenticationModule* definiert (Abbildung 3.11), die vom Rahmenwerk verwendet wird, um das *Principal* für eine Anfrage zu setzen. Weiterhin gibt es eine Implementierung der Schnittstelle (*SctAuthentication-*

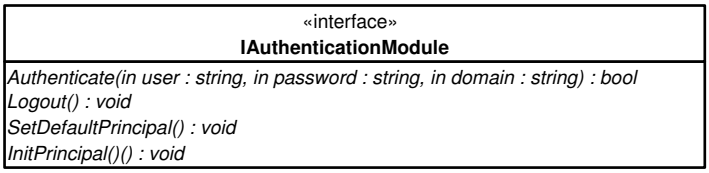

Abbildung 3.11: Schnittstelle *IAuthenticationModule*

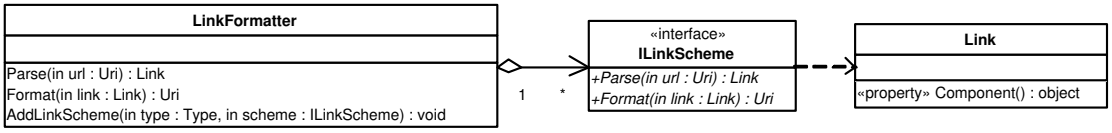

Abbildung 3.12: *ILinkSchemes* als Strategie für die *Link*/URL-Konvertierung

Module), die ein *FormsAuthenticationPrincipal* durch ein erweitertes *SctPrincipal* ersetzen kann, welches die Benutzerinformationen des *FormsAuthentication*-Moduls um die Gruppenhierarchie eines entsprechenden Benutzers im CMS ergänzt. Einzelheiten hierzu finden sich in Abschnitt 3.4.3. Die Einführung dieser Schnittstelle und ihre Verwendung durch das Rahmenwerk ermöglicht die Wiederverwendung von Authentisierungsmodulen in anderen Anwendungen und erleichtert den Austausch eines Moduls durch die klare Trennung von der übrigen Funktionalität.

Unabhängig von der Authentisierung gegen einen CMS Server kann eine Anwendung eine eigene Implementierung der *IAuthenticationModule*-Schnittstelle installieren, um eine eigene Autorisierungsmethode zu unterstützen. Es ist weiterhin anzumerken, dass eine Anwendung auch ohne Veränderung des *principal* auskommen kann, wenn für die Autorisierung eine Einschränkung des SCT-Rechtesystems akzeptabel ist.

3.3.7 Verweise

Abschnitt 3.2.3 beschreibt den Zusammenhang von URLs in Webressourcen und Verweisen zwischen Inhaltsobjekten des CMS. Inhaltsobjekte des CMS werden von der Anwendung in Präsentationsobjekte transformiert, die eine externe Darstellung in Form einer URL besitzen. Dazu werden die Klassen *Link* und *LinkFormatter* sowie die Schnittstelle *ILinkScheme* definiert (siehe Abbildung 3.12). Ein *Link* ist ein eindeutiger Verweis auf ein Präsentationsobjekt eines bestimmten Typs. Für die Konvertierung von URLs und Präsentationsobjekten wird das *Strategy*-Entwurfsmuster [21] wie folgt angewandt: *ILinkScheme* definiert die Schnittstelle für einen Konvertierungsalgorithmus von einem Präsentationsobjekt in eine URL-Repräsentation und zurück. *ILinkScheme*-Implementierungen für verschiedene Objekttypen werden bei einem *LinkFormatter* Objekt registriert, das abhängig vom Typ an die verschiedenen registrierten *ILinkSchemes* delegiert. *LinkSchemes* sind für jede Klasse von Präsentationsobjekten zu implementieren, auf die in einer Anwendung verwiesen wird.

Die Verarbeitung von Verweisen in Form von *Link*-Objekten anstelle von URLs soll es dem Programmierer einer Anwendung erleichtern, mit Verweisen zu arbeiten. Ein *Link* enthält zwar

derzeit nur einen Verweis auf das Präsentationsobjekt, er soll aber später um Parameter erweitert werden. Das *ILinkScheme* ist zustandslos und kapselt die Logik, wie ein solcher Link in Form einer URL dargestellt werden soll. Die Darstellung eines Verweises in einer Anwendung kann somit leicht geändert werden. Die getrennte Kapselung von Verweisinformation und ihrer Darstellung macht nicht nur Fehler unwahrscheinlicher, sondern vermeidet bei der Verarbeitung von Verweisen auch eine ständig wiederholte Interpretation einer URL-Zeichenkette, um die notwendigen Informationen zu extrahieren.

Das *ComponentModule*, das jede Anfrage abfängt, versucht zunächst, die angefragte URL mittels des *LinkFormatters* in ein *Link*-Objekt zu konvertieren, dessen *Component*-Eigenschaft das gewünschte Präsentationsobjekt ist. Umgekehrt kann bei der Ausgabe ein *Link*-Objekt für ein Präsentationsobjekt erstellt und vom *LinkFormatter* in eine URL umgeformt werden.

Zum Erstellen einer konkreten Anwendung auf Basis dieses Frameworks gehört die Implementierung von *Link*-Klassen für die Klassen von Präsentationsobjekten, für die später in der Ausgabe Verweise erzeugt werden sollen. Die Implementierung der *ILinkScheme*-Schnittstelle muss dieses Präsentationsobjekt – meist unter Verwendung von Inhaltsobjekten aus dem CMS – erzeugen. Die Auflösung des Verweises ist als *Evaluate*-Operation eines *CacheKey* implementiert und das Ergebnis somit als Wert im Cache verfügbar.

3.3.8 Personalisierung

In 3.2.4 wurden bereits die Anforderungen an die Personalisierungsfunktion beschrieben. Entsprechend dieser Anforderungen muss es möglich sein, den Komponentengraphen, der aus dem Cache gelesen wird, vor dem Verknüpfen mit Vorlagen an den Kontext der aktuellen Benutzersitzung anzupassen. Als spezieller Anwendungsfall wurde die gefilterte Darstellung nach Leserechten genannt.

Da die Wurzel des Komponentengraphen ein Wert im Cache ist, darf der gesamte Graph nicht modifiziert werden. Für die Personalisierung des Komponentengraphen werden daher die zu ändernden Knoten ausgetauscht. Für einen zu personalisierenden Knoten P wird eine Kopie mit entsprechend veränderten Werten angelegt. Auch alle Knoten auf einem Pfad von der Wurzel zu P werden kopiert und die Objektreferenz auf den nächsten Knoten des Pfads angepasst. Alle Komponenten, die auf keinem Pfad von der Wurzel zu einem zu personalisierenden Knoten liegen, bleiben unverändert. Der Algorithmus für diese Personalisierung des Komponentengraphen ist durch Framework-Klassen implementiert, so dass ein Anwendungsentwickler nur die Personalisierung lokal für eine bestimmte Komponente, also die Anwendungslogik selbst, implementieren muss. Dazu bedarf es für jeden Knoten nur des Hinweises, entlang welcher Kanten von der Wurzel aus nach personalisierten Komponenten gesucht werden soll (siehe Abschnitt 3.4.6).

Die Möglichkeit der Cache-Komponente zur Verwaltung von Attributen für Cache-Werte erlaubt eine automatische Filterung nach Leserechten für verwendete Inhaltsobjekte. Dazu definiert das Framework ein Attribut für die Inhaltsobjekte, die während der Berechnung eines Cache-Wertes ausgelesen werden. Bei der Berechnung aggregierter Werte, die auf andere Werte mit Hilfe von *Cache.Get* zugreifen, wird den aggregierten Werten automatisch als Attribut die Vereinigung aller verwendeten Inhaltsobjekte zugewiesen. Das Framework stellt nun eine Klasse bereit, die

in Form eines *decorators* [21] einen Cache-Wert kapselt und während der Personalisierungsphase die Filterung nach Leserechten auf den verwendeten Inhaltsobjekten vornimmt. Die Aufgabe des Programmierers beim Entwurf der Komponenten für eine Anwendung ist es, an den Kanten, an denen nach Zugriffsrechten gefiltert werden soll, diese Dekoratoren zu verwenden. Der Implementierungsabschnitt geht auf die Dekoratoren und ihre Verwendung im Zusammenhang mit Listen von Komponentenverweisen im Detail ein.

3.3.9 Rendering

Die Rendering-Schicht des Frameworks hat die Aufgabe, von konkreten Umgebungen für die Auslieferung und Präsentation von Seiten und der Interaktion mit dem Benutzer zu abstrahieren. Sie stellt eine Reihe von Schnittstellen bereit, die für verschiedene Umgebungen implementiert werden können, um das Framework in diesen zu verwenden. Darüber hinaus werden die Schnittstellen für ASP.NET als Präsentationsschicht implementiert. Die Rendering-Schicht wurde vollständig im Rahmen dieser Arbeit implementiert, basiert aber auf ähnlichen, existierenden Komponenten von CoreMedia. Insbesondere verwendet sie das Konzept des *ViewDispatchers*, das sich im CoreMedia Active Delivery Server in ähnlicher Form bereits bewährt hat.

Zwei wesentliche Funktionen müssen durch die Rendering-Schicht zur Verfüng gestellt werden, um einen Komponentengraphen an eine Präsentationsschicht weiterzureichen:

1. Ein Objekt finden, das eine gegebene Komponente in einer konkreten Präsentationsumgebung darstellen kann.

2. Eine Möglichkeit, während der Berechnung der Darstellung einer Komponente – z. B. durch Verknüpfung mit einer umgebungsspezifischen Vorlage – die Darstellung anderer Komponenten des Komponentengraphen einzubinden.

Ein Objekt, das eine Komponente darstellen kann, nennen wir „Renderer“. Ein Renderer implementiert die *Renderer* Schnittstelle (siehe Abbildung 3.13), die eine Methode für die Darstellung einer Komponente definiert. Dabei wird ein umgebungsspezifischer *RenderingContext* übergeben. Des weiteren definieren wir die Schnittstelle *ViewDispatcher*, der einen Renderer für eine gegebene Komponente finden kann. Sie bietet ausserdem Methoden zum Registrieren von Renderer-Implementierungen für bestimmte Typen von Komponenten.

