Generierung plattformunabhängiger Inhaltsverwaltungssysteme am Beispiel von Zope3

Henner Carl

13.4.2006

Gutachter: Prof. Dr. Joachim W. Schmidt

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Motivation und Einleitung

Durch die Nachfrage nach immer leistungsfähigeren und benutzerfreundlicheren computergestützten Informationssystemen wurde die Arbeit der Softwareentwicklung zunehmend anspruchsvoller. Die Ursache dafür liegt nicht nur in den immer komplexeren fachlichen Anforderungen aus der Anwendungsdomäne begründet, sondern auch in der Vielzahl miteinander konkurrierender und kooperierender Technologien.

Gerade im Anwendungsfeld der Inhaltsverwaltung ist man auf viele verschiedene Technologien angewiesen. Es werden Datenbanken für die persistente Speicherung der Inhalte und Frameworks für die Erstellung von Benutzeroberflächen benötigt. Diese Basisfunktionen werden häufig zu einem Framework für Inhaltsverwaltungssysteme kombiniert. Als Folge sind die entstehenden Anwendungen eng mit dem verwendeten Framework verknüpft und können nur schwer auf andere Plattformen portiert werden. Dadurch besteht der Bedarf, Software unabhängig von bestimmten Technologien, plattformunabhängig zu entwickeln.

Eine Lösung dieses Problems bieten Ansätze der generativen Programmierung, durch die konkrete Anwendungen aus einer abstrakten Spezifikation erzeugt werden. Dadurch ist es möglich, aus einer Spezifikation Implementierungen für unterschiedliche Plattformen zu generieren. Dieses Vorgehen nutzt der CCM (*concept oriented content management*) Modellkompiler zur Erzeugung von Inhaltsverwaltungssystemen, die auf einem konzeptionellen Modell der Anwendungsdomäne basieren.

Wesentlicher Grund für den Einsatz der generativen Programmierung in den CCM Systemen ist nicht die Plattformunabhängigkeit. Nur indem der Anwender selbst seine Modelle spezifizieren kann, lassen sich offene und dynamische Erkenntnissprozesse optimal unterstützen. (siehe Abschnitt 2.2.1) In dieser Arbeit soll untersucht werden, ob sich die theoretische Plattformunabhängigkeit der Systemgenerierung mit dem Modellkompiler auch über Programmiersprachengrenzen hinweg nutzen lässt.

Ausgangspunkt

Grundlage dieser Arbeit sind die CCM Systeme, eine Familie von generierten Inhaltsverwaltungssystemen. Diese Systeme werden in einem modellgetriebenen (*model driven*) Entwicklungsansatz aus einem konzeptionellen Modell der Anwendungsdomäne von einem Modellkompiler erzeugt.

Neben den konzeptionellen Vorteilen der Systemgenerierung - der Anwender selbst kann das Domänenmodell anpassen - werden mit ihr technologische Vorteile verbunden. Einer der wichtigsten ist die Plattformunabhängigkeit, sie ist neben Produktivitätsüberlegungen der Grund für das zur Zeit breite Interesse an der modellgetriebenen Architektur (MDA, *model driven architecture*).

Der allgemeine Begriff der Plattform muss im Zusammenhang differenziert betrachtet werden. Im Fall der CCM Systeme sind vor allem zwei Ebenen des Plattformbegriffs von Interesse: Zum einen ist dies die grundlegende Plattform für ein generiertes System, die verwendete Programmiersprache mit ihrer Laufzeitumgebung. Darüber hinaus stellen auch Basiskomponenten wie Frameworks und Datenbanken, die von den erzeugten Systemen genutzt werden, eine Plattform dar.

Auch die Plattformunabhängigkeit kann durch diese Unterscheidung differenziert werden. In der zur Zeit verfügbaren Implementierung des CCM Modellkompilers wird eine Plattformunabhängigkeit von Basiskomponenten bereits unterstützt. So ist es zur Zeit möglich, die Persistenz der Inhalte durch eine relationale oder XML Datenbank zu realisieren. Prinzipiell kann die Persistenz auf beliebige Weise verwirklicht werden, in der aktuellen Implementierung des Kompilers sind die genannten Basiskomponenten die wichtigsten.

Eine Plattformunabhängigkeit in Bezug auf die Programmiersprache der Zielsysteme wird bislang noch nicht unterstützt. Alle bislang verwendeten CCM Systeme basieren ausschließlich auf der Programmiersprache Java, in der auch der Modellkompiler selbst implementiert ist.

Aufgabe

In dieser Arbeit soll die Plattformunabhängigkeit generierter Systeme ausgenutzt werden, indem erstmalig CCM Systeme für eine andere Programmiersprache als Java erzeugt werden. Ziel der praktischen Arbeit ist es, konzeptorientierte Inhaltsverwaltungssysteme für Zope zu erzeugen, einem Server für webbasierte Applikationen, der in der Sprache Python programmiert wird. Dazu wird der Modellkompiler um sogenannte *Generatoren* für Zope basierte Systeme erweitert.

Um die Aufgabe zu bearbeiten, werden zunächst Zope und die CCM Systeme analysiert und betrachtet, wie man sie miteinander kombinieren kann. Es folgt eine Untersuchung darüber, wie es möglich ist, die Domänenmodelle

der CCM Systeme auf Zope abzubilden. Anschließend wird die Systemgenerierung mit Hilfe des Modellkompilers entworfen und realisiert. Da es für Java noch kein geeignetes Werkzeug für die Erzeugung von Python Kode gibt, muss ein solches entworfen und implementiert werden.

Die Wahl der Zielplattform fiel auf Zope, da es eine frei verfügbare und verbreitete Plattform für Inhaltsverwaltungssysteme ist und den Ruf genießt, sowohl performant als auch flexibel zu sein. Alternative Zielsysteme für diese Arbeit sind Plone und Silva, die beide auf einem älteren Zope basieren. Zope ist in den 2.X Versionen allerdings eine schwer zu handhabende Plattform, ein wichtiger Grund für die vollständige Neuentwicklung von Zope3. ([KW05])

Ziel

Durch die Erfahrungen im praktischen Teil der Arbeit sollen Erkenntnisse darüber gewonnen werden, inwieweit sich die Plattformunabhängigkeit für CCM Systeme in verschiedenen Programmiersprachen ausnutzen lässt. Darüber hinaus soll beurteilt werden, ob Zope eine geeignete Zielplattform für die konzeptorientierte Inhaltsverwaltung ist und inwieweit eine Integration von nicht Java basierten Modulen in CCM Systeme sinnvoll ist.

Ein Nebenprodukt der praktischen Arbeit ist ein Werkzeug zur Erzeugung von Python Kode. Daher sollen die Erfahrungen, die mit dem Werkzeug und der Kodeerzeugung für die schwach typisierte Programmiersprache gemacht wurden, bewertet werden. Dabei wird auch der Frage nachgegangen, ob ein kombiniertes Kodegenerierungswerkzeug für mehrere Sprachen, in diesem Fall Python und Java, sinnvoll erscheint.

Kapitel 2

Vorstellung der Systeme

In diesem Kapitel werden die Systeme vorgestellt, welche die Grundlage dieser Arbeit bilden. Zunächst wird das generische System Zope und anschließend die Klasse der generierten CCM Systeme beschrieben. Es folgt eine Diskussion, welche Möglichkeiten es gibt, Zope-basierte CCM Systeme zu erzeugen, und die Entscheidung, welcher Ansatz in dieser Arbeit verfolgt wird.

2.1 Zope

2.1.1 Einführung

Zope ist ein Server für webbasierte Anwendugen (*web application server*) der 1997 von der Firma Zope Cooperation (früher Digital Creations) entwickelt wurde. Seit der Offenlegung des Quellkodes (*open source*) beteiligen sich Firmen und individuelle Entwickler an der Weiterentwicklung des Systems. Zope ist kein unmittelbar einsetzbares Produkt, sondern ein generisches System, das als Framework eine Basis für eigene Anwendungen darstellt.

Zope wird als **Anwendungsserver** bezeichnet und als Alternative zu J2EE Servern wie JBoss [JBoss] vermarktet. Es gilt unter Zope-Entwicklern als Vorteil, dass die Zope basierte Anwendungsentwicklung in der Programmiersprache Python erfolgt, in der auch das System selbst implementiert ist. Der Einsatz dieser interpretierten, schwach typisierten Sprache vereinfache sich im Vergleich zu Java die Implementierung von Anwendungen [Ric05].

Agile Methoden sind bei der Entwicklung von Zope Anwendungen allgegenwärtig. Das System stellt eine webbasierte Administrationsoberfläche bereit, durch die eigene Anwendungen interaktiv und inkrementell entwickelt werden können. Die Möglichkeiten dieser TTW Entwicklung (*through the web development*) sind mit denen von PHP [KHM05] vergleichbar. Da diese Art der Entwicklung für diese Arbeit keine Bedeutung hat, wird nicht näher auf sie eingegangen. Auch bei der Entwicklung von Zope3 kamen mehrere

Praktiken agiler Methoden zum Einsatz. Das System wurde auf Entwicklertreffen (sogenannten *Sprints*) mit Methoden des *extreme programming* wie Programmierung in Paaren (*pair programming*) und Testgesteuerte Programmierung (*test first programming*) entwickelt. Diese Methoden werden auch für die Entwicklung eigener Anwendungen empfohlen und teilweise durch das System unterstützt, z.B. durch ein Framework zum Testen eigener Komponenten.

Der Anwendungsserver Zope stellt eine geeignete Plattform für **Inhaltsverwaltungssysteme** (*content management systems, CMS*) dar. Zu diesem Zweck wurde eine Erweiterung für Zope2 entwickelt: das *Content Management Framework, (CMF)*. Aufbauend auf Zope und dem CMF wurden mehrere freieverfügbare Inhaltsverwaltungssysteme, z.B. Plone und Silva, realisiert. (siehe [Mck04, Silva])

In dieser Arbeit kommt das aktuelle, noch wenig verbreitete Zope3 zum Einsatz, das von Grund auf neu entwickelt wurde und inkompatibel zu seinen Vorgängern ist. Weil Zope in der Vergangenheit am häufigsten für Inhaltsverwaltungssysteme eingesetzt wurde, sind die durch das CMF bereitgestellten Funktionen in die Basis der neuen Version integriert worden. Dokumentation zu den System findet sich in [Ric05, Wei05, FulPZ, Zope3].

2.1.2 Systemaufbau und Struktur

Obwohl die Dokumentation für Zope3 sehr umfangreich ist, bleibt sie dem Leser eine Übersicht über die Architektur des Systems schuldig. Dieser Abschnitt soll ein Versuch sein, die Architektur und Komponenten des Systems zu beschreiben. Er sollte unter dem Vorbehalt gelesen werden, dass er nicht von den Autoren von Zope3 stammt. Die Korrektheit jedes Details kann deshalb nicht garantiert werden.

Das Framework Zope3 ist in einer **Komponentenarchitektur** realisiert: Anwendungen werden implementiert, indem man das System durch eigene Komponenten erweitert, die auf die Basisfunktionen des Systems zurückgreifen können. Auch die meisten Basisfunktionen von Zope sind als Komponenten realisiert. Eine Übersicht über die Architektur findet sich in Abbildung 2.1: Neben der Komponentenarchitektur - auf die später eingegangen wird - sind in dem System Server für die Protokolle *http* und *ftp* vorhanden. Die Server erhalten über den Kern der Komponentenarchitektur Zugriff auf die Komponenten, welche zur Bearbeitung einer Anfrage benötigt werden. Für die Datenhaltung ist eine objektorientierte Datenbank integriert, die ZODB (*Zope Object Database*). Der Vorteil dieser Datenbank ist, dass sie gut in die Programmiersprache Python integriert ist und beinahe transparent ACID Transaktionen ermöglicht. (*Atomicity Consistency Isolation Durability*, siehe [BN96]) Persistente Komponenten greifen direkt auf die ZODB zu.

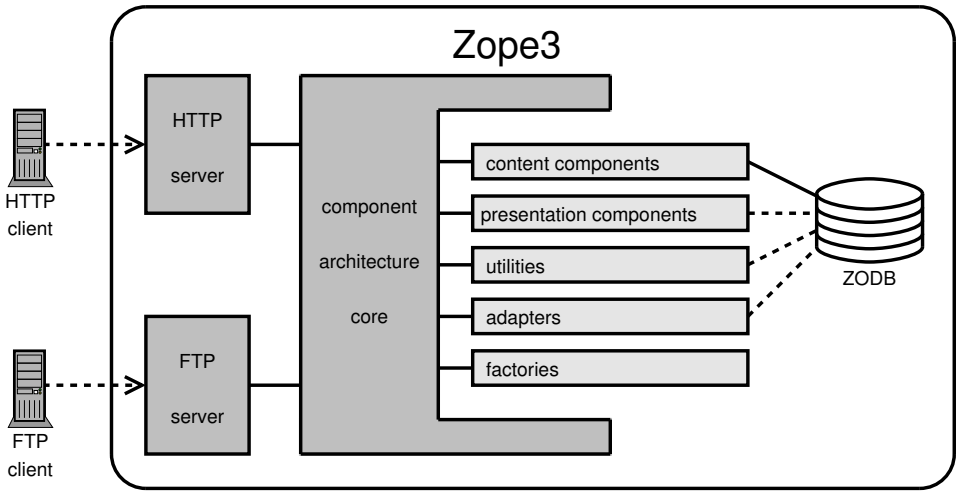

Abbildung 2.1: Zope3 Architekturdiagramm

Obwohl Zope3 als Komponentenarchitektur bezeichnet wird, bleibt mangels einer Definition in der Dokumentation unklar, was genau eine Zope3 Komponente ist. Der Begriff wird hier in folgendem Sinne verwendet (angelehnt an [FulPZ]): „Eine Komponente (*component*) ist ein Objekt, das eine Schnittstelle (*interface*) implementiert und mit anderen Objekten über Schnittstellen kommuniziert.“ Eine Schnittstelle bezeichnet in diesem Zusammenhang das *interface*-Konstrukt von Zope und nicht das allgemeine Konzept einer Schnittstelle. Schnittstellen sind in der schwach typisierten Sprache Python nicht vorhanden, sie sind mit den *interfaces* von Java vergleichbar. Folgende Arten von Komponenten lassen sich in Zope3 unterscheiden:

Inhaltskomponenten (*content components*) sind meist persistente Objekte, die anwendungsrelevante Information speichern. Sie erfüllen eine ähnliche Aufgabe wie die *entity beans* der J2EE Architektur. Inhaltskomponenten sind in einer Ordnerstruktur organisiert, so dass sie sich über individuelle URLs (*uniform resource locators*) ansprechen lassen.

Präsentationskomponenten (*presentation components*) sind zuständig für die Darstellung (z.B. als HTML) von Inhaltskomponenten.

Hilfsdienste (*utilities*) stellen grundlegende Systemdienste zur Verfügung, wie z.B. Fehlerbehandlung oder Verbindungen zu SQL Datenbanken.

Fabriken (*factories*) sind Komponenten die bei einem Aufruf andere Komponenten erzeugen, ähnlich dem in [GHJV94] beschriebenen Erbauer (*builder*).

Adapter (*adapters*) entsprechen in ihrer Struktur dem Entwurfsmuster Adapter aus [GHJV94]. Sie gehen aber über das bekannte Entwurfsmuster hinaus, weil sie nicht nur der Anpassung inkompatibler Schnittstellen dienen, sondern auch dazu verwendet werden, Komponenten um zusätzliche Funktionen zu erweitern. Durch die Adapterkomponenten soll sogar aspektorientiertes Programmieren ermöglicht werden.

Das wesentliche Anliegen einer Komponentenarchitektur, die Wiederverwendung von Kode, wird durch Ausnutzung der schwachen Typisierung von Python auf eine besondere Weise erfüllt. Da Komponenten keine einheitliche, sondern lediglich irgendeine Schnittstelle implementieren müssen, können beliebige Python Klassen als Implementierung einer Komponente verwendet werden. Dies gilt für Inhaltskomponenten und Hilfsdienste, bei anderen Komponentenarten gibt es geringfügige Einschränkungen.

Der **Kern der Komponentenarchitektur** (ansprechbar über die *ZAPI*) dient als ein Verzeichnis für Komponenten, siehe *registry pattern* in [FRF03]. Nach registrierten Komponenten kann unter Angabe von Name und Schnittstelle gesucht werden. Je nach Art der Komponente können weitere Parameter erforderlich sein, z. B. kann nach einem Adapter gesucht werden, der ein gegebenes Objekt an eine gewünschte Schnittstelle anpasst.

2.1.3 Entwicklung von Inhaltskomponenten

Wie bereits erwähnt wurde entsprechen Inhaltskomponenten den *entity beans* der J2EE Architektur. Da die Entwicklung dieser Komponenten bei der Programmierung von Zope3 im Vordergrund steht, soll anhand eines einfachen Beispiels gezeigt werden, wie eine Klasse von Inhaltskomponenten implementiert wird. In dem Beispiel sollen Informationen zu Personen verwaltet werden, die sich über eine webbasierte Benutzeroberfläche darstellen und bearbeiten lassen. Abbildung 2.2 zeigt die wichtigsten Bestandteile der Inhaltskomponentenklasse mit ihrem Quellkode. Der gezeigte Kode ist vollständig und funktionsfähig, nur das Einbinden der Bibliotheken wurde der Übersichtlichkeit halber ausgelassen.

Als erstes muss die **Schnittstelle** der Komponenten festgelegt werden. Da das Konzept von Schnittstellen in der Programmiersprache unbekannt ist, werden sie in Zope durch Python Klassen spezifiziert, die von der Klasse `Interface` erben. Neben den einfachen Schnittstellen, die Methodensignaturen festlegen, besitzt Zope eine erweiterte Form, durch die auch Objekteigenschaften (*properties*) und Instanzvariablen spezifiziert werden. Die Schnittstelle `IPerson` aus dem Beispiel ist ein solches Schema (*schema*). Objekteigenschaften sind ein Konzept in Python, das auch in C# bekannt ist: Akzessoren- und Modifikatorenaufrufe („getter“ und „setter“) werden mit der gleichen Syntax wie ein Attributzugriff durchgeführt (*dot operator*). Die Objekteigenschaften werden in den Schemata durch Felder (*fields*)

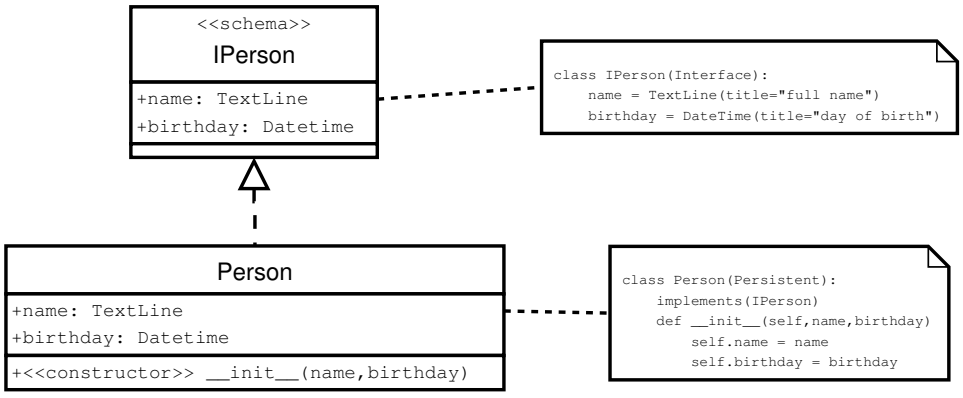

Abbildung 2.2: Realisierung einer Inhaltskomponente

spezifiziert, für die Feldtypen (ebenfalls *fields* genannt) für alle Python Standarddatentypen zur Verfügung stehen. Sie werden in Abschnitt 3.2.2 genauer behandelt.

Nachdem die Schnittstelle der Komponenten festgelegt wurde, muss diese durch eine **Klasse** implementiert werden. Dies geschieht durch Aufruf der Zope Funktion `implements` in der Definition der implementierenden Klasse. Da in Python Instanzvariablen nicht deklariert werden, reicht eine leere Klasse als Implementierung der Schnittstelle aus. Auf diese Besonderheit der Sprache Python wird in Kapitel 4.2.3 eingegangen. Um die Komponenten persistent zu halten, erbt die implementierende Klasse von der Basisklasse `Persistent`. Ohne zusätzlichen Aufwand werden ihre Instanzen in der Datenbank gespeichert und können an Transaktionen teilnehmen.

Abschließend muss die Inhaltskomponente durch Konfigurationskode in ZCML (*Zope Configuration Markup Language*, einer XML basierten Sprache) bei der Komponentenarchitektur registriert werden. Die **Konfiguration** wird hier ausgelassen, da sie nicht unmittelbar verständlich ist. Damit die Inhaltskomponenten dargestellt und verändert werden können, müssen ihnen Präsentationskomponenten zugeordnet werden: Wenn ein Schema die Komponentenschnittstelle spezifiziert, können entsprechende Präsentationskomponenten automatisch aufgrund der Felder des Schemas erzeugt werden. Es ist auch möglich HTML Vorlagen (*templates*) zu verwenden, wenn eine genauere Kontrolle über die Darstellung gewünscht wird.

