Technische Universität Hamburg Harburg
Arbeitsbereich Softwaresysteme
Prof. Dr. Joachim W. Schmidt

Erzeugung von Web Services aus konzeptionellen Inhaltsmodellen

Projektarbeit

Verfasser: Paulus Sentosa
22946

Gutachter: Prof. Dr. Joachim W. Schmidt

Betreuer: Dr. Hans-Werner Sehring

Hamburg, den 6.5.2005

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde weder einer anderen öffentlichen oder privaten Institution vorgelegt noch veröffentlicht.

Hamburg, den 6.5.2005

Inhaltsverzeichnis

1. Einleitung

1.1 Einführung in das Thema

Innovative Informationssysteme wie z.B. Inhaltsverwaltungssysteme werden entwickelt, um eine komplexe Mischung verschiedener Inhalte, wie z.B. Texte, Bilder, Videos, usw. zu organisieren und sie durch domänenspezifische konzeptuelle Modelle zu präsentieren. Eine traditionelle Implementierung solcher Systeme spiegelt aber diese Komplexität durch die Komplexität der Software, die z.B. durch die heterogene Mischung konventioneller Datenbanktechnologie verursacht wird. Daher ist ein integriertes Inhalt-Konzept-Modell, das auf so genannten *Assets* basiert, vorgeschlagen [HWS04]. Es ist eine Erweiterung des herkömmlichen Container Modells, damit noch mehr wichtige Informationen aus dem Modell gewonnen werden können, und dadurch eine höhere Bedeutung des Konzeptes erreicht wird.

Ein System, das auf dem Assetmodell basiert, wird konzeptorientiertes Inhaltsverwaltungssystem genannt. Die wesentlichen Grundlagen solches Systems sind die Prinzipien der Offenheit und Dynamik, mit denen das System durch neu angelegte Konzepte beliebig erweiterbar sein soll und Zugriffe auf alle Assets, insbesondere das Verändern von diesen, ermöglicht.

Diese dynamische Offenheit wird von einem generativen Implementierungsansatz unterstützt, für den ein so genanntes Asset Compiler Framework entwickelt wird. Zum Framework gehört ein Modellcompiler, der aus den Modellen Zwischenmodelle erzeugt, aus denen wiederum eine Menge von Modulen anhand der so genannten Generatoren generiert werden. Die Komponenten eines konzeptorientierten Inhaltsverwaltungssystems sind dann aus diesen Modulen aufgebaut.

Die verschiedenen Module haben ihre eigene Funktion. Einer der Generatoren ist zuständig dafür, dass er ein Modul generiert, das nach dem Prinzip der dynamischen Offenheit den Zugriff von außen auf das System ermöglicht. Dabei spielt

Interoperabilität eine wichtige Rolle. Das Modul soll daher in einer Art vorliegen, die diese wirklich unterstützt. Durch Web Services wird dies gewährleistet.

1.2 Ziel und Aufbau der Arbeit

Das Ziel dieser Arbeit ist es, den Zugriff auf ein konzeptorientiertes Inhaltsverwaltungssystem im Sinne der Web Services zu ermöglichen. Eine konkrete Vorgehensweise wird im Folgenden beschrieben. Innerhalb des vorhandenen Asset Compiler Framework wird ein für dieses Ziel vorgesehenes Modul generiert – das Web Services-Modul. Die Generierung des Moduls erfolgt in zwei Stufen. Zuerst muss ein Modul-Generator gemäß dem gültigen Framework-Protokoll implementiert werden. Dieses Modul soll Operationen nach außen anbieten, die einem Client ermöglichen, neue Assets anzulegen, oder diese zu modifizieren. Eine Anfrage des Clients wird an ein anderes Modul delegiert, das dann die anzulegenden bzw. zu bearbeitenden Assets annimmt bzw. zurückgibt. Damit dieses Modul plattformübergreifend arbeiten kann, generiert man anschließend eine Web Services-taugliche Form des Moduls, die *Web Service Description Language (WSDL)*-Beschreibung, anhand eines externen *plug ins*.

Der Aufbau der Arbeit entspricht der oben beschriebenen Vorgehensweise. Zum eindeutigen Verständnis der Begriffe, wird im Kapitel 2 zuerst eine kurze Einleitung in das Thema „Konzeptorientierte Inhaltsverwaltung“ gegeben. Danach folgt eine ausführliche Erläuterung zur Funktionsweise von Modellcompiler und Generatoren. Im Kapitel 3 wird das Konzept von Web Services betrachtet, um einen Überblick über das allgemeine Konzept zu geben. Kapitel 4 und Kapitel 5 beinhalten den Kern der Arbeit, nämlich die Implementierung des Generators, der die Web Services-Schnittstelle (Web Services-Modul) generieren soll und die Realisierung der WSDL-Beschreibung des generierten Moduls anhand eines von *Systinet* bereitgestellten *plug ins* für die Entwicklungsplattform *Eclipse*. Eine Zusammenfassung und ein Ausblick auf weitere Realisierungsmöglichkeiten im Kapitel 6 schließen die Arbeit ab.

2. Konzeptorientierte Inhaltsverwaltung

2.1 Einführung

Im Kapitel 1 wurde die Idee der konzeptorientierten Inhaltsverwaltung vorgestellt. Bei dieser werden die bereits existierenden Modellierungsansätze vereinigt, nämlich inhaltsorientierten Ansatz, der sich auf die Darstellung einer Entität konzentriert und konzeptorientierten Ansatz, der sich um die Beschreibung kümmert. Beide Sichtweisen ergeben ein neues Konzept zur Entitätsbeschreibung, das so genannte *Asset*.

Diese Art von Modellierung muss bestimmte Forderungen erfüllen, welche bereits C.S. Peirce [PE31] als wichtig erkannt hat. Pierce stellt fest, dass die Entitätsbeschreibung drei verschiedene Perspektiven umfassen soll, nämlich

- die inhärenten Charakteristika einer Entität
- Beziehungen zwischen Entitäten
- die Gesetzmäßigkeit der oben genannten Modellierungsdarstellungen

Damit hat die Entitätsbeschreibung die Eigenschaft der **Ausdrucksfähigkeit** (auf Engl. *Expressiveness*), welche ein *Asset* erfüllen soll.

Für das Anlegen derartiger Entitätsbeschreibungen gibt es desweiteren die Forderungen nach **Offenheit** und **Dynamik**. Offenheit bedeutet, dass es keine fest vorgegebene Menge von Konzepten gibt. Das heißt, dass ein Anwender Entitätsbeschreibungen je nach Bedarf bzw. Anforderungen der zu modellierenden Entitäten anlegen kann. Die Forderung nach Dynamik umfasst nicht nur das Anlegen, sondern auch das Ändern der offen angelegten Entitätsbeschreibungen. Die Eigenschaft der so genannten ***Responsiveness*** [HWS04] wird durch diese Forderungen verwirklicht.

Damit ein konzeptorientiertes Inhaltsverwaltungssystem all diesen Forderungen nachkommen kann, basiert der Systementwurf auf zwei Maßnahmen, nämlich

1. einem generativen Implementierungsansatz und
2. einer Architektur, welche die Evolution der individuellen Modelle durch kooperierende Systemkomponenten ermöglicht.

Durch diesen Ansatz kann die Forderung nach Offenheit erfüllt werden, indem Inhaltsverwaltungssysteme gemäß der Definition von Entitätsbeschreibungen erzeugt werden. Die Architektur realisiert die Dynamik, indem sie die Veränderung der generierten Systeme auch zur Laufzeit ermöglicht.

In den nächsten Abschnitten wird tiefer auf das Konzept eingegangen, und es wird der Implementierungsansatz vorgestellt, der sich nach den weiteren Anforderungen solcher Konzepte als ein passender Ansatz herausstellt.

2.2 Konzept-Inhalt-Paaren als Basis der konzeptorientierten Inhaltsverwaltung

In diesem Abschnitt wird das Inhaltsmodell vorgestellt, in dem Konzepte und Inhalte als untrennbare Einheiten betrachtet werden. Ein Konzept-Inhalt-Paar wird als *Assets* bezeichnet. Die folgende Abbildung illustriert die doppelte Beschreibungsleistung eines Assets.

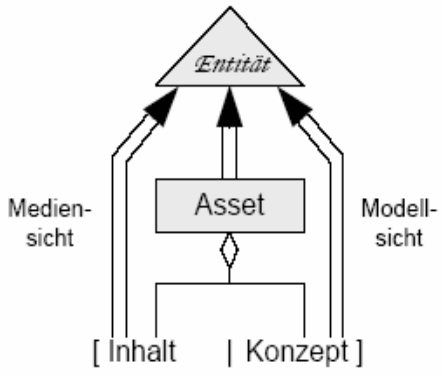


Abb. 2.1: Assets stellen Entitäten als Inhalt-Konzept-Paar dar ([HWS04])

Weder das Konzept noch der Inhalt kann einzeln existieren. Der Konzept-Teil wird benutzt, um zu verdeutlichen, wie ein Inhalt eine Entität referenziert, während der Inhalt als Existenzbeweis der Gültigkeit eines Konzeptes dient.

2.3 Assetdefinitionssprache : Sprache für die konzeptorientierte Inhaltsverwaltung

Damit die Offenheit realisiert werden kann, sind möglichst wenige Sprachkonzepte fest vorgegeben. Es wird stattdessen ein einheitliches Modellierungsmittel, die so genannte *Assetdefinitionssprache* vorgestellt, die das Asset in den Mittelpunkt stellt. Assetdefinitionen sind zugleich Grundlage der dynamischen Modellierung.

Grundsätzlich basieren Assets auf der Definition von Klassen, die ihre Struktur beschreiben. In der sprachlichen Notation der Assetdefinitionssprache werden Assetklassen in folgender Weise definiert.

```
class name {
        content ....
        concept ....
}
```

Eine Klassendefinition wird durch das Schlüsselwort **class** eingeleitet, nach dem der Name der Assetklasse und die Definition der Struktur in geschweiften Klammern angegeben werden. Die Assetklassendefinition besteht aus zwei Sektionen: **content** und **concept** (Inhalt und Konzept). Die Inhaltssektion definiert den Bezug zu Inhalten. In der Konzeptsektion werden die Attribute **characteristic** (Charakteristik) und **relationship** (Beziehung) und Bedingungen aus der Definition von Assets im Abschnitt 2.1 entsprechend eingeführt. Die beiden Attributarten beziehen sich auf Assetinstanzen, d.h., sie werden strukturell in einer Assetklasse definiert und jede Assetinstanz hat ihre eigenen Ausprägungen der Attribute. Bedingungen hingegen beziehen sich auf alle Instanzen einer Assetklasse.

Durch das Attribut *characteristic* werden die immanenten Eigenschaften einer Entität beschrieben. Diese Eigenschaften sind z.B. der Titel eines Bildes, der Zeitpunkt der Erstellung des Bildes, und der Erstellungsort. Das Attribut *relationship* definiert die Beziehungen zwischen Assets. Es ist zu beachten, dass Beziehungen nicht auf Objekte verweisen, da solche keine Entitäten sondern nur Werte darstellen. Ein Beispiel hierzu ist die Beziehung *Besitzer* zwischen den Assets *Bild* und *Person*.

2.4 Assetmanipulationssprache und Assetanfragesprache

Für das Anlegen, Verändern und Auffinden von Assets gibt es die entsprechenden Ausdrucksmittel, nämlich die Assetmanipulationssprache und Assetanfragesprache. Hierzu gibt es die Operationen `create`, `modify`, `delete` und `lookfor`, mit denen sich der Abschnitt 4.3 beschäftigt.

2.5 Implementierungsansatz : Generierung

Wie es oben bereits erwähnt ist, wird zur Erreichung der offenen Dynamik vorgeschlagen, konzeptorientierte Inhaltsverwaltungssysteme erzeugen zu lassen, so dass sich ein solches System dynamisch an sich ändernde Assetmodelle anpassen kann. Dieses Ziel lässt sich durch einen generativen Implementierungsansatz realisieren, der vorsieht, die Implementierung nicht in einer Programmiersprache vorzunehmen, sondern eine abstrakte Beschreibung der gewünschten Software zu erstellen. Diese Beschreibung wird dann einem so genannten Generator übergeben, der die Software spezifikationsgemäß erstellt.

2.6 Ein Modellcompiler für konzeptorientierte Inhaltsverwaltungssysteme

Für den generativen Ansatz müssen vorher die Assetmodelle von einem Modellcompiler in eine Zwischenrepräsentation (Zwischenmodell) umgewandelt werden. Die Basisstruktur dieses Modellcompilers ist der klassischen Architektur eines Compilers ähnlich, die aus einem Frontend und einem Backend besteht.

Diese beiden kommunizieren miteinander, indem sie das Zwischenmodell (auf Engl.: *Intermediate Model*) austauschen. Der Modellcompiler in sich ist ein Rahmenwerk (auf Engl.: *Framework*), das durch die so genannten Modul-Generatoren erweiterbar ist. Dieses ist dann auch zuständig für die Koordination der Generatoren.

2.6.1 Frontend des Modellcompilers

Aufgabe des Frontends ist es, ein vom Nutzer definiertes Modell in die Zwischenrepräsentation zu übersetzen, die von den verschiedenen Generatoren verwendet werden kann. Außerdem ist das Frontend auch zuständig für die Sicherstellung der Konformität des Modells, d.h., ob das Zwischenmodell in einer Form vorliegt, aus der die Generatoren des Backends die jeweiligen Komponenten konsistent erzeugen können. Die Eingabe ist ein Modell, das in der im Abschnitt 2.3 definierten Sprache gegeben ist. Als konkrete Objektsprache kommt die Programmiersprache Java zum Einsatz, in der auch der Modellcompiler geschrieben ist.

2.6.2 Compiler-Backend : der Modulschnittstellen-Generator „*API-Generator*“

Nach der Erstellung des Zwischenmodells durch das Frontend wird aus diesem im Backend zunächst eine Menge von Schnittstellen erzeugt. Diese sind unter anderen eine einheitliche Schnittstelle, die von allen Modulen, welche Erzeugnisse des Modellcompilers sind, und die Komponenten eines konzeptorientiertes Inhaltsverwaltungssystems aufbauen, implementiert wird, und Schnittstellen zum Umgang mit Instanzen. Die letzteren werden durch Methoden zum Zugriff auf die Attribute, Veränderung des Zustands und Verwaltung des Lebenszyklus der Assets ergänzt. Dazu gehören z.B. solche zum Löschen, Personalisieren etc. Diese Erzeugung von Schnittstellen wird von dem so genannten Schnittstellengenerator zur Anwendungsprogrammierung (auf Engl.: *Application Programming Interface-Generator, API-Generator)* erledigt.

Für die Modulschnittstelle werden die folgenden Schnittstellendefinitionen aus einem Modell erzeugt:

- Schnittstellen für die verschiedenen Lebenslagen der Assets
- Iteratoren über Mengen von Objekten
- Schnittstellen von Fabriken für Instanzen verschiedener Typen
- Besucherschnittstellen
- Schnittstellen für Anfrageobjekte
- Schnittstellen für die Darstellung der Typinformation

Die von diesem Generator generierten Methoden werden im Kapitel 4 noch ausführlicher behandelt.

Die möglichen Lebenslagen eines Assets und die Methoden, durch welche die Lebenslagen sich ändern, werden in Abbildung 2.4 als Zustandsdiagramm gezeigt. Diese Erkenntnis ist äußerst wichtig, da die zu realisierende Web Services-Schnittstelle sich diese Methoden, und dadurch die verschiedenen Lebenslagen, zu Nutze macht, wie später im Kapitel 4 zu sehen sein wird.