Eine *ViewDispatcher*-Implementierung findet für eine Komponente den speziellsten Renderer, der zum Typ der Komponente passt. Die Darstellung einer Komponente durch einen Renderer entspricht damit einem polymorphen Methodenaufruf auf einem Objekt, wie er aus objektorientierten Programmiersprachen bekannt ist. Falls kein entsprechender Renderer gefunden werden kann, ist dies ein Fehler, der einem Aufruf einer nicht definierten Methode auf einem Objekt entspricht. Renderer werden üblicherweise während der Initialisierung der Anwendung an einem *ViewDispatcher* angemeldet und ändern sich während der Laufzeit der Anwendung nicht. Die Verwendung eines *ViewDispatchers* erlaubt es Anwendungsentwicklern, allgemeine Renderer auch für Spezialisierungen von Komponenten einzusetzen, wodurch die Gesamtzahl der zu entwickelnden Renderer gesenkt wird. So ließe sich für Komponentenklassen, die an die Dokumenttypen aus Tabelle 3.1 auf Seite 24 angelehnt sind, ein Renderer für Objekte vom Typ *Content* entwickeln, der

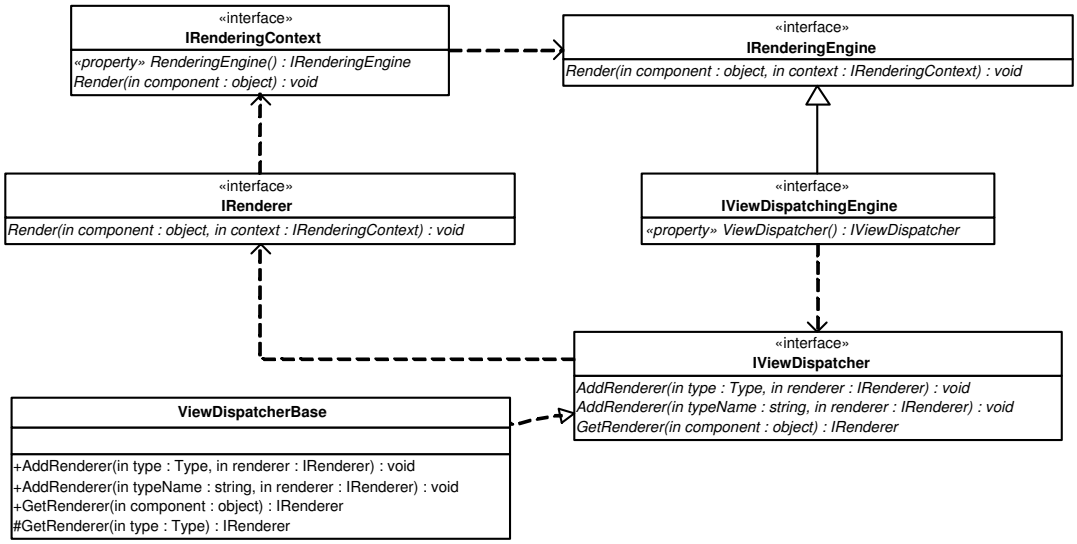

Abbildung 3.13: Schnittstellen und abstrakte Klassen der Rendering-Schicht

die Kurzbeschreibung und das zugeordnete Bild anzeigt. Dieser kann für alle Subtypen von *Content* gleichermaßen verwendet werden. Falls ein speziellerer Render für *Results* registriert ist, wird der *ViewDispatcher* diesen zurückgeben, während für die Darstellung aller anderen Subtypen von *Content* weiterhin der allgemeine Renderer verwendet wird.

Eine *RenderingEngine* bildet den Einstiegspunkt einer Anwendung für die Ausgabe in einer bestimmten Umgebung oder in einer bestimmten Form. So kann eine Anwendung mehrere *RenderingEngines* verwenden, um z. B. verschiedene Ausgabeformate zu produzieren. Eine *RenderingEngine* besitzt eine *factory method* [21] für die Erzeugung des umgebungsspezifischen *RenderingContext* und eine Methode zur Darstellung einer Komponente. Die Spezialisierung *ViewDispatchingEngine* verwendet einen *ViewDispatcher*, um einen geeigneten Renderer zu finden. Der Algorithmus für das Finden eines geeigneten Renderers ist in der abstrakten Klasse *ViewDispatcherBase* implementiert, die einen Cache für die Suchergebnisse benutzt. Implementierungen von *ViewDispatchingEngine* erzeugen einen *ViewDispatchingContext*, so dass während der Berechnung der Ausgabe eine Referenz auf den verwendeten *ViewDispatcher* weiterhin zur Verfügung steht.

Für die Ausgabe ruft das Framework die Methode *Render* auf der registrierten *RenderingEngine* mit der Wurzel des personalisierten Komponentengraphen auf. Es ist dann Aufgabe des Renderers, während der Berechnung der Darstellung wiederum *Render* für andere Komponenten des Graphen aufzurufen, um so die gesamte Ausgabe für die angefragte Seite zu generieren.

Die beschriebenen Schnittstellen sind in Abbildung 3.13 dargestellt. Ihre Implementierung für ASP.NET ist im Abschnitt 3.4.7 beschrieben.

3.4 Implementierung

Aufgrund des Umfangs des in dieser Arbeit implementierten Frameworks ist hier nicht die gesamte Implementierung darstellbar. Statt dessen habe ich einige interessante Aspekte ausgewählt, insbesondere solche, bei denen die Besonderheiten der Zielplattform oder des verwendeten CMS eine Rolle spielen.

3.4.1 Zugriff auf das Content Management System

Die CoreMedia SCT bietet als Protokoll für eine direkte Verbindung mit dem CMS Server das *Internet Inter-ORB Protocol* (IIOP) der *Common Object Requst Broker Architecture* (CORBA) [33] an. Es galt daher zunächst für die Zielumgebung der Microsoft .NET Plattform eine geeignete Protokollimplementierung zu finden. Drei Kandidaten wurden evaluiert:

Remoting.CORBA „Remoting.CORBA“[5] ist eine frei verfügbare IIOP-Implementierung, die allerdings scheinbar nur wenig weiterentwickelt wird. So ist die letzte Version 1.3.1 vom Mai 2003. Nach einem zunächst guten Eindruck stellte sich jedoch heraus, dass Remoting.CORBA viele Konstrukte der *Interface Definition Language* (IDL) wie z. B. Konstanten nicht unterstützt und somit für die Kommunikation mit der CoreMedia SCT nicht einsetzbar ist.

Borland Janeva „Janeva 6“ [40] von Borland ist eine kommerzielle Lösung für die Verknüpfung von Java- und .NET-basierten Systemen. Sie basiert auf füheren Borland Produkten für Java und hat somit eine hohe Reife und Qualität. „Janeva 6“ ist im Hinblick auf die Zuverlässigkeit und Geschwindigkeit sicherlich die beste der drei evaluierten Varianten. Aufgrund seines Preises ist ein Einsatz des Produkts für diese Arbeit jedoch nicht möglich.

IIOP.NET „IIOP.NET“[6] ist wie „Remoting.CORBA“ eine frei verfügbare Implementierung des Protokolls IIOP, wird jedoch zumindest derzeit kontinuierlich weiterentwickelt. Während zu Beginn der Implementierungsphase für diese Arbeit noch einige Funktionen fehlten, sind in der aktuellen Version 1.6.1 alle notwendigen Voraussetzungen für die Kommunikation mit dem CMS Server vorhanden. Als einzige negative Punkte ist der niedrige Durchsatz aufgrund der Implementierung des *Marshallers* sowie unregelmäßige Probleme mit der Zuverlässigkeit aufgefallen. Für diese Arbeit ist IIOP.NET ausreichend geeignet und wurde für die Implementierung des Zugriffs auf das CMS ausgewählt.

Die Inhaltsobjekte des CMS liegen im Fall der CoreMedia SCT in Form von Ressourcen, entweder als Ordner oder als Dokumente, vor (siehe CoreMedia SCT, 2.5.2). Sowohl die Inhalte als auch die Metadaten von Ressourcen werden nach dem ersten Lesen vom CMS als Werte im Cache abgelegt und sind dort verfügbar bis sie invalidiert oder bereinigt werden. Zur Invalidierung dienen Ereignisse, die der CMS Server an alle angeschlossenen Klienten schickt. Jede Änderung am *content repository* löst eine solche Änderung aus, so dass die entsprechenden Cache-Werte auf dem Klienten invalidiert werden können. Anwendungsobjekte im Cache, die direkt oder indirekt

[5] Remoting.CORBA 1.3.1: http://sourceforge.net/projects/remoting-corba
[6] IIOP.NET 1.6.1: http://sourceforge.net/projects/iiop-net

Wert	Beschreibung	Wurzeln
IncomingLinks	Die Menge der Dokumente, die auf ein Dokument verweisen.	resource:id, incominglinks:id
Content	Alle Dokumenteigenschaften einer Version und alle versionsbezogenen impliziten Eigenschaften eines Dokuments. Die Wurzel „content:id:version“ enthält die Versionsnummer, wobei „0“ eingesetzt wird, falls nicht eine bestimmte, sondern die aktuellste Version des Dokuments angefragt wurde.	resource:id, content:id:version
Children	Die Menge aller in einem Ordner enthaltenen Unterordner und Dokumente.	resource:id, folder:id
ImpliedProperties	Eingebaute Eigenschaften, die jede Ressource besitzt und die versionsunabhängig sind. Beinhaltet nicht die *folderId_* Eigenschaft, die den Ort der Ressource im *content repository* bezeichnet.	resource:id, implied:id
Parent	Die ID des übergeordneten Ordners dieser Ressource.	resource:id, parent:id

Tabelle 3.3: Cache-Werte für Inhaltsobjekte

von Ressourceninformationen abhängen, werden beim Eintreffen eines solchen Ereignisses ebenfalls invalidiert. Da Anwendungsobjekte häufig jedoch nicht von allen Daten einer Ressource abhängen, sondern beispielsweise nur von den Inhalten oder nur vom Namen, ist es sinnvoll, die zu einer Ressource gehörenden Informationen in logische Blöcke zu unterteilen, die durch unterschiedliche Cache-Werte repräsentiert werden und somit unabhängig voneinander invalidiert werden können. Wenn zum Beispiel nur eine Abhängigkeit vom Namen eines Dokuments besteht, muss das Anwendungsobjekt beim Anlegen einer neuen Version des Dokuments nicht invalidiert werden. Die Aufteilung der Informationen wird durch die Ereignisse bestimmt, die das CMS versendet. In Tabelle 3.3 ist die Aufteilung der Ressourceinformationen in unterschiedliche logische Blöcke dargestellt. Als Wurzeln des Abhängigkeitsgraphen werden Zeichenketten verwendet, die jeweils die ID der Ressource enthalten. Die rechte Spalte zeigt die Wurzeln von denen die Blöcke abhängen.

Um dem CMS Server anzuzeigen, dass der Klient noch aktiv ist und die Sitzung aufrecht erhalten werden soll, ruft der Klient in einer Endlosschleife die Methode *ping()* auf dem (entfernten) Sitzungsobjekt auf. Der Methodenaufruf kehrt zurück, sobald Ereignisse für den Klienten vorliegen. Diese werden als Rückgabewert der Methode *ping()* übertragen. Tabelle 3.4 zeigt die relevanten Ereignisse, die das *content repository* schickt, und welche Wurzeln des Cache-Abhängigkeitsgraphen daraufhin invalidiert werden, um potentiell ungültige Werte aus dem Cache zu entfernen. Für Benutzer, Gruppen, Mitgliedschaften und Regeln gibt es entsprechende Ereignisse, die hier nicht dargestellt sind.

Für den Anwendungsprogrammierer ist die Aufteilung der Ressourcendaten in verschiedene Cache-Einträge transparent. Dazu wurden die Schnittstellen *Document* und *Folder* (Abbildung 3.9) als *facade* [21] implementiert, die bei der Abfrage von Eigenschaften den entsprechenden Wert aus dem Cache laden und die Abfrage an diesen delegieren.