2.1.4 Stärken & Schwächen

In dieser Arbeit wird Zope3 als Basis für die Inhaltsverwaltung eingesetzt, deshalb sind Systeme wie CoreMedia und der InfoAsset Broker als Maßstab für die Beurteilung heranzuziehen.

Die **Stärken** von Zope liegen zum einen in der *einfachen Programmierung* des Systems. Eine Klasse von Inhaltskomponenten konnte im vorigen Abschnitt durch wenige Zeilen Programmkode realisiert werden. Durch die einfache Programmierung des Systems und das integrierte Framework für Komponententests eignet sich das System gut für den Einsatz agiler Methoden. Auch Zopes Komponenten zur *Präsentation* von Inhalten sind hervorzuheben, da sich HTML Dokumente direkt aus den Inhaltkomponenten mithilfe der Schemata erzeugen lassen. Wird dies nicht gewünscht, kann zwischen zwei Sprachen zur Formulierung von Präsentationsvorlagen (*templates*) gewählt werden. In diesem Zusammenhang ist auch die Unterstützung zur Internationalisierung und Lokalisierung zu erwähnen. Neben den Schnittstellen zu Standardwerkzeugen (z.B. KBabel [VK03]) ist sogar eine Anwendung für die Lokalisierung in die Administrationsoberfläche integriert. Auch die *integrierte objektorientierte Datenbank* erwähnenswert. Sie ist ein Grund für das gute Lastverhalten von Zope, da sie auf einem Transaktionsmodell ohne Sperren (*locks*) basiert. Sie verwendet ein optimistisches, auf Zeitstempeln basiertes Verfahren, um nebenläufige Transaktionen zu synchronisieren.

Herrausragende **Schwäche** des Systems sind die beschränkten Möglichkeiten, mit Java oder .NET Systemen zu kommunizieren. Eine direkte Kommunikation auf Ebene der Programmiersprachen ist nicht möglich. Die Kommunikation auf Basis von Webservices funktioniert nur in einer Richtung: Es ist möglich, von Zope aus auf SOAP Dienste zuzugreifen. Sollen allerdings Zope Anwendungen als Webservice zur Verfügung gestellt werden, so ergeben sich Probleme, da hierfür nur das veraltete XML-RPC Protokoll unterstützt wird.

2.2 Konzeptorientierte Inhaltsverwaltung (CCM)

2.2.1 Einführung

Bei den CCM Systemen (*concept oriented content mangagement*) handelt es sich nicht wie bei Zope um ein generisches System, sondern um eine Klasse generierter Inhaltsverwaltungssysteme (*content management systems*). Die Systeme werden aus einer Spezifikation, einem konzeptionellen Modell, von dem CCM Kompiler erzeugt, dieses Szenario veranschaulicht Abbildung 2.3. Die Idee, Inhaltsverwaltungssysteme aus einem Modell zu generieren, stammt aus interdisziplinären Projekten wie der WEL (siehe [Seh04, SSW02]). Die monotonen und sich wiederholenden Vorgänge bei der Programmierung gängiger Inhaltsverwaltungssysteme (im Fall der WEL CoreMedia) legten nahe, dass sich diese Programme gut für eine automatische Generierung eignen.

CCM ist damit den Methoden zur **modellgetriebenen Softwareentwicklung** zuzuordnen, wie auch die zur Zeit vieldiskutierten MDA Verfah-

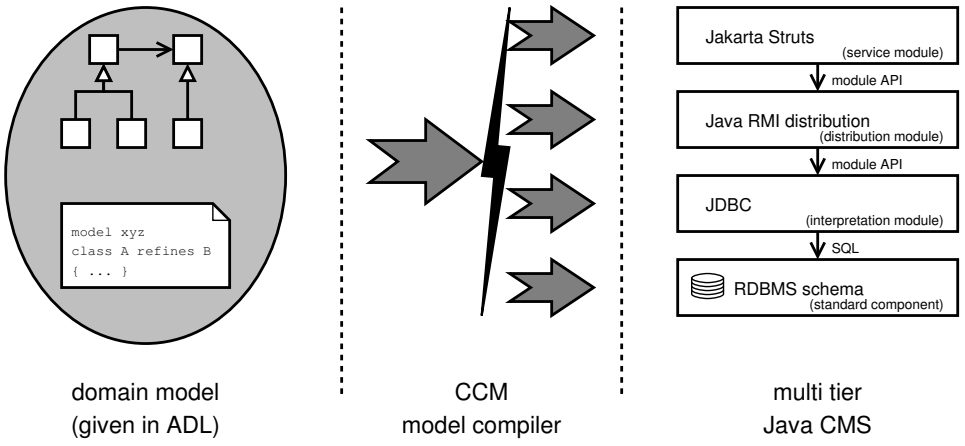

Abbildung 2.3: Szenario: Generierung von CCM Systemen

ren (*model driven architecture*). Der wesentliche Unterschied zur MDA ist, dass diese von einem Softwaremodell ausgehen, die CCM Systeme jedoch aus einem Modell der Anwendungsdomäne (*domain model*) heraus erzeugt werden.

Eine weitere Erfahrung aus der WEL ist die Erkenntnis, dass sich die Sicht der Anwender auf ihre Anwendungsdomäne mit zunehmenden Erkenntnisstand ändert, wodurch es notwendig wird, die verwendeten Inhaltsverwaltungssysteme laufend anzupassen. Die wichtigsten Forderungen an die CCM Systeme sind deshalb **Offenheit und Dynamik** : Offenheit gegenüber Veränderungen an den ihnen zugrundeliegenden Konzepten (Modelle) und die Dynamik, diese Konzepte jederzeit einzusehen und zu verändern. Eine genauere Beschreibung der offenen und dynamischen Erkenntnisprozesse findet sich in [Seh04]. Die Vision sind Inhaltsverwaltungssysteme, die von ihren Anwendern jederzeit um neue Konzepte erweitert und personalisiert werden können.

2.2.2 Aufbau eines generierten Systems

Bevor die Sprache zur Definition der Domänenmodelle erläutert wird, soll kurz ein Ausblick auf die Struktur der generierten Systeme gegeben werden. Ein CCM System setzt sich aus Modulen zusammen, wobei durch jedes Modul nur ein kleiner Teil der Gesamtfunktionen des Systems realisiert wird. Als Beispiel sei das generierte System aus Abbildung 2.3 gegeben, das aus folgenden Modulen besteht:

- Das Dienstschnittstellenmodul (*server module*) ist für die Präsentation der Daten in einer webbasierten Oberfläche zuständig. In diesem Fall wird das Java basierte Struts Framework [Hus02] verwendet.

- Das Distributionsmodul (*distribution module*) ist für die räumliche Verteilung der Anwendung zuständig. Es verwendet RMI (*Java remote method invocation*), um die Datenhaltung und Präsentation auf verschiedene Rechner zu verteilen.

- Das Interpretationsmodul (*client module*) dient dem Zugriff auf eine Standardkomponente, hier eine relationale Datenbank. Seine Aufgabe ist es, das Datenmodell der Standardkomponente zu kapseln.

- Die relationale Datenbank ist eine Standardkomponente und kein CCM Modul. Sie ist trotzdem dargestellt, da sie von dem Kompiler durch ein modellspezifisches Datenbankschema konfiguriert wird.

Die Module kommunizieren über die **einheitliche Modulschnittstelle** (*module API*) miteinander, dabei handelt es sich um eine Repräsentation des Domänenmodells in Form von Java Schnittstellen. Über die einheitliche Modulschnittstelle ist es möglich, auf die gespeicherten Inhalte zuzugreifen und diese transaktional zu verändern.

Es gibt noch weitere Modularten, die für die Personalisierung oder die Koordinierung mehrerer Interpretationsmodule eingesetzt werden, sie werden in [Seh04] eingehend beschrieben.

2.2.3 Assetdefinitionssprache

Die Domänenmodelle für die Systemgenerierung beschreibt man in einer dem Anwender angepassten Sprache, der Assetdefinitionssprache (ADL, *asset definition language*). Assets stellen die zentrale Informationsabstraktion von CCM dar: Sie verweisen auf Entitäten der Anwendungsdomäne durch einen Inhalt (z.B. eine mediale Repräsentation) und eine konzeptionelle Beschreibung der Entität.

Die Struktur von Assets wird durch die Deklaration von **Assetklassen** festgelegt, vergleichbar mit Objekten und Klassen in der objektorientierten Modellierung. Als Beispiel soll das Modell, dass in dem Klassendiagramm aus Abbildung 2.4 gezeigt wird, in der ADL formuliert werden. Dabei ist zu beachten, dass in der UML im Gegensatz zur ADL keine Unterscheidung zwischen Inhalt und Konzept vorgenommen wird.

```
class Print {
    content preview : Image
    concept characteristic title : String
            relationship artist : Artist
}
```

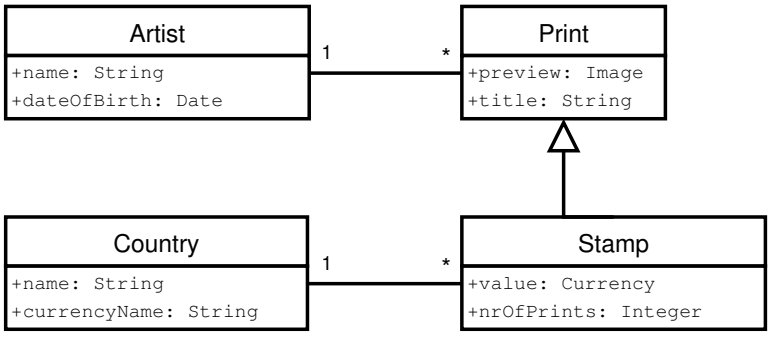

Abbildung 2.4: Beispiel: Klassendiagramm einer Anwendungsdomäne

```
class Stamp refines Print {
    concept characteristic value : Currency
            characteristic nrOfPrints : Integer
            relationship country : Country
}
class Artist { ... }
class Country { ... }
```

Jede Assetklasse besitzt je eine Inhalts- und eine Konzeptsektion, die auch leer sein dürfen. Auffällig ist, dass in der Konzeptsektion im Beispiel zwei Arten von Attributen auftreten: Charakteristika (`characteristic`) sind immanente Eigenschaften eines Assets, die untrennbar mit ihm verbunden sind und keine eigene Identität besitzen. Im Gegensatz dazu verweist eine Beziehung (`relationship`) auf ein anderes Asset, welches auch für sich alleine existiert und eine eigene Identität besitzt. Auf die Modellierungskonzepte der ADL wird in Kapitel 3 genauer eingegangen. Dort wird diskutiert, wie die Konzepte der ADL auf Zope3 abgebildet und realisiert werden können.

2.2.4 Kompiler Framework

Der CCM Modellkompiler ist wie die meisten Kompiler zweistufig aufgebaut: Das Frontend liest die Assetdefinitionen, überprüft sie auf Korrektheit und erzeugt aus ihnen ein Zwischenmodell (*intermediate model*). Daraus generiert das Backend den Quellkode eines Inhaltsverwaltungssystems. Die Besonderheit an dem CCM Modellkompiler ist, dass das Backend modular aufgebaut ist und durch *Generatoren* ergänzt wird. Generell sind mehrere Generatoren an der Erzeugung eines Systems beteiligt. Je nach Auswahl der verwendeten Generatoren können verschiedene Arten von Systemen erzeugt werden.

Das Backend ist als **Framework** realisiert: Der Kompiler bestimmt selbst, in welcher Reihenfolge die einzelnen Generatoren ausgeführt werden. Um seine Aufgabe wahrzunehmen, kann ein Generator neben dem Zwischenmodell noch Informationen von anderen Generatoren benötigen, die durch *Symboltabellen* übermittelt werden. Jeder Generator erzeugt eine Symboltablle und kann von beliebig vielen anderen abhängen. Anhand der Symboltabellen, die von den Generatoren benötigt werden, kann der Kompiler eine Ausführungsreihenfolge festgelegen. Die eigentliche Aufgabe der Generatoren - die Erzeugung von Kode - geschieht unabhängig von dem Austausch der Symboltabellen als Seiteneffekt.

Die Zusammenhänge sollen anhand des Aktivitätsdiagammes in Abbildung 2.5 veranschaulicht werden. In dem **Beispiel** sind zwei Generatoren als Aktionen dargestellt: Ein Generator erzeugt SQL Schemata, dazu benötigt er nur das Zwischenmodell. Der andere generiert JDBC (*Java Database Connectivity*) Kode, der von dem Datenbankschema abhängt - dieses wird ihm über die Symboltabelle `schemaSymbolTable` zur Verfügung gestellt. Es ist offensichtlich, dass der Schemagenerator zuerst ausgeführt werden muss. Zu dem Diagramm in Abbildung 2.5 sei angemerkt, dass es eine kombinierte Darstellung von Kontroll- und Objektfluss zeigt, wie sie in der UML 2 möglich ist. Aktionen mit mehreren eingehenden Kanten vollziehen im Unterschied zu früheren Versionen eine implizite Synchronisation (*join*). (siehe [Oes06])

2.2.5 Vorteile gegenüber generischen Systemen

Die Unterschiede zwischen der Verwendung eines generischen Systems und CCM Systemen sind zu zahlreich, als dass sie hier genau diskutiert werden könnten. Trotzdem sollen einige wichtige Punkte Erwähnung finden, die im Rahmen dieser Arbeit von Interesse sind.

Aus **Sicht des Anwenders** eines CCM Systems ergibt sich der Vorteil, dass das Datenmodell für ihn einsehbar und verständlich ist. Besteht für ihn der Bedarf, das Modell zu verändern oder zu personalisieren, ist keine Weiterentwicklung generischer Komponenten notwendig. Er selbst kann die gewünschten Änderungen unmittelbar vornehmen. Dadurch wird zum einen der Anwender optimal in seinem Arbeitsprozess unterstützt, zum anderen entfallen langwierige und kostpielige Entwicklungsprozesse, die bei der Anpassung eines generischen Systems durch Fachkräfte erforderlich sind.

Aus **technischer Sicht** werden durch den Einsatz generierter Systeme viele Probleme gelöst: Es können mehrere Basistechnologien effektiv und effizient miteinander kombiniert werden. Z. B. eignen sich relationale Datenbanken gut für die Speicherung der Konzeptattribute, für mediale Inhalte (z.B. Filme) ist die Speicherung im Dateisystem effizienter. Durch geschickte Kombination der Generatoren kann man erreichen, dass die konzeptionellen Attribute eines Assets in einem RDBMS, Inhalte jedoch direkt in dem

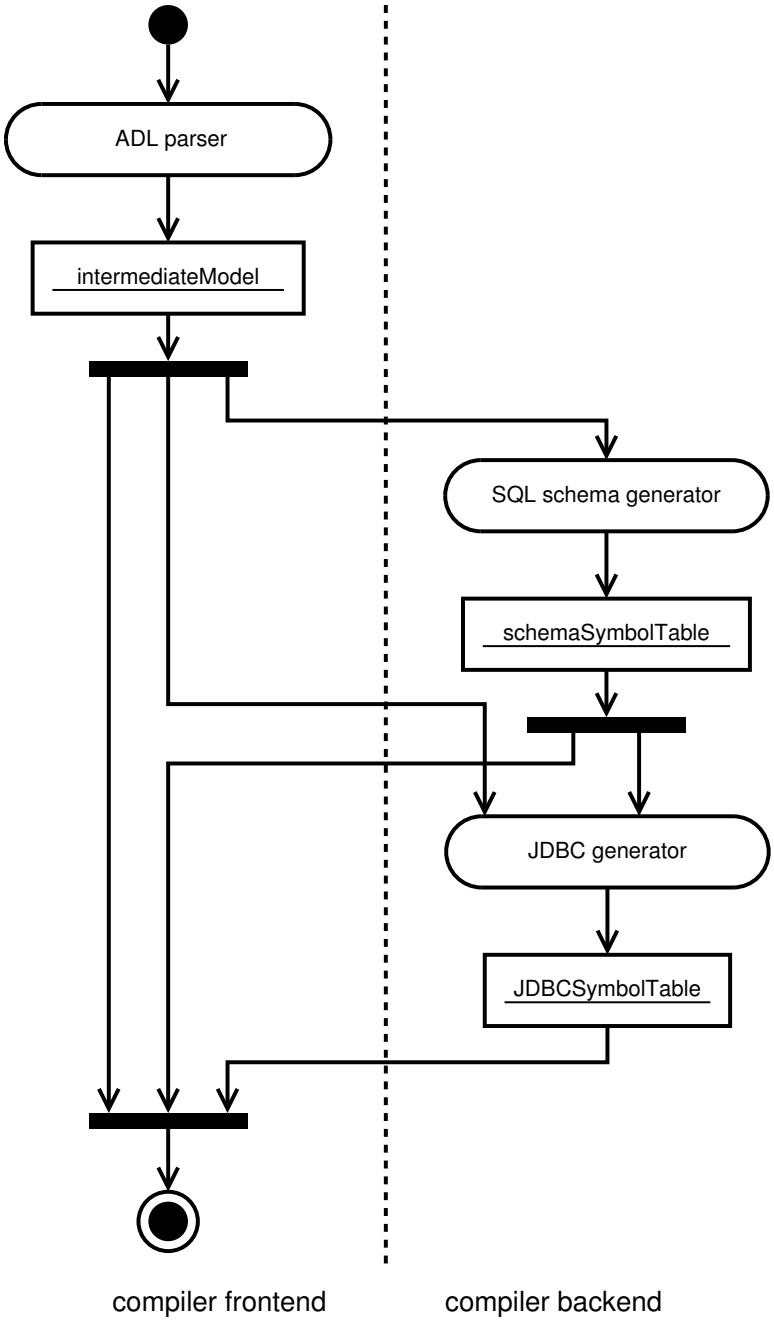

Abbildung 2.5: Abläufe im Modellkompiler

Dateisystem gespeichert werden. Ein anderer technischer Vorteil der Systemgenerierung ist für diese Arbeit von besonderer Bedeutung: die Plattformunabhängigkeit. Aus dem abstrakten Assetmodell können Implementierungen für verschiedene Plattformen erzeugt werden, die auch miteinander kooperieren können, da sie auf dem gleichen Datenmodell basieren. (Vergleiche [Seh04])

2.3 Generierung Zope3 basierter CCM Systeme

2.3.1 Eigenständige Zope Systeme

Eine einfache Möglichkeit, die Fähigkeiten von Zope für CCM auszunutzen, ist die Generierung eigenständiger Zope Systeme, deren Funktionsumfang sich an typischen Zope Anwendungen orientiert. Die erzeugten Inhaltsverwaltungssysteme sind auf einem Zope Server lauffähig, können allerdings nicht mit anderen CCM Modulen kommunizieren. Um diesen Ansatz zu verwirklichen muss analysiert werden, inwieweit sich Assetmodelle auf Zope übertragen und realisieren lassen.

Dieser Ansatz ist interessant, weil so die Stärken von Zope in der Präsentation von Inhalten ausgenutzt werden. Die Probleme der automatischen Generierung von Präsentationsmodulen in CCM - egal ob webbasiert oder als grafische Oberfläche - sind in dem Modellkompiler noch nicht hinreichend gelöst. Darüber hinaus ist ein eingenständiges System notwendige Grundlage für die Entwicklung ausgefeilterer CCM-Zope Systeme. Durch die Erfahrungen mit eigenständigen Zope Systemen kann beurteilt werden, ob Zope eine geeignete Plattform für die konzeptorientierte Inhaltsverwalung darstellt und ob eine Fortführung dieser Arbeit vielversprechend erscheint.

In einer aufbauenden Arbeit könnte Zope selbst angepasst und erweitert werden: Es ist denkbar, Zope um Konzepte aus CCM zu erweitern, um die Realisierung der Assetmodelle zu vereinfachen. Auch die Präsentationskomponenten könnten um weitere Sichten (*views*) erweitert werden, die es erlauben, Assets angepasst darzustellen. Durch die flexible Struktur von Zope lassen sich solche Erweiterungen als eigene Komponenten realisieren.

2.3.2 Offene und dynamische Inhaltsverwaltung mit Zope

Der wichtigste Anspruch der CCM Systeme, die Verwirklichung von Offenheit und Dynamik, wird durch typische Zope Systeme nicht erfüllt. Daher ist es interessant zu überprüfen, inwieweit offene und dynamische Inhaltsverwaltung mithilfe des Modellkompilers und Zope3s möglich ist, ohne dass die generierten Systeme mit anderen CCM Modulen kooperieren.

Dazu müssen die generierten eigenständigen Zope Systeme eine Evolution der Assetmodelle unterstützen können. Auch die mit Offenheit und Dynamik zusammenhängende Forderung nach Personalisierungsmöglichkeiten muss nachgegangen werden. Eine Behandlung dieser Themen findet sich in Abschnitt 5.2.