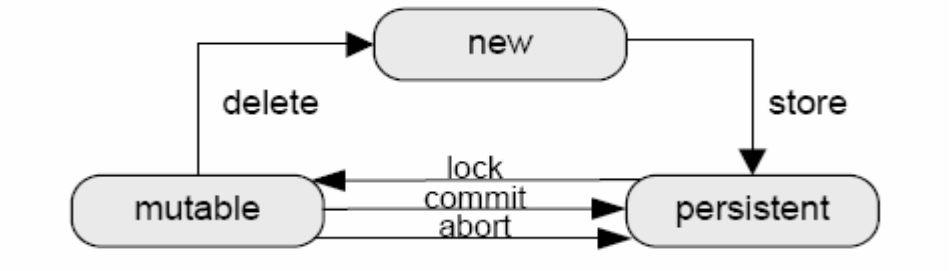


Abb. 2.4: Zustandsdiagramm der Übergänge zwischen den Lebenslagen

2.6.3 Compiler-Backend : Generatoren der Modulen

Nachdem aus dem Zwischenmodell die im vorigen Abschnitt beschriebenen Schnittstellen erzeugt wurden, werden Module implementiert, welche (Java-) Klassen sind und die Schnittstellen implementieren. Zur Erzeugung der Module wird das Backend des Modellcompilers durch Zuweisung einer Menge von Generatoren konfiguriert. Jeder Generator ist für eine Modulart zuständig.

Für jede Modulart stehen verschiedene Generatoren zur Wahl, um ein konkretes Modul auf eine bestimmte Art zu erzeugen. Folgende Abbildung zeigt eine Übersicht der verschiedenen Module.

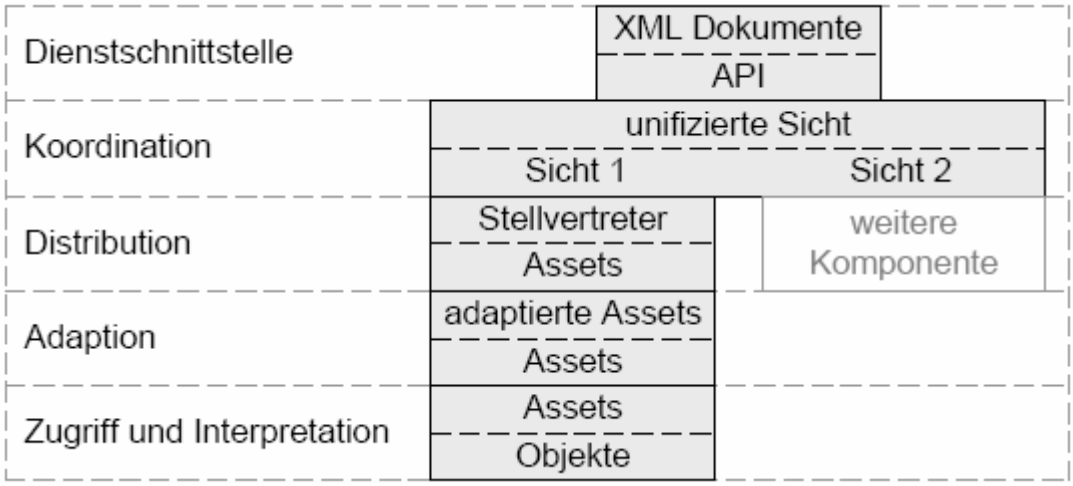


Abb. 2.5: Übersicht über die Arten der Module [HWS04]

Die Aufgaben der verschiedenen Module sind unter anderen

- Dienstschnittstellen-Modul

  Die Funktionalität des Inhaltsverwaltungssystems wird von Dienstschnittstellen-Modulen nach außen angeboten, z.B. über den Austausch von XML-Dokumenten.

- Koordinations-Modul

  Koordinations-Module binden verschiedene Module zusammen. Sie sind in der Lage, Aufrufe an andere Module zu delegieren, und deren Antwort in verschiedenen Kombinationen zu formulieren .

- Distributions-Modul

  Eine Zusammenarbeit zwischen Modulen, die sich in verschiedenen Bereichen eines verteilten Systems befinden, wird von Modulen dieser Art ermöglicht. Der Dienst dieses Moduls ist dem RPC ähnlich.

- Adaptions-Modul

  Adaptions-Module unterstützen die Abbildung zwischen Modellen. Daher ist das Transformieren von Assets eines Modells auf ein anderes möglich.
- Zugriff- und Interpretations-Modul

  Zugriff- und Interpretations-Module dienen dem Zugriff auf Standardkomponenten zum Verwalten von Assets. Durch diese Module können Assets persistent gemacht werden und zum Modifizieren wieder aufgerufen werden.

Der Schwerpunkt dieser Arbeit ist der, einen Generator eines Moduls zu erstellen, das das Zugreifen auf Assets von außen über Internetprotokolle, z.B. HTTP, SMTP, usw., ermöglicht, also ein Dienstschnittstellen - Modul.

3. Web Services

3.1 Einführung

Heutzutage gibt es mehrere Standardtechnologien zur Entwicklung verteilter Systeme, unter anderen Java Remote Method Invocation (RMI), Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA) und Web Services. Sie ermöglichen Applikationen, einen so genannten entfernten Prozeduraufruf auszuführen (auf Engl.: Remote Procedure Call (RPC)) [DOSB05].

In dieser Arbeit wird die Technologie der Web Services angewandt, um die Dienstschnittstelle konzeptorientierter Inhaltsverwaltungssysteme zu realisieren. Einer der Vorteile der Web Services ist der, dass Web Services auf Standardtechnologien des Internets basieren, nämlich dem HTTP Protokoll (oder anderen Protokollen, wie z.B. SMTP), das zurzeit das stark verbreitete Kommunikationsprotokoll ist und weitgehend über das Internet als Transportprotokoll eingesetzt wird. Web Services benutzen plattformunabhängige Technologien wie z.B. XML. Daher ist die Plattformunabhängigkeit eine sehr wichtige Eigenschaft von Web Services. Diese Eigenschaft spielt auch eine wichtige Rolle im *World Wide Web*. Eine Webseite wird zum Beispiel anhand verschiedener Programme bzw. Programmiersprachen und Plattformen erstellt, die von den Browsern der Clients unabhängig sein könnten. Mit Hilfe des HTTP Protokolls erhält ein Browser Daten von den Webseiten im HTML Format. Der Browser muss die dahinter steckende Technologie der Webseiten nicht kennen.

Das gleiche Prinzip gilt für Web Services. Ein Client muss nur die URL der Web Services und die für die Prozeduraufrufe benötigten Datentypen kennen. Die Web Services können dann ohne Kenntnis über die Plattform, auf dem die Services bereitgestellt werden, benutzt werden. Die detaillierte Implementation ist für Clients unsichtbar.

Bei der Entwicklung von Applikationen werden oft Web Services eingesetzt, da diese eine verteilte Umgebung bereitstellen, in der beliebig viele Applikationen bzw. Komponenten einer Applikation nahtlos kommunizieren und zusammenarbeiten können. Diese Applikationen können sich an den verschiedensten Standorten in den Netzwerken befinden, können aber dann zusammengefasst werden, um als Teile eines einzigen, großen Softwaresystems zu funktionieren. Automatisierte Businesstransaktionen, Stock Trading und Order Tracking System sind ein paar Beispiele für Applikationen, die durch Web Services ermöglicht werden.

3.2 Definition der Web Services

Viele Organisationen und Software- bzw. Hardware Anbieter haben Web Services entwickelt und somit den Begriff „Web Service“ selbst definiert.
Im Folgenden sind einige Definitionen zusammengefasst:

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. (World Wide Web Consortium [W3C04])

Web service is a set of related application functions that can be programmatically invoked over the Internet. (IBM [IBM04])

A Web Service is a piece of business logic, located somewhere on the internet, that is accessible through standard-based Internet protocols such as HTTP or SMTP. (Java Web Services, O’Reilly [CHA02])

3.3 Hauptmerkmale der Web Services

Einige der wichtigsten Merkmale der Web Services sind ([CHA02], [ABC02]):

- Web Services basieren auf XML
  Dies ist ein sehr wichtiges Merkmal, da dadurch Interoperabilität erreicht wird.
- Unterstützung des Dokumentenaustauschs
  Durch die Nutzung von XML als Basistechnologie ist nicht nur der Austausch einfacher Dokumente möglich, sondern auch komplexe Dokumente können übertragen werden.
- synchrone und asynchrone Ausführung
  Die Bindung zwischen den Clients und der Ausführung der Services kann sowohl synchron als auch asynchron erfolgen. Im Synchronfall blockiert sich der Client und wartet darauf, bis die Ausführung der Operation zu Ende ist, bevor er sich mit anderen Aufgaben beschäftigen kann. Im Asynchronfall dagegen ist es einem Client erlaubt, einen Service bzw. eine Prozedur aufzurufen und gleich danach andere Funktionalitäten zu erledigen. Das Ergebnis erhält ein asynchroner Client zu einem späteren Zeitpunkt, während ein synchroner Client gleich nachdem der Service ausgeführt ist.
- *Loosely coupled*
  Die Kopplungsstufe eines Systems beeinflusst dessen Modifizierbarkeit sehr stark. Je fester die Kopplung ist, desto mehr Änderungen müssen auf der Seite des Clients vorgenommen werden, wenn der Serviceanbieter in seinem System etwas ändert. Ein Serviceverbraucher ist nicht direkt an einen bestimmten Service gebunden. Die Schnittstellen der Web Services können sich beliebig ändern, ohne sich vorher darüber mit den Clients verständigen zu müssen.
- Web Services unterstützen einen entfernten Prozeduraufruf (RPC)
  Web Services erlauben den Clients, Prozeduren, Funktionen und Methoden auf entfernten Systemen mit Hilfe eines XML basierten Protokolls aufzurufen.

3.4 Technologie der Web Services

Web Services benötigen hauptsächlich auf XML basierende Technologien, um Daten von einem zum anderen Programm und zu Datenbanken zu übertragen und zu transformieren.

3.4.1 WSDL (Web Services Description Language)

WSDL ist eine XML basierte Beschreibungssprache, die die Schnittstellen der Web Services beschreibt. WSDL setzt einen Standard ein, der beschreibt, wie die Web Services die Eingabe- und Ausgabeparameter eines Aufrufes, die Struktur einer Funktion, den Prozeduraufruf und die Protokollbindung der Services darstellen.

3.4.2 SOAP (Simple Object Access Protocol)

SOAP ist eine Kollektion von XML basierten Technologien, die einen „Umschlag“ für die Web Service Kommunikation darstellt. Dies kann auf HTTP und andere Übertragungsprotokolle abgebildet werden, stellt ein Format zum Übertragen von XML Dokumenten über das Netzwerk, und eine Konvention zur Darstellung von RPC Interaktionen zur Verfügung. Heterogene Clients und Server werden interoperabel, wenn sie diesen einheitlichen Übertragungsmechanismus anwenden.

3.4.3 UDDI (Universal Description, Discovery and Integration)

UDDI stellt eine weltweite Registrierung zur Publikation, Findung und Integration von Web Services zur Verfügung. Man kann UDDI benutzen, um vorhandene Web Services zu finden, indem sie nach Namen, Bezeichnern, Kategorien oder implementierten Spezifikationen der Web Services suchen. Außerdem bietet UDDI eine Struktur zur Darstellung der Art des Business an, der Beziehung zwischen Businesses, der Spezifikationen der Metadaten und des Zugriffspunkts der Web Services.

Die Kombination von SOAP, WSDL und UDDI liefert eine dynamische und einheitliche Infrastruktur für dynamische Businesses.

In der folgenden Abbildung sind die Beziehungen zwischen WSDL, SOAP und UDDI graphisch dargestellt.

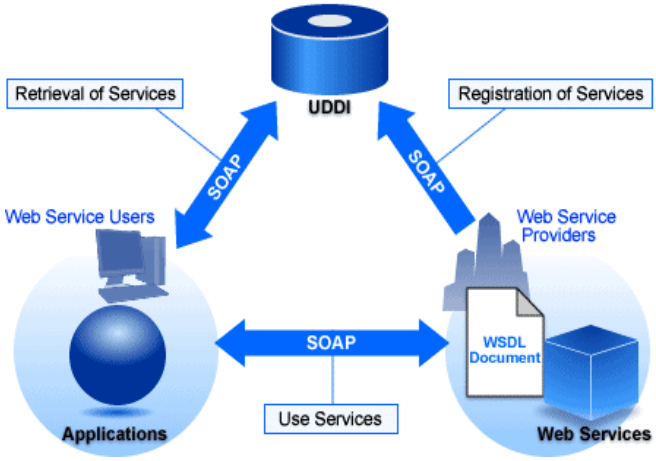


Abb. 3.1 : WSDL, SOAP und UDDI ([FUJ04])

WSDL, SOAP und UDDI werden in einem späteren Abschnitt ausführlicher erklärt.

3.5 Architektur der Web Services

Um das Zusammenarbeiten im Rahmen der Web Services zu ermöglichen, haben Web Services eine schlichte Struktur, die Serviceorientierte Architektur (auf Engl.: *service-oriented architecture* (SOA)) genannt wird. Sie ist im Wesentlichen eine Kollektion von Services, die miteinander kommunizieren. SOA ist ein simples Konzept, so dass es in verschiedenen Situationen der Web Services anwendbar ist.

In jeder serviceorientierten Architektur gibt es drei Hauptrollen, nämlich den Serviceanbieter, den Serviceanforderer und die Registrierung der Services (auf Engl.: *Service Provider, Service Requestor, Service Registry*) ([GSB02]).

Folgende Abbildung zeigt die Hauptrollen in einem SOA und deren Operationen.

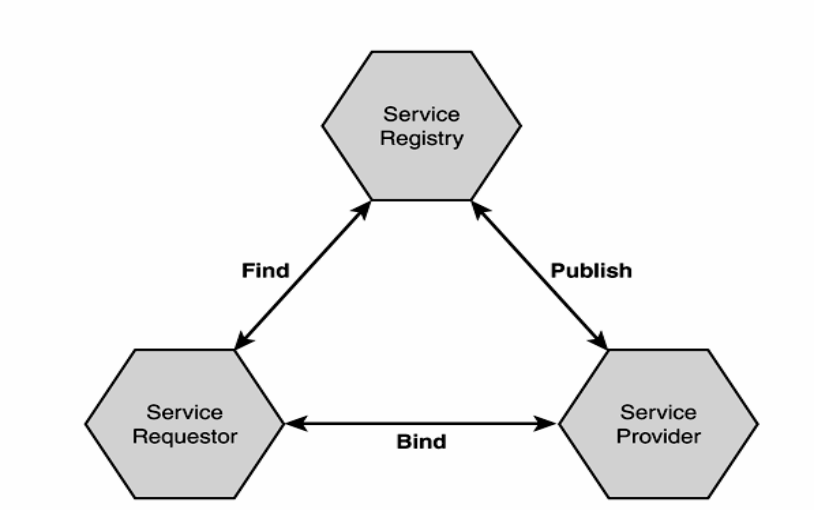


Abb. 3.2 : Serviceorientierte Architektur ([GSB02])

Ein *Service Provider* ist zuständig für das Erstellen der Servicebeschreibung (auf Engl.: *Service Description*), die Publikation der Beschreibung in einer oder mehreren *Registries* und für das Empfangen von Aufrufnachrichten von einem oder mehreren *Service Requestor*. Ein *Service Provider* kann z.B. ein Unternehmen sein, das ein Web Service in bestimmten Netzwerken bereitstellt. Dieser ist vergleichbar mit einem Server in der Client-Server Beziehung.