Ereignis	**Invalidierte Wurzeln**
Created	folder:id (für übergeordneten Ordner)
Destroyed	resource:id (für zerstörtes Dokument), folder:id (für übergeordneten Ordner)
Renamed	implied:id (für umbenannte Ressource), folder:id (für übergeordneten Ordner)
Moved	parent:id (für bewegte Ressource), folder:id (für Quellordner), folder:id (für Zielordner)
Deleted, Undeleted, PlaceApproved, PlaceDisapproved, Synced	implied:id
VersionCreated	content:id:0
Published, CheckedIn, Approved, Disapproved, VersionDestroyed, VersionSaved	content:id:version
IncomingLinksChanged	incominglinks:id

Tabelle 3.4: Ereignisse für Änderungen an Ressourcen

Eine Anforderung an die Zugriffsschicht ist es, Ereignisse des *content repository* für Applikationen verfügbar zu machen. Klassen und Schnittstellen können im CLS-Typsystem Ereignisse definieren, um Applikationskomponenten lose zu koppeln. Es ist daher wünschenswert, die vom CMS empfangenen Ereignisse aufzubereiten und als Ereignisse des Sitzungsobjekts anzubieten. Die Behandlung von Serverereignissen ist in der Klasse *EventCollector* implementiert. Ihre Aufgabe ist es, durch wiederholtes Aufrufen der Methode *Ping* Ereignisse zu empfangen und Objekte zu erstellen, die die Ereignisse beschreiben. Daraufhin wird zunächst ein internes Ereignis mit der Beschreibung als Argument ausgelöst, welches von der *ResourceFactory* empfangen und dort in Cache-Invalidierungen umgesetzt wird. Anschließend wird ein Ereignis ausgelöst, das *EventCollector* über eine öffentliche Schnittstelle exportiert, und das somit für andere Komponenten der Anwendung zur Verfügung steht. Der Zwischenschritt über das interne Ereignis ist notwendig, damit die Cache-Werte bereits invalidiert sind, wenn externe Komponenten die Ereignisse bearbeiten. Um die verschiedenen Schritte voneinander zu entkoppeln, startet *EventCollector* drei voneinander unabhängige Threads:

1. Thread 1 ruft in einer Endlosschleife *Ping* auf dem entfernten *Session*-Objekt auf, erstellt Objekte mit den Ereignisinformationen und stellt sie in die Warteschlange *internalQueue*.

2. Thread 2 entnimmt die Ereignisbeschreibungen aus der *internalQueue* und löst die internen Ereignisse aus. Diese werden von *ResourceFactory* empfangen und invalidieren die Cache-Einträge. Danach werden die Ereignisbeschreibungen in die Warteschlange *externalQueue* gestellt.

3. Thread 3 entnimmt die Ereignisbeschreibungen aus *externalQueue* und löst die öffentlichen Ereignisse aus, die von externen Anwendungskomponenten weiterverarbeitet werden können.

Durch Auslösung der internen und externen Ereignisse in unterschiedlichen Threads kann verhindert werden, dass ein Fehler in einer externen Anwendungskomponente die weitere Invalidierung von Cache-Einträgen verhindert. Auf diese Weise wird außerdem sichergestellt, dass *Ping* kontinuierlich aufgerufen wird, da andernfalls der Server die Sitzung abbrechen könnte.

3.4.2 Steuerung des Ablaufs in einer Webanwendung

In diesem Abschnitt geht es darum, wie die bisher beschriebenen Verarbeitungsschritte des Frameworks in den Ablauf einer Webanwendung auf Basis einer gegebenen Technologie – hier ASP.NET – einzubinden ist.

Tabelle 3.2 auf Seite 39 zeigt die Verarbeitung einer HTTP-Anfrage durch ASP.NET. Ein zentraler Aspekt ist die Ausführung des *HttpHandler*, der die eigentliche Ausgabe erzeugt. Der am häufigsten verwendete Handler ist der *PageHandler* für die Verarbeitung sogenannter *web forms*. Dies sind Dateien mit der Endung „.aspx“, denen ein Komponentenmodell für die Erstellung interaktiver Webseiten zugrunde liegt, wie in Abschnitt 3.2.6 beschrieben. *web forms* bieten eine Reihe von Möglichkeiten, die es einfach machen, mit ihrer Hilfe Webseiten zu definieren, selbst wenn sie nicht interaktiv sind und daher von ihrem umfangreichen Ereignismodell kein Gebrauch gemacht wird. So definieren sie das Layout einer Seite getrennt von eventueller Applikationslogik und bieten die Möglichkeit, die Eigenschaften der dargestellten visuellen Komponenten an beliebige Eigenschaften von Anwendungsobjekten zu binden. Darüber hinaus gibt es eine sehr gute Unterstützung des Entwicklungsprozesses durch kommerzielle und frei verfügbare Werkzeuge sowie einen großen Markt für wiederverwendbare visuelle Komponenten. Aus diesen Gründen liegt es nahe, bei der Implementierung des Frameworks für ASP.NET auf *web forms* für die Definition von Vorlagen, d. h. für die Beschreibung der Ausgabe, zu verwenden. Es wird also das Ziel sein, als Antwort auf eine angefragte URL (die einen Link auf ein Präsentationsobjekt darstellt), ein *web form* auszuführen, das über den im letzten Kapitel beschriebenen Rendering-Mechanismus gefunden wurde.

Um die Verarbeitungsschritte des Frameworks an bestimmten Stellen der ASP.NET-Pipeline auszuführen, implementiert die Klasse *ComponentModule* die Schnittstelle *IHttpModule*. Ein solches Modul kann über eine Konfigurationsdatei zu einer bestehenden Anwendung hinzugefügt werden und die in oben genannter Tabelle aufgeführten Ereignisse empfangen.

Zu Beginn einer Anfrage (Ereignis *BeginRequest*) werden bereits die wichtigsten Schritte ausgeführt. Zunächst löst ein konfigurierter *LinkFormatter* die angefragte URL in ein Präsentationsobjekt auf. Dabei wird anwendungsspezifischer Code ausgeführt, der für das Erstellen des Präsentationsobjekts verantwortlich ist. Falls es sich bei der URL nicht um eine gültige Anfrage für ein Präsentationsobjekt handelt, wird die Anfrage vom *ComponentModule* ignoriert. Die ASP.NET-Pipeline wird dann weiter ausgeführt und es ist so auch möglich *web forms* in der Anwendung zu verwenden, die nicht durch das Framework verwaltet werden.

Ein wichtiger Punkt ist an dieser Stelle, dass das Auflösen der URL zu einem Präsentationsobjekt in Form eines *CacheKey* (3.3.3) implementiert ist. Dadurch wird bei erneuter Anfrage mit der

gleichen URL das Präsentationsobjekt häufig sofort im Cache gefunden. Auch das anwendungsspezifische Erstellen eines Präsentationsobjekts läuft somit im Kontext einer *CacheKey*-Evaluierung ab, wodurch alle Abhängigkeiten auf dabei verwendete Inhaltsobjekte des CMS automatisch aufgezeichnet und dem Präsentationsobjekt zugeordnet werden. Bei einer relevanten Änderung im *content repository* wird die Abbildung der URL auf das Präsentationsobjekt daher ungültig und es ist beim nächsten Zugriff erneut zu berechnen. Das Finden bzw. Erstellen des Präsentationsobjekts, das alle darzustellenden Informationen enthält oder referenziert, ist durch die Verwendung der Cache-Komponente mit Abhängigkeitsverwaltung hoch effizient. Wegen der ereignisbasierten Invalidierung der Cache-Werte sind die dargestellten Informationen ausserdem immer aktuell. Damit sind zwei wichtige Anforderungen an das Framework erfüllt.

Als nächstes erzeugt das Modul einen *RenderingContext* durch Aufruf der Methode *CreateContext* auf einer konfigurierten *RenderingEngine*. Für die weitere Verarbeitung legt das *ComponentModule* ein Objekt vom Typ *ComponentRequestContext* an, das das erstellte Präsentationsobjekt, die ursprünglich angefragte URL, die bei der Berechnung aufgezeichneten Cache-Attribute und den *RenderingContext* enthält. Dieses Objekt wird der aktuellen Anfrage zugewiesen, so dass diese Informationen späteren Verarbeitungsschritten der Pipeline zur Verfügung stehen. Durch den Aufruf von *RenderingContext.Render(component)* wird die Verarbeitung des *BeginRequest*-Ereignisses abgeschlossen. Wie ein späterer Abschnitt noch zeigen wird, ändert die Methode *Render* indirekt den Pfad der aktuellen Anfrage auf das auszuführende *web form*, das dem Präsentationsobjekt *component* zugeordnet ist. Dies führt dazu, dass das *web form* später in der Verarbeitungskette als Handler ausgeführt wird.

Beim Auslösen des Ereignisses *AuthenticateRequest* leitet das *ComponentModule* dieses an ein evtl. konfiguriertes *AuthenticationModule* weiter (siehe nächster Abschnitt).

Das Ereignis *AuthorizeRequest* kann von einer Anwendung behandelt werden, um zu prüfen, ob der authentisierte Benutzer zugriffsberechtigt für das anzuzeigende Präsentationsobjekt ist. Dazu stehen ihr die Informationen im *ComponentRequestContext* zur Verfügung. Das Framework selbst führt keine Autorisierung durch, bietet aber eine Möglichkeit, ein *AuthorizationModule* zu konfigurieren, anhand dessen Berechtigungen für CMS-Inhaltsobjekte abgefragt werden können.

Das letzte vom *ComponentModule* behandelte Ereignis ist *AcquireRequestState*. Es wird von ASP.NET ausgelöst, sobald der Zustand der zu der Anfrage gehörenden Benutzersitzung wiederhergestellt ist. Dies ist der Zeitpunkt für die Personalisierung des Komponentengraphen, dessen Wurzel das gefunde Präsentationsobjekt ist. Die Referenz auf das Präsentationsobjekt im *ComponentRequestContext* wird durch das personalisierte Objekt ersetzt.

Die Abfolge der Verarbeitungsschritte und die Interaktion des *ComponentModule* mit anderen Komponenten ist im Sequenzdiagramm 3.14 dargestellt. Die einzelnen Schritte sind in den folgenden Abschnitten im Detail beschrieben.