2.3.3 Zope als CCM Dienstschnittstellenmodul

Um alle Möglichkeiten der CCM Systeme auszuschöpfen, ist es unumgänglich, eine Kommunikation zur Laufzeit zwischen Zope Komponenten und CCM Modulen zu ermöglichen. Zope eignet sich für den Einsatz als Dienstschnittstellenmodul am besten, da so seine Stärken in der Präsentation für CCM Systeme genutzt werden können. Ein Einsatz als Interpretationsmodul ist auch denkbar, aber weniger interessant, da Zope als Standardkomponente zur Datenhaltung keine entscheidenden Vorteile gegenüber einer relationalen oder XML Datenbank bietet.

Aufgrund der Autonomie der Standardkomponenten (vgl. [Seh04]) ist auch ein anderes Szenario denkbar: Zope Systeme können eigenständig funktionieren und die Aufgaben der Datenhaltung und Präsentation wahrnehmen. Es wäre jedoch möglich, diese Systeme durch weitere CCM Module zu ergänzen, z.B. durch Dienstschnittstellenmodule für Web Services oder grafische Benutzeroberflächen - mit Zope allein können diese Funktionen nicht realisiert werden.

Für beide Varianten ist es notwendig, dass Zope3 und die Java basierten Module über die einheitliche Modulschnittstelle miteinander kommunizieren. Es ergeben sich hierbei sowohl prinzipielle Probleme, da Zope und CCM auf verschiedenen Konzepten basieren, wie auch rein technische: z.B. die Realisierung der Kommunikation zwischen den verschiedenen Programmiersprachen. Diese Probleme werden im Ausblick in Kapitel 5.3 genauer erläutert.

2.3.4 Entscheidung

Vor allem aus Gründen der einfachen Umsetzbarkeit, sollen im Rahmen dieser Arbeit eigenständige Zope Systeme generiert werden. Dieser Ansatz bringt auch den Vorteil, dass so die Stärken von Zope3, die Datenbank und die automatische Präsentation von Inhalten, ausgenutzt werden können.

Offenheit und Dynamik sind die herausragenden Eigenschaften von CCM Systemen, ihre Realisierung auf Zope würde jedoch den Umfang dieser Arbeit sprengen. Zope als ein CCM Modul zu verwenden bringt das Problem mit sich, dass zur Laufzeit eine Kommunikation zwischen den Java-basierten und den Zope-basierten Modulen des Systems erforderlich ist. Es wurde bereits erläutert, dass es nur begrenzte Möglichkeiten gibt, Zope3 in Java Systeme zu integrieren.

Grundlage für alle Zope basierten CCM Systeme ist die Abbildung des Assetmodells auf Zope3 Inhaltskomponenten. Die Realisierung der Abbildung ist ein zentrales Anliegen dieser Arbeit. Wie man darauf aufbauend Offenheit und Dynamik realisieren kann, entweder direkt auf Zope oder durch Java basierte CCM Module, wird im Ausblick erläutert. Die praktische Umsetzung wird jedoch nicht angestrebt.

Kapitel 3

Abbildung von Assetmodellen auf Zope3 Inhaltskomponenten

Nachdem im vorigen Kapitel CCM Assets und Zope3 Inhaltskomponenten vorgestellt wurden, soll in diesem Kapitel untersucht werden, wie sich in der Assetdefinitionssprache (ADL) formulierte Assetmodelle auf Zope Schemata übertragen lassen. Nachdem der Umfang dieser Abbildung festgelegt wurde, folgt eine Diskussion, wie sich die grundlegenden Kozepte der ADL und ihre Sprachmittel für konzeptionelle Beschreibungen auf Zope übertragen lassen. Jedes Modellierungskonzept der ADL wird gesondert in einem eigenen Abschnitt betrachtet.

3.1 Umfang der Abbildung

Wie im Abschnitt 2.3 festgelegt wurde, ist das Ziel dieser Arbeit die Erzeugung **eigenständiger Zope3 Systeme** , die auf einem Assetmodell basieren. Aus der Eigenständigkeit des zu erzeugenden Systems folgt, dass die Teile der Assetsprache, die für das Auffinden und Manipulieren von Assets zuständig sind, nicht umgesetzt werden können und deshalb nicht weiter beachtet werden.

Darüber hinaus werden in dieser Arbeit Systeme mit **für Zope typischer Funktionalität** erzeugt. Es muss bedacht werden, dass Zope als Anwendungsserver zwar eine Basis für Inhaltsverwaltungssysteme darstellt, aber selbst noch keines ist. Die Notwendigkeit, das System zu erweitern, ist daher gegeben, ihr wird jedoch nicht nachgegangen, da eine Erweiterung von Zope nicht Gegenstand dieser Arbeit ist. Mögliche Erweiterungen von Zope werden im Ausblick, Unterkapitel 5.2, wieder aufgegriffen.

Die Generierung typischer Zope Systeme hat noch eine weitere Folge: Der **Lebenszyklus eines Assets,** wie er in der einheitlichen Modulschnittstelle

implementiert ist, wird hier nicht verfolgt. Die ZODB (*Zope Object Database*) realisiert für in ihr gespeicherte Objekte einen eigenen Lebenszyklus, dem auch die generierten Inhaltskomponenten unterliegen. Ein Zustandsdiagramme, welche die Lebenszyklen eines in der ZODB gespeicherten Objektes und eines Assets veranschaulichen, finden sich im Anhang A.

3.2 Grundlegende Konzepte

3.2.1 Typkonstruktion

Konzept in CCM

Wie in objektorientierten Modellen Mengen von Objekten zu Klassen abstrahiert werden, werden in der Assetdefinitionssprache Mengen von Assets zu Klassen zusammengefasst. Die Erstellung eines Domänenmodells mit der Assetdefinitionssprache ist ähnlich der konzeptuellen Modellierung einer Anwendungsdomäne in der objektorientierten Analyse, z. B. durch ein Klassendiagramm. ([Bal01])

Die Modellierung mit Assets weist bei der Konstruktion von Typen zwei wesentliche Unterschiede zum objektorientierten Ansatz auf: Das Fehlen der Verhaltensmodellierung und die Trennung zwischen Inhalt und Konzept. Im Gegensatz zu Objektklassen wird in der konzeptuellen Modellierung kein Verhalten berücksichtigt. In dieser Hinsicht entsprechen die Assetklassen der ER (*entity relationship*) Modellierung nach Chen. ([Che75]) In Assetklassen wird allerdings eine Unterscheidung zwischen Inhalt (*content*) und Konzept (*concept*) getroffen, in diesem Aspekt erweitern sie die bisher genannten Modellierungsansätze.

Die Unterscheidung zwischen Inhalt und Konzept geht auf den Wissenschaftsphilosophen Ernst Cassirer zurück, der zu dem Schluss gelangte, dass sich Wissen am besten in Einheiten von Inhalt und Konzept beschreiben lässt. Diese Erkenntnis stimmt mit den Erfahrungen aus der WEL überein, die gezeigt haben, dass Inhalte immer mit einem abstrakten Konzept verknüpft sind. Ein Asset ist demzufolge definiert als eine Einheit von Inhalt und Konzept.

Als Beispiel dient hier eine Assetklasse für Drucke. Die Sektionen für Inhalt und Konzept werden in den folgenden Abschnitten erläutert.

```
class Print {
    content ...
    concept characteristic title ...
}
```

Entsprechung in Zope

Die Typkonstruktion in Zope geschieht durch die Schemata, welche die Struktur von Inhaltskomponenten definieren. Da die Schemata eine Erweiterung der Zope Schnittstellen sind, kann man in ihnen sowohl Eigenschaften wie auch Methoden der sie implementierenden Klassen festlegen. Es wird in Zope allerdings nicht zwischen Inhalt und Konzept eines Objektes unterschieden. Die Möglichkeiten mit diesem Umstand umzugehen werden im Folgenden erläutert.

```
class Print (Interface):
    title = ...
```

Realisierungsvarianten

In dieser Arbeit wird die Trennung zwischen Inhaltsobjekten und Konzeptattributen vernachlässigt. Sie werden gemeinsam als Felder in das Schema zu übernommen. Die Interpretation, ob ein Feld einen Inhalt oder ein Konzept repräsentiert, bleibt dem Anwender überlassen. Durch diese Lösung wäre der generierte Kode dem für Zope handgeschriebenen sehr ähnlich und wäre für mit dem System vertraute Programmierer leicht zu erweitern und zu warten.

Obwohl Zope diese Unterscheidung nicht unterstützt, ist es möglich, sie für den Anwender des Systemes vorzutäuschen. Es können unveränderliche Felder in die Schemata eingeführt werden, die als Überschriften für den Inhalts- und Konzeptteil eines Inhaltsobjektes verwendet werden. Der Anwender sieht beim Betrachten der Inhaltskomponente mit einem Browser Abschnitte, die mit `content` und `concept` betitelt sind. Diese Lösung bereitet bei der Subklassenbildung von Assets allerdings Schwierigkeiten, weil zusätzliche Attribute von Schemata nach den geerbten Attributen erscheinen. Dieses kann durch das Feldattribut `order` verhindert werden, durch das die Ordnung der Felder untereinander festgelegt werden kann.

```
class Print (Interface):
    contentSectionTitle = TextLine( order=0,
                                    value="content",
                                    readonly=True)
    ...
    conceptSectionTitle = TextLine( order=10,
                                    value="concept",
                                    readonly=True)
    title = TextLine( order=11, ...
```

Eine aufwändigere Lösung ist es, die in Zope verwendeten Schemata um die Möglichkeit zu erweitern, ihre Felder nach Inhalt und Konzept zu unterscheiden. Da diese Unterscheidung nicht aufgrund des Typs eines Feldes

getroffen werden kann - es gibt auch mediale Darstellungen von Konzepten - müssen die Schemata um ein syntaktisches Konstrukt ergänzt werden, um Inhalts- bzw. Konzeptfelder zu kennzeichnen. Da die Schemata lediglich durch Pythonklassen realisiert sind, ist diese Möglichkeit gegeben. Die generierten Systeme sind so mit dem Assetmodell konsistent. Diese Lösung ist allerdings aufwändig, weil die Präsentationskomponenten von Zope angepasst werden müssen, so dass sie Inhalt und Konzept voneinander unterscheidbar darstellen können.

```
class Print (CCMExtendedSchema):
    picture = ...
    declareAsContent(picture)
    title = ...
```

3.2.2 Eingebettete Sprache

Konzept in CCM

Die Assetdefinitionssprache definiert selbst keine Basistypen für Inhaltsobjekte und Konzeptattribute. Stattdessen können an dieser Stelle beliebige Java Klassen verwendet werden, sie müssen lediglich in ausführbarer Form vorliegen. (d. h. als Java Bytekodedateien) Da die Module eines CCM Systems auf Java basieren, ist man durch die Verwendung beliebiger Typen der Basissprache sehr flexibel. Allerdings stellte sich bis zu dieser Arbeit noch nicht die Frage, wie man mit anderen Zielsprachen als Java verfahren kann.

Entsprechung in Zope

Im Gegensatz zu den Assetklassen ist man für die Felder der Schemata auf eine kleine Menge an Feldtypen beschränkt, die wichtigsten sind in Tabelle 3.1 aufgeführt.

Diese Beschränkung hat zwei Ursachen: Feldtypen müssen persistent sein, was nicht für alle Python Klassen gegeben ist. Die integrierte Datenbank ZODB hält nur Objekte persistent, die entweder Python Basistypen sind, oder deren Klassen von `persistent` erben. Die Verwendung nicht persistenter Klassen als Felder hat zur Folge, dass Attributwerte nicht gespeichert werden.

Der andere Grund für die Typbeschränkung ist, dass den meisten Feldtypen Widgets zugeordnet sind, die ermöglichen, für eine Inhaltskomponente automatisch Präsentationskomponenten zu erzeugen. Diese Funktion von Zope wird in dieser Arbeit genutzt. Darüber hinaus bieten die Widgets die Möglichkeit, Eingaben zu validieren, oder Bedingungen (*constraints*) an Felder zu knüpfen. Neben den in Tabelle 3.1 aufgeführten gibt es noch weitere Feldtypen, die hier nur von geringem Interesse sind, für eine vollständige Übersicht wird hier auf Kapitel 8 in [Ric05] verwiesen.

`TextArea`	eine mehrzeilige Zeichenkette
`Bytes` `BytesLine`	Zeichenketten, allerdings kein Unicode, eine BytesLine darf keine Zeilenumbrüche enthalten
`Text` `TextLine`	Unicode Zeichenketten, eine `TextLine` darf keine Zeilenumbrüche enthalten
`SourceText` `Password` `URI`	Zeichenketten mit zusätzlichen Nebenbedingungen oder alternativen Widgets zur Darstellung
`Int`	ganze Zahlen
`Float`	Gleitkommazahlen, entspricht `double` in Java
`Datetime`	Datum mit Uhrzeit
`Tuple` `List`	Liste von Feldern eines einheitlichen Typs
`Choice`	Enumerationstyp, entspricht einer `enum` in Java
`Bool`	Wahrheitswert, `True` oder `False`, entspricht einem `boolean` in Java

Tabelle 3.1: Zope3 Feldtypen

Realisierungsvarianten

Die erste Variante besteht darin, **Java als eingebettete Sprache** beizubehalten und Javaklassen auf Zope Schematypen abzubilden. Da die Zope Schemata nur über wenige Basistypen verfügen, gibt es viele Java Typen, für die keine Entsprechung in Zope gefunden werden kann. In anderen Arbeiten wurde dieses Problem gelöst, indem Javaklassen, für die keine Entsprechung im Zielsystem existiert, serialisiert und als BLOBs (binary large objects) gespeichert werden. Weil das zu erzeugende Zope System eigenständig laufen soll, kann diese Lösung hier nicht verwendet werden. Es ist notwendig, die verwendbaren Javaklassen so einzuschränken, dass für jede Klasse ein entsprechender Zope Feldtyp existiert. Der Vorteil dieses Ansatzes ist, dass die in der Assetdefinitionssprache vorliegenden Modelle auch zur Generierung von Java basierten CCM Systemen verwendet werden können.

Alternativ können, statt Javaklassen die **Zope Feldtypen in die Assetdefinitionssprache eingebettet werden.** Die oben genannten Probleme fallen dann weg, allerdings können die Assetdefinitionen dann nicht zur Generierung Java basierter CCM Module verwendet werden. Die Beschränkung auf die Feldtypen scheint im Vergleich zu Java eine große Einschränkung zu sein. Werden zusätzliche Typen benötigt, kann Zope um diese erweitert werden, da die Programmierschnittstelle für das Erstellen von

Feldtypen und Widgets frei zugänglich ist. Dieser Ansatz wird realisiert, weil dadurch für die Assetdefinitionen die größtmögliche Menge an Basistypen zur Verfügung steht.

3.2.3 Inhaltsobjekte

Konzept in CCM

Wie bereits in dem Abschnitt „Typkonstruktion“ diskutiert wurde, wird in der ADL zwischen Inhalt und Konzept unterschieden. Es soll hier geklärt werden, ob die Inhalte der Assets auch in den zu erzeugenden Zope3 Systemen besonders gehandhabt werden sollten. Wie zuvor erläutert, erlaubt es CCM in der Inhaltssektion, beliebige Javaklassen als Typen für Inhaltsobjekte zu verwenden. Im Fall der Inhaltsobjekte sind dies typischerweis mediale Typen, z.B. für Bilder, Grafiken, Video- oder Audiosequenzen. Je nach Anwendung ist es auch sinnvoll, Dokumente von Büroanwendungen oder HTML-Seiten als Inhaltsobjekte zu verwenden, wie es z.B. in Dokumentenverwaltungssystemen üblich ist.

```
class Print {
    content preview : Image
... }
```

Entsprechung in Zope

In den Zope3 Schemata gibt es nur eine geringe Anzahl von Basistypen. Es ist nicht möglich, mediale Inhalte in ein Schema aufzunehmen, da für solche keine Feldtypen vorhanden sind. Es gibt lediglich die Möglichkeit, beliebige Dateien als eigenständige Inhaltskomponenten vom Typ `File` zu speichern.

Realisierungsvarianten

Die einfachste Lösung ist, sämtliche **Inhalte als Datei in der Datenbank abzulegen** , da diese Funktion bereits in der Zope3 Administrationsoberfläche vorhanden ist. Weil sich alle Inhaltskomponenten in Zope über eine URL auffinden lassen, kann das Schema der Assetklasse durch einen Link auf Inhalte verweisen. Dies ist einfach zu implementieren, weil in das Schema lediglich die URL aufgenommen werden muss. Allerdings existieren so die Inhaltskomponente und ihr Inhalt unabhängig voneinander und können einzeln gelöscht werden. Darüber hinaus ist das Anlegen eines Assets mit einem Inhalt für den Benutzer umständlich, weil mehrere Schritte nötig sind. Trotz dieser Nachteile soll diese Variante realisiert werden, da die Erweiterung von Zope nicht das Ziel dieser Arbeit ist.

```
class IPrint (Interface)
    preview = URI( name="preview" ) # ...
```

Diese Nachteile entfallen, wenn man die Datei mit dem Inhalt als Zeichenkette in der Inhaltskomponente speichert. Dies ist mit der Verwendungs von **BLOBs** in relationalen Datenbanken vergleichbar. Zusätzlich zu den eigentlichen Daten muss noch der Dateiname, oder wenigstens Typinformation wie die Dateiendung oder der MIME Typ gespeichert werden, damit der Inhalt interpretierbar bleibt. Die notwendige Funktionalität ist in den Feldtypen und Widgets bereits vorhanden. Allerdings muss daraus ein neuer Feldtyp entwickelt werden, damit Dateiname und die Typinformation gespeichert wird. (im Beispiel `Content`)

```
class IPrint (Interface)
    preview = Content( name="preview" ) # ...
```

Am interessantesten ist jedoch, die Anpassungsfähigkeit von Zope auszunutzen und **Schematypen für mediale Daten** zu definieren. Ähnlich wie in den Java basierten CCM Systemen kann man hierbei auf Python Bibliotheken zurückgreifen und auf ihrer Basis Widgets implementieren. Dies ermöglicht interessante Perspektiven für die Integration medialer Inhalte: Durch Widgets für die Darstellung von Bildern können diese bei Bedarf in verschiedenen Auflösungen und Bildformaten bereitgestellt werden. Durch die Integration entsprechender Software könnten Widgets auch Filmdaten skalieren und komprimieren.

3.2.4 Spezialisierung

Konzept in CCM

Das Konzept der Spezialisierung von Klassen ist in der ADL und objektorientierten Methoden identisch. Wird eine Assetklasse von einer anderen abgeleitet (Schlüsselwort `refines`), erbt sie alle strukturellen Eigenschaften der Basisklasse: Inhalte und konzeptionelle Beschreibungen (Attribute, Bedingungen)

```
class Stamp refines Print {
    concept characteristic value : Currency }
```

Es soll hier darauf hingewiesen werden, dass in CCM noch eine andere Form der Subklassenbildung existiert. Bei der *extensionalen Definition von Subklassen* werden Klassen durch die Angabe ihrer Instanzen definiert. Dies kann durch Aufzählung aller Assets dieser Klasse (statisch extensional) oder durch eine Angabe typischer Klassenvertreter (dynamisch extensional) geschehen. Extensionale Klassenbildung ist jedoch nicht Gegenstand dieser Arbeit, daher wird für weitere Details auf Abschnitt 3.4.3 in [Seh04] verwiesen.

Entsprechung in Zope

Auch die Zope Schemata können spezialisiert werden. Wie bei den Python Klassen ist Mehrfachvererbung auch mit Schemata möglich. Dies wird für diese Arbeit nicht genutzt, wird jedoch für den Fall bedeutsam, sollte die ADL um das Konzept der Mehrfachvererbung erweitert werden.

Bei den Klassen, welche die Schemata implementieren, ist darauf zu achten, dass der Konstruktor der Elternklasse aufgerufen wird, um eventuell notwendige Initialisierungen vornehmen zu können. Es folgt ein Beispiel wie dies im Python Kode bewerkstelligt werden kann.

```
class IStamp (IPrint):
    value = Float( name="value" )

class Stamp (Print): implements(IStamp)
    def __init__(self, *args, **kargs):
        super(Print,self).__init__(*args, **kargs)
        self.value = None
```

Weil das Konzept der Spezialisierung in der ADL und in Zope gleichermaßen vorhanden ist, bedarf es keiner besonderen Behandlung.

3.3 Konzeptionelle Beschreibung

3.3.1 Charakteristika

Konzept in CCM

In der ADL beschreiben Charakteristika (`characteristic`) immanente Eigenschaften von Assets, solche die untrennbar mit der beschriebenen Entität verbunden sind. Sie entsprechen den Objektattributen bei der objektorientierten Modellierung.