Ein *Service Requestor* sucht nach den Servicebeschreibungen, die in einer oder mehreren *Registries* publiziert sind, und benutzt diese entsprechend, um die Web Services der *Service Provider* anzufordern. Jeder Benutzer der Web Services ist ein *Service Requestor*. In der Client-Server Beziehung ist dieser vergleichbar zu einem Client.

Die *Service Registry* ist zuständig für das Ankündigen der auf ihr publizierten Servicebeschreibungen, und erlaubt den *Service Requestors*, die in ihr enthaltenen Servicebeschreibungen zu durchsuchen. Wenn eine Servicebeschreibung von einem *Service Requestor* gefunden wird, ist die Rolle der *Service Registry* nicht mehr

wichtig, da der Rest der Interaktion nur noch zwischen dem *Service Provider* und dem *Service Requestor* verläuft.

Jede dieser Rollen kann von irgendeinem Programm oder Knoten innerhalb des Netzwerkes übernommen werden. In bestimmten Fällen kann ein einziges Programm sogar mehrere Rollen haben, z.B. ein Programm, das gleichzeitig den Clients Web Services zur Verfügung stellt und selbst ein von anderen Programmen bereitgestelltes Service anfordert.

Wie in Abbildung 3.2 zu erkennen ist, gibt es drei Operationen in der Serviceorientierten Architektur. Diese sind *publish, find* und *bind*.

Die *publish* Operation wird von einem *Service Provider* ausgeführt, um die Details über seinen Web Services in der *Registry* zu publizieren, die öffentlich erreichbar ist. Damit werden die Web Services bekannt und können dadurch ein größeres „Publikum“ an sich ziehen.

Ein *Service Requestor* führt die Operation *find* in der Registry aus, um Details über die Funktionalität der Web Services zu ermitteln. Mit dieser Operation gibt der *Service Requestor* die Suchkriterien an, wie z.B. den Typ der Services. Die *Registry* vergleicht die angegebenen Kriterien mit ihrer eigenen Kollektion von den in ihr publizierten Beschreibungen der Web Services. Das Ergebnis ist dann eine Liste von Servicebeschreibungen, die den Suchkriterien entsprechen.

Der Komplexität dieser Operation, d.h. Anzahl der Parameter, Formulierung der Suchkriterien, usw., hängt von der Implementation der *Registry* ab. Eine einfache *Registry* kann die Operation *find* mit nur einem ohne Parameter versehenen HTTP GET bereitstellen. Dies führt dazu, dass diese Operation immer alle in der *Registry* publizierten Web Services zurückliefert, und es ist die Aufgabe des *Service Requestors*, die Beschreibung der gesuchten Web Services herauszufinden. Im Gegensatz dazu stellt UDDI einen sehr leistungsfähigen Suchmechanismus zur Verfügung.

Die Client-Server Beziehung wird von der Operation *bind* verkörpert. Diese kann je nach Servicebeschreibungen sowohl komplex und dynamisch sein, wie z.B. *on-the-fly* Generierung eines Proxys auf der Client-Seite, als auch statisch, wobei ein Entwickler den Ablauf eines Serviceaufrufs manuell programmieren muss.

3.6 Schichtstapel der Web Services

Web Services bestehen aus mehreren Schichten. Wenn diese Schichten gemeinsam einen Stapel bilden, ergibt sich die Basis eines einheitlichen Mechanismus zum Ermitteln, Beschreiben und Aufrufen der von einem Web Service bereitgestellten Funktionalität.

In Abbildung 3.3 wird der Schichtstapel der Web Services gezeigt.

3.6.1 Protokollschicht *(Protocol Layer)*

Auf der untersten Ebene ist die Protokollschicht. Diese ist zuständig dafür, dass die Web Services von überall aus erreichbar sind, indem sie die allgegenwärtigen Applikationsprotokolle, wie z.B. HTTP, SMTP und FTP als Übertragungsprotokolle der Web Services einsetzt.

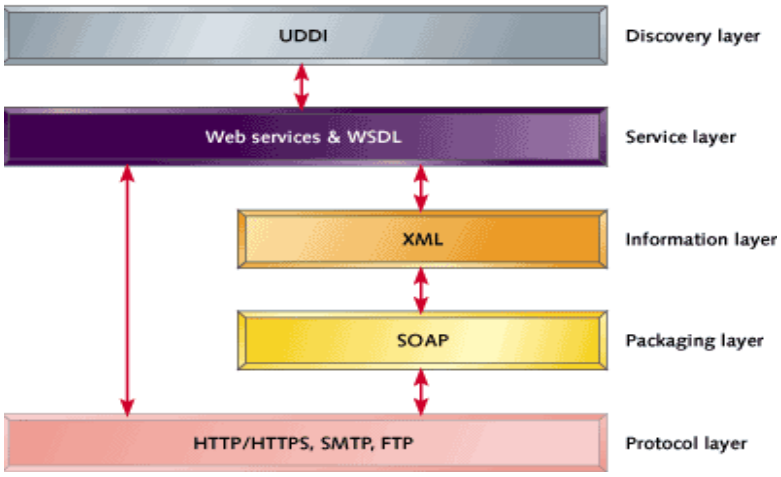


Abb. 3.3: Schichtstapel der Web Services ([CAN04])

3.6.2 Verpackungsschicht (*Packaging Layer)* – SOAP

In dieser Schicht wird SOAP als einheitliches Nachrichtenformat für die Kommunikation zwischen Applikationen eingesetzt. Da SOAP für die verteilten, dezentralisierten Umgebungen vorgesehen ist, stellt es einen Rahmen bereit, der für die Serviceaufrufe über das Internet verwendet werden kann. Dazu besitzt SOAP einen Mechanismus, der plattformübergreifende Integration ermöglicht, die unabhängig ist von den Programmiersprachen und Infrastrukturen der verteilten Objekte.

SOAP definiert weder ein neues Programmierungsmodell noch eine Implementation. Stattdessen definiert es ein modulares Verpackungsmodell und einen Kodierungsmechanismus zum Kodieren von Daten innerhalb der Module. Dies führt dazu, dass SOAP in verschiedenen Systemen einsetzbar ist, wie z.B. in einem Nachrichtenübertragungssystem oder für RPCs.

Abb. 3.4 : Struktur einer SOAP-Nachricht ([FUJ04])

In Abbildung 3.4 ist die Struktur einer SOAP-Nachricht dargestellt. Die SOAP-Nachricht ist ein wohlgeformtes XML-Dokument, das ein Element *Envelope*, ein optionales Element *Header*, und ein Pflichtelement *Body* enthält.

Das Element *Envelope* dient als Behälter für die Elemente *Header* und *Body* und als Indikator, der angibt, dass dieses XML-Dokument eine SOAP-Nachricht ist. Es gibt außerdem den Anfang und das Ende einer Nachricht an.

Das Element *Header* ist optional. Wenn es aber vorhanden ist, dann muss dieses ganz am Anfang der Nachricht stehen. *Header* werden manchmal vom Empfänger der Nachricht ignoriert.

Im Element *Body* steht die eigentliche Nachricht, die übergeben werden soll. Sie kann XML-Elemente enthalten, die einen Prozeduraufruf definieren. Beispielsweise gibt es XML-Elemente, die Prozedurnamen, Argumente und Parameter beinhalten. Es könnte aber auch sein, dass die Nachricht z.B. eine komplette Warenbestellung ist. Diese zwei verschiedenen Modelle werden RPC-orientierte bzw. Dokumentorientierte SOAP-Nachrichten genannt.

Zusätzlich definiert die Spezifikation des SOAP eine Bindung, um SOAP-Nachrichten mithilfe des HTTP-Protokolls zu übertragen. Die SOAP-Nachricht ist unidirektional. Von daher müssen individuelle Nachrichten kombiniert werden, damit der *Request/Response* Mechanismus entsteht.

Die Nutzung des SOAP hat einige Vorteile. Diese sind unter anderen ([BCGH01]):

- SOAP kann eine Firewall durchlaufen, da dieses auf HTTP basiert
- SOAP hat eine XML-Struktur.
- SOAP kann in Kombination mit etlichen Übertragungsprotokollen, wie z.B. HTTP, SMTP und Java Message Service (JMS) benutzt werden
- SOAP wird auf das Request/Response - Muster des HTTP abgebildet
- SOAP ist durch XML erweiterbar
- Viele Anbieter unterstützen SOAP

Es gibt allerdings auch Nachteile, wenn man SOAP als Übertragungsprotokoll einsetzt, z.B.:

- Mangel an Interoperabilität verschiedener SOAP-Toolkits
- Mangel an Sicherheitsmechanismen
- Keine Garantie, ob eine Nachricht erfolgreich geliefert wurde
- Kein publish and subscribe-Modell

3.6.3 Informationsschicht (*Information Layer)* – XML

XML ist eine Metasprache, die durch Nutzung einheitlicher Methoden zur Kodierung und Formatierung von Informationen den plattformübergreifenden Datenaustausch ermöglicht.

Im Rahmen der Web Services ist ein Konzept von höherem Level, das auf XML basiert, sehr wichtig, da XML nur eine Beschreibung über etwas gibt, aber nicht wie dieses behandelt werden kann.

3.6.4 Dienstschicht (*Service Layer*) – Web Servcies und WSDL

Die Schnittstellen der Web Services sind in der XML-basierten Beschreibungssprache der Web Services (auf Engl.: *Web Service Description Language* (WSDL)) definiert. Diese bietet alle nötigen Informationen an, die eine Applikation zum Aufrufen der Services braucht, wie z.B. die Signatur der Services, den Standorten der Services, das für den Prozeduraufruf benötigte Protokoll, usw. Die ganzen Informationen werden in zwei logischen Teilen organisiert. Der erste Teil ist die übertragungsprotokollunabhängige abstrakte Definition eines Web Services und der zweite Teil ist die netzwerkspezifische Bindungsdeskription für die Übertragung [MTSM03].

Abb. 3.5 zeigt die Struktur eines WSDL-Dokumentes.

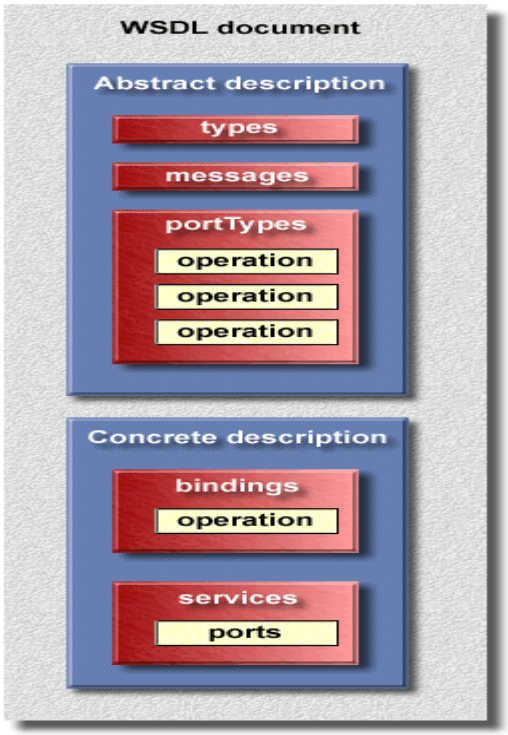


Abb. 3.5: Eine konzeptuelle Repräsentation eines WSDL-Dokumentes

Die Elemente eines WSDL-Dokumentes sind

Data types

Data types sind die plattform- und sprachenunabhängigen Definition der Datentypen in der Form eines XML Schemas oder anderen Mechanismus. Diese werden in der *message* verwendet.

Message

Message bietet eine abstrakte Definition der Daten. Die Daten können aus einem ganzen Dokument oder aus Argumenten, sowie aus Input- bzw. Outputparametern eines Prozeduraufrufs bestehen. Dieser ist vergleichbar mit der Signatur einer Methode.

Operation

Operation stellt bestimmte Interaktionen mit den Services dar und beschreibt die möglichen Eingaben, Ausgaben und Ausnahmen während der Interaktion. In der Programmiersprache ist dies mit einem Prozeduraufruf vergleichbar.

Port type

Eine abstrakte Menge von Operationen wird auf einen oder mehrere Endpunkte (*ports*) abgebildet. Ein Endpunkt stellt die von dem Service unterstützte Menge von Operationen dar. Diese Menge ist vergleichbar mit einer Schnittstelle in Java.

Binding

Binding spezifiziert die Bindung für jede Operation in dem *port type*-Element und ordnet die abstrakte Deskription eines *port types* zu einem Protokoll, wie z.B. HTTP zu.

Port

Port ist eine Kombination von einer Bindung und einer Netzwerkadresse, die die Zieladresse der Servicekommunikation angibt.

Service

Services stellen eine Kollektion von *ports* zur Verfügung.

Die logische Trennung der abstrakten und konkreten Informationen ermöglicht die Wiederverwendung des abstrakten Teils der Web Services mit ganz anderen Implementationen.

3.6.5 Ermittlungsschicht (*Discovery Layer*)

Die optionale Schicht im Protokollstapel der Web Services ist die Spezifikation der *Universal Description, Discovery, and Integration (UDDI).* UDDI bietet die Möglichkeit, Informationen über Web Services gleichzeitig zu publizieren und auch zu ermitteln.

Im Wesentlichen ist UDDI ein Registrierungssystem, das durch eine Reihe von XML-Dateien und ein assoziiertes Schema realisiert wird und eine Deskription einer Businessentität und deren Services enthält. Außerdem bietet die Spezifikation des UDDI eine programmatische Schnittstelle an, die es erlaubt, ein Web Service zu registrieren und/oder durch die *Registry* nach bestimmten Web Services zu suchen. Wenn das gesuchte Web Service identifiziert ist, wird ein Zeiger auf dem Standort des WSDL-Dokumentes gelegt. Es ist wichtig zu wissen, dass UDDI nur optional ist. Unternehmen, deren Web Services nur für bestimmte Leute bzw. Geräten vorgesehen sind, müssen diese nicht bekannt geben.

Abb. 3.6: Struktur des UDDI ([FUJ04])

Die UDDI *Registry* funktioniert wie ein Telefonbuch. Sie beinhaltet Informationen, die in drei Kategorien klassifiziert werden können [CHA02].

White pages

White pages enthalten die Kontaktinformationen und die Identifikationen eines Unternehmens, wie z.B. den Geschäftsnamen, die Adresse, die Steuernummer, usw. Dies wird für die Suche nach der Identifikation des Business verwendet.

Yellow pages

Informationen, die Web Services nach verschiedenen Kategorien beschreiben, sind in den *Yellow Pages* beschrieben.

Green pages

Green pages beinhalten technische Informationen, die das Verhalten und die unterstützen Funktionen eines Web Services beschreiben.

Aus Sicht eines Nutzers der Web Services hat ein UDDI Registry vier Arten von Informationen. Diese sind ([MTSM03]):

Business entity

Jede Businessentität besitzt einen eindeutigen Bezeichner, den Namen des Business, simple Kontaktinformationen, eine kurze Beschreibung des Business, eine Liste von Kategorien, die dieses Business beschreiben und klassifizieren, und ein URL, das auf zusätzliche Informationen über das Business zeigt.