3.4.3 Authentisierung

Um eine personalisierte Anwendung zu implementieren ist es notwendig, Benutzer zu authentisieren. ASP.NET bietet in Verbindung mit dem IIS drei Authentisierungsmethoden an, die jeweils die *User*-Eigenschaft der aktuellen Anfrage auf ein *principal* setzen, das den authentisierten Be-

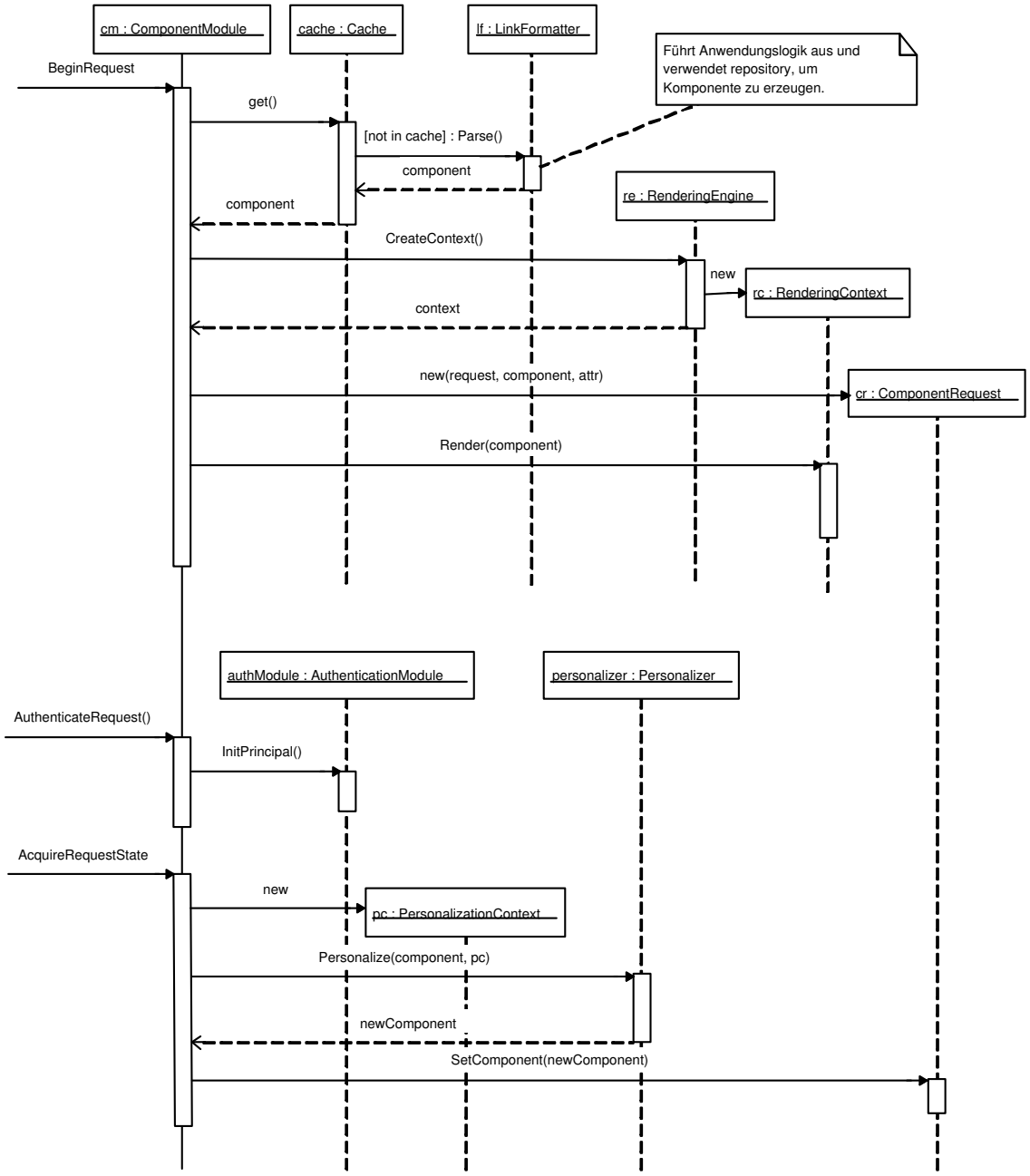

Abbildung 3.14: Sequenzdiagramm der Verarbeitungschritte durch das *ComponentModule*

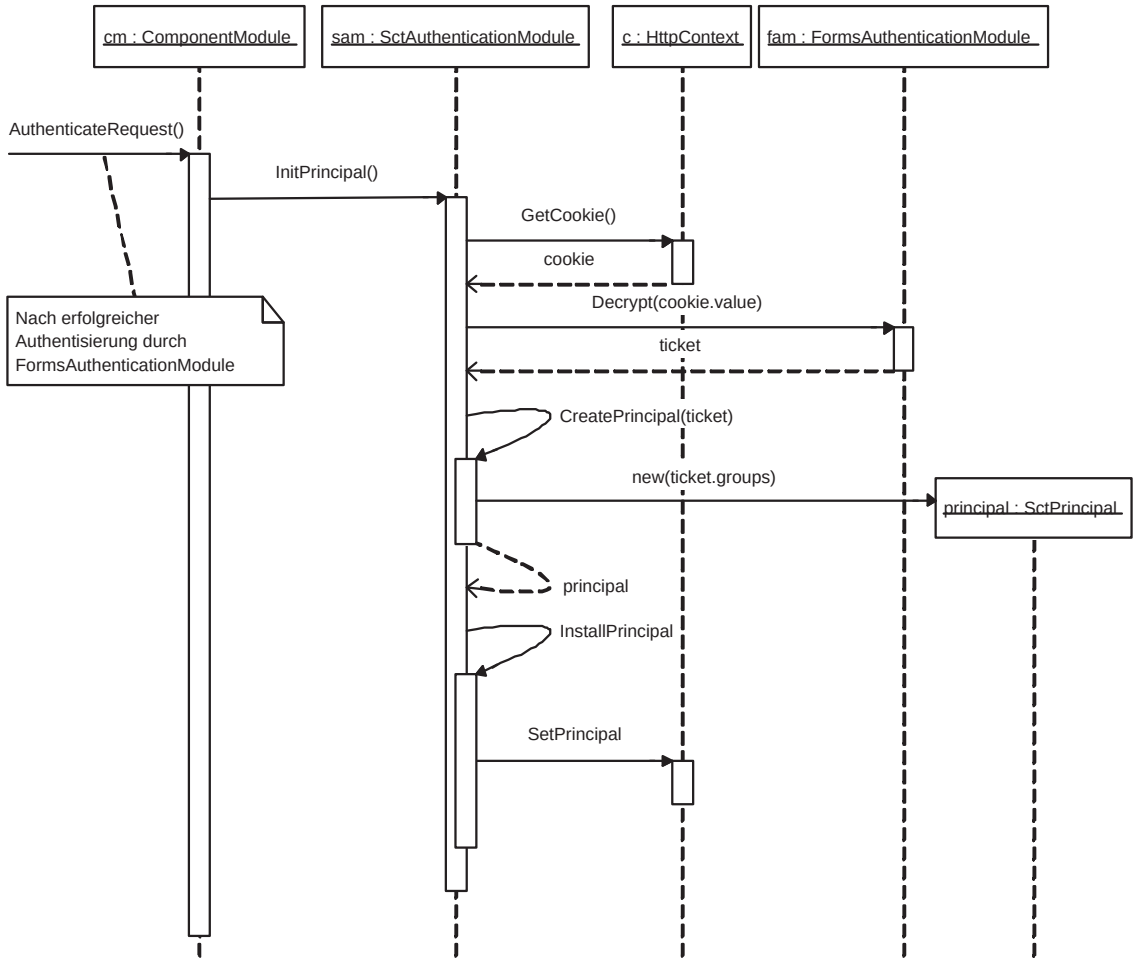

Abbildung 3.15: Sequenzdiagramm der Authentisierung durch das *SctAuthenticationModule*

nutzer mit seiner Identität und seinen Rollenzugehörigkeiten repräsentiert. Dabei ist das *FormsAuthenticationModule* (siehe 3.2.6) dazu geeignet, eine eigene Authentisierung zu implementieren. Abschnitt 3.3.6 beschreibt wie eine Implementierung der Schnittstelle *AuthenticationModule* bereitgestellt werden kann, um eine anwendungsspezifische Authentisierung zu implementieren. So prüft die Klasse *SctAuthenticationModule* einen Benutzernamen und ein Passwort gegen die CoreMedia SCT Benutzerverwaltung und setzt die *User*-Eigenschaft der Anfrage auf ein *rincipal*, das auch alle Gruppen enthält, deren Mitglied der Benutzer ist. Ausserdem kann eine Menge von Gruppen definiert werden, die anonymen Anfragen, also Anfragen von nicht authentisierten Benutzern, zugeordnet wird. *SctAuthenticationModule* nutzt die Funktionalität des *FormsAuthenticationModule*, um einen *cookie* mit einem verschlüsselten Ticket auf dem Klienten abzulegen, das die Benutzerinformationen mit der Gruppenzugehörigkeit enthält. Für eingehende Anfragen, die über *FormsAuthentication* authentisiert sind, werden die Gruppeninformationen aus dem *cookie* ausgelesen und das *principal* ersetzt (Abbildung 3.15).

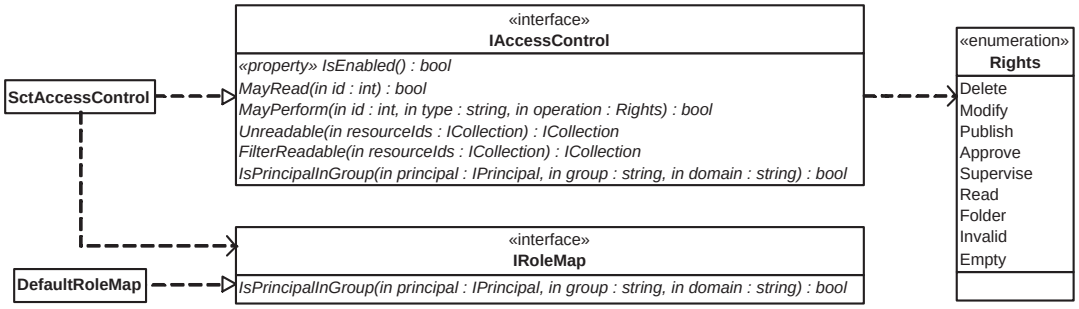

Abbildung 3.16: Schnittstellen und Klassen für die Autorisierung

3.4.4 Autorisierung

Das Rechtesystem der CoreMedia SCT ist sehr komplex. Für bestimmte Anwendungen kann es im Rahmen der Personalisierung wünschenswert sein, Informationen für einen Benutzer auszublenden. Wenn diese Entscheidung aufgrund der im CMS definierten Berechtigungen erfolgen soll, sind zwei Voraussetzungen nötig:

1. Da die CoreMedia SCT Rechte für Gruppen verwaltet, muss einer Anfrage eine Menge von SCT-Gruppen zugeordnet werden, für die die Rechte geprüft werden.

2. Um nicht für jede Rechteprüfung mit dem CMS Server kommunizieren zu müssen, ist das komplexe Rechtesystem auf der Seite der Webanwendung nachzubilden, wobei Benutzer, Gruppen und Regeln im Cache abgelegt und bei Änderungen invalidiert werden.

Die erste Anforderung kann auf zwei Arten erfüllt werden. Die erste Möglichkeit ist die Verwendung des im vorhergehenden Abschnitt beschriebenen *SctAuthenticationModule*, da es ein *SctPrincipal* mit den notwendigen Gruppeninformationen erzeugt. Eine zweite Möglichkeit ist die Verwendung eines beliebigen anderen Authentisierungsmoduls, z. B. des *WindowsAuthenticationModule* (siehe 3.2.6), und die Angabe einer Abbildung von SCT-Gruppen auf Rollen, die zu dem durch das Authentisierungsmodul erstellten *principal* passen. Im Fall des genannten *WindowsAuthenticationModule* wären diese Rollen beispielsweise die Gruppen der Windows-Domäne. Eine solche Abbildung nennen wir *RoleMap*.