Die ADL ermöglicht es, Charakteristika Initialwerte zuzuweisen. Diese müssen entweder durch ein Literal oder durch einen Ausdruck der eingebetteten Sprache gegeben sein, der zur Konstruktionszeit der Instanz ausgewertet werden kann.

```
class Print {
    content ...
    concept characteristic title : String := "<untitled>"
    ... }
```

Entsprechung in Zope

Die Charakteristika der Assetdefinitionssprache sind konzeptionell identisch mit den Feldern der Schemata. Lediglich die Fähigkeit Charakteristika von

Inhaltsobjekten zu unterscheiden ist nicht vorhanden und wurde bereits in Abschnitt 3.2.1 diskutiert. Es ist in den Schemata möglich, Initialwerte in Form von Literalen anzugeben, aber nicht Felder durch Ausdrücke zu initialisieren. Charakteristika können direkt auf die Zope Felder abgebildet werden, allerdings sind Initialwerte nur in Form von Literalen zulässig.

```
class IStamp (IPrint):
    title = TextLine ( name="title", default=u"<untitled>" )
    # (Anmerkung: u".." kennzeichnet einen Unicode String)
    ...
```

3.3.2 Beziehungen

Konzept in CCM

Assets verfügen neben Charakteristika noch über eine andere Art von Attributen: den Beziehungen (`relationship`). Im Gegensatz zu Charakteristika, die keine eigene Identität besitzen, verweisen Beziehungen auf eigenständige Assets. Aus diesem Grund sind für die Typen für Beziehungen nur Assetklassen und keine Typen der eingebetteten Sprache zulässig. Es ist auch möglich, Initialwerte für Beziehungen durch benannte Assets anzugeben.

```
class Print {
    content ...
    concept relationship artist : Artist := Picasso
    ... }
```

Entsprechung in Zope

Obwohl die in Zope3 verwendete Datenbank es ermöglicht, beliebige Objektgraphen persistent zu halten, gibt es in den Schemata keine Möglichkeit, in einem Schema Beziehungen zwischen Inhaltskomponenten auszudrücken. Dieser Mangel ist erstaunlich, wenn man bedenkt, dass Zope als Basis für verschiedenen Inhaltsverwaltungssysteme verwendet wird. Es ist lediglich möglich, URLs zu speichern, die auch auf andere Inhaltskomponenten verweisen können.

Realisierungsvarianten

Weil jede Inhaltskomponente in Zope über eine URL ansprechbar ist, kann man **Webverweise (HTML links)** dazu verwenden, um auf andere Assets zu verweisen. Diese Lösung ist einfach zu realisieren, bringt aber viele Nachteil mit sich. Durch das Löschen oder Umbenennen von Inhaltskomponenten kann es passieren, dass bestehende Verweise ungültig werden. (broken links) Systematisches Überprüfen der Verweise ist keine Lösung, da neue Inhaltskomponenten gleich benannt werden können.

Eine elegantere Lösung ist es, einen **Feldtyp für Beziehungen** zu implementieren, der eine Datenbank-Referenz auf die in Beziehung stehende Inhaltskomponente speichert. Zur Darstellung werden Widgets benötigt, die es dem Anwender ermöglichen, die gewünschten Beziehungen anzulegen. Der Implementierungsaufwand liegt wesentlich höher als bei der vorigen Lösung, allerdings verschwinden ihre Nachteile fast vollständig. Nur das Löschen von Inhaltskomponenten bleibt ein Problem: Werden Inhaltskomponenten gelöscht, die noch von Anderen referenziert werden, bleiben diese in der Datenbank bestehen, sind jedoch nicht mehr über über eine URL ansprechbar. Trotz dieses Problems wird diese Lösung implementiert, da sie für weiterführende Arbeiten die Möglichkeit bietet, über das Widget eine komfortable Benutzeroberfläche zu implementieren.

```
class IStamp (IPrint):
    artist = Relationship ( name="artist",
                            schemaName="IArtist" )
    # ...
```

Durch die URLs einer Inhaltskomponente ist es auch möglich, Initialwerte für Beziehungen anzugeben. Vorraussetzung dafür ist, dass die vollständige URL eines Assets angegeben wird oder zur Laufzeit hergeleitet werden kann. Dies kann dadurch erreicht werden, dass Assets der Klasse `Artist` immer in dem gleichen Zope Ordner gespeichert werden. Da es nicht möglich ist, die Sprachmittel zur Assetmanipulation in dieser Arbeit zu realisieren, kann nicht sichergestellt werden, dass die genannten Assets bereits existieren. Da Inkonsistenzen mit dem Assetmodell die Folge wären, werden Initialwerte für Beziehungen nicht implementiert. In diesem Zusammenhang sei darauf verwiesen, dass der Lebenszyklus der Assets, wie er in der einheitlichen Modulschnittstelle realisiert ist, hier keine Verwendung findet. Siehe Abschnitt 3.1.

3.3.3 Mengenwertige Attribute

Konzept in CCM

Ähnlich wie in der UML, wo Kardinalitäten für Attribute und Assoziationen angegeben werden können, ist es auch in der ADL möglich, mengenwertige Attribute zu verwenden. So kann zum Beispiel durch ein Beziehungsattribut statt eines einzelnen eine Menge von Assets referenziert werden. Initialwerte können auch für mengenwertige Attribute angegeben werden, es müssen nur jeweils statt eines einzelnen Mengen des Attributes angegeben werden. Im Beispiel wird es ermöglicht, Drucke mit mehreren Künstlern in Beziehung zu setzen.

```
class Print { ...
    concept relationship artists : Artist*
    ... }
```

Entsprechung in Zope

Mit dem Zope Feldtyp `List` kann man Listen beliebiger Länge eines einheitlichen Feldtyps anlegen. Die Darstellung von Listen in Webseiten geschieht durch die vorhandenen Widgets automatisch. Allerdings gibt es konzeptuell zwei Unterschiede zu den mengenwertigen Attributen in der ADL: Eine `List` kann Duplikate enthalten und es wird die Reihenfolge der enthaltenen Elemente berücksichtigt. Letzteres stellt kein Hindernis dar, allerdings stellt sich die Frage, wie mit Duplikaten umgegangen werden soll.

Realisierungsvarianten

Die einfachste Lösung ist es, das **Duplikatproblem** zu **vernachlässigen.** Für mengenwertige Attribute werden Zope3 `List`en verwendet, in der Hoffnung, dass durch mögliche Redundanz der Attribute keine Probleme entstehen werden. Dieser Ansatz soll im praktischen Teil dieser Arbeit realisiert werden.

Es gibt auch die Möglichkeit, einen **eigenen Feldtyp** für Zope zu implementieren, der für die Darstellung von Mengen geeigneter ist. Es wäre möglich, den Typ `List` zu einer *UniqueList* zu modifizieren, welche die Zuweisung von Duplikaten verweigert. Effizienter, aber auch aufwändiger, ist es, einen unabhängigen Feldtyp z.B. auf Basis der Python Standardklasse `Set` zu implementieren.

```
class IStamp (IPrint):
    artists = UniqueList( name="artists",
                value_type=Relationship )
    # ...
```

3.3.4 Bedingungen

Konzept in CCM

Die Deklarationen von Assetklassen können Bedingungen (*constraints*) enthalten, die den Wertebereich ihrer Attribute einschränken. Eine Bedingung ist ein logischer Ausdruck auf den Assetattributen, der stets erfüllt sein muss. Es gibt zwei Möglichkeiten, um Bedingungen zu formulieren: Wenn sich eine Bedingung nur auf ein einzelnes Attribut bezieht, kann die Bedingung der Attributdeklaration angehängt werden. Die andere Möglichkeit ist, Bedingungen eigenständig in Assetklassen aufzunehmen. In diesem Fall kann sich die Bedingung auch auf mehrere Attribute des Assets sowie klassenübergreifend auf Attribute durch Beziehungen referenzierter Assets beziehen.

```
class PreciousStamp refines Stamp {
    concept
        characteristic pricePaid : Currency > 1000 EUR
        relationship owner : Collector
        constraint owner.collectionValue >= pricePaid
}
```

Die logischen Ausdrücke für Bedingungen bestehen aus sieben Vergleichsoperatoren `=`, `#` (ungleich), `~` (ähnlich, ist optional), `<`, `<=`, `>`, `>=` und drei logischen Operatoren `not`, `and`, `or`. Die Vergleichsoperatoren müssen von den Attributtypen in der Zielsprache implementiert werden, und auch für mengenwertige Attribute verfügbar sein, wobei die Operatoren `<` und `>` dann auf echte Teil- und Obermengenbeziehung prüfen.

Entsprechung in Zope

Auch die Feldtypen in Zope bieten die Möglichkeit, Bedingungen an sie zu knüpfen. Bei der Deklaration eines Feldes wird eine Bedingung in Form einer Funktion vorgegeben. Diese Funktion wird bei jeder Änderung des Feldes über die Weboberfläche überprüft und muss immer einen logisch wahren Rückgabewert liefern. Präzise formuliert, handelt es sich nicht um Funktionen, sondern um aufrufbare Objekte (*callable objects*). Sie sind ein Bestandteil der Sprache Python und beinhalten Funktionen, Klassen und andere Objekte wie z.B. reguläre Ausdrücke, die auch als Bedingung eingesetzt werden.

Neben der allgemeinen Form können spezielle Bedingungen in Zope auch direkt angegeben werden, z. B. die Beschränkung numerischer Feldtypen auf ein Intervall und Längenbeschränkungen von Zeichenketten und Listen.

```
value = Float( title="value" , min=0.0 , max=10.0 )
positiveValue = Float( title="value" ,
                       constraint=lambda val : val>=0.0 )
```

Es ist allerdings nicht möglich, Bedingungen zwischen mehreren Feldern eines Schemas zu definieren.

Realisierungsmöglichkeit

Grundsätzlich stellt die Abbildung der einem Attribut anhaftenden Bedingungen von CCM auf ihre Zope3 Pendants keine Schwierigkeit dar. Die logischen Ausdrücke in der Assetdefinitionssprache können relativ einfach in Python Ausdrücke übersetzt werden. Eine Ausnahme sind die mengenwertigen Attribute, weil es für die Zope3 `List` keine Möglichkeit gibt, auf Teil- und Obermengenbeziehungen zu überprüfen. Diese Funktionen müssen

zuvor manuell implementiert werden. Bedingungen auf Beziehungen sind allerdings nicht möglich, da Zope diese nicht als Feldtyp unterstützt.

Eine Schwierigkeit ergibt sich, wenn Bedingungen in abgeleiteten Assetdefinitionen verschärft werden, oder zusätzliche Bedingungen eingeführt werden. In diesem Fall muss sichergestellt werden, dass sowohl die Bedingungen der Elternklasse, wie auch die der abgeleiteten Klasse berücksichtigt werden. Werden Schemafelder neu definiert, müssen beide Bedingungen konjunktional verknüpft werden.

Eigenständige Bedingungen haben in Zope keine Entsprechung, so dass ihre Implementierung Probleme aufwirft. Es ist möglich Methodendeklarationen in ein Schema aufzunehmen, diese in der implementierenden Klasse zu realisieren und als als Bedingungen zu interpretieren. Allerdings müssten diese systematisch überprüft werden, entweder immer wenn ein Attribut verändert wird oder bevor eine Transaktion abgeschlossen wird. Da hierfür jedoch noch kein Mechanismus zur Verfügung steht, ist die Realisierung aufwändig.

Bedingungen werden in dieser Arbeit vernachlässigt, weil ihre Realisierung den Umfang dieser Arbeit übersteigt. Darüber hinaus sind sie zur Zeit nicht vollständig im Modellkompiler implementiert.

Kapitel 4

Realisierung von Generatoren für Zope3 Inhaltskomponenten

Im vorigen Kapitel wurde diskutiert, wie Assetmodelle auf Zope3 Inhaltskomponenten abgebildet werden können. Darauf aufbauend soll in diesem Kapitel erläutert werden, wie diese Abbildung mithilfe des CCM Modellkompilers realisiert werden kann. Dazu werden zunächst Anforderungen an das zu generierende System analysiert, die bislang nicht berücksichtigt wurden. Anschließend wird ein Satz von Generatoren und ein Hilfsmittel zur Generierung von Python Kode entworfen.

4.1 Anforderungen an das zu generierende System

4.1.1 Bestandteile einer Inhaltskomponente

In Kapitel 3 wurde diskutiert, wie das Assetmodell von CCM im Rahmen dieser Arbeit auf die Schemata der Zope3 Inhaltskomponenten abgebildet wird. Die Schemata sind der für diese Aufgabe bedeutendste Bestandteil einer Inhaltskomponente. Damit die generierten Systeme lauffähig sind, müssen alle Bestandteile einer Inhaltskomponente generiert werden. Die fehlenden Bestandteile werden in diesem Unterkapitel erläutert.

Zur Übersicht ist das Paketdiagramm Abbildung 4.1 gegeben. Es soll eine einfache Inhaltskomponente `Stamp` realisiert werden. Kodebeispiele finden sich in den folgenden Erläuterungen. Das Schlüsselwort [1] `<<ZCML>>` wird hier verwendet, um eine Konfigurationsdatei in der Zope3 eigenen Konfigurationssprache *Zope Configuration Markup Language* zu kennzeichen.

[1] Die Bedeutung der Begriffe *Schlüsselwort* und *Stereotyp* hat sich mit Einführung der UML2 geändert, siehe [Fow04, Lar05].

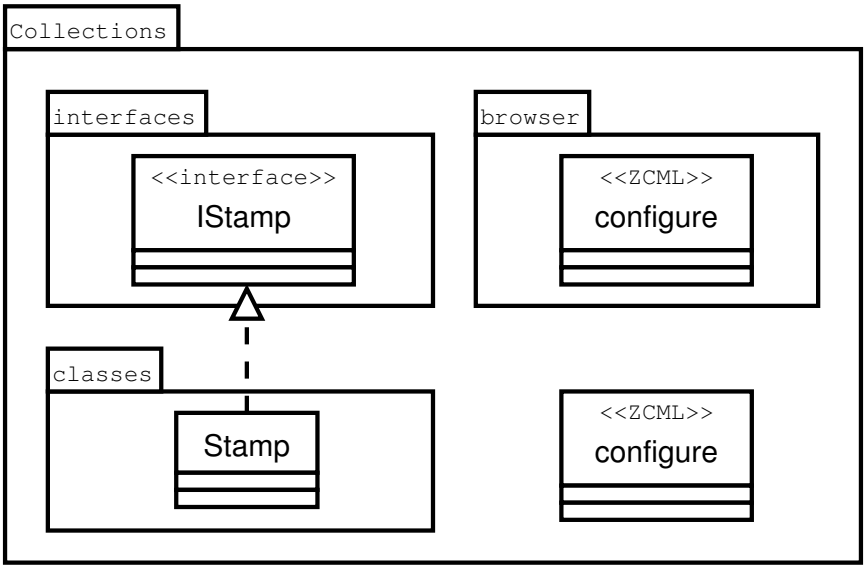

Abbildung 4.1: Paketdiagramm: Bestandteile einer Inhaltskomponente

Implementierung des Schemas

Dass die Implementierung des Schemas bei Zope trivial ist, folgt aus Besonderheiten der Programmiersprache Python: Zum einen gibt es in Python keine Möglichkeiten der Datenkapselung: Private Attribute werden durch Namenskonventionen kenntlich gemacht. Darüber hinaus werden Instanzattribute von Objekten nicht deklariert. Man erzeugt ein Attribut durch die Zuweisung eines Wertes an einen Bezeichner. Dies geschieht auch in Zope, so dass eine leere Klasse als Implementierung eines Schemas ausreicht. Es ist allerdings Konvention, einen Konstruktor zu implementieren. Dieser ist nur dann relevant, wenn man eine neue Instanz eines Inhaltskomponententyps durch eigenen Programmkode erzeugt. Für die Erzeugung neuer Instanzen über die Präsentationslogik von Zope ist er ohne Bedeutung. Durch das Erben von der Basisklasse `persistent.Persistent` werden Instanzen dieser Klasse in der Zope eigenen Datenbank gespeichert. Ein Schema `IStamp` mit einem Attribut `value` könnte wie folgt implementiert werden:

```
class Stamp (persistent.Persistent):
    implements(IStamp)
    __init__(self, value=0.55):
        self.value = value
```

Präsentationskomponente

Um eine Inhaltskomponente als HTML Seite darstellen zu können, wird eine zugehörige Präsentationskomponente benötigt. Diese kann durch eine HTML Vorlage (*Template*) gegeben sein, oder es kann aus dem Schema eine Darstellung generiert werden. In beiden Fällen ist es nötig, das Zope System zur Verwendung der entsprechenden Inhaltskomponente zu konfigurieren. In dem folgenden Kodebeispiel wird das System konfiguriert, eine automatisch generierte Präsentationskomponente zum Editieren zon `IStamp` Instanzen zu verwenden. Der Konfigurationskode zum Erzeugen und Darstellen von Inhaltskomponenten ist ähnlich und wird hier ausgelassen.

```
<configure xmlns="http://namespaces.zope.org/browser">
    <editform schema=".interfaces.IStamp"
              label="change Stamp information"
              name="edit.html"
              menu="zmi_views" title="Edit"
              permission="EditStamp" />
    <!-- ... -->
</configure>
```

Testkode für die Inhaltskomponente

Es ist üblich, für alle Bestandteile einer Zope Anwendung Testfälle zu erzeugen. Obwohl man bei bei einem sorgfältig implementiertem Kodegenerator davon ausgehen kann, dass dieser fehlerfreien Kode erzeugt, werden grundlegende Testfälle generiert. [2] Dies hat zwei Gründe: Zum einen werden die Tests benötigt, wenn der erzeugte Kode manuell verändert wird. Zum anderen unterscheidet sich der Ablauf der Kodegenerierung gegenüber dem Ablauf bei den bisher implementierten Generatoren für den CCM Kompiler. Durch diese Unterschiede, welche in Abschnitt 4.2.1 erläutert werden, ist eine zusätzliche Überprüfung des erzeugten Kodes wünschenswert.

Als Beispiel dient der folgende Kode, mit dem überprüft wird, ob die Instanzen der Klasse `Stamp` das Schema `IStamp` korrekt implementieren.

```
class InterfaceTest(unittest.TestCase):
    def test_Stamp(self):
        self.assert_( zope.interface.verify.verifyObject(
            IStamp,Stamp() ) )
```

Konfigurationsdateien

Eine Inhaltskomponentenklasse muss neben ihrer Spezifizierung durch ein Schema und ihrer Implementierung durch eine Klasse noch beim System

[2] Es ist sogar möglich, die syntaktische Korrektheit des generierten Kodes durch statische Analyse des Generators zu gewährleisten, siehe [HZS05].

angemeldet werden. Dieses wird durch die Konfigurierung in ZCML (*Zope Configuration Markup Language*), einer XML basierten Sprache, erreicht.

Die Anmeldung einer Inhaltskomponente beim System geschieht in zwei Schritten. Zuerst deklariert man, dass es sich bei Komponenten der Schnittstelle `IStamp` um Inhaltskomponenten handelt. Diese Klassifizierung von Schnittstellen geschieht über die ZCML Direktive `interface`. Anschließend wird die Klasse, welche die Komponentenschnittstelle implementiert, durch die `content` Direktive beim System registriert. In diesem Zusammenhang werden eine Fabrik (*factory*), in diesem Fall die Klasse `Stamp` selbst, und deren Rechte (*permissions*) angegeben, die zum Zugriff auf diese Komponente benötigt werden. Eine besondere Bedeutung hat die Direktive `implements`. Sie ermöglicht, dass Instanzen der Klasse `Stamp` die Schnittstelle `IAttributeAnnotatable` implementieren. Dadurch wird erreicht, dass die Komponente in der Lage ist bestimmte Metadaten zu speichern, wie z. B. das Datum ihrer Instanziierung.

```
<configure xmlns="http://namespaces.zope.org/zope">
    <interface interface=".interfaces.IStamp"
               type="zope...IContentType" />
    <content class=".classes.Stamp">
        <implements interface="zope...IAttributeAnnotatable" />
        <factory id=".classes.Stamp"
                 description="Stamp" />
        <require permission="application.Stamp.View"
                 interface=".interfaces.IStamp" />
        <!-- ... -->
    </content>
</configure>
```

4.1.2 Zope Sicherheitskonzept

Zope überwacht alle Zugriffe auf die gespeicherten Inhalte. Ein Anwender kann nur dann eine Operation ausführen, wenn er über das Recht (*permission*) dazu verfügt. Zope verfügt über eine integrierte Benutzer- und Rechteverwaltung, die es erlaubt, Benutzern (*principals*) Rechte zu gewähren. Wie den Benutzern Rechte gewährt werden, ist abhängig von der Sicherheitsstrategie (*security policy*), die von dem Administrator eines Zope Systems frei gewählt und angepasst werden kann.

Um eine Inhaltskomponente verwenden zu können, muss festgelegt werden, welches Recht für den Zugriff auf eine Inhaltskomponente (z. B. Betrachten, Verändern) nötig ist. Es gibt zwei Möglichkeiten, die aus der Assetdefinition generierten Inhaltskomponenten - im folgenden auch Assets genannt - in das Zope3 Sicherheitskonzept zu integrieren.