Business service

Jede Businessserviceentität besitzt eine Beschreibung des Services, eine Liste von Kategorien, die dieses Service beschreiben und klassifizieren, und ein URL, das auf zusätzliche Informationen über das Service zeigt.

Specification pointers

Zusätzlich hat jede Businessserviceentität eine Liste von Bindungsvorlagen und Informationen, die das Service mit einem Servicetyp assoziieren.

Service types

Ein Servicetyp wird durch ein technisches Modell (kurz: tModell) definiert. Ein tModell definiert die Informationen über die Services und einen Zeiger, der auf die Spezifikation des Servicetyps zeigt.

4. Ein Generator für eine Web Services-Schnittstelle für konzeptorientierte Inhaltsverwaltungssysteme

4.1 Allgemeine Entwurfmuster

In dieser Arbeit sind mehrere Entwurfmuster [GHJV94] zum Einsatz gekommen. Davon sind mehrere Entwurfmuster im Framework bereits vorgegeben. Das Fabrikmuster, Iteratormuster und Besuchermuster, die in den folgenden Unterabschnitten zu sehen sind, sind einige Beispiele. Zusätzlich wird das Adaptermuster verwendet, um die Schnittstelle `java.util.Iterator` an der generischen Schnittstelle `AssetIterator` anzupassen.

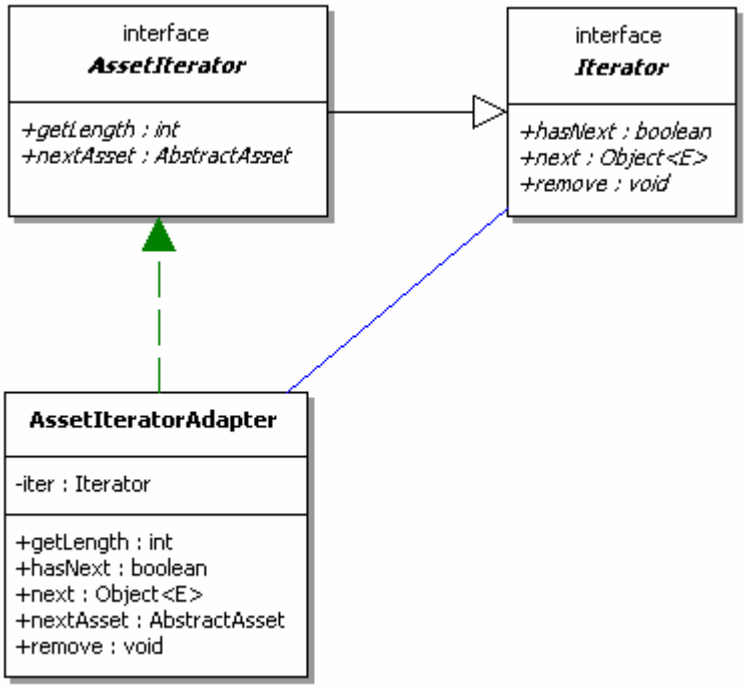


Abb. 4.6: Adaptermuster

Dieses Muster wird z.B. benötigt, wenn die Methode einen `AssetIterator` zurückliefern muss, aber als Argument einen herkömmlichen `java.util.Iterator` bekommt. Durch diese Entwurfmuster sind sowohl generische als auch generierte Klassen abgebildet.

4.2 Basis der Modul-Generierung

Dieses Kapitel widmet sich der ersten Stufe der Modul-Generierung, nämlich der Generierung des Moduls in Form der eingesetzten Programmiersprache (Java). Die nächste Stufe der Generierung, die eine WSDL-Beschreibung aus dem Modul generiert, wird im Kapitel 5 erörtert.

Um einen Modul-Generator zu implementieren, ist es wichtig, die grundlegenden Aspekte beim Implementieren eines Generators zur Kenntnis zu nehmen. Im Abschnitt 2.6 ist zu erfahren, dass das Backend des Modellcompilers aus verschiedenen Generatoren besteht. Diese Generatoren können miteinander kommunizieren, indem sie einander die so genannte Symboltabelle (auf Engl.: *Symbol Table*) austauschen. Im Fall des API-Generators beinhaltet die Symboltabelle z.B. Methoden, durch deren Aufruf ein Generator nicht nur die Namen der aus den vom Nutzer definierten Modellen generierten Klassen, sondern auch deren Meta-Objekte bekommen kann. Jeder Generator kann eine beliebige Anzahl von Symboltabellen nutzen, erstellt aber nur genau eine Symboltabelle, deren Typ vom erstellenden Generator abhängt. Jede Symboltabelle ist von der Klasse `SymbolTable` abgeleitet. Diese Klasse und viele anderen Klassen und Schnittstellen sind Teile der zur Verfügung gestellten Bibliotheken des Frameworks.

Ein Generator könnte außerdem bestimmte Parameter benötigen. Parameter werden in einer XML-basierten Konfigurationsdatei des Frameworks definiert, die vom Generator zur Laufzeit benutzt wird. Die vordefinierte Methode `getRequestedParameter` gibt die vom Generator benötigten Parameter zurück.

Wenn mehrere Generatoren vorhanden sind, dann werden diese vom Modellcompiler koordiniert, und zwar bzgl. der Reihenfolge, in der die Generatoren laufen sollen und des Austausches der Symboltabellen zwischen den Generatoren. Abbildung 4.1 zeigt ein Beispiel dieses Ablaufs. Es stehen zwei Generatoren zur Verfügung, die jeweils bestimmte Symboltabelle und Parameter benötigen. Anschließend generieren diese ihre eigene Symboltabelle.

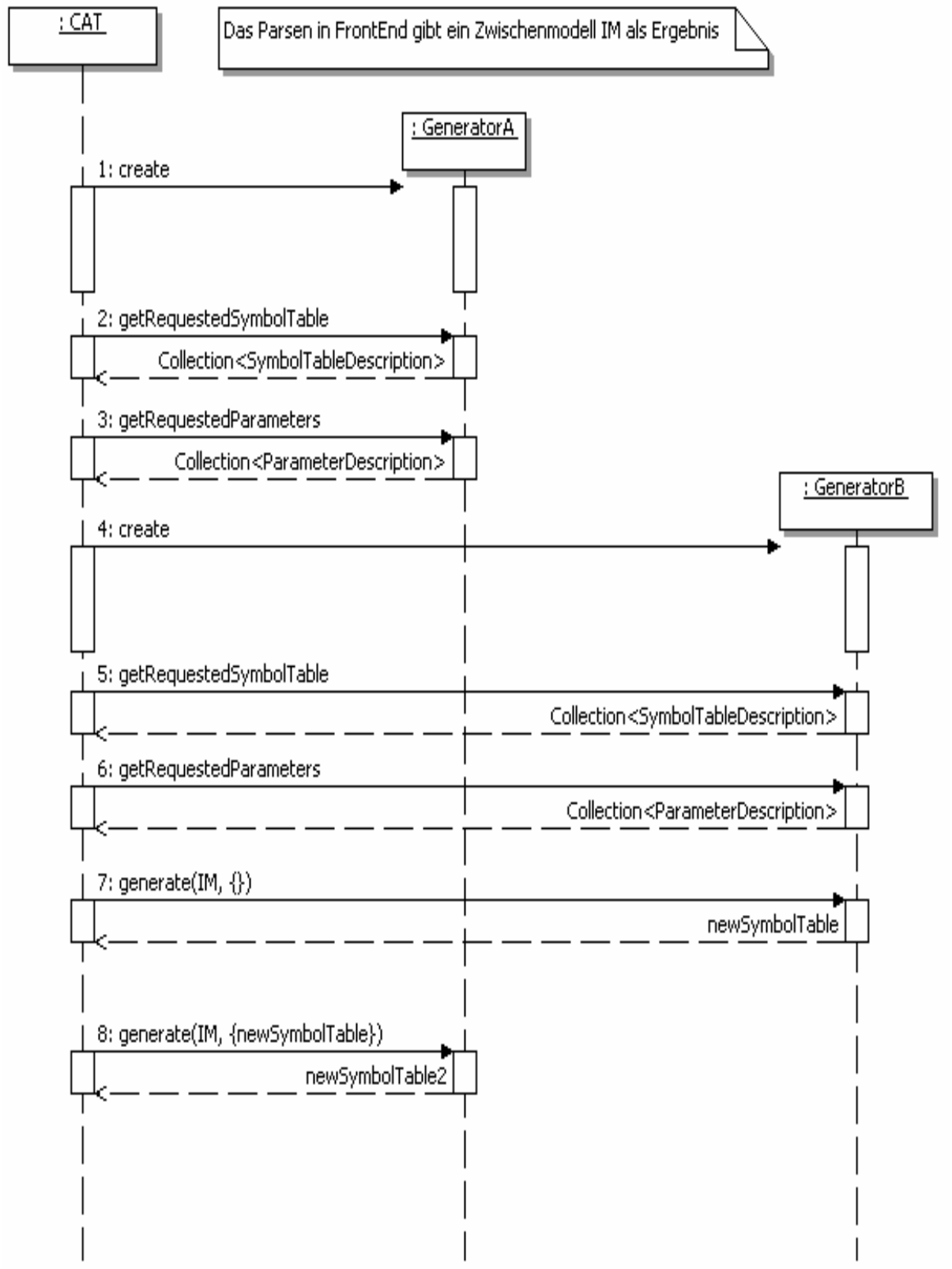


Abb. 4.1: Koordination der Modul-Generatoren

4.2.1 Die zu implementierenden Methoden der Basisklasse

Es wurde am Anfang erklärt, dass man einen Generator implementieren muss, um das Asset Compiler Framework um eine neue Modulart zu erweitern. Das Framework hat bereits etliche vordefinierte Klassen, mit denen man die Implementierung eines Generators beginnen kann. Hierzu ist der neue Generator von der abstrakten Klasse `Generator` abzuleiten. Generator beinhaltet einige Methoden, die von den Unterklassen, also den Modul-Generatoren, (neu) implementiert werden sollen. Dazu zählen:

- `getRequestedSymbolTable`
  Diese Methode gibt den Namen und Typen der von dem Generator benötigten Symboltabelle an.
- `getProducedSymbolTable`
  Diese Methode gibt an, wie die von diesem Generator erstellte Symboltabelle und deren Typ heißen.
- `getRequestedParameter`
  Um ausgeführt werden zu können, brauchen Generatoren bestimmte Parameter. Anhand dieser Methode weiß der Compiler, welche Parameter er benötigt.
- `generate`
  `generate` ist die Hauptmethode jedes Generators. Hier werden das Zwischenmodell und die benötigte Symboltabelle geholt. Als Ergebnis wird eine Symboltabelle dieses Generators zurückgegeben, wie in Abbildung 4.1 zu sehen ist. Außerdem werden die gewünschten Klassen bzw. Schnittstellen generiert.

Es gibt noch weitere Methoden, die zum Anzeigen des Ablaufs des Generierungsprozesses dienen, z.B. `addProgressListener` und `removeProgressListener`. Diese beiden Methoden registrieren so genannte Beobachter (auf Engl.: *Observer*), die zusammen mit `ProgressEvent` bereitgestellt

sind. Solche Beobachter sind eine Implementierung der Schnittstelle `ProgressListener`. Wenn der Prozess Fortschritte gemacht hat, dann wird der Generator die Beobachter darüber informieren.

4.2.2 Modulschnittstelle eines API-Generators

Üblicherweise läuft der API-Generator zeitlich vor anderen Generatoren, da dieser die Java-Schnittstelle für die spezifischen Schnittstellen generiert, die von allen Modulen erfüllt werden müssen.

Jeder Generator erstellt eine Symboltabelle. Wie bereits erwähnt, erstellt API-Generator auch eine Symboltabelle (`APISymbolTable`). Anhand dieser Symboltabelle können Generatoren Methoden aufrufen, die ihnen dann die aus Modellen generierten Java Meta-Objekte und deren Methoden zurückliefern. Die aufrufbaren Methoden dienen unter anderen zum Herbeiholen der im Unterabschnitt 2.6.2 beschriebenen Zustandsschnittstellen und anderen Schnittstellen.

Für die Zustandsschnittstellen stehen folgende Methoden zur Verfügung.

- `getAbstractInterface`
  gibt die Schnittstelle für alle Assetszustände zurück
- `getVolatileInterface`
  gibt die *volatile* - Schnittstelle zurück
- `getAbstractMutableInterface`
  gibt die Basisschnittstelle für alle *mutable*-Zustände zurück
- `getLockedInterface`
  gibt die *mutable* - Schnittstelle zurück
- `getPersistentInterface`
  gibt die *persistent* - Schnittstelle zurück

Methoden, die die Factory-, Iterator-, Anfrage-, und Besucherschnittstelle liefern, sind

- `getFactoryInterface`
  gibt ein *factory* - Objekt zurück zum Kreieren neuer Instanzen
- `getIteratorInterface`
  gibt die *iterator* - Schnittstelle einer Menge von Assets zurück
- `getQueryInterface`
  gibt die *query* - Schnittstelle der angegebenen Klasse zurück
- `getTypeVisitor`
  gibt den Besucher für die Subtypen der angegebenen Klasse zurück

4.3 Generierung eines Web Services-Modul als Implementierung der Web Services-Schnittstelle

Die Grundlagen für die Erstellung neuer Generatoren sind bereits erläutert worden. Als nächstes überleget man sich, was ein Generator im Endeffekt zu generieren hat, also die Aufgabe(n) des Generators. Für diese Projektarbeit lautet die Aufgabe, einen Generator zu entwickeln, der den Zugriff auf ein bereits oben definierte konzeptorientiertes Inhaltsverwaltungssystem gemäß Web Services ermöglicht, also eine Serviceorientierte Architektur auf Basis eines Inhaltsverwaltungssystems. Dazu werden aus den Assetdefinitionen Schnittstellen generiert, die dann in der *Web Services Description Language (WSDL)* bereitgestellt sind. Diese bilden das Web Services-Modul (WS-Modul), das für die Interaktion im Sinne der Web Services die Hauptrolle spielt. In den folgenden Unterabschnitten wird die für die Erstellung nötige Vorgehensweise ausführlich erklärt. Die Realisierungsmöglichkeit der Schnittstellenbeschreibung in WSDL wird im nächsten Kapitel vorgestellt.

4.3.1 Allgemeiner Aufbau des Web Services-Moduls

Im Kapitel 2 ist zu sehen, dass es verschiedene Arten von Modulen gibt. Im Fall des Web Services-Generators ist eine Zusammenarbeit mit den anderen Modulen notwendig, weil das WS-Modul nur eine Schnittstelle nach außen zum Kreieren bzw. Modifizieren von Assets darstellt. Die von der Schnittstelle angebotenen Methoden beruhen tatsächlich auf der Implementierung eines bestimmten Basismoduls. Aus diesem Grund muss es eine Methode geben, durch die das Basismodul mit dem WS-Modul in Verbindung gebracht wird.