Um die Personalisierung anhand von Zugriffsrechten auf CMS-Inhaltsobjekten zu unterstützen, bietet das Framework die Schnittstelle *IAccessControl* und eine Implementierung *SctAccessControl* (siehe Abbildung 3.16), die sowohl mit den Gruppeninformationen eines *SctPrincipal* als auch mit einer *IRoleMap* für beliebige *principals* arbeiten kann. Bei der Konfiguration ist darauf zu achten, dass *SctAccessControl* entweder in Verbindung mit *SctAuthenticationModule* verwendet oder eine *IRoleMap* bereitgestellt wird. *IRoleMap* ist wiederum eine Schnittstelle, für die eine Implementierung *DefaultRoleMap* existiert, die die Abbildungsinformationen aus der Konfigurationsdatei *web.config* liest. Eine Anwendung kann statt dessen auch eine eigene Implementierung zur Verfügung stellen, um beispielsweise Rollen dynamisch auf CMS Gruppen abzubilden.

SctAccessControl enthält eine Implementierung des komplexen SCT-Rechtesystems, um Rückfragen zum CMS Server zu vermeiden. Das SCT-Rechtesystem sieht eine Spezialisierung von Regeln

entlang dreier Achsen vor: der Ordnerhierarchie im *content repository*, der Dokumenttyphierarchie und der Mitgliedschaft in Gruppen. Eine Gruppe erbt alle Regeln der Gruppen, deren transitives Mitglied sie ist, kann diese aber spezialisieren. Daraus folgt, dass für eine exakte Berechnung der Rechte eines Benutzers für ein gegebenes Inhaltsobjekt nicht nur dessen Pfad im *content repository*, sein Typ und alle im System definierten Regeln bekannt sein müssen, sondern auch die genaue Hierarchie der Gruppen, deren Mitglied der Benutzer ist. Falls das *SctAuthenticationModule* verwendet wird, ist diese Gruppenhierarchie bekannt, da sie bei der Anmeldung vom Server gelesen und im *SctPrincipal* abgelegt wird. Dieses besitzt neben der Methode *IsInRole* aus der *IPrincipal*-Schnittstelle auch eine Eigenschaft *GroupHierarchy*, die die entsprechende Information enthält. Wenn jedoch ein anderes Authentisierungsmodul in Verbindung mit einer *RoleMap* verwendet wird, steht nur die *IsInRole*-Methode zur Verfügung, die Gruppenhierarchie ist dann nicht bekannt. Daher kann eine solche Konfiguration nur mit einer flachen Gruppenhierarchie verwendet werden, bei der die Information über die direkte Mitgliedschaft in einer Gruppe für das Finden aller relevanten Regeln ausreicht.

Um zu verdeutlichen welcher Art die Regeln des SCT-Rechtesystems sind, werden diese im Folgenden beschrieben. Aus Platzgründen ist der in *SctAccessControl* implementierte Algorithmus zur Berechnung der Rechte eines Benutzers auf einer Ressource anhand der Gruppenhierarchie, des Dokumenttyps und des Pfads sowie die Verwendung der Cache-Komponente in diesem Zusammenhang nicht dargestellt.

Eine Regel (*Rule*) ist ein Tripel R mit

$$R = (r, g, rt, rights), r \in R, g \in G, rt \in \{*, +\} \cup DT, rights \subseteq Rights(rt)$$

.

Dabei ist R die Menge aller Ressourcen, G die Menge aller Gruppen und DT die Menge aller definierten Dokumenttypen. Die durch eine Regel vergebenen Rechte beziehen sich nur auf den durch rt bestimmten Typ. Dabei ist $* = DT$, mit $rt = *$ gilt die Regel also für alle definierten Dokumenttypen. Falls r ein Dokument ist, ist rt implizit der Typ dieses Dokuments. Die möglichen Rechte sind defininiert als

$$Rights(rt) = \begin{cases} Rights_{Folder} & , rt = + \\ Rights_{Document} & , rt \in \{*\} \cup DT \end{cases}$$

Tabelle 3.5 zeigt die möglichen Rechte für die verschiedenen Ressourcetypen und ihre Abkürzungen.

3.4.5 XML-Unterstützung

Das Rahmenwerk beinhaltet einige Hilfsklassen, um die Verarbeitung von Verweisen in XML-Dokumenten zu vereinfachen. Die CoreMedia SCT verwendet XML-Eigenschaften, um längere Texte zu speichern. In diesen XML-Dokumenten sind Verweise auf andere Inhaltsobjekte und auf Eigenschaften vom Typ „Blob“ in Form von XLink-Attributen enthalten. Verweise nach der XLink-Empfehlung des *World Wide Web Consortium* (W3C) sind `<a>` Elemente, die mindestens

$Rights_{Folder}$	$Rights_{Document}$	Beschreibung
R	R	Lesen (*read*)
	M	Modifizieren (*modify*)
	D	Löschen (*delete*)
A	A	Zur Publikation freigeben (*approve*)
P	P	Publizieren (*publish*)
S	S	Rechte vergeben (*supervise*)
F		Ordner anlegen/löschen (*folder*)

Tabelle 3.5: Mögliche Berechtigungen für Ordner und Dokumente

ein `xlink:href` und ein `xlink:type` Attribute enthalten.

Der wichtigste Anwendungsfall für eine Transformation der XML-Inhalte ist die Ausgabe als HTML, wobei interne Verweise – das sind in den Text eingebettete Verweise auf andere Inhaltsobjekte oder BLOB-Eigenschaften – geändert werden müssen. In der endgültigen Ausgabe dürfen nur solche Verweise zu finden sein, die auch als Anfrage von der Anwendung verstanden werden, und das sind wiederum *Link*-Objekte, die auf ein Präsentationsobjekt verweisen und durch einen entsprechenden *LinkFormatter* in eine Zeichenkettenrepräsentation überführt wurden. Die im Framework enthaltenen Klassen ermöglichen eine Verarbeitung dieser Verweise als *Link*-Objekte.

Elemente mit XLink-Verweis können durch eine Subklasse von *LinkFilterBase* verarbeitet werden, die eine von zwei *GetHandlerFor* Methoden überschreibt (Abbildung 3.17). Beide bekommen Informationen über das Element und seine Attribute sowie den Inhalt des `xlink:href` Attributs und ggf. das daraus erzeugte *Link*-Objekt übergeben. Sie geben eine Implementierung von *Handler* zurück, die speziell implementiert oder durch Aufruf einer der anderen geschützten Methoden erzeugt werden kann:

CreateRewriteElementHandler Ändert den Namen, das Präfix, den Namensraum und die Attribute des Elements.

CreateElementHrefHandler Ändert den Namen, das Präfix, den Namensraum und die Attribute des Elements, wobei der geänderte Verweis je nach verwendeter Methode als *Link* oder als Zeichenkette übergeben werden kann.

CreateDropElementHandler Entfernt das Element mit dem Verweis, behält aber die Kindelemente bei.

CreateDropElementAndChildrenHandler Entfernt das Element mit dem Verweis und alle seine Kindelemente.

Eine Anwendung für einen selbst implementierten *Handler* ist z. B. das dereferenzieren des Verweises und das Ersetzen des Elements durch referenzierte Inhalte.

Die Subklasse *HrefRewriter* ist für den einfachen Fall geeignet, dass nur die Inhalte der `xlink:href` Attribute geändert werden sollen. Eine Subklasse von *HrefRewriter* überschreibt dazu eine der drei *Rewrite* Methoden. Je nach verwendeter Überladung wird der Inhalt des Attributs als Zeichenkette, als *Link* oder beides übergeben und kann von der Subklasse nach Belieben transformiert werden. Die Klasse *HtmlLinkRewriter* transformiert XLink-Verweise in eine HTML-Darstellung.

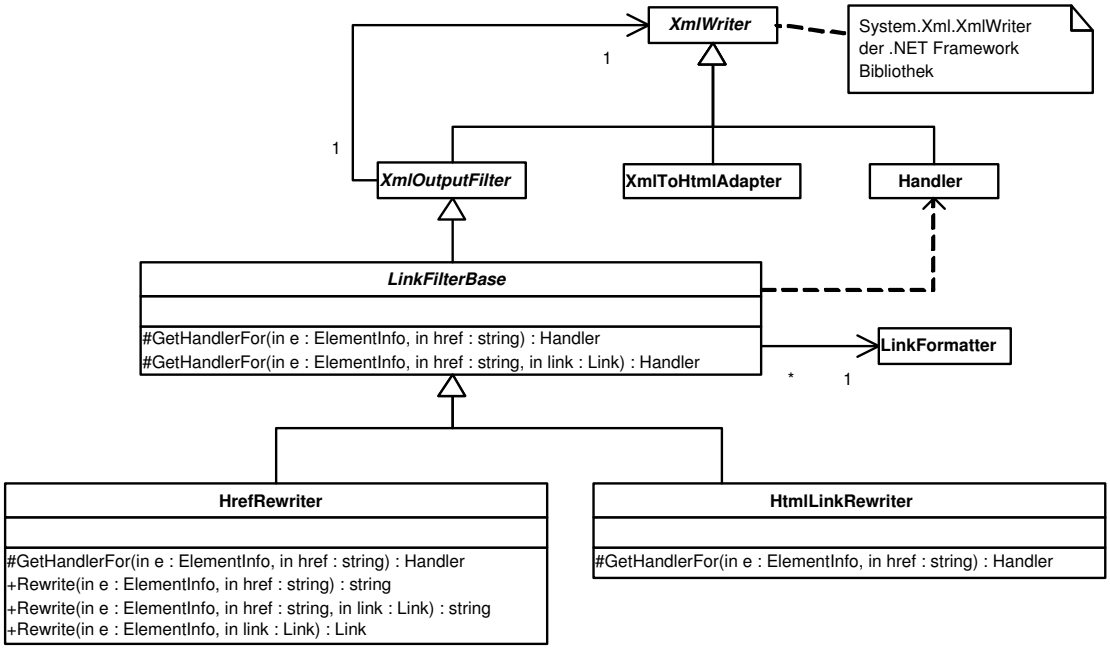

Abbildung 3.17: Klassen für die Verarbeitung von Verweisen in XML-Dokumenten

Alle genannten Klassen erweitern *XmlOutputFilter* und können daher in einer Filterkette aneinandergehängt werden, um mehrere Transformationen bei der Ausgabe von XML auszuführen (*pipes and filters* Architekturmuster, [5]). Die Klasse *XmlToHtmlAdapter* erweitert *XmlWriter* und gibt das XML als HTML aus. Sie sollte bei der Ausgabe von XML als HTML-Fragment das letzte Glied einer Filterkette sein.

3.4.6 Personalisierung

Wie in 3.3.8 beschrieben besteht die Grundidee des Personalisierungsschritts darin, einen Teil des Komponentengraphen durch eine Variante zu ersetzen, deren Werte für den aktuellen Benutzer angepasst oder gefiltert sind.

Der Implementierung liegen folgende Überlegungen zugrunde:

- Der Komponentengraph ist ein Graph mit Präsentationsobjekten als Knoten, die durch Objektreferenzen verbunden sind. Die öffentlichen Eigenschaften eines Objekts entsprechenden den Kanten zu den Kindknoten. Falls der Typ einer öffentlichen Eigenschaft die Schnittstelle *ICollection* implementiert oder erweitert, entspricht dies mehreren Kanten, da die Eigenschaft auf mehrere Objekte verweist.