Da der Fokus dieser Arbeit auf der Abbildung von CCM Assets auf Zope3 Inhaltskomponenten liegt, ist es angemessen, das Sicherheitskonzept außer Acht zu lassen. Das ist möglich, indem man vordefinierte Rechte verwendet wie die folgenden: `zope.Public` ist ein Zugriffsrecht, über das jeder, auch anonyme, Nutzer verfügt. Im Gegensatz dazu wird das Recht `zope.ManageContent` nur Zope Administratoren gewährt. Diese Rechte können so verwendet werden, dass es jedem Benutzer erlaubt ist, Assets zu betrachten, aber nur der Administrator darf sie erzeugen und verändern.

Für den Produktiveinsatz sind die so generierten Systeme nicht geeignet. Um den Zugriff auf eigene Komponenten zu reglementieren, wird es in der Zope Dokumentation empfohlen, lediglich eigene Rechte zu definieren. Es sei Aufgabe des Administrators die Rechte den Benutzern zuzuordnen. Nach dieser Empfehlung sollten für jedes Asset folgende Rechte definiert werden:

- `model.Asset.View` zum Betrachten von Assets,
- `model.Asset.Edit` zum Verändern von Assets,
- `model.Asset.Add` zum Erzeugen von Assets,
- `model.Asset.Delete` zum Löschen von Assets.

Damit die erzeugten Inhaltskomponenten lauffähig sind, müssen die Rechte (z. B. durch die Verwendung von Benutzerrollen, *roles*) den Benutzern zugeordnet werden. Damit ein solches System unmittelbar nach seiner Generierung lauffähig ist, kann eine separate Konfigurationsdatei erzeugt werden, in der sämtliche Rechte dem Administrator gewährt werden. Durch Anpassung dieser Datei kann der Administrator die erzeugten Inhaltskomponenten in sein gewähltes Sicherheitskonzept integrieren. Das zuletzt geschilderte Vorgehen ist vorzuziehen, da die erzeugten Inhaltskomponenten ohne Modifikation in einem Produktivsystem verwendet werden können.

4.1.3 Zope Ordnerstruktur

Beschreibung

Die Stärke der in Zope integrierten objektorientierten Datenbank (ZODB) ist es, dass sie beliebige Objektgraphen ohne zusätzlichen Implementierungsaufwand persistent halten kann. Eine Nachteil ist, dass sie über keine Suchfunktion oder eine Anfragesprache verfügt (wie z. B. OQL, SQL). Dadurch ergibt sich das Problem, wie bestimmte Objekte in der Datenbank aufgefunden werden können.

Gelöst wurde dieses Problem durch die Verwendung von Ordnern, die stark an ihre Pendants in einem Dateisystem erinnern. Ein Ordner ist eine Container-Datenstruktur, die es erlaubt, in ihr enthaltene Elemente unter Angabe ihres Namens aufzufinden. Ein Ordner ist mit einer

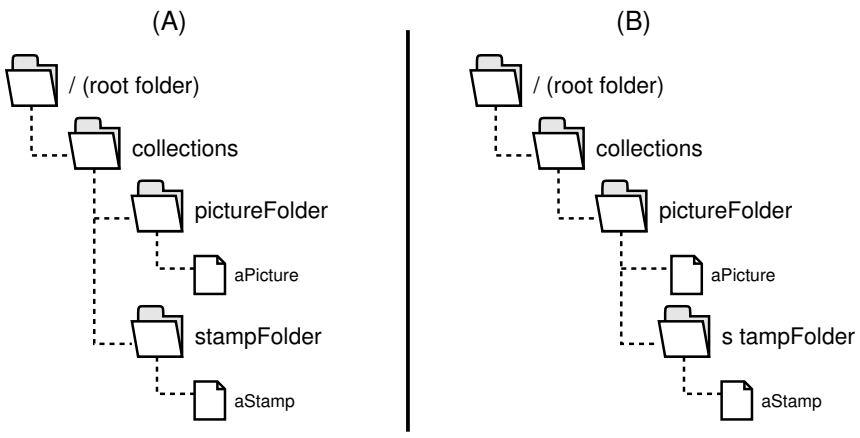

Abbildung 4.2: Alternative Ordnerstrukturen

`Map<String,Object>` in Java vergleichbar, wenn diese persistent gehalten wird. Der Zugriff auf ein Objekt in Zope erfolgt über eine URL. Deren Pfad ergibt sich aus dem Namen des Objektes und den Namen der Ordner, in denen es enthalten ist. In der Abbildung 4.2 (A) kann man das Objekt `aPicture` über die URL `/collections/pictureFolder/aPicture` ansprechen. Gibt man diese URL in einem Browser ein, erhält man eine Standardansicht (*default view*) des Objektes.

Entwicklung einer Ordnerstruktur für Assets

In CCM gibt es das Konzept von Ordnern nicht. Auch das Konzept von benannten Assets in der Assetmanipulationssprache ist nicht mit den benannten Objekten in den Zope Ordnern vergleichbar. Die Benennung der Assets in den Ordnern stellt kein Problem dar, weil Zope selbstständig eindeutige Namen für ein Objekt verwendet, sofern keiner explizit angegeben wird. Daher stellt sich die Frage, wie die Assets auf Zope Ordner verteilt werden können.

Vernachlässigt man bei der Implementierung von Inhaltskomponenten die Ordner, kann man Instanzen in jedem Ordner anlegen. Die Folge für Assets wäre, dass sie über alle Ordner des Systems verstreut werden könnten und ein Auffinden bestimmter Assets schwierig würde. Daher ist es sinnvoller, für jede Assetklasse einen speziellen Ordner anzulegen, der sämtliche Assets einer Klasse enthält. Es gibt zwei naheliegende Alternativen, die Assets nach ihrer Klassenzugehörigkeit auf Ordner zu verteilen.

Eine einfache Möglichkeit für eine Ordnerstruktur wird in Abbildung 4.2 (A) veranschaulicht. Innerhalb eines Ordners, der das gesamte Assetmodell enthält, existiert für jede Klasse ein eigener Ordner. Allerdings ist in dieser Struktur die Vererbungshierarchie nicht mehr offensichtlich. Im Beispiel

sei `Stamp` eine Unterklasse von `Picture`. Soll unter allen Assets der Klasse `Picture` einschließlich ihrer Unterklassen gesucht werden, müssen auch die Ordner der Unterklassen berücksichtigt werden.

Bei der Variante in Abbildung 4.2 (B) tritt dieses Problem nicht mehr auf. Es können alle Assets des Typs `Picture` durch eine Suche in `pictureFolder` und in den enthaltenen Unterordnern gefunden werden. Bei dieser Variante ist die Realisierung komplizierter, da sichergestellt werden muss, dass für jede Assetklasse genau ein Ordner existiert. Über die Administrationsschnittstelle von Zope ist es möglich, mehrere Ordner für eine Assetklasse zu erzeugen. Um die Eindeutigkeit der Ordnerstruktur zu wahren, muss dies verhindert werden. Die Lösung dieses Problems wird hier nicht weiter diskutiert, dazu sei auf die Literatur zu Zope verwiesen. ([Ful00]) Diese tief geschachtelte Ordnerstruktur soll realisiert werden, weil der Erhalt der Vererbungshierarchie Vorteile beim Auffinden von Assets bietet.

Der Ausgangspunkt in den vorangegangenen Überlegungen war eine fiktive Suche nach Assets. Zope3 verfügt bisher über keine Möglichkeit, nach Objekten in der Datenbank zu suchen. In den Vorgängern Zope 2.X ist eine Suche durch die Komponente *ZCatalog* möglich. Es ist zu erwarten, dass diese Funktion bald auf Zope3 portiert wird.

Realisierung der Ordnerstruktur

Nachdem die Entscheidung für eine bestimmte Ordnerstruktur getroffen wurde, soll kurz erläutert werden, wie die benötigten Ordner in Zope realisiert werden können. Ordner selbst sind auch Komponenten, die den bisher erläuterten Inhaltskomponenten ähnlich sind. Ordner, die in diesem Fall nur Assets einer einzigen Klasse aufnehmen können, werden durch die gleichen Schnittstellen, Klassen und Konfigurationsdateien wie eine Inhaltskomponente realisiert. Die Zusammenhänge zwischen den Schnittstellen und Klassen einer Inhaltskomponente der Klasse `Stamp` und ihrer Containerkomponente sind in Abbildung 4.3 dargestellt.

Zwei Randbedingungen müssen immer für die Inhalts- und die Containerkomponente gelten: Erstens müssen *alle* `Stamp`s in dem `StampContainer` enthalten sein und zweitens dürfen in dem Container *ausschliesslich* `IStamp` Komponenten enthalten sein. Ersteres wird durch die Bedingung `{contains}` zwischen den Schnittstellen der Containerkomponente und der Inhaltskomponente festgelegt. Der Container wird nun keine Inhaltskomponenten anderer Typen aufnehmen. Durch die Bedingung `containers` wird sichergestellt, dass Klassen, welche die Schnittstelle `IStampContained` implementieren, sich ausschließlich in dem angegebenen Container instanziieren lassen. Diese Bedingung kann man theoretisch auch an die Schnittstellen `IStamp` und `IStampContainer` stellen, so dass sich alle `IStamp` Instanzen ausschließlich in dem `IStampContainer` ablegen lassen. Zyklische Abhängigkeiten zwischen Klassen sind jedoch in Python nicht möglich. Man behilft sich mit der

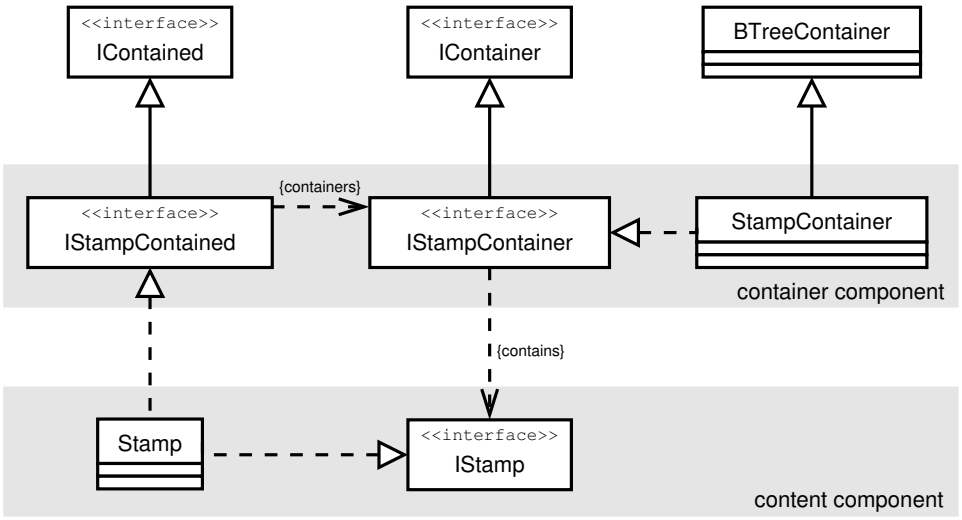

Abbildung 4.3: Beispiel: Realisierung eines Ordners

Einführung einer Schnittstelle, die bis auf die Bedingung leer ist, um die zyklische Abhängigkeit aufzulösen. Für Details sei auf die Zope Dokumentation verwiesen. ([Wei05])

4.2 Entwurf von Generatoren für den CCM Kompiler

4.2.1 Phasen der Kodegenerierung

Die meisten Generatoren für den CCM Kompiler erzeugen Programme in Form von Java Quellkode. Bevor so ein Programm ausgeführt werden kann, muss es zu Java Bytekode kompiliert und in das Zielsystem (z. B. InfoAsset Broker) integriert werden - dies ist in Abbildung 4.4 (A) dargestellt. Neben der Erzeugung des Bytekodes ist es Aufgabe des Java Kompilers, den Quellkode auf syntaktische und semantische Fehler zu analysieren. (siehe Kompilerbau Literatur z. B. [ASU86]) Generell kann man davon ausgehen, dass ein gewissenhaft realisierter Kodegenerator fehlerfreie Programme erzeugt. Trotzdem ist es sinnvoll, den erzeugten Kode auf Fehler zu überprüfen, da bestimmte Fehlerquellen nicht grundsätzlich ausgeschlossen werden können.

- Durch eine neue Version der Programmiersprache kann bislang korrekter Kode unbrauchbar werden. z. B. wurde mit Java 1.5 das Schlüsselwort `enum` eingeführt, welches zuvor als Bezeichner verwendet werden durfte.

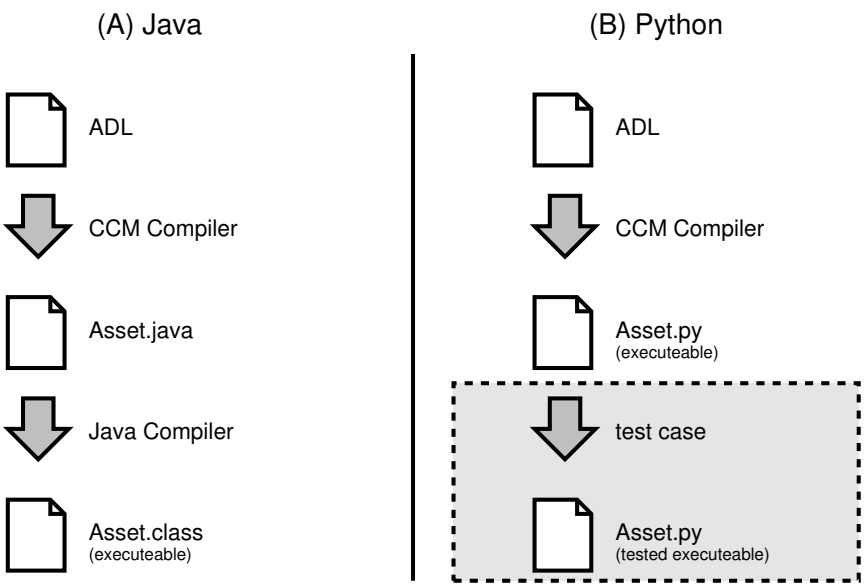

Abbildung 4.4: Phasen der Systemgenerierung

- Der CCM Kompiler befindet sich in ständiger Entwicklung. Durch Änderungen kann der generierte Kode unbrauchbar werden, z. B. wenn Leerzeichen in Bezeichnern zugelassen werden.

- Die Programmierschnittstellen (*API*) des Zielsystemes können bei neuen Programmerversionen verändert werden und dadurch Inkompatibilitäten erzeugen. Dies ist z. B. bei CoreMedia geschehen.

Darüber hinaus ist eine abschließende Überprüfung der Korrektheit des generierten Kodes bei der Entwicklung des Generators von großem Nutzen.

Die in dieser Arbeit zu entwickelnden Generatoren erzeugen Kode in Python, einer interpretierten Programmiersprache. Weil der Kode ohne eine Kompilierung lauffähig ist, wird der Kode nach seiner Generierung nicht mehr auf Fehler überprüft, siehe Abbildung 4.4 (B). Dies ist problematisch, da zwei der oben genannten Fehlerquellen hier besonders zum Tragen kommen: Zum einen wird die Programmiersprache Python wesentlich offensiver weiterentwickelt als z. B. Java oder C++. Mit jeder Version kommen neue syntaktische Konstrukte hinzu. Zum anderen ist auch Zope3 großen Änderungen unterworfen. Es ist ein neues System und wird von seinen Entwicklern noch als experimentell eingestuft. Inkompatibilitäten des generierten Kodes zu neuen Versionen von Zope und Python sind die Folge. Darüber hinaus kann die Realisierung der offenen Dynamik [Seh04], die herrausragenden Eigenschaft der CCM Systeme, die Systemstabilität gefährden.

Statt durch den Aufruf eines Kompilers kann die Korrektheit des erzeugten Kodes auch durch Tests sichergestellt werden. Zum Testen können bestimmte Funktionen von Zope3 genutzt werden, welche die Automatisierung von Komponententests unterstützen. Diese kann man nutzen, indem man Testfunktionen für die Inhaltskomponenten generiert. Viele Fehler (z. B. syntaktische Fehler, fehlende Bibliotheken) werden von Python bereits beim Laden eines Programmes gefunden. Daher ist es ausreichend, dass die Testfunktion nur eine einfache Aufgabe ausführt, z. B. das Erzeugen einer Assetinstanz. Tritt dabei keine Ausnahme (*exception*) auf, kann der Kode als syntaktisch korrekt angesehen werden.

4.2.2 Entwurf der Grobstruktur

Wie in den Abschnitten 2.1.3 und 4.1 erläutert wurde, sind zur Generierung einer Inhaltskomponente aus einer Assetdefinition folgende Aktivitäten notwendig:

- Generierung eines Schemas,
- Generierung einer Implementierung,
- Generierung einer Testroutine,
- Generierung eines Containers,
- Konfigurierung der Präsentation,
- Konfigurierung der Zugriffsrechte,
- Konfigurierung der Inhaltskomponente.

Die Generierung einer Inhaltskomponente ist ein komplexer Vorgang, der einer Unterteilung bedarf, um ihn strukturiert und übersichtlich zu implementieren. Als Basis für die Unterteilung bieten sich die oben genannten Aktivitäten an, die durch Anwendung des Entwurfmusters *Befehl* in einem objektorientierten System realisiert werden können. (siehe [GHJV94]) Der CCM Kompiler ist als Framework realisiert, welches durch sogenannte *Generatoren* ergänzt wird. (siehe Abschnitt 2.2.4) Da ein Generator eine erweiterte Umsetzung des Musters *Befehl* ist, liegt es nahe, jede Aktivität als eigenen Generator zu implementieren. Diese Lösung ist einer eigenen Implementierung des Befehlsmusters aus Gründen der Transparenz und Wartbarkeit vorzuziehen, da sie jedem, der an dem CCM Projekt gearbeitet hat, unmittelbar verständlich ist.

Die einzelnen Generatoren kommunizieren über Symboltabellen miteinander: Jeder Generator erzeugt eine Symboltabelle und kann von beliebig vielen anderen abhängen. Aus der Abhängigkeit der Generatoren von den Symboltabellen kann auf die Abhängigkeiten zwischen den Generatoren und

auf eine mögliche Ausführungsreihenfolge geschlossen werden. (siehe Abschnitt 2.2.4 und [Seh04]) In Tabelle 4.2.2 sind die Generatoren mit den Abhängigkeiten von den Symboltabellen in einer möglichen Ausführungsreihenfolge dargestellt. Die in den Symboltabellen gespeicherte Information ist sehr unterschiedlich: In der `SchemaSymbolTable` wird jeder Assetklasse ein Zope Schema zugeordnet. Dagegen enthält die `ViewSymbolTable` nur den Namen des Unterpaketes mit den Präsentationsskomponenten, das in der Komponenenkonfiguration vom `ConfigurationGenerator` eingebunden werden muss. Ein Klassendiagramm der Symboltabellen findet sich in Anhang B.

Die Unterteilung der Kodegenerierung in die genannten Aktivitäten erscheint sehr feinstrukturiert, ihr liegen neben dem bisher genannten noch zwei weitere Überlegungen zugrunde: Zum einen ist es wünschenswert, einzelne Generatoren austauschen zu können, um Systeme für andere Anwendungsbereiche generieren zu können. Man kann zum Beispiel den `ViewGenerator` durch einen anderen ersetzen, der die Assets als *Webservices* über Protokolle wie XML-RPC oder SOAP zugänglich macht. Wenn jeder Generator nur eine einfache Aufgabe übernimmt, wird ihre Wiederverwendbarkeit in anderen Anwendungsbereichen verbessert. Zum anderen ist es bei der Implementierung der Generatoren von Vorteil, wenn jede Kode- und Konfigurationsdatei von nur einem Generator erzeugt wird. Würden zwei Generatoren in eine Konfigurationsdatei schreiben, dann müsste man dafür sorgen, dass bereits vorhandene Konfigurationen nicht überschrieben werden.

4.2.3 Python Kodeerzeugung

Ansatz

Die Erzeugung von Python Kode kann nach dem gleichen Schema funktionieren wie das *Java Code Generation Toolkit (JCGTk)*, das von den Generatoren des CCM Projekts für die Erzeugung von Java Kode eingesetzt wird. Die Funktionsweise dieser Bibliothek wird deshalb kurz erläutert, Details finden sich in der Dokumentation zum JCGTk in [JCGTk].

Das JCGTk basiert auf der Idee den Vorgang der Kompilierung umzukehren. Ein Kompiler analysiert den Quellkode und erzeugt daraus eine Repräsentation des Kodes im Hauptspeicher, den Syntaxbaum, aus dem später Maschienenkode erzeugt wird. Details zum Vorgang der Kompilierung finden sich in der Literatur zum Kompilerbau, z. B. in [ASU86].

Zur Erzeugung von Kode wird zunächst eine dem Syntaxbaum vergleichbare Datenstruktur im Hauptspeicher aufgebaut, aus der anschließend z. B. durch Verwendung eines Besuchers (*Visitor*, [GHJV94]) Quellkode erzeugt wird. Zum Aufbau der Datenstruktur wurde ein Metamodell der Programmiersprache Java implementiert, in dem Klassen für alle Sprachkonzepte

	SchemaSymbolTable	PermissionSymbolTable	ContainerSymbolTable	ImplementationSymbolTable	ViewSymbolTable	ConfigurationSymbolTable	TestSymbolTable
SchemaGenerator	E						
PermissionGenerator	A	E					
ContainerGenerator	A	A	E				
ImplementationGenerator	A		A	E			
ViewGenerator	A	A	A		E		
ConfigurationGenerator	A	A	A	A	A	E	
TestGenerator	A			A			E

Legende

E Generator erzeugt die Symboltabelle

A Generator hängt von der Symboltabelle ab

Tabelle 4.1: Generatoren und Symboltabellen

vorhanden sind, z. B. für die Konzepte Klasse, Typ, Variable, Variablenreferenz, Anweisung usw.