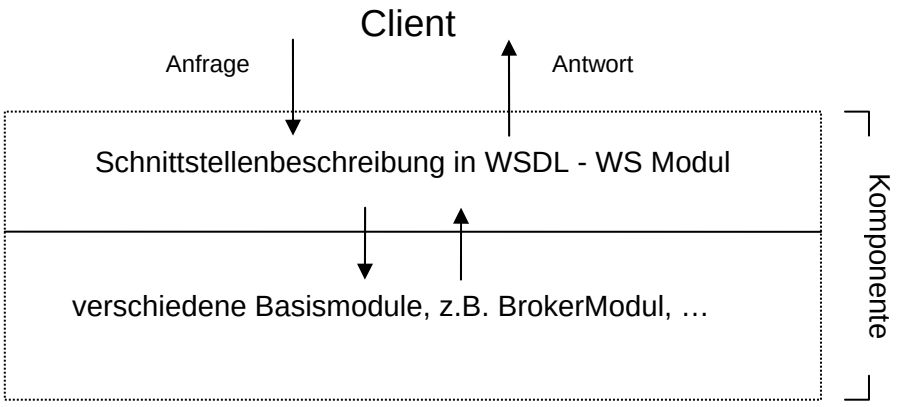


Abb. 4.2: WS-Modul auf Basis anderer Module

Einige Module, die zusammenarbeiten, bilden eine Komponente auf, die durch die Klassen `Component` dargestellt ist. Die Komponente stellt die Methode `getBaseModule` zur Verfügung, mit deren Hilfe ein Modul ein darunterliegendes Modul, also das Basismodul, holen kann. Dieses Modul kann anschließend die angefragten Assets zurückliefern. Kode-Beispiel 4.1 zeigt die Methode `getBaseModule`.

```
private Component comp;

public de.tuhh.sts.cocoma.generic.AssetClass getClass(java.lang.String name) {
          de.tuhh.sts.cocoma.generic.Module baseModule = comp.getBaseModule(this);
          return baseModule.getClass(name);
     }
```

Kode-Beispiel 4.1: `getBaseModule` liefert das Basismodul

4.3.2 Generische zu implementierende Schnittstelle mit Operationen aus der Assetmanipulationssprache und Assetanfragesprache

Das Framework stellt eine Menge von vordefinierten Klassen bzw. Schnittstellen bereit, um Integration in das Framework zu ermöglichen. Darunter gibt es die Schnittstelle `Module`, die etliche zu implementierenden Methoden definieren, wenn ein neues Modul generiert werden soll. Davon abgeleitet ist die Schnittstelle `ServerModule`, die für diese Arbeit vorgesehen ist, und deren Implementierung je nach angegebenen Modellen dem Generator überlassen werden soll.

Die Methoden dieser Schnittstelle sind im Grunde genommen die im konzeptorientierten Inhaltsverwaltungssystem definierten Operationen, nämlich die Standardoperationen von Assetmanipulationssprache und Assetanfragesprache `create`, `modify`, `delete` und `lookfor` hinsichtlich der aktuell angegebenen Assetdefinition. Die Implementierungen der Methoden, die diesen Operationen entsprechen, werden von `ServerModule` als Web Services-Schnittstelle nach außen angeboten.

Es werden verschiedene Varianten der Operationen implementiert, um die möglichen Anfragemodelle eines Clients behandeln zu können. Aus der Assetmanipulationssprache sind die Operationen `create`, `modify`, `delete` bekannt.

`Asset create (AssetClass ac, AttributeInitialization [] arg)`
Erzeugen eines Assets mit vordefinierten Attributen

`Asset create (AssetClass ac, AbstractAsset arg)`
Erzeugen eines Assets nach angegebenem Prototyp

`AssetIterator create (AssetClass ac, AssetIterator arg)`
Erzeugen einer Menge von Assets gemäß Prototypen

`Asset modify (Asset a, AttributeInitialization [] arg)`
Modifizieren eines Assets mit neuen Attributen

`Asset modify (Asset a, AbstractAsset arg)`
Modifizieren eines Assets nach angegebenem Prototyp

`AssetIterator modify(AssetIterator ai,AttributeInitialization[]arg)`
Modifizieren einer Menge von Assets mit neuen Attributen

`AssetIterator modify (AssetIterator ai, AbstractAsset arg)`
Modifizieren einer Menge von Assets nach angegebenem Prototyp

`NewAsset delete (Asset a)`
Löschen eines Assets

`AssetIterator delete (AssetIterator a)`
Löschen einer Menge von Assets

Die Assetanfragesprache hat die `lookfor`-Operation. Folgende sind die entsprechenden Methoden.

`Asset lookfor (ID id)`
Suchanfrage nach Asset mit angegebener ID

`AssetIterator lookfor (AssetClass a, QueryConstraint[] arg)`
Suchanfrage nach Asset mit bestimmten Bedingungen der Anfrage

`AssetIterator lookfor (AssetClass a, AbstractAsset arg)`
Suchanfrage nach Asset mit angegebenem Prototyp

`AssetIterator lookfor (AssetClass a, AssetIterator arg)`
Suchanfrage nach einer Menge von angegebenen Assets

Andere Methoden dienen z.B. zum Initialisieren, Starten oder Stoppen des Moduls und, wie oben bereits erwähnt, auch zum Aufrufen eines Basismoduls.

4.3.3 Generierte Methodenimplementierung einer Web Services-Schnittstelle

Die offene Dynamik wird dadurch realisiert, dass die Implementierung gemäß den angegebenen Modellen generiert wird. Dafür gibt es das so genannte Werkzeug zur Generierung von Java Code (auf Engl.: *Java Code Generation Toolkit (JCGT)*), das einem Programmierer eine praktische Generierung von Code ermöglicht. Dieses Vorgehen wird als Metaprogrammierung bezeichnet, wobei der Code (in diesem Fall das des Moduls) von anderem Programmcode (Generator) erzeugt wird [WIKI05]. Das JCGT stellt Java Klassen bzw. Schnittstellen bereit, die mit den Klassen der Java Standardbibliotheken `java.lang.reflect` vergleichbar sind. Aber im Gegensatz zu diesen bietet es auch die Möglichkeiten an, neue Klassen anzulegen und zu manipulieren.

An dieser Stelle werden zuerst die Varianten der vier Hauptmethoden untersucht. Bei den Implementierungen der Methoden wird sich der Generator die vom API-Generator bereitgestellten Methoden zu Nutze machen, weil erst durch diese auf die benötigten Informationen zugegriffen werden kann.

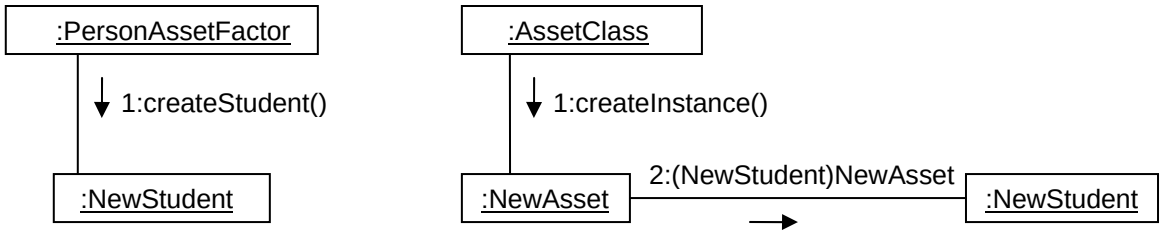
Damit die Erklärungen der Methoden besser zu verstehen sind, wird das folgende Beispielmodell angelegt.

```
model Person
class Student {
    concept characteristic name    : String
            relationship   friend  : Student
          }
```

Kode-Beispiel 4.2: Beispielmodell

4.3.3.1 Methode zum Anlegen neuer Assetinstanzen

Die `create`-Methode wird benutzt, um eine neue Instanz anzulegen. Um von der Implementierung der Erzeugung zu abstrahieren, wird das Entwurfmuster **Fabrikmethode** [GHJV94] umgesetzt. Z.B. kann eine Instanz von `Student` durch Verwendung der Methode `createStudent` der Fabrik `PersonAssetFactory` erzeugt werden, wobei `Person` eine Superklasse von `Student` ist. Da man aber in diesem Fall die generelle `AssetClass` als Argument der Methode übergeben bekommt, kann auch alternativ deren `createInstance`-Methode zuerst benutzt werden. Danach wird die neue Instanz in die tatsächliche Klasse typumgewandelt.



(a) (b)

Abb. 4.3: (a) Erzeugung neuer Instanz direkt aus der „Fabrik“ der Superklasse (b) Unter Benutzung der Methode der AssetClass

Als Erstes muss immer eine neue Instanz erzeugt werden. Je nach Variante der Methode werden dann der Instanz neue Attribute aus einem Array mit Basistyp `AttributeInitialization` (*Intentional Create-Method*) oder Attribute eines vorhandenen Prototyps (*Extensional Create-Method)* hinzugefügt. Das Array dient zum Kapseln von angegebenen Attributen, das für jede Instanz durchgelaufen wird, um die Attribute einzusetzen. Dafür ist eine Methode `getAttributeName` definiert, die den Namen eines Attributes zurückliefert.

Bei der intentionalen `create`-Methode muss man unterscheiden, ob ein Attribut vom Typ `characteristic` oder `relationship` ist. Für beide Typen gibt es eine entsprechende Klasse, zu der ein Attribut aus dem `AttributInitialization` zuerst typumgewandelt wird. Wenn es eine Charakteristik ist, wird die Klasse `CharacteristicInitialization` initialisiert, dann kann dessen Wert direkt als Attribut der neuen Instanz gesetzt werden. Im Fall einer Beziehung kommt die Klasse `RelationshipInitialization` zum Einsatz. Es wird eine weitere Unterscheidung gemacht, und zwar ob es sich um eine 1:n- oder m:n-Beziehung handelt. Bei einer 1:n-Beziehung wird entsprechend die `setter`-Methode verwendet, um die referenzierten Assets in Verbindung zu setzen. Ansonsten verwendet man die für eine m:n-Beziehung vorgesehene `adder`-Methode.

In beiden Fällen ist eine `accept`-Methode eingesetzt, durch die ein Besucher zur Unterscheidung der Lebenslagen (eine Implementierung der Schnittstelle `LifeCycleVisitor`) benutzt werden kann, der eine Fallunterscheidung aufgrund des Zustandes im Lebenszyklus eines Assets macht. Die Fallunterscheidung ist hier nötig, da `RelationshipInitialization` ein `AbstractAsset` liefert, so dass man aus dieser Abstraktion verschiedene mögliche Zustände behandeln muss.

Dafür hat der Besucher je eine `visit`-Methode für die verschiedenen Zustände, also jeweils mit einem Parameter vom Typ `Persistent`, `Locked` und `Volatile`. Je nach Methode wird das entsprechende Vorgehen unternommen: ist ein Asset vom

Typ `Persistent`, muss das Asset zuerst in den gesperrten Zustand `Locked` gebracht werden, bevor es als Attribut eingesetzt werden kann. In diesem Zustand ist das Asset für eine exklusive Bearbeitung gesperrt, d.h. dieses Asset kann während dieser Sperrung von keinem anderen verarbeitet werden. Wenn ein Asset bereits den Typ `Locked` hat, dann kann dies gleich eingesetzt werden. Im Fall des `volatile`-Zustands kann keine Sperrung erfolgen und somit ist ein Asset in diesem Zustand nicht einsetzbar.

Eine dritte Variante der Methode bekommt einen Iterator einer bestimmten Kollektion von Assets als Argument übergeben. Diese Methode verwendet dann die oben genannte extensionale `create`-Methode, um die vorhandenen Assets beim Erzeugen neuer Assets nacheinander zu verarbeiten.

Am Ende jeder `create`-Methode wird die neue Instanz durch den Aufruf der `store`-Methode persistent gemacht. Kode-Beispiel 4.3 zeigt eine Variante der Methode.

4.3.3.2 Methode zum Modifizieren von Assets

Mit der `modify`-Methode kann ein Asset bzw. können dessen Attribute modifiziert werden. Es wurde im Kapitel 1 gezeigt, wie sich die Lebenslagen der Assets durch Anwendung verschiedener Methoden ändern können. Diese Erkenntnis wird in den `modify`-Methoden eingesetzt, um transaktionsartige Operationen durchzuführen. Da von „offenen“ Web Services gesprochen wird, ist es möglich, es mit mehreren nebenläufigen Anfragen auf ein Asset zu tun zu haben. Dabei ist es sehr wichtig zu beachten, dass die Eigenschaften der Transaktion (Atomarität, Konsistenz, Isolation und Dauerhaftigkeit-auf Engl.: *ACID: Atomacity, Consistency, Isolation, Durabilty*) erfüllt werden.

```
public Asset create ( AssetClass assetCls, AttributeInitialization[] attrInit) {
   if ("Student".equals(assetCls.getName())) {
     final NewStudent newStudent = (NewStudent) assetCls.createInstance();
     for (int j = 0; j < attrInit.length; j++)
        if ("friends".equals((attrInit[j]).getAttributeName())) {
          RelationshipInitialization ri = RelationshipInitialization)(attrInit[j]);
          AbstractStudent abstStudent = (AbstractStudent) ri.getValue();
          Throwable e = (Throwable)abstStudent.accept(new LifeCycleVisitor() {
             public Object visit(Asset asset) {
               if (!(asset instanceof Student))
                  throw new Error();
               MutableStudent mutStu;

               try {
                  mutStu = ((MutableStudent) asset  .lockAsAsset());
                    } catch (Throwable e) {
                          return null;}
               newStudent.addFriends(mutStu);

               try {
                   asset = mutStu.abortAsAsset();
                    } catch (StateException exc) {
                          return null;}

                    return null;
             }
             ......
        } else if ("age".equals((attrInit[j]).getAttributeName())) {
          CharacteristicInitialization    ci    =    (CharacteristicInitialization)
          (attrInit[j]);
          newStudent.setAge(((Integer) ci.getValue()).intValue());
     ... ... ...
     return newStudent.store();
```

Kode-Beispiel 4.3: `create`-Methode

Die Varianten der Methoden sind den `create`-Methoden ähnlich. Hier wird das Modifizieren der Assets aber grundsätzlich nur innerhalb einer privaten `modify`-Methode definiert. Das Einsetzen von Attributen erfolgt nach Unterscheidung der Lebenslagen eines als Attribut einzusetzenden Assets, die in der `create`-Methode (Kode-Beispiel 4.3) auch zu sehen ist, innerhalb dieser Methode. Kode-Beispiel 4.4 zeigt die `private modify`-Methode.

Auf die `private modify`-Methode greifen die anderen modify-Methoden zu, die z.B. für das Modifizieren eines einzelnen Assets oder einer Menge von Assets benutzt werden. Vorher muss das Asset für die Bearbeitung durch die Methode `lockAsAsset` gesperrt werden, bevor es an die `private modify`-Methode übergeben wird, da diese nur Argument vom Typ `MutableAsset`, was dem Zustand `Locked` entspricht, annimmt. Nach dem Modifizieren wird dieses Asset durch die Methode `commitAsAsset` oder `abortAsAsset` wieder freigegeben.