- Die Wurzel des Graphen und somit alle transitiv erreichbaren Objekte wurde als Wert aus dem Cache gelesen. Da auf Cache-Werte potentiell nebenläufig zugegriffen wird und sie eine konstante Identität (bzgl. der *Equals*-Methode) besitzen müssen, dürfen sie nicht verändert werden. Jede Veränderung an einem Objekt des Komponentengraphen erfordert daher das

Anlegen einer Kopie des ursprünglichen Objekts und Setzen der neuen Eigenschaftswerte. Da auch die Kante auf das neue Objekt, also die Eigenschaft eines anderen Objekts, zu ändern ist, bedeutet das Ersetzen eines Objekts gleichzeitig das Kopieren und Ersetzen aller Objekte auf dem Pfad zur Wurzel.

- Nicht alle Objekte eines Komponentengraphen müssen personalisiert werden. Zur Entwicklungszeit ist bereits bekannt, welche Objekte potentiell an den aktuellen Benutzer angepasst werden. Beispielsweise weiß ein Entwickler, ob eine Liste zur Laufzeit gefiltert werden muss.
- Von den Objekten, die für die Personalisierung in Frage kommen, werden im konkreten Fall nicht alle wirklich verändert. Dies ist zum Beispiel der Fall, wenn eine Liste im Allgemeinen zu filtern ist, der aktuelle Benutzer aber alle Einträge sehen darf.
- Um einen Knoten im Graphen zu personalisieren, reicht es aus, die ursprünglichen Informationen, die personalisierten Kindknoten sowie allgemeine Informationen über die Umgebung, den aktuellen Benutzer und die aktuelle Anfrage zu kennen.

Die Schnittstelle *IPersonalizable* (Abbildung 3.18) besitzt eine Methode *Personalize*. Diese Schnittstelle muss von allen Präsentationsobjekten im Komponentengraphen implementiert werden, die zu personalisieren sind. Eine Implementierung dieser Methode kann dasselbe Objekt (*this*) zurückgeben, falls keine Änderungen notwendig sind. Andernfalls muss sie das aktuelle Objekt mittels der *Clone*-Methode kopieren, die Eigenschaften des neuen Objekts ändern und es als Ergebnis zurückgeben. Damit alle zu personalisierenden Objekte gefunden, aber nicht mehr als nötig traversiert werden müssen, wird das Attribut *Personalized* definiert. Es ist nur für die Markierung von Eigenschaften gültig und wird für die Eigenschaften verwendet, deren Referenzen für die Suche nach *Personalizable*-Typen verfolgt werden sollen. Falls keine Eigenschaft eines Präsentationsobjekts mit *Personalized* ausgezeichnet ist, bricht der Personalisierungsalgorithmus an dieser Stelle ab und der Teilbaum wird nicht weiter durchsucht.

Der Personalisierungsalgorithmus durchläuft für einen Knoten drei Schritte. Er beginnt mit der Wurzel des Komponentengraphen.

1. Für einen Knoten wird zunächst festgestellt, ob sein Typ *ICollection* implementiert. Falls dies der Fall ist, wird jedes der enthaltenen Elemente durch den Algorithmus personalisiert und die Ergebnisse auf Identität mit den ursprünglichen Objekten getestet. Falls sich mindestens ein Element geändert hat, wird eine neue *collection* gleichen Typs erzeugt, die neuen Elemente hinzugefügt, und diese Kopie im nächsten Schritt weiterverarbeitet.
2. Für jede Eigenschaft, die mit *Personalized* markiert ist, wird nun wiederum der Algorithmus aufgerufen und die Ergebnisse auf Identität mit den ursprünglichen Werten getestet. Falls sich mindestens eines der referenzierten Objekte geändert hat, wird per *Clone* eine Kopie des aktuellen Knotens angelegt und die Werte der Eigenschaften auf die personalisierten Werte gesetzt. Der nächste Schritt wird dann mit dieser Kopie ausgeführt.
3. Falls der aktuelle Knoten *Personalizable* implementiert, wird die *Personalize*-Methode der Schnittstelle auf ihm aufgerufen (siehe oben). Das Ergebnis dieser Methode wird als Ergebnis der Personalisierung dieses Knotens zurückgegeben.

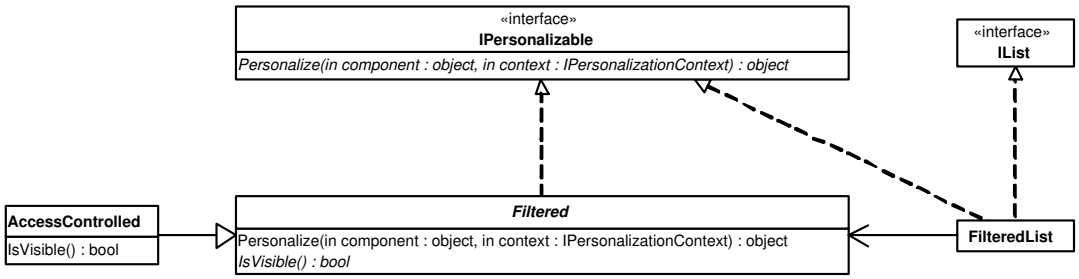

Abbildung 3.18: Schnittstellen und Klassen für die Filterung nach Leserechten

Für das Filtern von Listen und als Spezialfall das Filtern anhand von Leserechten gibt es drei weitere Klassen (siehe Abbildung 3.18). Die abstrakte Klasse *Filtered* implementiert *IPersonalizable* und hat eine abstrakte Methode *IsVisible*. *Filtered* erzeugt ein dynamisches Proxy-Objekt für ein zu filterendes Element. Ein dynamisches Proxy-Objekt ist ein Objekt, dass eine zur Laufzeit bestimmte Schnittstelle implementiert und alle Methodenaufrufe an einen Handler weiterleitet. Diese Klasse verwendet dynamische Proxy-Objekte, um das *Decorator*-Entwurfsmuster für beliebige zu dekorierende Typen zu implementieren. Die Implementierung von *Personalize* gibt *null* zurück, falls *IsVisible false* liefert. Andernfalls personalisiert sie das dekorierte Objekt und gibt das Ergebnis dieser Personalisierung zurück.

Filtered wird zusammen mit *FilteredList* verwendet. Diese Klasse implementiert die Schnittstellen *IList* und *IPersonalizable*. Die Implementierung von *Personalize* gibt eine neue Liste zurück mit allen Elementen der alten Liste, die ungleich *null* sind, oder sich selbst, falls dies für alle Elemente zutrifft. Da der Personalisierungsalgorithmus für *ICollections* zunächst die Elemente und dann die *collection* selbst verarbeitet, werden Objekte, die durch ein *Filtered* dekoriert wurden und ein einer *FilteredList* enthalten sind, bei der Personalisierung aus der Liste entfernt, falls *Filtered.IsVisible false* liefert.

Die Subklasse *AccessControlled* von *Filtered* kann nun als Dekorator für Präsentationsobjekte in einer *FilteredList* verwendet werden. Wenn ein Inhaltsobjekt des CMS über die Zugriffsschicht gelesen wird, fügt diese die ID des Inhaltsobjekts zu einem Cache-Attribut hinzu. Jedem Cache-Wert ist dieses Attribut zugeordnet, das die IDs aller während seiner Berechnung gelesenen CMS Ressourcen enthält. Die Methode *IsVisible* von *AccessControlled* ruft dieses Cache-Attribut für das dekorierte Objekt ab, und ruft mit den enthaltenen IDs die Methode *MayRead* des im *Component-Module* konfigurierten *AccessControl*-Objekts auf. Nur wenn der aktuelle Benutzer Leserechte auf allen verwendeten CMS Ressourcen hat, gibt *IsVisible true* zurück. Dementsprechend wird dieses Objekt andernfalls bei der Personalisierung aus der *FilteredList* entfernt.

3.4.7 Rendering-Implementierung für ASP.NET

Das vorgestellte Framework kollaboriert mit umgebungsspezifischen Ausgabetechnologien über die allgemeinen Rendering-Schnittstellen wie sie in 3.3.9 beschrieben sind. Für jede Ausgabetechnologie sind diese Schnittstellen neu zu implementieren oder entsprechende Subklassen von vorhandenen

Klassen zu entwickeln. Hier wird eine Implementierung für ASP.NET vorgestellt (siehe Abbildung 3.19).

In ASP.NET soll die Darstellung einer Komponente durch *web forms*, Benutzersteuerelemente oder Serversteuerelemente beschrieben werden. Die ersten beiden Formen verwenden eine deklarative Beschreibung des Ausgabeformats, ermöglichen das Einfügen dynamischer Elemente und werden zu *controls* kompiliert. Serversteuerelemente sind ebenfalls *controls*, jedoch ohne deklarative Beschreibung der Ausgabe. Abschnitt 3.2.6 enthält nähere Informationen zu ASP.NET.

Renderer für eine Komponente werden in diesem Modell als Klassen implementiert, die dem Baum von *controls* eines *web forms* ein Benutzersteuerelement oder ein Serversteuerelement hinzufügen. Das zugefügte *control* erhält eine Referenz auf die darzustellende Komponente und kann ihre Eigenschaften in geeigneter Form ausgeben sowie wiederum eine Darstellung referenzierter Komponenten in die Ausgabe einfügen.

Um an den entprechenden Stellen im Baum von Steuerelementen neue Elemente hinzuzufügen, verwaltet die Klasse *ControlRenderingContext* einen Stapel von *controls*. Das oberste Element auf dem Stapel stellt das *control* dar, dem gerade ein neues Element hinzugefügt wird. Die Klasse *ControlRenderingEngine* erzeugt einen solches Kontextobjekt in der Methode *CreateContext*. *ControlRenderingEngine* erbt von *ViewDispatchingEngineBase* und verwendet zum Finden eines geeigneten Renderers für eine Komponente einen *ControlViewDispatcher*. Dieser erbt von *ViewDispatcherBase* die Möglichkeit zur Registrierung von Renderer-Klassen zur Darstellung bestimmter Typen von Komponenten und die im Entwurfskapitel beschriebene Strategie der Renderer-Auswahl. Darüber hinaus sucht *ControlViewDispatcher* – falls kein Renderer für eine Komponente registriert ist – in einem Unterverzeichnis „Renderers/<*Namensraum des Komponententyps*>“ nach einem Benutzersteuerelement mit dem Namen „<*Komponententypname*>.ascx“. Dabei wird die gleiche Strategie für die Suche entlang der Typhierarchie verwendet wie sie für den *ViewDispatcherBase* beschrieben wurde. Falls eine entsprechende Datei existiert, erzeugt *ControlViewDispatcher* eine Instanz von *ControlRenderer* mit dem Pfad der Datei und gibt diese als Ergebnis zurück. *ControlRenderer* lädt das Benutzersteuerelement beim Aufruf von *Render* und fügt es unterhalb des *controls*, das oben auf dem Stapel des *ControlRenderingContext* liegt, in den Baum der Steuerelemente ein. Für das dynamische Laden wird die Methode *LoadControl* der Klasse *System.Web.UI.Page* verwendet. Um die Erstellung von Benutzersteuerelementen zum Rendern von Komponenten zu erleichtern, sucht sucht *ControlRenderer* nach dem Laden des Steuerelements nach einer öffentlichen Eigenschaft dieser Klasse mit dem Namen „self“ und setzt diese ggf. auf die darzustellende Komponente. Sie steht somit in einem Benutzersteuerelement ohne zusätzlichen Initialisierungsaufwand sofort zur Verfügung.