Um dieses Prinzip für die Erzeugung von Python Kode nutzen zu können, muss ein Metamodell für die Programmiersprache entworfen und implementiert werden. Der Entwurf des Metamodelles erfolgt unter zwei Gesichtspunkten: Aus formaler Sicht ist es erstrebenswert ein möglichst korrektes und vollständiges Modell der Programmiersprache zu verwenden. Aus einem pragmatischen Gesichtspunkt dient ein Werkzeug zur Kodeerzeugung dazu, syntaktisch und semantisch korrekten Kode zu erzeugen. Eine zu hohe Komplexität des Metamodells stellt eine Fehlerquelle dar und ist zu vermeiden. Es werden im folgenden die Sprachkonstrukte der Programmiersprache Python modelliert, die zum Generieren von Inhaltskomponenten benötigt werden.

Spracheigenschaften

Um ein zur Kodegenerierung geeignetes Metamodell zu erstellen, müssen die Besonderheiten der Programmiersprache Python beachtet werden. Es werden im folgendem herrausragende Unterschiede zwischen Java und Python beschrieben, da sie Hinweise dazu geben, wie das Konzept des Java Code Generation Toolkit für Python umgesetzt und angepasst werden kann. Für eine detailiertere Darstellung der Sprache Python sei auf die Dokumentation, vor allem die Sprachreferenz verwiesen. [Ros05]

Der auffälligste Unterschied zwischen Python und Java betrifft die lexikalische Struktur der Programme: Die Einrückung des Quellkodes hat in Python im Gegensatz zu Java eine Bedeutung. Ein gleichmäßig eingerückter Block von Python Kommandos hat die gleiche Bedeutung, wie eine durch geschweifte Klammern begrenzte Verbundanweisung (*compound statement*) in Java oder C.

```
if false:
    print "cats & "  # won't be executed, part of if-statement
print "dogs"         # will be executed, statement of it's own
```

Syntaktisch unterscheiden sich die Sprachen vor allem in ihren Ausdrücken (*expressions*), die in Python komplexer sind als in Java. Der Grund dafür liegt in der Vielfalt von mächtigen Typen, die fest in die Sprache integriert sind wie z. B. mehrere Arten von Listen, Mengen und Hashtabellen. Für jeden dieser Typen sind spezielle syntaktische Konstrukte in der Sprache vorhanden, um mit ihnen zu arbeiten, z. B. Listenschnitte (*slices*). Dabei handelt es sich um einen speziellen Operator mit dem aus Listen Abschnitte extrahiert werden können. Ein weiteres Beispiel für komplexe Ausdrücke in Python sind die sogenannten *list comprehensions* die ihren Ursprung in funktionalen Programmiersprachen wie z. B. *Haskell* haben. (siehe [Tho99])

Das folgende Beispiel verwendet sie, um eine 10x2 Matrix zu erzeugen, die Argumente und Funktionswerte der Exponentialfunktion enthält.

```
fibonacci = [1, 1, 2, 3, 5]   # list
counting  = fibonacci[1:-1]   # remove first & last element
# counting == [1, 2, 3]

expTable = [ [x, exp(x)] for x in range(10) ]
# expTable == [ [0, 1.0], [1, 2.718...], ... ]
```

Auf semantischer Ebene unterscheidet sich Python von Java dadurch, dass es die dynamischere Programmiersprache ist: Da es sich um eine interpretierte Sprache handelt, finden viele Vorgänge, die in Java der Kompiler durchführt, erst zur Laufzeit statt. Die Dynamik der Sprache äußert sich zum einen in ihrer *schwachen* und *dynamischen Typisierung*, zum anderen darin, dass es kaum einen Bestandteil eines Programmes gibt, der nicht zur Laufzeit verändert werden kann. Dies wird im folgenden weiter erläutert.

Die Dynamik der Sprache wird vor allem dadurch zu einer Schwierigkeit für die Kodeerzeugung, dass trotz objektorientierter Ansätze die Attribute einer Klasse nicht deklariert werden und es keine Möglichkeit zur Datenkapselung gibt. Private Attribute werden in Python lediglich durch eine Namenskonvention gekennzeichnet. Es kann durch Analyse der Klasse eines Objektes nicht bestimmt werden, über welche Attribute das Objekt zur Laufzeit verfügt. Die Dynamik der Sprache äußert sich auch in einem weiteren Unterschied zu Java: Klassen- und Methodendefinitionen sind Anweisungen (*statements*) und werden zur Laufzeit ausgeführt. Diese Aspekte der Sprache werden in dem folgenden Beispiel veranschaulicht.

```
class Dummy(object):
    pass                # empty class statement
d = Dummy()             # d is an instance of class Dummmy
d.name, d.age = "Dilbert", 42  # multiple assignment
# => d.name=="Dilbert", d.age==42

location = "wild west"
if location == "wild west":
    def greet():        # function statement
        print "Howdy"
else:
    def greet():        # another function statement
        print "Hi"
greet()  # => "Howdy"
```

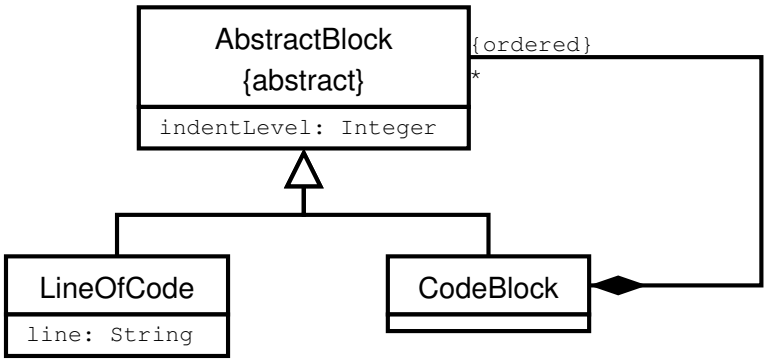

Abbildung 4.5: Python Kodeerzeugung: Kodeblöcke & Kodezeilen

Modellierungsalternative: Kodeblöcke & Kodezeilen

Die einfachste Möglichkeit, der Kodeerzeugung besteht darin, ihn mit Hilfe der Dateiausgabe- und Zeichenkettenoperationen in eine Datei zu schreiben. Dabei ist es problematisch die konsistente Einrückung zu gewährleisten, die für Python Programme zwingend erforderlich ist.

Dies kann gelöst werden, indem man die Struktur der Einrückungen auf ein *Kompositum* ([GHJV94]) abbildet. In Abbildung 4.5 ist dieser Entwurf in einem Klassendiagramm dargestellt. Die zentrale Abstraktion ist die des Kodeblockes (`CodeBlock`), der eine Komposition aus Kodezeilen (`LineOfCode`) und untergeordneten Kodeblöcken ist. Dabei ist zu beachten ist, dass die enthaltenen Kodeblöcke eine Ebene tiefer eingerückt sind als der umgebene Block (Variable `indentLevel`).

Dieser Ansatz wurden in einem Prototyp implementiert, mit dem problemlos korrekt eingerückter Kode erzeugt werden konnte. Dieses einfache Modell ist noch kein Metamodell der Programmiersprache Python und für die den Einsatz zur Kodeerzeugung unzureichend, aber es zeigt, dass ein Kodeblock eine sinnvolle Abstraktion zur Erzeugung von Python Kode ist.

Modellierungsalternative: Pakete & Anweisungen

Ein einfaches Metamodell von Python findet sich in Abbildung 4.6. Es beschränkt sich auf die Modellierung von Paketen, darin enthaltenen Dateien und Anweisungen. Das Konzept des Kodeblockes wurde in abgewandelter Form beibehalten. Das Modell ist bei weitem nicht vollständig, es fehlen viele Anweisungen der Sprache, die bei Bedarf ergänzt werden müssen.

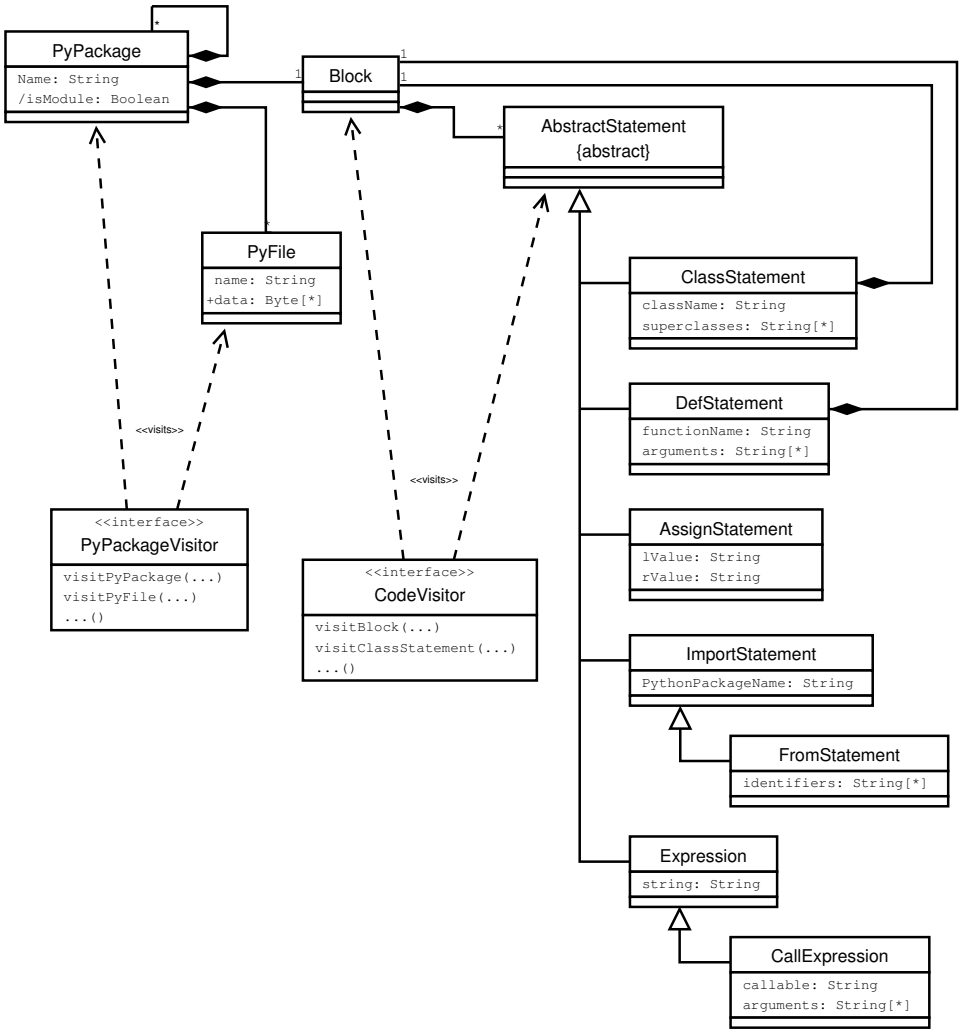

Abbildung 4.6: Python Kodeerzeugung: Pakete & Anweisungen

Der Verzicht auf eine genauere Modellierung von Ausdrücken (*expressions*) ist aber beabsichtigt: Ausdrücke werden hier durch Zeichenketten repräsentiert, da eine exakte Abbildung zu einem wesentlich komplexeren Modell führen würde. Eine Erläuterung findet sich im folgenden Unterabschnitt „Entscheidung“.

Trotz der Unvollständigkeit beinhaltet das Modell sämtliche Konstrukte, die zur Erzeugung von Zope Inhaltskomponenten notwendig sind. Die dargestellten Konzepte der Sprache Python werden im folgenden kurz erläutert.

Paket, Modul werden durch die Klasse `PyPackage` modelliert. Python Pakete sind mit denen von Java vergleichbar. Ein Modul ist ein Spezialfall eines Paketes, es kann keine Resourcen oder untergeordnete Pakete enthalten. (siehe [Mar03])

Resourcen sind gewöhnliche Dateien, die selbst keinen Quellkode enthalten aber in Python-Paketen enthalten sind. Um Verwechslung mit anderen Java Standardklassen zu vermeiden wurde der Name `PyFile` gewählt. Die Klasse kann dazu verwendet werden, den erzeugten Paketen die Zope3 Konfigurationsdateien hinzuzufügen.

Kodeblock Kodeblöcke wurden bereits erläutert, hier sind sie allerdings ein Kompositum von Anweisungen, sie enthalten keine untergeordneten Kodeblöcke. (`Block`)

Anweisungen Python Kode besteht nur aus Anweisungen, Deklarationen oder Definitionen gibt es in der Sprache nicht. In Python werden die Aufgaben von Methodendefinitionen und Klassendeklarationen durch Anweisungen zur Laufzeit wahrgenommen (`DefStatement`, `ClassStatement`), ähnlich wie in anderen schwach typisierten objektorientierten Sprachen wie Lisp mit CLOS. (*Common Lisp object system*, siehe [Gra95])

Zuweisung Zuweisungen sind in Python Anweisungen, keine Ausdrücke. (`AssignStatement`)

Importierung Das Verwenden von Kode in anderen Paketen wird durch deren Importierung ermöglicht. Es gibt dafür zwei Anweisugen (`ImportStatement`, `FromStatement`), deren Unterschied in der Python Literatur erläutert wird.

Ausdruck Wie in Java können Ausdrücke (`Expressions`) auch als Anweisungen verwendet werden. Ausdrücke sind hier durch Zeichenketten dargestellt, Unterklassen sind nur als Hilfsmittel für die Konstruktion komplexer Ausdrücke zu verstehen.

Nachdem aus den Klassen des Metamodells eine Repräsentation des zu gewünschten Kodes erzeugt wurde, muss diese noch in den eigentlichen

Quellkode transformiert werden. Dies kann durch Verwendung eines *Besuchers* [GHJV94] geschehen. In Abbildung 4.6 sind zwei Besucherschnittstellen gezeigt, die von den eigentlichen Besuchern implementiert werden müssen. Ein `CodeVisitor` dient der Generierung des eigentlichen Kodes, während ein `PyPackageVisitor` die Paketstruktur auf Ordner in dem Dateisystem abbildet.

Modellierungsalternative: vollständige Modellierung

Ähnlich wie im Java Code Generation Toolkit (JCGTk) geschehen, kann auch die Sprache Python so detailliert wie möglich modelliert werden. Ein Klassendiagramm dazu wird hier nicht gegeben, stattdessen wird erläutert, inwieweit das in Abbildung 4.6 dargestellte Modell erweitert werden müßte.

Um ein vollständiges Modell der Sprache zu erhalten, müssen - von den fehlenden Anweisungen abgesehen - vor allem die Ausdrücke genauer modelliert werden. Ausdrücke bestehen in Python wie in Java aus Operatoren, Literalen und Variablenreferenzen, die alle in das Modell aufgenommen werden müssen. Ausdrücke können wie im JCGTk als Kompositum modelliert werden, wobei das Modell komplex wird, da Python über mehr Operatoren als Java verfügt: arithmetische Operatoren, Klammern, Operatoren zum Zugriff auf Datenstrukturen, Methodenaufrufe und Spezialfälle wie den *list comprehension* Operator und den *lambda* Operator.

Um literale Werte und Konstruktoraufrufe in Ausdrücken zu ermöglichen, muss auch das Typsystem der Sprache modelliert werden. Python verfügt im wesentlichen über die gleichen primitiven Typen wie Java und darüber hinaus über Container und Klassentypen. Die Klassentypen stellen eine Besonderheit dar: Der Mechanismus zur mehrfachen Vererbung wurde zur Python Version 2 überarbeitet, der alte Mechanismus blieb aus Kompatibilitätsgründen in der Sprache erhalten. Seitdem existieren zwei semantisch unterschiedliche Arten von Klassen, die in der Literatur als *old-style classes* und *new-style classes* bezeichnet werden.

Der letzte fehlende Bestandteil zur vollständigen Modellierung von Ausdrücken sind Variablen, bzw. Variablenreferenzen. Durch die Dynamik von Python wie sie im vorigen Abschnitt erläutert wurde unterscheidet sich das Metamodell hier deutlich von Java: Das einzige, was von einer Variable zur Zeit der Kodeerzeugung bekannt ist, ist ihr Name. Es lässt sich weder der Typ des referenzierten Objektes feststellen, noch ob zur Laufzeit eine Variable dieses Namens überhaupt existiert. Das gleiche gilt für Klassen- und Methodennamen, denn obwohl eine „`class Widget:`“ Anweisung im Kode vorkommt, kann nicht mit Sicherheit davon ausgegangen werden, dass die Klasse zur Laufzeit unter diesem Namen ansprechbar ist. (siehe letztes Beispiel im vorigen Unterabschnitt „Spracheigenschaften“)

Entscheidung

Aus dem vorigen Abschnitt wird klar, dass die Verwendung eines vollständigen Metamodells zur Kodeerzeugung offene Fragen mit sich bringt: Erstens ist es unklar, inwieweit die Modellierung von Referenzen auf Klassen und Objekte zur Kodeerzeugung hilfreich ist. Zum Beispiel würde das Metamodell einer Klassenreferenz (nicht zu verwechseln mit der `class` Anweisung) im wesentlichen nur ihren Namen beinhalten. Zweitens ist das Metamodell der Ausdrücke sehr komplex, es ist jedoch fraglich, inwieweit die aufwändigere Modellierung der Programmiersprache Vorteile bringt. So kann zwar sichergestellt werden, dass nur syntaktisch korrekte Ausdrücke generiert werden können, die Ausführbarkeit des Kodes zur Laufzeit lässt sich so allerdings nicht garantieren, z. B. ist der Ausdruck „`[x*x for x in range(n)]`“ [3] syntaktisch korrekt, es kann jedoch im Allgemeinen nicht garantiert werden, dass die Variable `n` existiert und auf eine natürliche Zahl verweist.

Weil für die Realisierung der Generatoren lediglich ein Werkzeug zur Kodeerzeugung benötigt wird, ist das Modell, welches sich auf Pakete und Anweisungen beschränkt, am besten geeignet. Der Entwurf aus Abbildung 4.6 erscheint unvollständig, weil Ausdrücke nur als Zeichenketten modelliert werden, allerdings wird gerade diese unvollständige Modellierung den Besonderheiten der Sprache Python gerecht, deshalb soll eine auf diesem Metamodell basierende Bibliothek zur Kodeerzeugung im Rahmen dieser Arbeit erstellt werden. Es ist zwar möglich, mithilfe der Bibliothek fehlerhaften Kode, z. B. falsch geklammerte Ausdrücke, zu erzeugen, allerdings wird dieser Nachteil in Kauf genommen, weil diese Variante einfach zu realisieren ist. Die Lauffähigkeit des erzeugten Kodes kann durch ein detaillierteres Metamodell allein nicht gewährleistet werden.

In diesem Zusammenhang soll erwähnt werden, dass es für die Sprache Java Methoden gibt, die Korrektheit von Kodegeneratoren zu verifizieren. Dies wird durch die Metaprogrammiersprache *SafeGen* [HZS05] realisiert, die durch eine statische Überprüfung eines Generators sicherstellen kann, dass dieser ausschließlich wohlgeformten Java Kode erzeugt. Ob durch ähnliche Methoden die Lauffähigkeit von Pythonkode gewährleistet werden kann, bleibt zu klären. Ein Ansatz ist es, in der Metaprogrammiersprache Zusicherungen über die Existenz von Variablen und den Typ der durch sie referenzierten Objekte zu machen.

4.2.4 Konfigurierung von Zope

Wie alle Komponenten von Zope müssen auch die generierten Inhaltskomponenten mithilfe der XML Sprache *ZCML* konfiguriert werden. Die Details wurden in Abschnitt 4.1.1 erläutert. Hilfsmittel zum Erzeugen von XML sind für die Java Plattform zahlreich vorhanden. *DOM*, *SAX* und *JDOM*

[3] Dieser list comprehension Ausdruck erzeugt eine Liste der Quadratzahlen bis $(n-1)^2$.

sind nur die verbreitesten. Für die Konfigurierung von Zope ist DOM am besten geeignet: Zum einen sind die DOM Bilbliotheken im Gegensatz zu JDOM fester Bestandteil der Java Plattform. Zum anderen ist der Aufbau der XML Baumstruktur im Hauptspeicher bei DOM weniger fehleranfällig als der Einsatz der ereignisbasierten SAX Bibliothek. Auf die Erzeugung von XML durch DOM wird hier nicht weiter eingegangen, da Literatur zu dem Thema ausreichend vorhanden ist, z. B. [SG01, HM04].