Wie beim Kreieren des Assets der Fall ist, macht man auch hier die Fallunterscheidung zwischen Charakteristik- und Beziehungs-Attributen, so wie zwischen 1:n- und m:n-Beziehungen. Man soll auch bedenken, dass ein zu modifizierendes Asset eventuell schon Beziehungen hat, die aber nach einer Modifikation nicht mehr existieren. Solche Beziehungen muss man explizit löschen, indem man die `remove`-Methode aufruft.

```
private MutableAsset privateModify(MutableAsset mutAsset, AttributeInitialization[]
attrInit) {
     if ("Student".equals(mutAsset.getType().getName())) {
          final MutableStudent mutStudent = (MutableStudent) mutAsset;
          for (int j = 0; j < attrInit.length; j++)
               if ("friends".equals((attrInit[j]).getAttributeName())) {
                    RelationshipInitialization ri = (RelationshipInitialization)
                    (attrInit[j]);
                    AbstractStudent abstStudent = (AbstractStudent) ri.getValue();
                    if (!mutStudent.hasFriends(abstStudent)) {
```

```
                    Throwable e = (Throwable) abstStudent.accept(
                        new LifeCycleVisitor() {
                         public Object visit(Asset asset) {
                              if (!(asset instanceof Student))
                                   throw new Error();
                              MutableStudent mutStu;
                              try {
                                   mutStu = ((MutableStudent)asset.lockAsAsset());
                                   } catch (Throwable e) {
                                        return null;    }
                              mutStudent.addFriends(mutStu);
                    ......
               } else if ("age".equals((attrInit[j]).getAttributeName())) {
                    CharacteristicInitialization               ci               =
                    (CharacteristicInitialization) (attrInit[j]);
                    ......
          StudentIterator iterStudent = mutStudent.getFriends();
          while (iterStudent.hasNext()) {
               AbstractStudent absStu = iterStudent.nextStudent();
               boolean found = false;
               for (int j = 0; j < attrInit.length; j++) {

                    if (((attrInit[j]) instanceof RelationshipInitialization)
                    && ((attrInit[j]).getAttributeName().equals("friends")
                    && (((RelationshipInitialization) (attrInit[j])).getValue() ==
                    absStu)))
                         {found = true;  break;}
                    if (!found)
                         absStu.accept(new LifeCycleVisitor() {
                         public Object visit(Asset asset) {
                           MutableStudent mutStu;
                           try {
                              mutStu = ((MutableStudent) asset.lockAsAsset());
                               } catch (Throwable e) {return e;}
                           mutStudent.removeFriends(mutStu);
                           return null;
                         }
                         ......
     return mutStudent;
}
```

Kode-Beispiel 4.4: `private modify`–Methode

Im Kode-Beispiel 4.5 ist zu sehen, wie der oben genannte Ansatz mit den `lockAsAsset-`, `privateModify-` und `commitAsAsset-`Methoden zum Einsatz gekommen ist.

```
public Asset modify(Asset asset, AttributeInitialization[] attrInit) {
          MutableAsset mutAsset;
          try {
               mutAsset = asset.lockAsAsset();
          } catch (StateException exc) {
               return null;
          }
          MutableAsset ma2 = this.privateModify(mutAsset, attrInit);
          Asset a2;
          try {
               a2 = ma2.commitAsAsset();
          } catch (StateException exc) {
               return null;
          }
          return a2;
}
```

Kode-Beispiel 4.5 : `modify` - Methode, die auf `private modify` zurückgreift

Die Idee beim Modifizieren einer Menge von Asset ist, die Assets immer als eine Menge zu sperren, zu bearbeiten und sie am Ende wieder freizugeben, anstatt ein einzelnes Asset nacheinander diesen dreistufigen Prozess durchlaufen zu lassen. Dadurch soll die Förderung nach der Eigenschaft der Transaktion „Atomarität“ erreicht werden. Um dieses Konzept zu veranschaulichen, wird folgendes Sequenzdiagramm gezeigt:

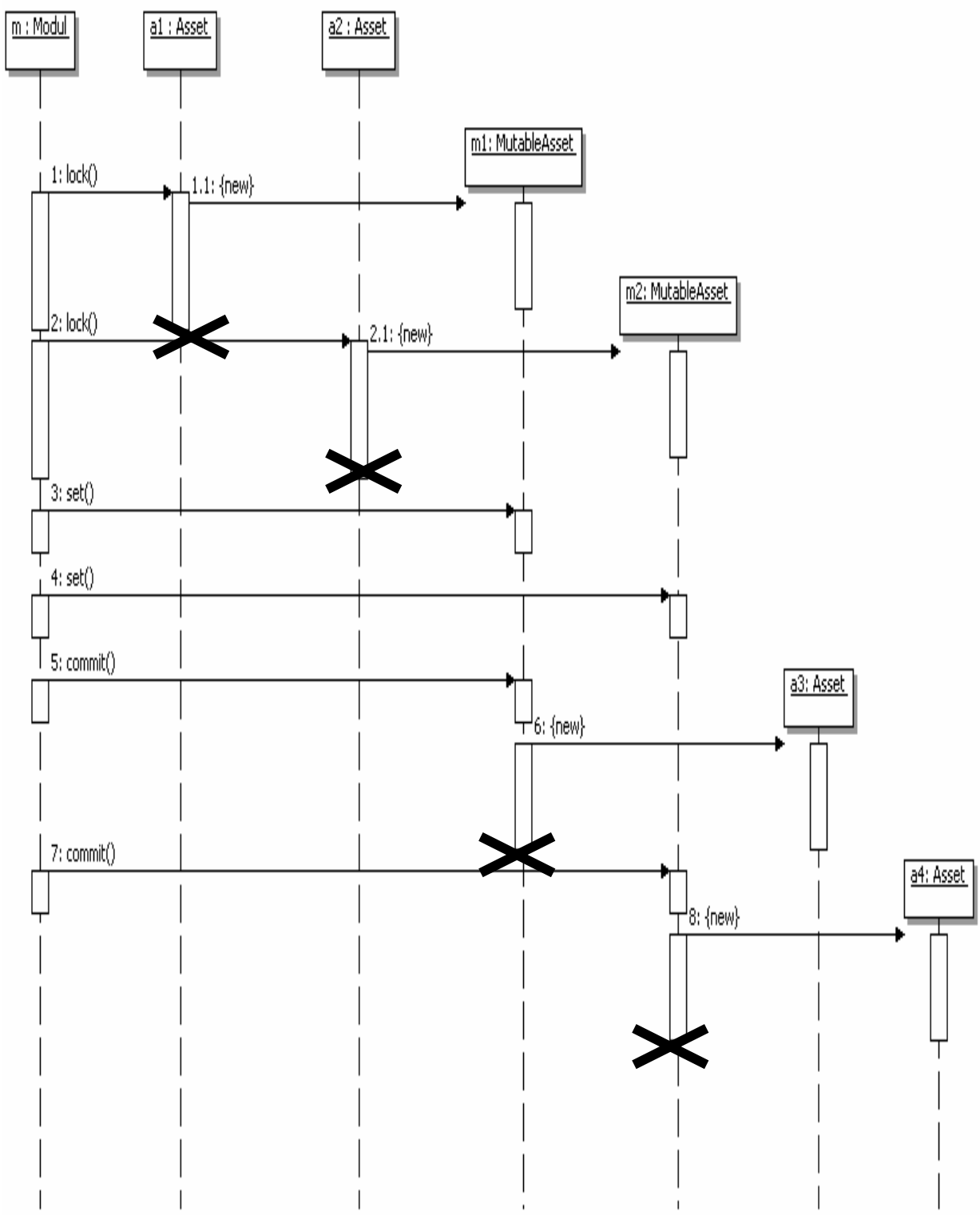


Abb. 4.4: Sequenzdiagramm des Ablaufes der Modifizierung einer Menge von Assets

Wie es aus dem Bereich der Datenbanken bekannt ist, könnten auch Probleme wie *Deadlock*, fehlgeschlagene Operationen, usw. auftreten. Folgendes Szenario kann man sich vorstellen:

`StudentA →addFriends → StudentB`

```
mutStuA = stuA.lock();
```

```
mutStuB = stuB.lock();
newStuB = mutStuB.delete();
```

```
mutStuB = stuB.lock();
mutStuA.addFriends(mutStuB);
stuA = mutStuA.commit();
stuB = mutStuB.abort();
```

Kode-Beispiel 4.6: Nebenläufige Transaktionen

Ein Asset `StudentB` soll als Attribut von einem anderen Asset `StudentA` gesetzt werden. Dafür wird zuerst `StudentA` gesperrt. Gleichzeitig wird aber `StudentB` von einem anderen User des Systems gesperrt, der diesen dann löschen möchte. Daraus ergibt sich die Situation, dass `StudentB` nicht mehr als Attribut gesetzt werden kann.

Man versucht diese Situation dadurch zu verhindern, dass man auch das als Attribut einzusetzende Asset zuerst sperren muss, bevor dieses an die `adder-` bzw. `setter`-Methode übergeben wird (Diese sind im Kode-Beispiel 4.5 enthalten und in Kode-Beispiel 4.6 hervorgehoben).

Durch diese Vorgehensweise kommt aber ein anderes Problem zustande, und zwar die Verklemmungen (auf. Engl. *Deadlock*). Das Sperren von `StudentB` könnte auch unter Umständen scheitern, was dazu führt, dass auf `StudentB` sehr lange gewartet werden muss, bis es, wenn überhaupt, wieder zur Verfügung steht. Einige Möglichkeiten hierzu sind das Freigeben der Sperrung von `StudentA`, um diesen Vorgang vom Anfang an zu wiederholen, und das Setzen eines *Time Out*, der

die Wartezeit auf `StudentB` einschränkt. Diese Möglichkeiten wären aber noch zu implementieren und in dieser Arbeit zu integrieren.

Ein anderes Problem, das auftreten könnte, ist ein durch fehlgeschlagene Operationen verursachtes Problem. Man stelle sich vor, dass eine `commit`-Methode bei der Modifizierung einer Menge von Assets fehlschlägt. Die Frage, die sich stellt, ist, was mit den bereits durch `commit`-Methoden persistent gemachten Assets passieren soll, weil die `commit`-Methoden noch nicht vollständig ausgeführt worden sind. Die hierzu nötigen Behandlungsschritte sind ebenfalls noch zu implementieren.

4.3.3.3 Methode zum Löschen von Assets

Wie es üblich bei den Inhaltsverwaltungssystemen ist, gibt es auch die Möglichkeit, die Assets wieder zu entfernen. Dieses wird anhand der `delete`-Methode erledigt. Es ist die Idee gewesen, die gelöschten Objekte wieder als flüchtige anzubieten; daher wird ein Objekt als `NewB` von der Methode `deleteAsAsset` der Schnittstelle `MutableB` zurückgeliefert.

Es stehen wieder drei Varianten zur Verfügung, von denen eine die private `delete` - Methode ist, auf die von außen nicht direkt zugegriffen werden kann. Die beiden anderen Methoden sind für das Löschen eines einzelnen Assets und einer Menge von Assets vorgesehen. Im Kode-Beispiel 4.6 ist die private `delete` - Methode zu sehen. Man beachte, dass das Argument der Methode vom Typ `MutableAsset` ist, d.h., dass das Asset vorher zur exklusiven Bearbeitung gesperrt worden ist. Dies ist besonders beim Löschen mehrerer Assets ganz wichtig, um sicherzustellen, dass alle Assets der Menge tatsächlich gelöscht werden können, und keine anderse Assets eines der zu löschenden Assets „in Besitz nehmen“ könnte[HWS04].

Eine erwähnenswerte Bemerkung hierzu ist die, dass zur Wahrung der referentiellen Integrität alle Beziehungen anderer Assets zu dem zu Löschenden entfernt werden müssen. Dies kann ein Anwender nicht selbst veranlassen, da die verweisenden

Assets nicht in seinem Besitz sind, so dass er keinen Zugriff auf sie hat. Daher ist die Löschung der Beziehungen eine Leistung, die das konzeptorientierte Inhaltsverwaltungssystem erbringen muss.

```
private NewAsset privateDelete( MutableAsset mutAsset) {
          try {
               return mutAsset.deleteAsAsset();
          } catch (StateException exc) {
               return null;
          }
     }
```

Kode-Beispiel 4.7: `private delete` - Methode

4.3.3.4 Methode zum Auffinden von Assets

Eine Methode, die zur Assetanfragesprache gehört und als Web Services angeboten wird, ist die Methode zum Auffinden von Assets. Die `lookfor`-Methode wird benutzt, um eine Assetinstanz abzurufen. Anhand dieser Methode können aber auch alle Instanzen einer angegebenen Klasse abgefragt werden.

Zur Suche nach Assets mit gewissen Eigenschaften gibt es die so genannten **Anfrageobjekte**. Für unser Beispielmodell aus Kode-Beispiel 4.2 wäre es die Schnittstelle `StudentQuery`, die durch Aufruf der `startQuery`-Methode aus `AssetClass` entstanden ist. Diese hat für die Charakteristik `name` und für die Beziehung `friend` die Methoden `constrainName...` und `constrainFriend...`, mit denen jeweils ein Teilterm zur Einschränkung des Suchergebnisses definiert wird. Das Suffix der Methode deutet an, wie der übergebene Wert bzw. das übergebene Asset die Suche einschränkt. Dafür gibt es eine Variante für jeden wohl bekannten Vergleichsoperator (`SIMILAR`,

NOT_EQUAL, LESS, LESS_OR_EQUAL, GREATER, GREATER_OR_EQUAL, EQUAL).

Das folgende Klassendiagramm zeigt die Schnittstelle und deren Methode.

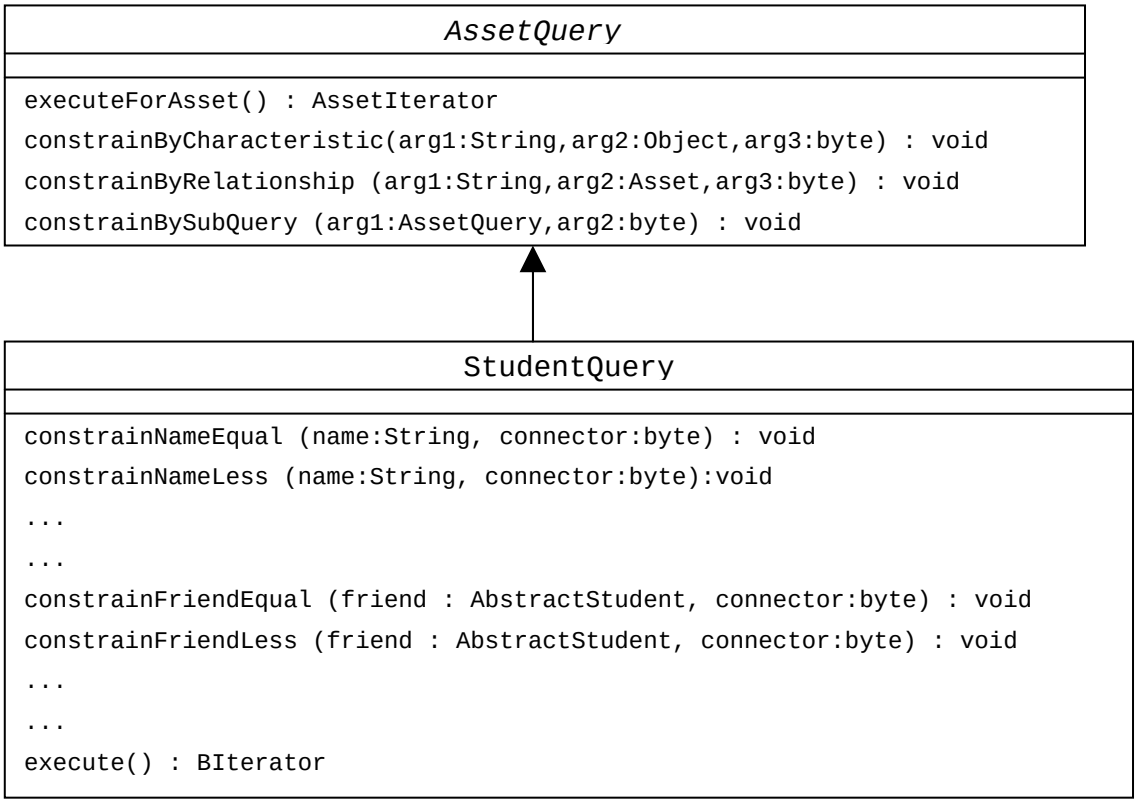


Abb. 4.5: Schnittstellen für Anfrageobjekte ([HWS04])

Eine Variante der `lookfor`-Methode verwendet die vom Anfrageobjekt angebotenen Methoden, um eine Abfrage mit bestimmten Einschränkungen, die im `QueryConstraint` - Objekt gekapselt sind, verarbeiten zu können. Die Methoden setzen danach die Werte der „Einschränkungen“ als Parameter ein, die das Anfrageobjekt später beim Ausführen benötigen wird. Dieses wird im Kode-Beispiel 4.7 gezeigt.

```
public AssetIterator lookfor(AssetClass assetCls,QueryConstraint[] queryConst) {
   if ("Student".equals(assetCls.getName())) {
     StudentQuery studentQuery = (StudentQuery) assetCls.startQuery();
     for (int j = 0; j < queryConst.length; j++)
        if ("friends".equals((queryConst[j]).getAttributeName()))
          if ((queryConst[j]).getComparator() == ConstraintDescription.SIMILIAR)

               studentQuery.constrainFriendsSimilar(
                (AbstractStudent) (queryConst[j]).getConstrainingValue());
          else if ...
        else if ("name".equals((queryConst[j]).getAttributeName()))
          if ((queryConst[j]).getComparator() == ConstraintDescription.SIMILIAR)
               studentQuery.constrainNameSimilar(
                 (String) (queryConst[j]).getConstrainingValue());
          else if ...
          ...
     return studentQuery.execute();
}
```

Kode - Beispiel 4.8: `lookfor` - Methode mit `QueryConstraint`

Man kann die Suche nach Assets nach einem bestimmten Prototypen ausführen. Aus diesem Prototypen, wie bei den create- und modify- Methoden zu sehen ist, wird lediglich durch die `getter`-Methode die Attribute geholt, und anschließend werden diese wie bei der ersten Variante anhand der `constrain`-Methode in das Anfrageobjekt gesetzt.