Um die Implementierung der Rendering-Schicht für ASP.NET abzuschließen sind zwei weitere Fälle zu betrachten: Das Laden des *web forms* für die Wurzel des Komponentengraphen und das Einbinden der Darstellung einer referenzierten Komponente aus einem Benutzersteuerelement.

Als Lösung für das erste Problem erstellt der *ControlViewDispatcher* für gefundene aspx-Dateien (also *web forms*) einen Renderer, der in seiner *Render*-Methode nur *RewritePath* auf dem aktuellen Anfrageobjekt aufruft. Der Grund dafür ist, dass der Renderer für die Wurzelkomponente von *ComponentModule* als letzter Schritt während der Behandlung des *BeginRequest*-Ereignisses

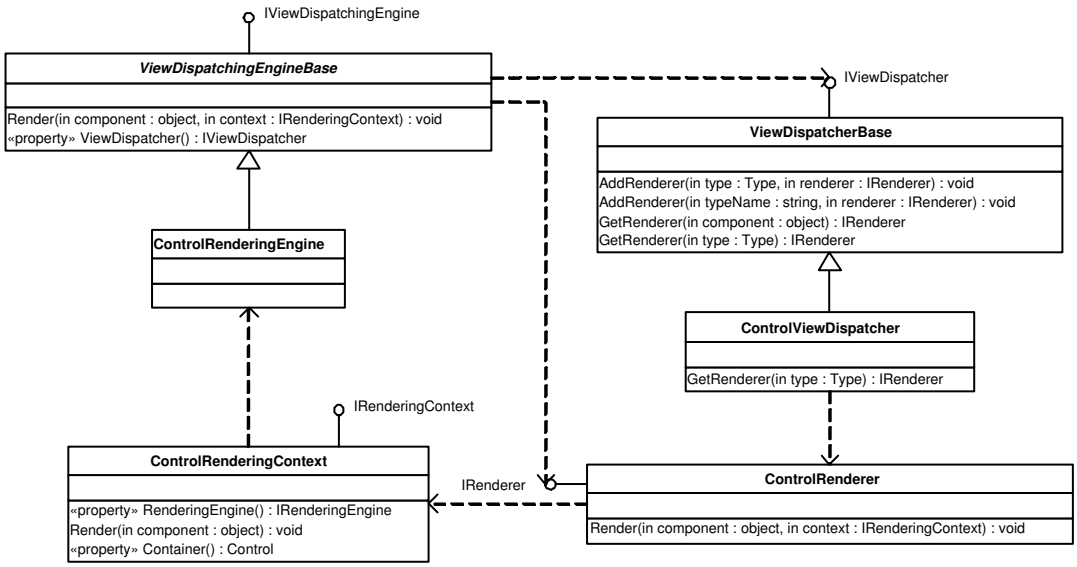

Abbildung 3.19: Rendering-Klassen für ASP.NET

ausgeführt wird. Durch das Umschreiben des Pfads der Anfrage auf den Pfad zu dem gefundenen *web form*, erzeugt ASP.NET im Laufe der Anfragebearbeitung einen *PageHandler* für dieses *web form*, so dass es als Einstieg für die Darstellung des Komponentengraphen verwendet wird. Das *web form* erhält eine Referenz auf die darzustellende Komponente aus dem aktuellen *ComponentRequestContext* (siehe 3.4.2).

Zum Einbinden der Darstellung für eine Komponente aus einem Benutzersteuerelement oder einem web form heraus gibt es ein Serversteuerlement *RenderComponent*, das wie alle anderen Steuerelemente in der deklarativen Ausgabebeschreibung von aspx- und ascx-Dateien verwendet werden kann. Es besitzt eine Eigenschaft *Component*, die auf die darzustellende Komponente verweist. Wenn es durch die ASP.NET-Umgebung geladen wird, fragt es den aktuellen *ControlRenderingContext* vom *ComponentRequestContext* ab, legt sich selbst als aktuelles Steuerelement auf den Stapel und ruft *Render* auf dem Kontext auf. Dieser Aufruf wird über die *ControlRenderingEngine* an den *ControlViewDispatcher* delegiert, so dass ein Renderer für die einzubindende Komponente gefunden werden kann, der wiederum ein neues Steuerelement unterhalb des *RenderComponent*-Elements in den Baum einfügt. Die Interaktion der verschiedenen Klassen ist in Abbildung 3.20 dargestellt.

Zu erwähnen sind an dieser Stelle noch drei weitere Hilfsklassen:

Form Dies ist eine Subklasse des sonst in *web forms* verwendeten *HtmlForm*. Das in ASP.NET übliche Verfahren der *PostBack*-Anfrage beim Betätigen von *controls* auf dem Klienten beruht auf der Annahme, dass das in jedem *web form* enthaltene Formular an dasselbe *web form* abgesendet wird. Die Klasse *HtmlForm* verwendet dazu den Pfad auf das aktuell ausgeführte *web form*. Dieser Pfad ist jedoch kein gültiger *Link*, der vom *ComponentModule* aufgelöst werden kann und führt somit zu einem Fehler. Die hier implementierte Subklasse

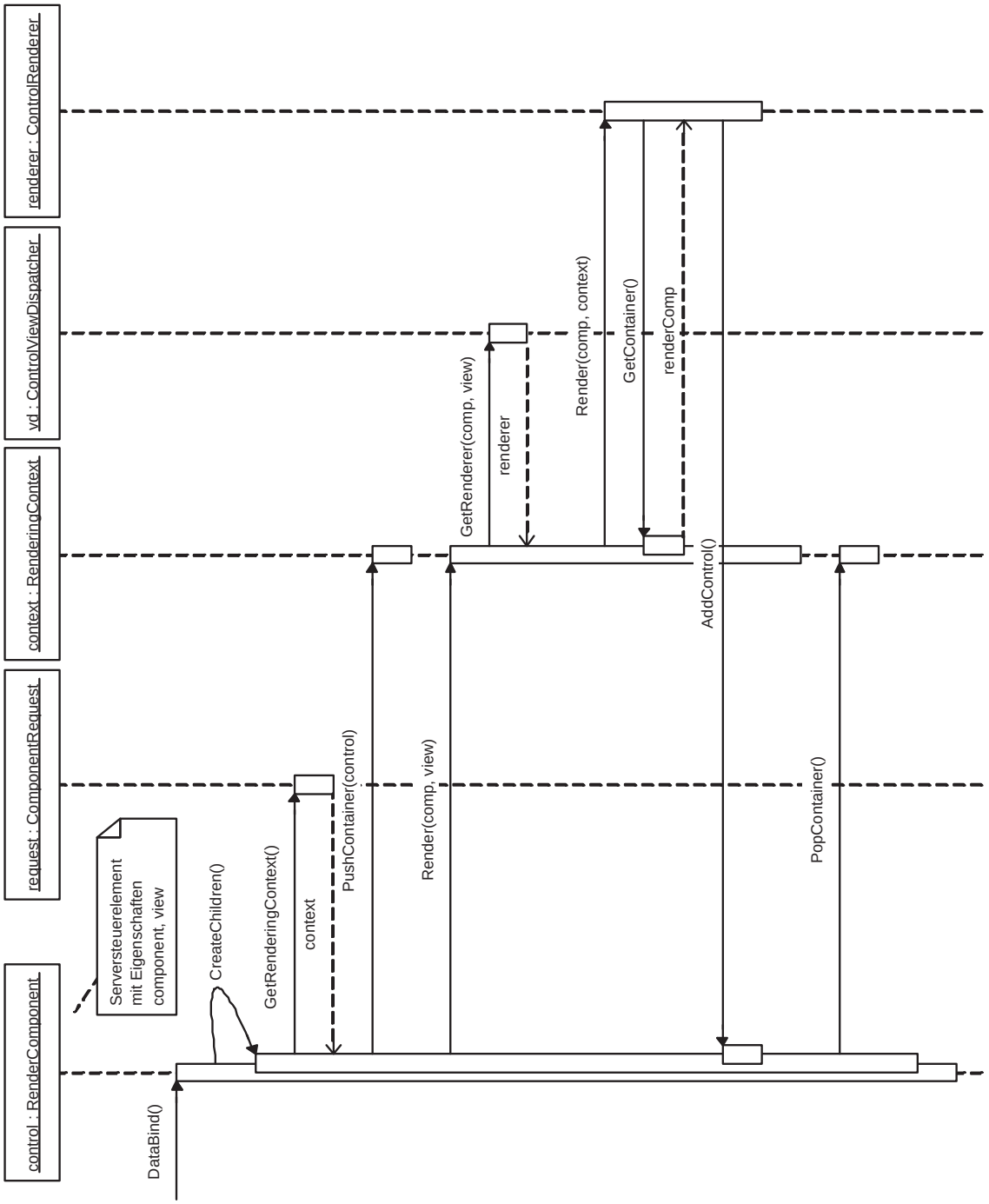

Abbildung 3.20: Einfügen eines Renderers für eine Komponente durch *RenderComponent*

überschreibt dieses Verhalten und verwendet statt dessen die ursprünglich angefragte URL, die es aus dem aktuellen *ComponentRequestContext* erhält. Diese Form-Klasse ist von allen *web forms* zu verwenden, die über den *ControlViewDispatcher* aufgerufen werden.

DefaultMarkupRenderer Ein Objekt dieser Klasse kann beim *ControlViewDispatcher* als Renderer für Objekte registriert werden, die ein XML-Dokument repräsentieren. Dieser Renderer fügt ein Serversteuerelement in den Elementbaum ein, das eine Reihe konfigurierbarer Filter auf dem XML-Dokument ausführt und das Ergebnis ausgibt.

DefaultBlobRenderer Dieser Renderer wird für Komponenten vom Typ *Blob* registriert. *Blobs* sind Typen der CMS Zugriffsschicht und kapseln den Inhalt einer Ressourceneigenschaft vom Typ „Blob“ mit binärem Inhalt, der unverarbeitet ausgegeben werden soll. Üblicherweise trifft dies auf Bilder aus dem CMS zu, deren Link zu einem Blob-Objekt aufgelöst wird, das somit die Wurzel und einzige Komponente im Komponentengraph ist. *DefaultBlobRenderer* schreibt in seiner *Render*-Methode den Pfad der aktuellen Anfrage auf einen wohldefinierten Pfad um, der in der Konfiguration der ASP.NET Webanwendung auf einen *BlobHandler* abgebildet ist. ASP.NET lädt daher keinen *PageHandler* für ein *web form*, sondern diesen *BlobHandler*, der den binären Inhalt der Blob-Komponente direkt auf den Ausgabestrom schreibt.

4 Zusammenfassung und Bewertung

Der erste Teil der Arbeit erläutert die Probleme, die zur Entwicklung von Content Management System geführt haben und beschreibt die Funktionen und Dienste, die diese Systeme bieten. Insbesondere der Betrieb dynamischer Webanwendungen mit vielen kontinuierlich aktualisierten Inhalten erfordert die Unterstützung durch entsprechende Werkzeuge. Für die Verwaltung der Inhalte bieten Content Management Systeme generische Dienste wie

- die Validierung nach festgelegten Regeln,
- Sicherstellung der Verweiskonsistenz,
- verteilte Bearbeitung von Inhalten durch mehrere Benutzer,
- Versionierung, Archivierung und
- eine Ablaufsteuerung für den Produktionsprozess.