Eine Schwierigkeit entsteht dadurch, dass es für ZCML Kode Richtlinien zur Formatierung und Einrückung der XML Elemente gibt. Es ist wichtig, dass die ZCML Dateien für Menschen lesbar bleiben, da es Zope Administratoren ermöglicht werden soll, durch Änderung der Konfigurationsdateien, Komponenten nach ihren Bedürfnissen anzupassen. Die für die ZCML Dateien geforderte Formatierung wird in keiner DOM Implementierung unterstützt. Es muss das DOM Dokument durch eigenen Kode serialisiert werden. Hilfreich sind die Funktionen der *JAXP (Java Api for XML Processing)*: Sie bieten die Möglichkeit, aus einem DOM Dokument eine Folge von SAX Ereignissen zu erzeugen. Diese Ereignisse werden von einer eigenen Implementierung der SAX-Schnittstelle `ContentHandler` gemäß den Richtlinien für ZCML in eine Datei geschrieben.

Kapitel 5

Bewertung und Ausblick

Diese Arbeit schließt mit einer Bewertung der praktischen Arbeit und Ausblicken, wie die Ansätze dieser Arbeit weiterentwickelt werden können. Dazu werden die generierten Zope Systeme bewertet, welche durch die in dieser Arbeit realisierten Generatoren erzeugt werden. Eines der Kernprobleme bei der Realisierung der Abbildung ist die Generierung von Python Kode. Daher soll geklärt werden, inwieweit sich der in Abschnitt 4.2.3 entwickelte Ansatz in der Praxis bewährt hat. Außerdem wird der Frage nachgegangen, inwieweit Zope eine geeignete Plattform für die konzeptorientierte Inhaltsverwaltung ist.

Ausgehend von den Überlegungen in Abschnitt 2.3, dass sich Zope auf verschiedene Weisen zur konzeptorientierten Inhaltsverwaltung nutzen lässt, gibt es zwei voneinander unabhängige Ausblicke. Im ersten Ausblick wird thematisiert, welche Möglichkeiten eigenständige Zope basierte CCM Systeme bieten. Kernpunkt der Überlegung ist, wie sich Offenheit und Dynamik, die Hauptansprüche der CCM Systeme, in einer reinen Zope Lösung realisieren lassen. Der zweite Ausblick beschäftigt sich mit dem Einsatz von Zope als Dienstschnittstellenmodul für Java basierte CCM Systeme und den Problemen, die in diesem Zusammenhang noch zu lösen sind.

5.1 Bewertung

5.1.1 Bewertung der generierten Systeme

Das Ziel dieser Arbeit, Zope Systeme mit typischer Funktionalität auf Grundlage eines CCM Assetmodells zu erzeugen, wurde erreicht. Grundlegende Operationen wie das Instanzieren, Modifizieren, Darstellen und Destruieren von Assets werden über die webbasierte Oberfläche von Zope unterstützt. Abstriche sind bei der Praxistauglichkeit der Systeme zu machen, vor allem was die Benutzerfreundlichkeit, den Umgang mit Inhaltsobjekten und die Stabilität angeht.

Abgesehen von dem trivialen Fall, dass Inhalte in Form von Zeichenketten vorliegen, ist es in Zope nur möglich, **Inhaltsobjekte** als Dateien in der integrierten Datenbank abzulegen. Die Dualität von Inhalt und Konzept, die Grundlage des Assetsbegriffs, geht damit verloren. Die Möglichkeiten, Zope um Fähigkeiten zum Umgang mit Inhalten zu erweitern, werden im Ausblick 5.2 erläutert.

Die **Benutzeroberfläche** der generierten Systeme wird über die Zope Administrationsoberfläche realisiert, die einen Zugriff auf alle in Zope vorhandenen Komponenten erlaubt. Für einen praktischen Einsatz der Systeme ist es empfehlenswert, die Menustruktur auf die Bedürfnisse der Anwender anzupassen. Die Zope basierten Inhaltsverwaltungssysteme Plone und Silva liefern Beispiele für angepasste Benutzeroberflächen. [Mck04, Silva]

Auch die **Stabilität** der generierten Systeme kann nicht immer gewährleistet werden, da das Problem von Namenskonflikten ignoriert wurde. Diese können auftreten, wenn z. B. Python Schlüsselwörter als Namen von Assetklassen verwendet werden. Um Produktivsysteme zu generieren, muss eine bijektive Abbildung zwischen den ADL und den Zope Bezeichnern vorgenommen werden, welche die erlaubte Zeichenmenge, Schlüsselwörter sowie dadurch entstehende Mehrdeutigkeiten berücksichtigt.

Schwierigkeiten bei der praktischen Arbeit hatten ihre Ursache zumeist in der Dokumentation von Zope. Die Bücher sind zwar umfangreich und verständlich, allerdings in ihrem Aufbau nicht immer zweckdienlich, weil sie größtenteils nur „Kochrezepte“ für eigene Anwendungen vorschlagen. Die Dokumentation im Quellkode des Systems liefert detailliertere Informationen, ist jedoch ohne fundiertes Grundwissen, das durch die Bücher nur schwer zu erlangen ist, kaum verständlich.

Weitere Probleme bei der Entwicklung mit Zope entstanden während der Implementierung des Feldtyps für Beziehungen. Es kam mehrmals vor, dass man durch das Framework andere Objekte erhielt, als zu erwarten war, z.B. Objekte eines anderen Typs oder durch Sicherheitsproxies geschützte Objekte. Ursache dieser Probleme war eine Kombination von ungeeigneter Dokumentation und der schwach typisierten Programmiersprache. Der Kompiler einer statisch typisierten Sprache hätte die Fehlersuche erleichtert.

Auch der Feldtyp `List` und seine Präsentationskomponenten offenbarten Schwächen. Eingabefehler in Listenelementen werden dem Anwender auf eine kryptische Weise angezeigt, die an einen Serverfehler erinnert. Im Zusammenhang mit dem Feldtyp für Beziehungen (siehe Abschnitt 3.3.2) offenbarte sich ein überraschendes Verhalten der Listen: Obwohl die Beziehungsfelder intern Datenbankreferenzen speichern, war dies nicht der Fall, als sie in Listen verwendet wurden. In diesem Fall wurden lediglich URLs als Zeichenketten gespeichert. Zu diesem Verhalten kommt es weil die unausgereifte Implementierung der `List`-Präsentationskomponenten Teile des Feld/Widget Framework umgeht.

5.1.2 Kodeerzeugung für Python

Eine Vorraussetzung für die Generierung von Zope3 Systemen war die Entwicklung eines Werkzeuges zur Kodeerzeugung für Python, wie es in Abschnitt 4.2.3 beschrieben wird. Die Erfahrungen, die mit dem Werkzeug gemacht wurden, sollen kurz aufgeführt werden, da sie sich eventuell auch auf andere schwach typisierte Programmiersprachen übertragen lassen.

Aus **praktischer Sicht** hat sich das PCGTk (*python code generation toolkit*) bewährt. Obwohl durch das stark vereinfachte Metamodell der Programmiersprache Fehler im generierten Kode zu erwarten waren, wie z. B. durch falsch geklammerte Ausdrücke, traten solche Probleme in der Praxis nicht auf. Die Kodeerzeugung gestaltete sich sehr übersichtlich, so dass Fehler gar nicht erst auftraten. Als Einschränkung muss erwähnt werden, dass der in dieser Arbeit erzeugte Kode nur von geringer Komplexität ist.

Aus **theoretischer Sicht** stellt sich aus den Erfahrungen dieser Arbeit die Frage, ob die Vereinheitlichung aller objektorientierten Programmiersprachen auf ein gemeinsames Metamodell sinnvoll ist. Diese Idee findet sich zum einen im JCKTk (*java code generation toolkit* [JCGTk]) wieder, zum anderen wird sie auch in der Forschung zur MDA (*model driven architecture*) im Projekt *Grasland* (Graphs for Software Language Definitions) an der Universität Twente verfolgt. [Trese] Leider wurden im Rahmen dieses junges Projektes noch keine Publikationen veröffentlicht. Zumindest darf bezweifelt werden, ob es für die Sprachen Java und Python ein sinnvolles gemeinsames Metamodell gibt. Als Begründung sei darauf hingewiesen, dass die Konzepte *Klasse* und *Instanz* in den beiden Sprachen unterschiedlich interpretiert werden. In Java kann durch die Kenntnis der Klasse eines Objektes auf Namen und Typ seiner Attribute und Methoden geschlossen werden. In Python können Instanzen Attribute und Methoden besitzen, die ihrer Klasse unbekannt sind. Die Bedeutung der Klasse hat in Python wenig mit einem Typ gemeinsam, vielmehr ist sie als Fabrik [GHJV94] zu verstehen.

5.1.3 Bewertung von Zope als CCM Plattform

Die Frage nach der Eignung von Zope für die konzeptorientierte Inhaltsverwaltung besteht aus zwei voneinander unabhängigen Fragen. *Wie gut lassen sich Assetmodelle auf Zope abbilden?* und *Wie gut lässt sich Zope in die modulare Systemarchitektur der* CCM *Systeme einbinden?* Eng verbunden mit diesen Fragen, ist die nach der Plattformunabhängigkeit von CCM Systemen, der anschließend nachgegangen wird.

Die **Abbildung von Assetmodellen auf Zope** gelang erstaunlich gut, wie die Analyse in Kapitel 3 zeigt. Die nennenswerten Defizite von Zope bestanden in dem Fehlen des Konzeptes der *Beziehung* und der fehlenden Trennung zwischen Inhalt und Konzept. Am meisten überraschte, dass sich die konzeptionellen Modelle der ADL in ähnlich verständlicher Form durch

die Zope Schemata ausdrücken lassen - die Schemasprache wurde schließlich im Gegensatz zur ADL nicht für die Anwender konzipiert.

Die **Einbindung von Zope in die CCM Systemarchitektur** bereitet größere Probleme. Bereits bei der Erläuterung in Abschnitt 2.3, wie man Zope und CCM gemeinsam einsetzen kann, wurde klar, dass sich Zope nicht ohne weiteres in den modularen Aufbau der CCM Systeme integrieren lässt. Das liegt zum einen an den Problemen, eine Laufzeitkommunikation zwischen Java und Python zu realisieren, Lösungsansätze werden im Ausblick 5.3 vorgestellt. Zum anderen sind einige Dienste, wie z. B. die Persistenz, sehr fest in Zope integriert und werden nach außen nicht zur Verfügung gestellt. Zope ist für die Realisierung vollständiger webbasierter Anwendungen ausgelegt, die Nutzung von Teilfunktionalität des Systems gestaltet sich aufwändig. Als Beispiel wird die Einbindung von Zope als CCM Präsentationsmodul im Anhang 5.3 erläutert.

Da die **Plattformunabhängigkeit** als einer der wichtigsten Vorteile der Systemgenerierung angesehen wird, soll, ausgehend von den Erfahrungen dieser Arbeit, erläutert werden, inwieweit sich dieser Vorteil für die CCM Systeme über Programmiersprachengrenzen hinweg nutzen lässt. Es hat sich gezeigt, dass die Integration von nicht Java basierten Komponenten in CCM Systemen mit nicht unerheblichen technischen Schwierigkeiten verbunden ist. Aus diesem Grund ist für die meisten Anwendungen eine Beschränkung auf Java als grundlegende Plattform sinnvoll. Es scheint interessanter, das Modulkonzept von CCM vollständig auf eine andere Programmiersprache zu übertragen, so dass die generierten Systeme unabhängig von Java Komponenten lauffähig sind. Dieser Gedanke wird im Anhang 5.2 aufgegriffen, in dem untersucht wird, wie sich durch Zope Komponenten Modellevolution und Personalisierung realisieren lässt.

5.2 Ausblick: eigenständige Zope Systeme

5.2.1 Basisfunktionen

Feldtypen für Inhaltsobjekte

Wie bereits erläutert, sind die Möglichkeiten, Inhaltsobjekte in Zope zu verwenden, begrenzt. Dabei wurde bereits in Abschnitt 3.2.3 festgestellt, dass Zope anbietet, dieses Defizit durch Implementierung spezieller Feldtypen zu beseitigen. Das Speichern der Inhaltsobjekte stellt dabei keine Schwierigkeit dar, vielmehr ist die Integration der Inhalte in die webbasierte Oberfläche durch *Widgets* eine anspruchsvolle Aufgabe.

Bilder und Grafiken können direkt in Webseiten dargestellt werden, wobei die Widgets eine Anpassung der Bilddaten vornehmen können. Z. B. kann die Auflösung von Bildern reduziert werden, um die Darstellung einer Bildvorschau zu beschleunigen. Auch andere Anpassungen, wie die Konvertierung zwischen verschiedenen Bildformaten ist denkbar.

Gerade bei **Filmen** bietet sich eine Reduktion der Datenrate für die Übertragung an. Darüber hinaus können Filme in Webseiten auch durch Bilder symbolisiert werden, z. B. durch die Darstellung von Schlüsselszenen.

Auch die Verwendung von **Dokumenten aus Büroanwendungen** als Inhaltsobjekte bietet interessante Möglichkeiten. So können für Dokumente Such- und Konvertierungsfunktionen bereitgestellt werden. Auch eine Vorschau in Webseiten in Form von Symbolen oder Inhaltsauszügen ist denkbar.

Feldtypen für Beziehungen

Wie bei den Inhaltsobjekten bereitet auch das Speichern von Beziehungen keine Probleme, sogar die Darstellung von Beziehungen in Webseiten gelingt durch HTTP Links auf in Beziehung stehende Assets problemlos. Die Schwierigkeit ist, wie Beziehungen über eine webbasierte Benutzeroberfläche eingegeben werden können. Zwei allgemein verwendbare Ansätze sind denkbar.

Eine Möglichkeit ist es, über ein Textfeld eine **Suchfunktion** bereitzustellen. Nachdem die Suche durchgeführt wurde, wird dem Anwender eine Ergebnisliste präsentiert, aus der er die zu referenzierenden Assets auswählt. Dieses Verfahren wird häufig bei Fahrplanauskunftdiensten verwendet, um Start- und Zielstationen einzugeben. Offen bleibt die Frage, nach welchen Kriterien gesucht werden soll. Es sind sowohl Volltextsuchen oder die Suche nach einem bestimmten Charakteristikum denkbar.

Alternativ kann ein **Klemmbrett** (*clipboard*) verwendet werden, welches Verweise auf Assets speichert. Der Anwender kann bei der Eingabe einer Beziehung aus den Assets vom Klemmbrett wählen.

Suchfunktion

Bisher gibt es in den generierten Zope Systemen keine Möglichkeit, nach Assets zu suchen - die Praxistauglichkeit der Systeme ist dadurch deutlich eingeschränkt. Während der Weiterentwicklung von Zope3 in den letzten Monaten wurde ein Katalog (*catalog*) implementiert. Er ist ein Grundgerüst, um Suchindizes auf eigenen Komponenten zu erstellen. Inwieweit sich hieraus praxisgerechte Suchfunktionen realisieren lassen, bleibt noch zu klären.

5.2.2 Modellevolution

Aus den Forderungen nach Offenheit und Dynamik folgt direkt die Notwendigkeit, eine Evolution der Assetmodelle zu unterstützen. Es gibt mehrere Möglichkeiten, diese auf einem reinen Zope System zu realisieren.

Inhärent gegebene Möglichkeiten

Durch besondere Eigenschaften der Programmiersprache und der integrierten Datenbank gibt es eine Möglichkeit, Inhaltskomponenten um weitere Felder zu ergänzen. Dies funktioniert allerdings nur, wenn ein konstanter Standardwert für das neue Feld angegeben werden kann.

Ermöglicht wird dies durch zwei Umstände: Die ZODB speichert unabhängig von seiner Klasse alle Attribute eines Objektes, - jedes Objekt ist in der Datenbank als ein assoziatives Feld mit Attributnamen und Attributwerten gespeichert. Da sich Klassen- und Instanzattribute im lesenden Zugriff nicht voneinander unterscheiden lassen, können Standardwerte in Klassendefinitionen vorgegeben werden. Auf Spracheigenschaften von Python wurde in Abschnitt 4.2.3 eingegangen.

ZODB Generationen

In Zope wurde eine experimentelle Funktion implementiert, um Datenbankevolutionen über Generationen zu unterstützen. Dazu werden die ZODB Datenbanken mit einer Generationskennziffer versehen. Wird eine Schemaevolution benötigt, muss dem System eine Routine zur Verfügung gestellt werden, welche die Konvertierung zwischen den verschiedenen Versionen vornimmt.

Ein Problem bei diesem Ansatz ist, dass die Konvertierung der Datenbank zu einer neuen Generation im ganzen erfolgt. Eine Evolution, wie sie für die CCM Systeme geplant ist, nämlich jedes Asset zu konvertieren, sobald es benötigt wird, ist nicht möglich.

Evolution durch Adapterkomponenten

Eine Modellevolution kann prinzipiell auch mit Hilfe der Adapterkomponenten realisiert werden. Wie schon in der Vorstellung von Zope erläutert, gehen diese über das Entwurfsmuster Adapter hinaus, da sie nicht nur zur Anpassung inkompatibler Schnittstellen eingesetzt werden können.

Adapterkomponenten werden in Zope über einen Konstruktoraufruf erzeugt, dem das zu adaptierende Objekt übergeben wird. Statt die Instanz einer Adapterklasse zu erzeugen, kann der Konstruktor selbst ein Asset der Klasse A in eines der Klasse A' überführen. In diesem Fall können komplexere Änderungen des Modells realisiert werden, ohne dass die Datenbank als ganzes in das neue Modell konvertiert wird.

Allerdings bedarf es einer genaueren Untersuchung, um festzustellen, ob die Persistenzmechanismen der ZODB eine solche Verwendung der Adapterkomponenten zulassen. Entscheidend ist hierbei, wie mit persistenten Objektreferenzen zwischen verschiedenen Evolutionsstufen umgegangen werden kann.

5.2.3 Personalisierung

Personalisierung durch Adapterkomponenten

Die Personalisierung von Assetmodellen kann durch den Einsatz von Adapterkomponenten mit eigenen Präsentationskomponenten realisiert werden. Eine Kombination dieser Komponentenarten ist ein für Zope typisches Muster. Durch einen Adapter können Assets der Klasse A in personalisierte Assets der Klasse A' überführt werden. Werden für A' Präsentationskomponenten bereitgestellt, lässt sich die personalisierte Form betrachten und verändern - Modifikationen werden von dem Adapter an die allgemeine Komponente (Klasse A) delegiert. Zu klären bleibt die Frage, was geschieht, wenn in A' zusätzliche Felder vorhanden sind bzw. wo diese Information gespeichert werden kann.

Auf diese Weise gibt es mehrere Sichtweisen (*views*) auf die Daten - direkte und adaptierte. Es bietet sich an die Sichtweisen für ein personalisiertes Modell in einer eigenen Präsentationsschicht (*layer*) zusammenzufassen, damit die Nutzer eine konsistente Darstellung des personalisierten Modells erhalten.

Personalisierung durch Delegation

Statt der Verwendung von Adapterkomponenten kann die Personalisierung auch durch angepasste Inhaltskomponenten erfolgen. Wird eine Inhaltskomponente I erzeugt, wird gleichzeitig eine personalisierte Variante I' in einem anderen Ordner abgelegt. Die Implementierung von I' delegiert Anfragen und Änderungen an I. Sind in I' zusätzliche Attribute vorhanden, werden diese direkt in der personalisierten Variante gespeichert.

5.3 Ausblick: Zope als CCM Dienstschnittstellenmodul

5.3.1 Laufzeitkommunikation zwischen Zope und Java

Problembeschreibung

Um Zope als Dienstschnittstellenmodul nutzen zu können, müssen Schnittstellen über Programmiersprachengrenzen adaptiert werden. (siehe Abbil-

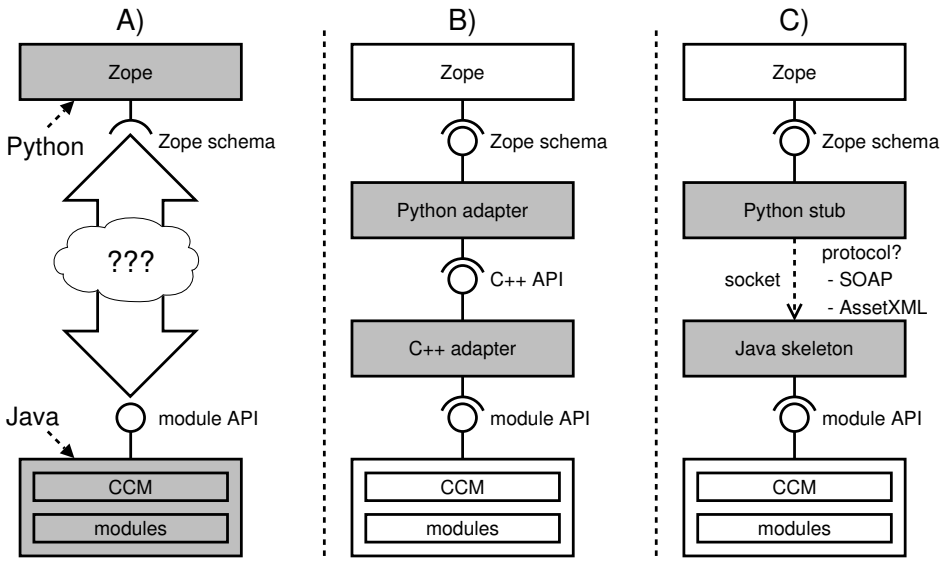

Abbildung 5.1: Laufzeitkommunikation zwischen Java und Zope

dung 5.1 A) Die CCM Module stellen ihre Dienste über die allgemeine Modulschnittstelle in Form von Java *interfaces* zur Verfügung, wohingegen die Zope Komponenten über Schemata miteinander kommunizieren.