Wenn man den Iterator einer Kollektion von Assets übergeben bekommt, dann ist eine Suche nach mehreren Assets möglich. Hier wird auf die zuletzt genannte Methode verwiesen.

Durch die `execute`-Methode wird am Ende jeder `lookfor`-Methode das Anfrageobjekt ausgeführt, und diese liefert das Ergebnis in Form eines `StudentIterators`.

5. Tools zur Generierung von WSDL-Beschreibung der Schnittstelle

5.1 Einsatz eines Tools als Hilfsmittel

Nachdem die als Webdienste angebotenen Methoden generiert werden, kann man sie jetzt programmiersprachen - und plattformunabhängig machen, indem man sie in der WSDL-Beschreibung vorlegt. Wie in den vorherigen Kapiteln zu sehen ist, ist die Implementierung eines Generators alles andere als eine triviale Aufgabe. Damit der Aufwand des gesamten Generierungsprozesses von einem Web Services-Modul nicht weiter wächst, bedient man sich für die Generierung der WSDL-Beschreibung der Schnittstelle einem Hilfsmittel, nämlich einem externen Tool zur Generierung von WSDL. Die Anpassung dieses Tools an die sich ständig ändernden Standards von Web Services wird dann vom Entwickler des Tools übernommen.

5.2 Kriterien zur Tool-Auswahl

Beim Auswählen von Tools zur Generierung der WSDL-Beschreibung soll man sich überlegen, ob das Tool einfach zu bedienen ist, d.h. das Tool soll die Generierung erleichtern und nicht mehr Aufwand fordern, z.B. für dessen Einbindung ins Framework. Außerdem ist die Möglichkeit, mit dem Tool vollständige und lauffähige Web Services zu implementieren, von Vorteil. Es sind zurzeit mehrere Tools zur Webentwicklung vorhanden. Diese sind unter anderem das *Java Web Service Developer Pack* (aktuelle Version JWSDP 1.5), *Systinet Developer* (aktuelle Version 5.5), und viele mehr. Die meisten von ihnen bieten ein vollständiges Framework mit umfangreichen Funktionalitäten. Die Generierung der WSDL-Beschreibung aus Java-Klassen ist nur eine davon.

In dieser Arbeit bedient man sich dem ***Systinet Developer for Eclipse*** von Systinet [SYS05], einem *plug-in*, das eine integrierte Entwicklungsumgebung (auf Engl.: *Integrated Development Environment - IDE)* um die erweiterte Unterstützung für Web Services-Entwicklung anreichert. Für diese Arbeit ist das *Eclipse IDE* zum Einsatz gekommen. Die Gründe für diese Auswahl sind:

- Mit JWSDP ist mehr API-Programmierung notwendig. Man muss bestimmte Konfiguration für den Server der Web Services vornehmen, bestimmte Funktionalitäten sind nur aus der Kommandozeile ausführbar, usw., was die Webentwicklung erschweren kann.
- *Systinet Developer* stellt verschiedene Tools bereit, mit denen auch anspruchsvolle Web Services einfach erstellt und verwaltet werden können.
- Einfache Integration von Systinet ins System

5.3 Das ausgewählte Tool : *Systinet Developer for Eclipse*

Systinet Developer for Eclipse (SDfE) wurde im vorherigen Abschnitt vorgestellt. Die Hauptfeatures von SDfE sind unter anderem:

- Unterstützung der aktuellen Standards der Web Services
- Automatische Generierung der WSDL - Datei aus Java - Klassen und andersherum
- Kapselung und Entwicklung der Web Services
- Inbetriebnahme der Web Services auf dem Server
- UDDI-Assistent für Publizieren, Suchen und Auffinden von Web Services
- Generierung von *Stub* und *Skeleton* aus WSDL-Datei
- Unterstützung von WS - Interoperabilität und WS -*Reliable Messaging*

5.4 Einbindung des Tools

Wenn man das *plug-in* erfolgreich installiert hat, dann ist das Ergebnis im Kontextmenü zu sehen und zwar als zusätzliche Optionen. Außerdem ist die Web Services-Ansicht jetzt verfügbar.

Folgende Abbildung zeigt die zusätzlichen Menü, die durch die Installation von SDfE verfügbar sind.

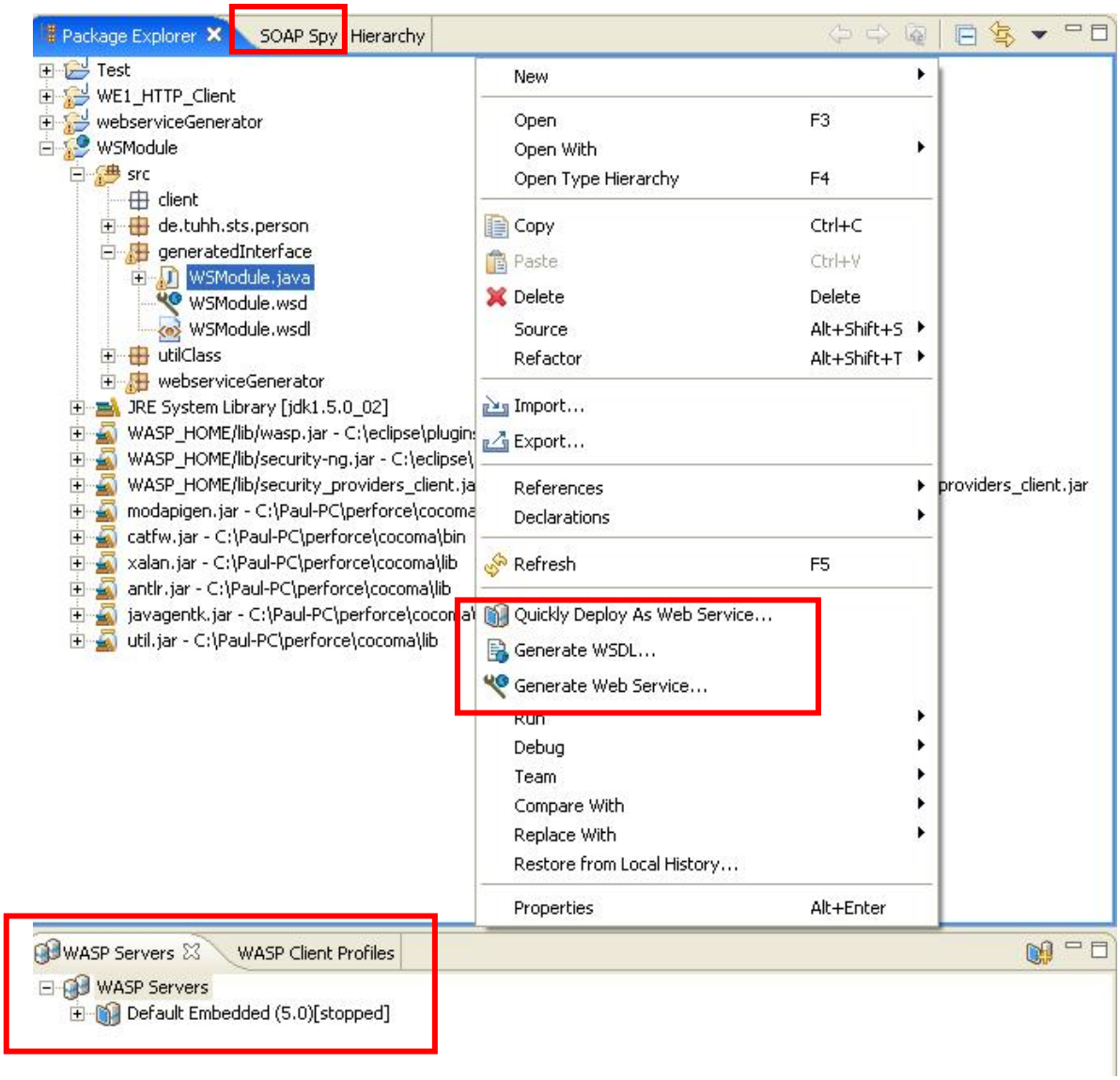


Abb. 5.1: Zusätzliche Optionen in der Web Services-Ansicht

SOAP Spy ermöglicht das Betrachten von Inhalten der gesendeten SOAP-Nachrichten. Die neuen Einträge *Quickly Deploy As Web Service, Generate WSDL, Generate Web Service* werden benutzt, um aus Java-Klassen lauffähige und vollständige Web Services zu erzeugen. Das *WASP Servers*-Fenster zeigt eine Liste von lokalen und entfernten Server, auf die Web Services eingesetzt werden können. Verschiedene Profile für Clients bestimmter Web Services bzgl. der Einstellungen für die Sicherheit der Web Services können gespeichert werden und diese werden im Fenster *WASP Client Profiles* aufgelistet.

Trotz der zahlreichen Features des SDfE wird für diese Arbeit zuerst nur die Funktion zur automatischen Generierung der WSDL-Datei aus Java Klassen verwendet. Für diese Aufgabe ist die Schnittstelle `Java2WSDL` aus der Systinet-API verantwortlich.

Durch *point-and-click* kann man aus der generierten Schnittstelle eine WSDL - Beschreibung generieren lassen, also eine manuelle Einbindung des Tools. Man wähle zuerst den Eintrag *Generate WSDL …* aus dem Kontextmenü. Im ersten Dialog wird gefragt, welche Methoden aus der Schnittstelle in die WSDL aufgenommen werden sollen; in diesem Fall sind dies alle Methoden.

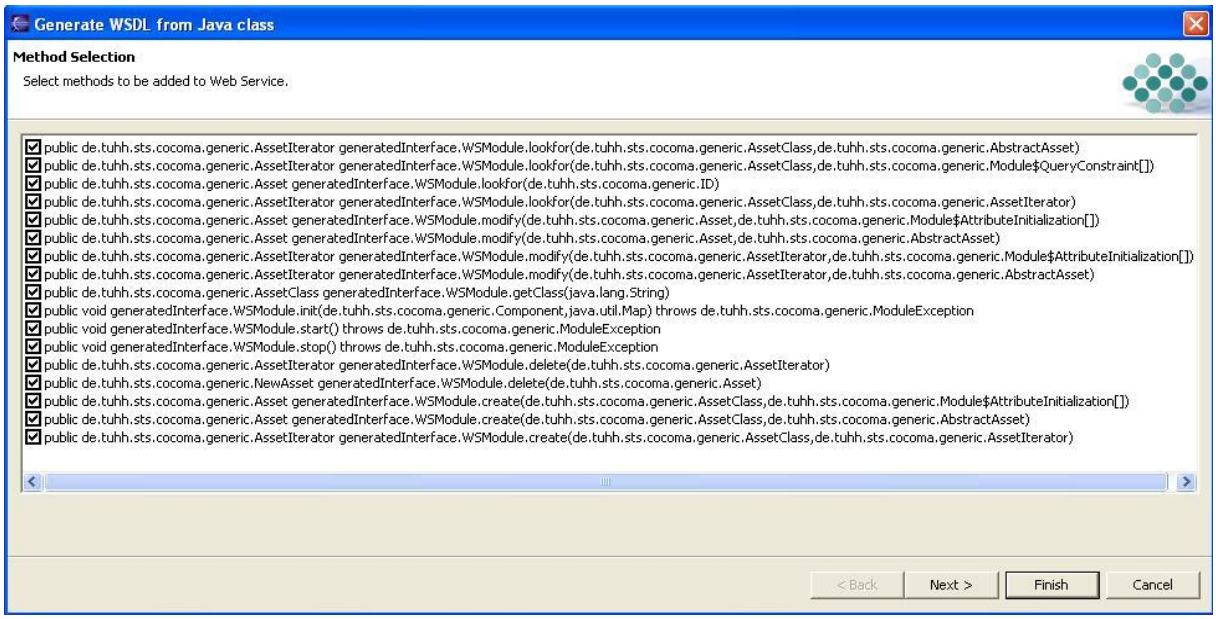


Abb. 5.2: Dialog zur Auswahl der Methoden, die in eine WSDL aufgenommen werden sollen

Im zweiten Dialog wird nach dem Target Namespace, Path und Servicename gefragt. Im letzten Dialog können die XML-Optionen festgelegt werden. Es stehen mehrere Optionen zum XML-Protokoll, Typ der Anhänge, Bindungsstil und zur SOAP–Kodierung zur Verfügung. Danach wird die WSDL-Datei generiert.

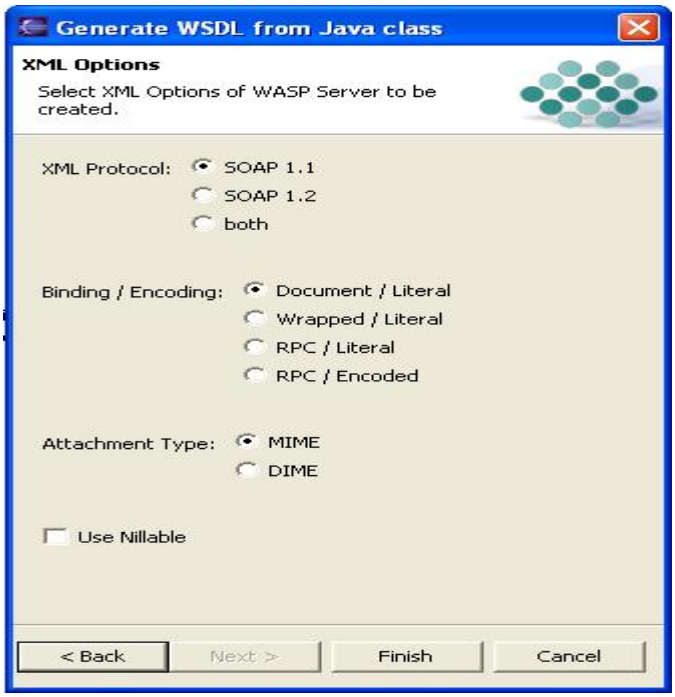


Abb. 5.3: Dialog zur Auswahl der XML - Optionen für die Erzeugung der WSDL - Datei

In der WSDL-Datei fehlen aber noch die Definition der Schnittstellen `NewA`, `AbstractA`, `MutableA`, usw. Es ist notwendig, dass es für diese Klasse auch Schema - Definitionen gibt, da ein potenzieller Client in der Lage sein muss, für diese Klassen Strukturen zu erzeugen. Diese werden bei der Generierung der WSDL-Datei zurzeit noch nicht berücksichtigt, da sie in der Web Services-Schnittstelle nicht direkt sichtbar sind. Vor der Generierung der WSDL-Datei muss also definiert werden wie Vererbung abgebildet werden soll.