Eine zentrale Funktion eines CMS besteht neben der Produktion und Verwaltung von Inhalten in deren Darstellung und Auslieferung an Klienten, üblicherweise in Form von Webseiten. Durch die Anwendung einmal definierter Regeln und Vorlagen erzeugt die Auslieferungskomponente eines CMS aus den verwalteten Inhalten verschiedene Ausgabeformate. Dabei kann sowohl ein Inhaltsobjekt in verschiedenen Kontexten und Formaten dargestellt, als auch unterschiedlichste Inhalte beliebig zu einer Ausgabe zusammengestellt werden. Der Vorteil der Verwendung von Vorlagen liegt in der getrennten Pflege von Inhalten und Darstellungsinformationen und dem damit deutlich reduzierten Aufwand gegenüber der direkten Bearbeitung der Seiten. Anhand zweier ausgewählter Produkte mit unterschiedlichen Modellierungsansätzen werden die beschriebenen Konzepte verdeutlicht.

Die Entwicklung einer Webanwendung auf Basis der Inhalte eines Content Management Systems ist im Allgemeinen eine komplexe Aufgabe, da oft verschiedene konkurrierende Anforderungen abgedeckt werden müssen:

- die flexible Zusammenstellung der Webseiten auf der Grundlage von CMS-Inhalten durch Vorlagen,
- eine effiziente Auslieferung der erstellten Seiten,
- die Personalisierung der Ausgabe und
- die sofortige Übernahme von Änderungen bei der Bearbeitung von Inhalten.

Diese Arbeit untersucht die Anforderungen an Webanwendungen und stellt ein Rahmenwerk vor, das die Entwicklung von CMS-basierten Webapplikationen unterstützt. Bestandteil der Arbeit ist die Implementierung dieses Rahmenwerks für die CoreMedia SCT in Verbindung mit Microsoft ASP.NET. Zunächst werden die Probleme dargestellt, die bei der Entwicklung einer Webanwendung typischerweise zu lösen sind. Ausgehend von einer allgemeinen Beschreibung einer Webanwendung als Transformation der Inhaltsobjekte des CMS in Webressourcen wird das Konzept des Komponentengraphen eingeführt, eines aus einer Menge von Inhaltsobjekten berechneten anwendungsspezifischen Objektgraphen für die Ausgabe einer einzelnen Webressource. Dazu wurde eine Komponente für den lesenden Zugriff auf Inhalte aus dem verwendeten CMS implementiert. Besondere Beachtung findet das Caching mit ereignisgesteuerter Invalidierung der erstellten Anwendungsobjekte. Anhand des Komponentengraphen für die Darstellung einer Webressource wird ein Verfahren zur Personalisierung der Ausgabe beschrieben. Weitere Bestandteile des implementierten Rahmenwerks sind die Konvertierung zwischen Anwendungsobjekten und ihrer Repräsentation als URL sowie die Verarbeitung von Verweisen in XML-Inhalten. Das Rahmenwerk definiert außerdem eine umgebungsunabhängige Rendering-Schicht für die Erstellung der Ausgabe der Anwendung. Diese wurde für ASP.NET als konkrete Umgebung implementiert.

Zur Bewertung des in dieser Arbeit implementierten Rahmenwerks wurde darauf aufbauend eine Beispielanwendung entwickelt. Das Rahmenwerk erwies sich dabei als eine gute Grundlage für die Erstellung von CMS-basierten Webanwendungen, die Inhalte effizient und personalisiert zusammenstellen und ausgeben. Die Verwendung der bereits vorhandenen CoreMedia Cache-Komponente mit Abhängigkeitsverwaltung ermöglicht ein für den Anwendungsentwickler transparentes Caching mit ereignisgesteuerter Invalidierung der Anwendungsobjekte.

Ebenso konnte festgestellt werden, dass mit Hilfe einer frei verfügbaren IIOP-Implementierung für .NET eine stabile – wenn auch nicht sonderlich performante – Kommunikation mit einem Java-CMS möglich ist und somit die angebotenen Dienste des CMS auch von .NET-basierten Klienten genutzt werden können. Die Anpassung der generischen Ausgabeschicht an ASP.NET bietet dem Entwickler eine intuitive Darstellung von Anwendungsobjekten als Benutzersteuerelemente, die hier als Vorlagen für die Ausgabe dienen. Insgesamt erwies sich das durch das Rahmenwerk unterstützte Programmiermodell als sehr angenehm und lässt dem Entwickler auch bei Nutzung der angebotenen Dienste viel Spielraum für die Implementierung der Anwendungslogik.

Als Weiterentwicklung des Rahmenwerks ist eine Überarbeitung des Personalisierungsverfahrens geplant. Das aktuelle Verfahren erfordert vom Anwender eine genaue Kenntnis des Komponentengraphen und das Einhalten eines relativ komplexen Protokolls. Außerdem soll untersucht werden, inwiefern die zugrundeliegende Cache-Komponente durch das Aufsetzen einer weiteren Abstraktionsschicht vor dem Benutzer verborgen werden kann, um so die Verwendung des Caches noch einfacher zu gestalten. Der Entwickler einer Anwendung sollte sich im Wesentlichen auf die Implementierung der Applikationslogik, also die Abbildung von CMS-Inhalten auf Anwendungsobjekte und deren Ausgabe, konzentrieren können.

Danksagung

Für die Unterstützung und Betreuung meiner Arbeit, inhaltliche Anregungen und interessante Diskussionen möchte ich Prof. Dr. Joachim W. Schmidt, Prof. Dr. Ralf Möller und meinen Kollegen der CoreMedia AG danken. Außerdem danke ich Prof. Dr. Florian Matthes und den Mitarbeitern des Arbeitsbereichs Softwaresysteme für die vielfältige Unterstützung während meines Studiums.

Referenzen

[1] K. Andrews, F. Kappe, and H. Maurer. Serving Information to the Web with Hyper-G, 1995.

[2] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. IETF Network Working Group, 1998. RFC 2396.

[3] Tim Berners-Lee, Robert Cailliau, Jean-Francois Groff, and Bernd Pollermann. World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 1(2):74–82, 1992.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium, Oktober 2000.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns*. Pattern-Oriented Software Architecture (1). Wiley, 1996.

[6] James Clark. *XSL Transformations (XSLT) Version 1.0*. World Wide Web Consortium, November 1999.

[7] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[8] Danny Coward. *Java Servlet Specification Version 2.3*. Sun Microsystems, Inc., 2001.

[9] Bundesverwaltungsamt Projektgruppe Dienstleistungsportal. Bund online 2005: http://www.bundonline2005.de.

[10] T. Dierks and C. Allen. *The TLS Protocol*. IETF Network Working Group, 1999. RFC 2246.

[11] ECMA International. *C# Language Specification*, 2nd edition, December 2002. Standard ECMA-334.

[12] ECMA International. *Common Language Infrastructure (CLI) Partitions I to V*, 2nd edition, December 2002. ECMA-335.

[13] David Booth et al. *Web Services Architecture*. World Wide Web Consortium, 2003. W3C Working Draft.

[14] Martin Gudgin et al. *SOAP Version 1.2 Part 1: Messaging Framework*. World Wide Web Consortium, 2003. W3C Recommendation.

[15] Tim Ewald and Keith Brown. HTTP Pipelines: Securely Implement Request Processing, Filtering, and Content Redirection with HTTP Pipelines in ASP.NET. *MSDN magazine*, September 2002.
http://msdn.microsoft.com/msdnmag/issues/02/09/HTTPPipelines/toc.asp.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, 1999. RFC 2616.

[17] Roy T. Fielding. Maintaining distributed hypertext infostructures: welcome to MOMspider's Web. *Computer Networks and ISDN Systems*, 27(2):193–204, 1994.

[18] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison Wesley, 2nd edition, 1999.

[19] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. *HTTP Authentication: Basic and Digest Access Authentication*. IETF Network Working Group, 1999. RFC 2617.

[20] N. Fried and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*. IETF Network Working Group, 1996. RFC 2046.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[22] Maria Ginsburg. Comparison of Web Visualization Frameworks for eBusiness Applications Using the Example of an Online Shop. Diplomarbeit, Technische Universität Hamburg-Harburg, Arbeitsbereich Softwaresysteme, 2004.

[23] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun Microsystems, Inc., 2000.

[24] .NET Languages Developers Group. About languages.
http://www.gotdotnet.com/team/lang.

[25] John L. Hennessy and David A. Patterson. *Computer architecture: A quantitative approach*, chapter 5. Morgan Kaufmann Publishers, 3rd edition, 2003.

[26] D. B. Ingham, M. C. Little, S. J. Caughey, and S. K. Shrivastava. W3Objects: Bringing Object-Oriented Technology to the Web. *World-Wide Web Journal*, 1, 1995.

[27] J. Kohl and C. Neuman. *The Kerberos Network Authentication Service (V5)*. IETF Network Working Group, 1993. RFC 1510.

[28] David M. Kristol and Lou Montulli. *HTTP State Management Mechanism*. IETF Network Working Group, 1997. RFC 2109.

[29] Michael Mattson. Object-Oriented Frameworks: A Survey of Methodological Issues. In *Proceedings of the 1996 Triennal IFAC World Congress, IFAC'96*, 1996.

[30] Craig McClanaham and Ed Burns. *JavaServer Faces Specification Version 1.0*. Sun Microsystems, Inc., 2004.

[31] Microsoft Corporation. *Design Guidelines for Class Library Developers*, 2004. http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp.

[32] Microsoft Corporation. *Designing Distributed Applications with Visual Studio .NET: IIS Authentication*, 2004. http://msdn.microsoft.com/library/en-us/vsent7/html/vxconIISAuthentication.asp.

[33] Object Management Group. *Common Object Request Broker Architecture: Core Specification*, 2002. Version 3.0.2.

[34] Object Management Group. *Unified Modeling Language Specification*, 2003. Version 1.5.

[35] Morag Ottens. Internet Usage by Individuals and Enterprises. Statistics in Focus - Theme 4. Eurostat, April 2004. KS-NP-04-016-EN-N, 15.04.2004.

[36] Eduardo Pelegrí-Llopart. *JavaServer Pages Specification Version 1.2*. Sun Microsystems, Inc., 2001. FCS.

[37] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*. World Wide Web Consortium, 1999.

[38] Ravi S. Sandhu and Pierrangela Samarati. Access Control: Principles and Practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[39] Abraham Silberschatz and Peter Galvin. *Operating System Concepts*, chapter 9. John Wiley, 5th edition, 1997.

[40] Borland Staff. Borland Janeva Overview. Technical report, Borland Software Corporation, 2003.

[41] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. *XML Schema Part 1: Structures*. World Wide Web Consortium, 2001. W3C Recommendation.

[42] J. van Gurp and J. Bosch. Design, Implementation and Evolution of Object Oriented Frameworks: Concepts and Guidelines. *Software - Practice and Experience*, 31(3):277–300, 2001.

[43] M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (v3)*. IETF Network Working Group, 1997. RFC 2251.