Die konzeptionellen Differenzen zwischen Modulschnittstelle und Schema bereiten weniger Probleme als die Laufzeitkommunikation, da eine direkte Kommunikation zwischen Python und Java Modulen nicht möglich ist. Auch Zope stellt keine Mittel bereit, um dieses Problem zu lösen - wie in Abschnitt 2.1.4 erläutert wurde, sind die Möglichkeiten, Zope in andere Systemumgebungen zu integrieren, begrenzt.

Kommunikation über C++

Sowohl Java wie auch Python verfügen über Möglichkeiten mit C/C++ Programmen zu interagieren. Daher ist es möglich, die allgemeine Modulschnittstelle auf eine C++ Schnittstelle abzubilden, die wiederum über die Python/C API auf Zope Schemata adaptiert wird. Dies wird in Abbildung 5.1 B) veranschaulicht. In diesem Zusammenhang stellt sich die Frage, inwieweit der SWIG Kodegenerator genutzt werden kann. (*Simplified Wrapper and Interface Generator*, [Swig]) Er erzeugt Adaptionskode, um C/C++ Bibliotheken unter anderen Programmiersprachen, auch Java und Python, nutzen zu können.

Trotzdem erscheint diese Möglichkeit eher theoretischer Natur zu sein, da durch die mehrfachen Übergänge zwischen den Programmiesprachen in der Praxis Probleme zu erwarten sind.

Kommunikation über Netzwerkprotokolle

Wenn eine direkte Kommunikation zwischen den Programmiersprachen keine Lösung ist, bleibt nur die Möglichkeit, Netzwerkprotokolle einzusetzen. Es bleibt die Frage nach einem geeigneten Protokoll, siehe Abbildung 5.1 C). In der Architektur der CCM Systeme sind Distributionsmodule vorgesehen, allerdings dienen diese nur der Verteilung der Systeme. Ergänzt man das Konzept der Distribution um eine programmiersprachenunabhängige Datenrepräsentation und Protokollebene, ist es möglich, eine Verteilung auch über Programmiersprachengrenzen hinweg zu realisieren.

Es gibt bereits eine programmiersprachenunabhängige Repräsentation von **Assetmodellen in Form von XML Schemata** . Diese werden zum Speichern von Assets in XML Datenbanken verwendet, eignen sich aber auch für den Austausch von Assetdaten. Auf Grundlage dieser Schemata kann ein Protokoll zur Distribution von Modulen entwickelt werden. Dazu müssen die vorhandenen XML Schemata um eine Kommandoschicht ergänzt werden, die Laufzeitoperationen auf Assets ermöglicht, wie z. B. das Erzeugen, Auffinden und Destruieren. Ein solches Protokoll wird zur Zeit in einer anderen Arbeit entwickelt.

Eine andere Möglichkeit ist es, **SOAP** als programmiersprachenunabhängiges Protokoll einzusetzen. Es gibt bereits Ansätze für CCM Systeme, Dienstschnittstellenmodule zu generieren, welche die Assets über SOAP-Webservices zugänglich machen. Obwohl als Dienstschnittstelle geplant, können diese Module als Skelett (*skeleton*) für eine sprachenunabhängige Verteilung verwendet werden.

5.3.2 Anpassung der Komponenten

Auch wenn die Daten nicht in der integrieren Datenbank gespeichert werden, muss jede Inhaltskomponente - und damit auch jedes Asset - über eine eigene URL ansprechbar sein. Dafür bietet sich das in dieser Arbeit entwickelte Prinzip, für jede Assetklasse einen eigenen Ordner in Zope zu verwenden, an. Für den letzten Teil der URL, den Namen, kann der Identifikator (*identifier*) der Assets verwendet werden.

Die Flexibilität der Zope Komponentenarchitektur kommt hier gelegen, da die Schemata, welche die zu dieser Arbeit realisierten Generatoren erzeugen, prinzipiell weiter verwendet werden können. Es soll kurz angedeutet werden, welche Aufgaben die neuen Implementierungen der Komponenten übernehmen müssen, um die Verteilung der Anwendung zu realisieren.

- Den Implementierungen der **Ordner** fällt die Aufgabe zu, über einen Proxy (bzw. einen *stub*) die Daten der Assets von den Java Modulen zu beschaffen und aus ihnen transiente Inhaltskomponenten zu erzeugen. Diese Inhaltskomponenten werden als Grundlage für die Präsentation der Inhalte und für Änderungsoperationen verwendet.

- Da die **Inhaltskomponenten** nicht mehr in der integrierten Datenbank gespeichert werden, muss ihre Implementierung Sorge für die Persistenz tragen. Modifizierte Inhaltskomponenten geben ihre Änderungen über einen Proxy an die Java Module weiter.

- Der **Zope Schema Generator** muss angepasst werden. Da die meisten Module in diesem Szenario auf Java basieren, wird auch Java als in die ADL eingebettete Sprache verwendet. Das in Abschnitt 3.2.2 diskutierte Problem, wie man eine unbegrenzte Menge von Java Typen auf die wenigen Zope Feldtypen abbildet, tritt wieder auf. Alternativ ist es möglich, die Zope Schemata auf Grundlage von XML Schemata des Assetmodells zu erzeugen - sofern diese zur Laufzeitkommunikation verwendet werden.

5.3.3 Transaktionen

Einer gesonderten Betrachtung bedürfen die unterschiedlichen Transaktionsmodelle von Zope und CCM. Die in Zope integrierte Datenbank verwendet einen optimistischen, auf Zeitstempeln basierenden Synchronisationsmechanismus. Sperren sind dabei nicht notwendig, sie werden weder explizit noch implizit gesetzt. In den CCM Systemen wird dagegen ein pessimistischer Mechanismus verwendet. Das Sperren von Assets muss über die Modulschnittstelle sogar explizit geschehen. (siehe [Ful00, Seh04])

Für die Verwendung von Zope als Dienstschnittstellenmodul bleibt zu klären, wie Zope Komponenten mit dem Transaktionsmodell der CCM Systeme umgehen können. Der Scheduler (*transaction manager*) der ZODB unterstützt zur Anbindung relationaler Datenbanken das *two-phase commit* Protokoll. Dieser könnte auch zur Bearbeitung von CCM Transaktionen verwendet werden. Da offen ist, wie Asset Transaktionen auf die Protokollschicht zwischen Zope und Java abgebildet werden, können hierzu keine endgültigen Aussagen gemacht werden.

Anhang A

Lebenszyklen

A.1 ZODB Objekt Lebenszyklus

Abbildung A.1 zeigt ein Zustandsdiagramm in Anlehnung an [Ful00]. In dem Diagramm sind die Zustände im Lebenszyklus eines in der ZODB (*Zope Object Database*) gespeicherten Objektes dargestellt. Dabei ist zu beachten, dass lediglich die Zustände eines Objektes im Hauptspeicher dargestellt sind, d.h. nicht mehr referenzierte Objekte können sich noch weiterhin in der Datenbank befinden.

Der Zustand *ghost* bezeichnet ein Objekt, dass persistent in der Datenbank gehalten wird, sein Zustand jedoch nicht im Hauptspeicher vorhanden ist. Beim Zugriff auf ein Attribut dieses Objektes wird sein Zustand geladen, dadurch wird das Muster *lazy load* aus [FRF03] implementiert Ein nicht mehr referenziertes Objekt kann, sofern sein Zustand im nichtflüchtigen Speicher abgelegt ist, weiterhin existieren. Ein *garbage collector* entfernt persistente Objekte, die nicht über Objektreferenzen erreicht werden können.

A.2 Asset Lebenszyklus

Das Zustandsdiagramm aus Abbildung A.2 ist [Seh04] entnommen. Es zeigt den Lebenszyklus eines Assets, wie er durch die Allgemeine Modulschnittstelle realisiert wird. Im Unterschied zu der ZODB ist ein explizites Sperren eines Assets notwendig, um seine Attribute zu verändern.

Es wird deutlich, dass sich die Lebenszyklen der ZODB Objekte und der Assets deutlich voneinander unterscheiden. Ein Vergleich der Persistenzmechanismen ist jedoch nicht Gegenstand dieser Arbeit (siehe Abschnitt 3.1) und wird erst im Ausblick (Abschnitt 5.3) aufgegriffen.

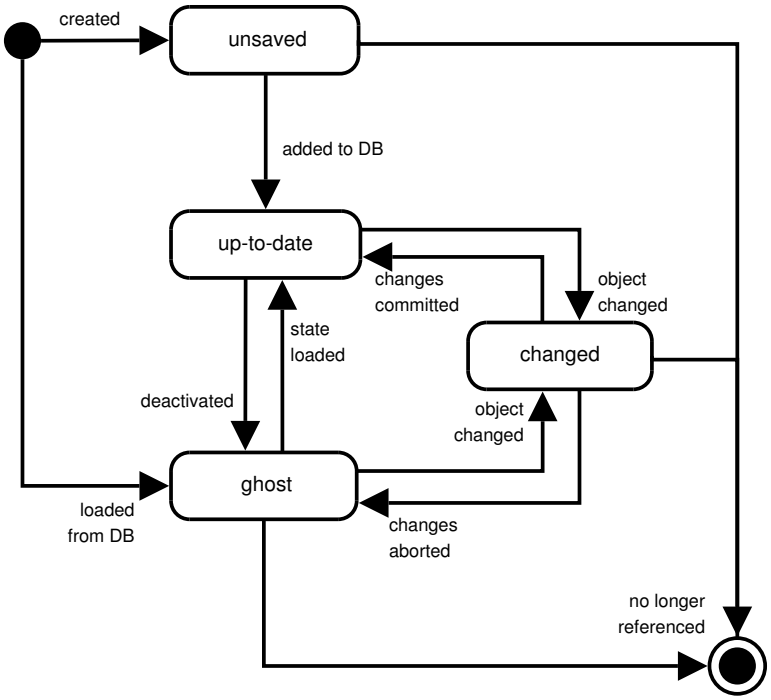

Abbildung A.1: Zustandsdiagramm: Lebenszyklus eines ZODB Objekt

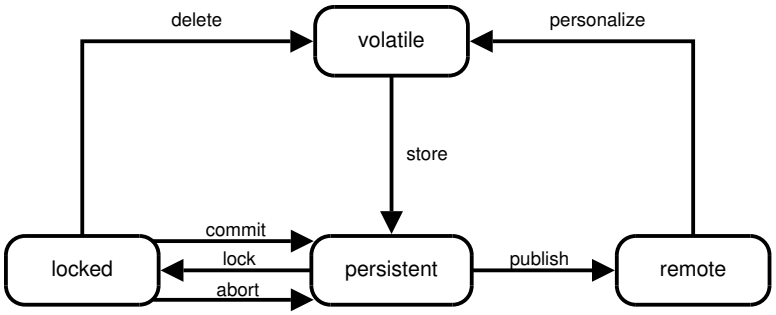

Abbildung A.2: Zustandsdiagramm: Lebenszyklus eines Assets

Anhang B

Symboltabellen der Zope Generatoren

Das Klassendiagramm in Abbildung B.1 gibt einen Überblick über die Symboltabellen der Zope3 Generatoren. Das Diagramm ist aus einer Entwurfssicht heraus erstellt, die konkrete Implementierung in Java unterscheidet sich dementsprechend. Aus Gründen der Übersicht wurden Attributtypen meist ausgelassen: Wenn keiner angegeben wurde handelt es sich um eine Zeichenkette.

Zwei Symboltabellen fehlen in der Darstellung: Bei dem `ConfigurationSymbolTable` und dem `TestSymbolTable` (siehe Abschnitt 4.2.2) handelt es sich um Instanzen der Klasse `EmptyZope3SymbolTable`. Es gibt keine relevanten Informationen, welche die entsprechenden Generatoren an nachfolgende Generatoren übermitteln könnten.

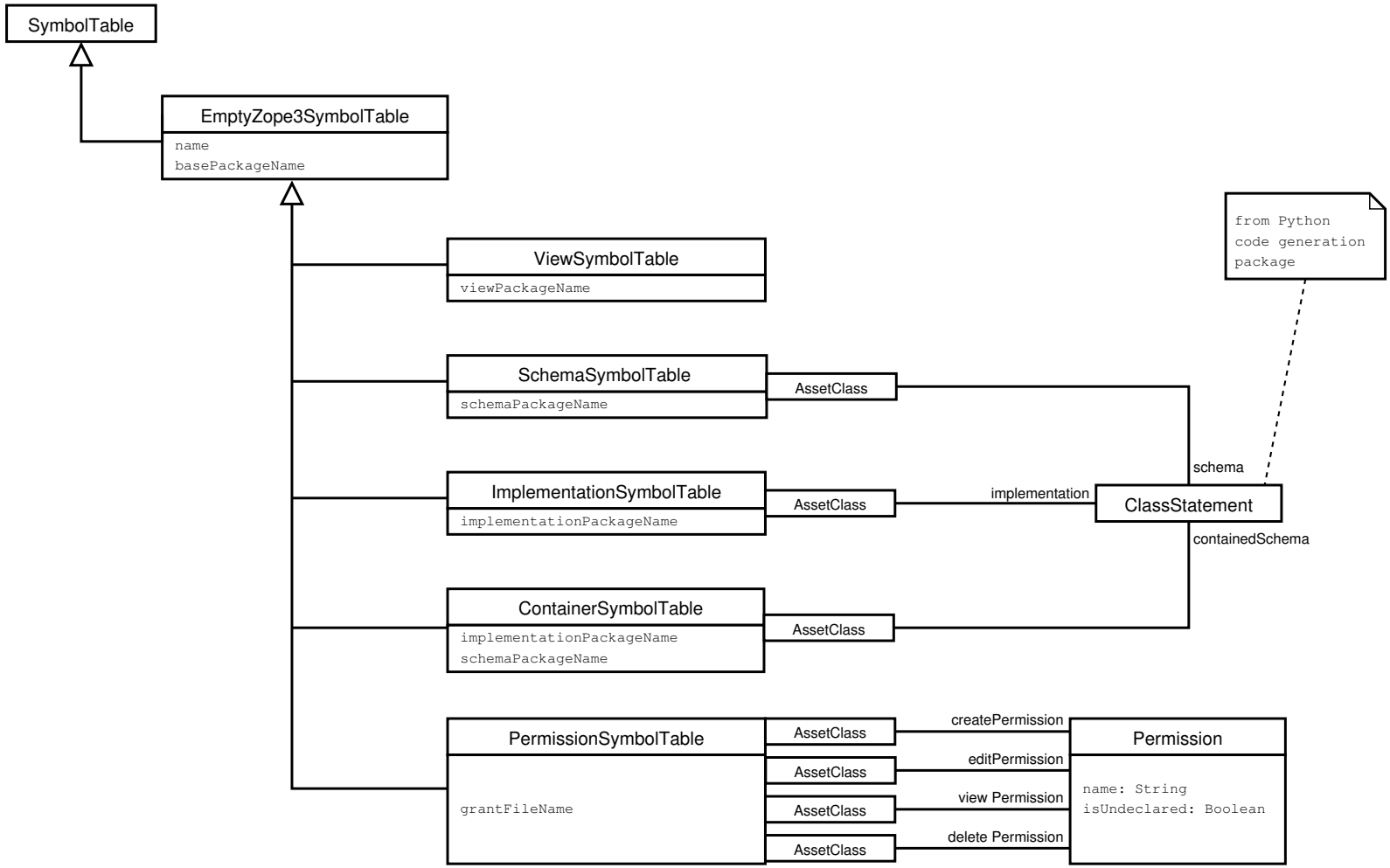

Abbildung B.1: Klassendiagramm: Symboltabellen der Zope3 Generatoren

Literaturverzeichnis

[ASU86] Aho, Alfred V., Ravi Sethi und Jeffrey D. Ullman: Compilerbau. Addison-Wesley, 1986

[Bal01] Balzert, Helmut: Lehrbuch der Software-Technik, Band 1: Software-Entwicklung. Spektrum Akademischer Verlag, 2. Auflage, 2001

[BN96] Bernstein, Philip A. und Newcomer, Eric: Principles of Transaction Processing. Morgan Kaufmann Publishers, 1996

[Che75] Chen, Peter P.: The Enity-Relationship Model: Toward a Unified View of Data. In: Kerr, Douglas S. (Herausgeber): Proceedings of the International Conference on Very Large Data Bases, September 22-24, 1975, Framingham, Massachusetts, USA, Seite 173. ACM, 1975

[Fow04] UML destilled: a brief guide to the standard object modeling language, 3rd edition. Addison-Wesley, 2004

[FRF03] Fowler, Martin, David Rice und Matthew Foemmel: Patterns of Enterprise Application Architecture. Addison Wesley, 2002

[Ful00] Fulton, James L.: Introduction to the Zope Object Database. Proceedings of the 8th Python Conference, Arlington, Va, 2000, erhältlich über:
`http://www.zope.org/Wikis/ZODB/FrontPage`

[FulPZ] Fulton, James L.: Programming with the Zope 3 Component Architecture. Folien verwendet auf mehreren Entwicklertreffen, erhältlich über:
`http://www.zope.org/Wikis/DevSite/Projects/..`
`../ComponentArchitecture/Documentation`

[GHJV94] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison- Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1994

[Gra95] Graham, Paul: ANSI Common Lisp. Prentice Hall, 1995

[HM04] Harold, Eliotte R. und W. Scott Means: XML in a Nutshell, A Desktop Quick Reference. O'Reilly, 2004

[Hus02] Husted, Ted: Struts in Action A Practical Guide to the Leading Java Web Framework. Manning Publications, 2002

[HZS05] Huang, Shan Shan, David Zook und Yannis Smaragdakis: Statically Safe Program Generation with SafeGen. Proceedings of the Fourth International Conference on Generative Programming and Component Engineering, September 29 - October 1 2005, Tallinn, Estonia

[JBoss] JBoss Application Server
`http://www.jboss.com/products/jbossas`

[JCGTk] Java Code Generation Toolkit
`http://www.sts.tu-harburg.de/~hw.sehring/cocoma/#jcgtk`

[KHM05] Kücükyilmaz, Hakan, Thomas M. Haas und Alexander Merz: PHP 5. dpunkt Verlag, 2005

[KW05] Krekel, Holger und Phillipp von Weitershausen: Agil serviert, Zope X3.0 mit Komponentenarchitektur. Artikel in der iX 8/3005, Heise Verlag

[Lar05] Larman, Craig: Applying UML and patterns, an introduction to object-oriented analysis and design and iterative development, 3rd edition. Prentice Hall, 2005

[Mar03] Martelli, Alex: Python in a Nutshell. O'Reilly, 2003

[Mck04] McKay, Andy: The Definitive Guide to Plone. Apress, 2004

[Oes06] Oestereich, Bernd: Analyse und Design mit UML 2.1, 8. aktualisierte Auflage. Oldenbourg Verlag, 2006

[Ric05] Richter, Stephan: Zope 3 Developer's Handbook. Sams Publishing, 2005

[Ros05] van Rossum, Guido: Python Reference Manual, Release 2.4.2, 2005, erhältlich über:
`http://docs.python.org/`

[Seh04] Sehring, Hans-Werner: Konzeptorientierte Inhaltsverwaltung - Modell, Systemarchitektur und Prototypen. Dissertation, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, Deutschland, 2004

[SG01] Skonnard, Aaron und Martin Gudgin: Essential XML Quick Reference. Addison-Wesley, 2001

[Silva] The Silva Content Management System
http://www.infrae.com/products/silva

[SSW02] Schmidt, J.W., H.-W. Sehring und M. Warnke: Der Bildindex zur Politischen Ikonographie in der Warburg Electronic Library – Einsichten eines interdisziplinären Projektes. In: Pompe, Hedwig und Leander Scholz (Herausgeber): Archivprozesse. Die Kommunikation der Aufbewahrung, Seiten 238-268. Dumont, 2002

[Swig] SWIG: Simplified Wrapper and Interface Generator
http://www.swig.org

[Tho99] Thompson, Simon: Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999

[Trese] Twente Research & Education on Software Engineering
http://trese.cs.utwente.nl

[VK03] Visnovsky, Stanislav und Matthias Kiefer: The KBabel Handbook. 2003, erhältlich über:
http://docs.kde.org/development/en/kdesdk/kbabel/

[Wei05] von Weitershausen, Philipp: Web Component Development with Zope 3. Springer Verlag Berlin, 2005

[Zope3] Zope.org: Zope3 Documentation
http://www.zope.org/Wikis/DevSite/Projects/..
../ComponentArchitecture/Documentation