Dies geschieht über eine Web Service-Datei (*.wsd). Diese lässt sich über das Kontextmenü *Generate Web Service …* erzeugen. Die Dialogabfolge ist entsprechend der Erzeugung der WSDL-Datei angeordnet. Nach dem letzten Dialog wird die WSD-Datei in einem eigenen Editor geöffnet. Die Konfiguration geschieht auf verschiedenen Seiten.

- *Overview* : Überblick über die getroffenen Einstellungen
- *Service Parts* : Implementierungsklasse, Schnittstelle …

- *Deployment Defaults* : Endpoint-Definition
- *Package Content* : zusätzliche Klassen und Bibiliotheken
- *Polymorphism* : Vererbungsinformationen
- *Compilers* : XML-Optionen für die WSDL - Datei

Auf der *Polymorphism*-Seite können nun die Vererbungsbeziehungen eingetragen werden.

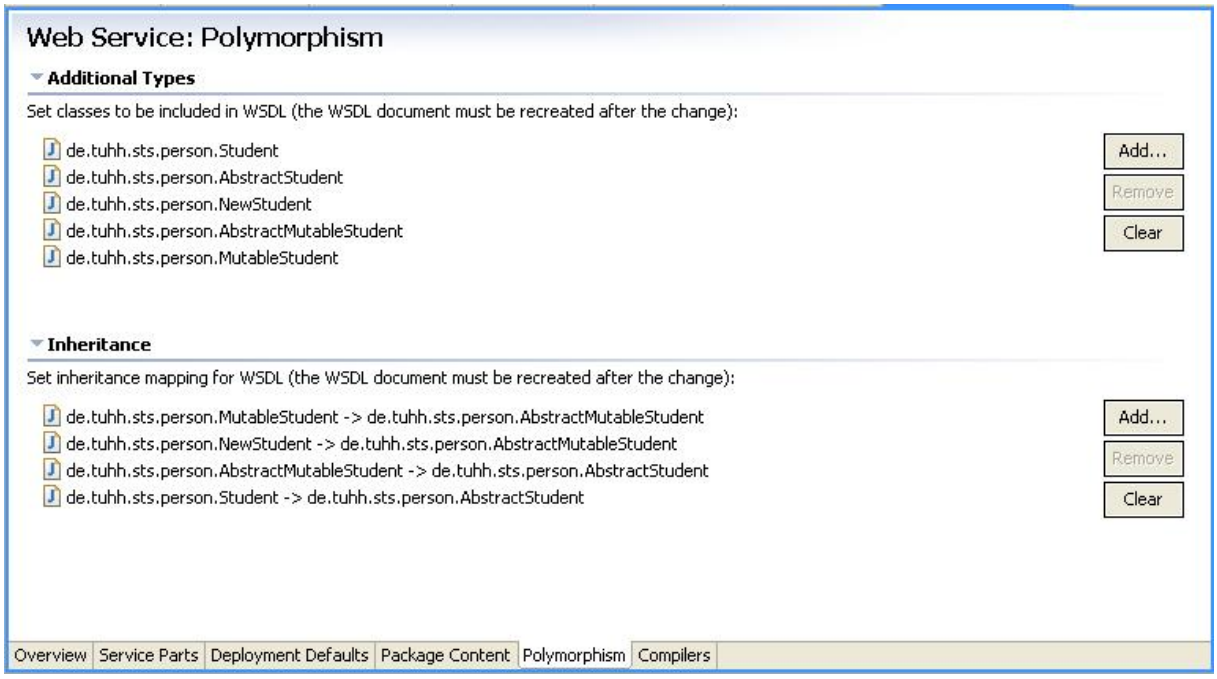


Abb. 5.4: Vererbungsbeziehungen

Es wird hier zuerst auf weitere Schritte beim Entwickeln der lauffähigen einsatzreifen Web Services verzichtet, bis die nötigen Komponenten fertig gebaut sind.

In der nächsten Abbildung ist die generierte WSDL-Beschreibung zu sehen.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="generatedInterface.WSModule"
    targetNamespace="http://systinet.com/wsdl/generatedInterface/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:map="http://systinet.com/mapping/"
    xmlns:ns0="http://systinet.com/xsd/SchemaTypes/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://systinet.com/wsdl/generatedInterface/">
    <types>
        <xsd:schema elementFormDefault="qualified"
            targetNamespace="http://systinet.com/wsdl/de/tuhh/sts/person/"
            xmlns:map="http://systinet.com/mapping/"
            xmlns:tns="http://systinet.com/wsdl/de/tuhh/sts/person/"
            xmlns:xns5="http://systinet.com/wsdl/de/tuhh/sts/cocoma/generic/"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">
            <xsd:import
            namespace="http://systinet.com/wsdl/de/tuhh/sts/cocoma/generic/"/>
```

.....

.....

```
    <xsd:complexType name="AbstractStudent">
        <xsd:annotation>
            <xsd:appinfo>
                <map:java-type
                name="de.tuhh.sts.person.AbstractStudent"/>
            </xsd:appinfo>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="xns5:AbstractAsset">
                <xsd:sequence>
                    <xsd:element name="age" type="xsd:int"/>
                    <xsd:element maxOccurs="1" minOccurs="0"
                        name="friends" type="tns:StudentIterator"/>
                    <xsd:element maxOccurs="1" minOccurs="0" name="name"
                        type="xsd:string"/>
                    <xsd:element maxOccurs="1" minOccurs="0"
                        name="partner" type="tns:AbstractStudent"/>
                </xsd:sequence>
            </xsd:extension>
        </xsd:complexContent>
    </xsd:complexType>
```

Abb. 5.5: <types>-Element

```
<operation name="create" parameterOrder="p0 p1">
    <input message="tns:WSModule_create_13_Request_Soap"
        name="create_13_input"/>
    <output message="tns:WSModule_create_13_Response_Soap"
        name="create_13_output"/>
</operation>
<operation name="delete" parameterOrder="p0">
    <input message="tns:WSModule_delete_9_Request_Soap"
        name="delete_9_input"/>
    <output message="tns:WSModule_delete_9_Response_Soap"
        name="delete_9_output"/>
</operation>
<operation name="delete" parameterOrder="p0">
    <input message="tns:WSModule_delete_10_Request_Soap"
        name="delete_10_input"/>
    <output message="tns:WSModule_delete_10_Response_Soap"
        name="delete_10_output"/>
</operation>
<operation name="getClass" parameterOrder="p0">
    <input message="tns:WSModule_getClass_8_Request_Soap"
        name="getClass_8_input"/>
    <output message="tns:WSModule_getClass_8_Response_Soap"
        name="getClass_8_output"/>
</operation>
<operation name="init" parameterOrder="p0 p1">
    <input message="tns:WSModule_init_9_Request_Soap"/>
    <output message="tns:WSModule_init_Response_Soap"/>
    <fault message="tns:ModuleException" name="ModuleException"/>
</operation>
<operation name="lookfor" parameterOrder="p0 p1">
    <input message="tns:WSModule_lookfor_4_Request_Soap"
        name="lookfor_4_input"/>
    <output message="tns:WSModule_lookfor_4_Response_Soap"
        name="lookfor_4_output"/>
</operation>
```

Abb. 5.6: <operation>-Element

In Abb. 5.5 ist zu sehen, wie die Schemadefinitionen mit den entsprechenden *namespaces* generiert worden sind. Außerdem werden mehrere komplexe Typen gemäß dem Modul-API definiert. Das <operation>-Element in Abb. 5.6 stellt die verschiedenen Operationen aus der generierten Schnittstelle.

6. Fazit

Für die Realisierung eines konzeptorientierten Inhaltsverwaltungssystems und die Generierung dessen Module wird der generative Implementierungsansatz verwendet. Dies wird damit begründet, dass bei einem solchen System sich Modelle ständig ändern und immer wieder neue Modelle im System angelegt werden können. Daher sind ständige Anpassungen erforderlich, die man mit Hilfe des generativen Ansatzes durchführen kann. Entsprechend soll auch die Web Services-Schnittstelle regelmäßig angepasst werden. Daher ist eine automatische Generierung der Schnittstelle für verschiedene Modelle sehr hilfreich.

Im Prinzip ist die Realisierung der Web Services-Schnittstelle eine Implementation der Standardoperationen des konzeptorientierten Inhaltsverwaltungssystems (`create`, `modify`, `delete`, `lookfor`) im Bezug auf die aktuellen Assetdefinitionen. Es ist zu beachten, dass die Implementation den möglichen Varianten der Anfrage entsprechen müssen.

Um Interoperabilität zu ermöglichen, muss eine WSDL-Beschreibung der Schnittstelle vorliegen, da erst durch diese die Interaktion im Sinne der Web Services zustande kommt. Daher erfolgt der Generierungsprozess von einem WS-Modul zweistufig. In der ersten Stufe wird die Schnittstelle in Form von Kode aus der verwendeten Programmiersprache generiert, deren WSDL-Beschreibung erst bei der nächsten Stufe mit Hilfe eines *Tools* von Systinet generiert wird. Das in dieser Arbeit verwendete Tool von Systinet ist ein *plug-in* für die Entwicklungsumgebung *Eclipse*.

Da man mit Hilfe dieses *plug-ins* die Generierung von WSDL und sonstige Aufgaben, die mit Web Services zu tun haben, einfach und schnell erledigen kann, liegt der eigentliche Schwerpunkt dieser Arbeit darin, die angebotenen Methoden so zu implementieren, dass sie sich auf verteilten Systemen transaktional verhalten.

Folgende Überlegungen wären für weitere Projekte zu diesem Thema denkbar. Die *point-and-click*-Methode bei der Erstellung von WSDL ist einfach, aber ungeschickt,

weil man dadurch die Bedeutung an Dynamik eines gesamten Systems verliert. Daher könnte man eine Abhängigkeit zwischen den *plug-ins* definieren, und zwar zwischen dem *plug-in* von Systinet und denen des Frameworks, z.B. durch einen *extension point* (gemäß Eclipse) bei dem Framework, in den man das *plug-in* einbauen kann.

Zum Zeitpunkt der Abgabe wurde noch das *SDfE 5.0* benutzt. Die neueste Version *SDfE 5.5* hat etliche neue Features, z.B. die Unterstützung der WS- Interoperabilität durch zusätzliche Tools, die sehr hilfreich sind, um sicherzustellen, dass die generierte WSDL-Beschreibung wirklich Plattform-, Betriebssystem-, und Programmiersprachenunabhängig ist. Im Bereich der Interoperabilität ist bis heute noch einiges zu klären. Ein neues Modul zu implementieren, das die Assets in XML serialisiert, wäre eine Möglichkeit zum Lösen des Problems der Interoperabilität.

Für die Generierung der WSDL-Beschreibung kann man einen eigenen WSDL-Generator implementieren, der der eigenen Vorstellung von einem WSDL-Generator wirklich entsprechen kann. Dafür muss man aber bei der Entwicklung der Web Services eigene zusätzliche Implementierungen von *Stubs* und Server-Klassen unternehmen.

Und wie im Kapitel 4 zu sehen ist, ist die Implementierung der transaktionalen Aspekte noch weiter zu verbessern, da diese den Kern des Web Services-Modul bilden.

7. Verzeichnisse

[ABC02] Eric Armstrong, Stephanie Bodoff, Debbie Carson, Maydene Fischer, Dale Green, Kim Haase, *The Java Web Services Tutorial*, Addison-Wesley, 2002

[BCGH01] S. Jeelani Basha, Scott Cable, Ben Galbraith, Mack Hendricks, Romin Irani, Jamer Milbery, Tarak Modi, Andre Tost, Alex Toussaint, *Professional Java Web Services*, Wrox Press Ltd., 2001

[BDSFMT05] Berthold Daum, Stefan Franke, Marcel Tilly, *Webentwicklung mit Eclipse*, dpunkt.Verlag, 2005

[CAN04] John Canosa, *Introduction to Web Services* www.embedded.com/story/OEG20020125S0103

[CAS01] Ernst Cassierer, Die Sprache, Band 11 Philosophie der symbolischen Formen der Reihe Gesammelte Werke, Felix Meiner Verlag GmbH, Hamburger Ausgabe Auflage, 2001, Text und Anm. bearb. von Claus Rosenkranz

[CAS02a] Ernst Cassierer, *Das mythische Denken*, Band 12 Philosophie der symbolischen Formen der Reihe Gesammelte Werke, Felix Meiner Verlag GmbH, Hamburger Ausgabe Auflage, 2002, Text und Anm. bearb. von Claus Rosenkranz

[CHA02] David Chappell, Tyler Jewell, *Java Web Services*, O'Reilly, 2002

[DOSB05] Dare Obasanjo, Sanjay Bhatia, *An Introduction to Distributed Object Technologies*, http://www.25hoursaday.com/IntroductionToDistributedComputing.html, abgerufen Mai 2005

[FPI96] Forschungsstelle Politische Ikonographie, Kunstgeschichtliches Seminar der Universität Hamburg (Herausgeber): *Bildindex zur politischen Ikonographie*, Privatdruck, 1996

[FUJ04] Fujitsu, abgerufen Oktober 2004 www.fujitsu.com/global/services/solutions/xml/tech/

[GE03] Jing Ge, Diplomarbeit: *A Web-Services Based Interface for An Information Portal Platform*, 2003

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

[GSB02] Steve Graham, Simeon Simeonov, Toufic Boubez, Doug Davis, Glen Daniels, Yuichi Nakamura, Ryo Neyama, *Building Web Services With Java-Making Sense of XML,SOAP, WSDL und UDDI*, SAMS, 2002

[HWS04] Hans-Werner Sehring, Dissertation : *Konzeptorientierte Inhaltsverwaltung: Modell, Systemarchitektur, Prototypen*, Technische Universität Hamburg Harburg, 2004

[JWSHWS03] Joachim W. Schmidt, Hans-Werner Sehring, *Conceptual Content Modelling and Management: The Rationale of An Asset Language*, Proc. PSI'03, 2003

[JWSHWS04] Joachim W. Schmidt, Hans-Werner Sehring, *Beyond Databases: An Asset Language for Conceptual Content Management*, Proc. ADBIS 2004, 2004

[MTSM03] James McGovern, Sameer Tyagi, Michael E. Stevens, Sunil Mathew, *Java Web Service Architecture*, Morgan Kaufmann Publishers, 2003

[NEW02] Eric Newcomer, *Understanding Web Services*, Addison Wesley, 2002

[ODCCM04] Open Dynamic Conceptual Content Management
http://www.sts.tu-harburg.de/~hw.sehring/cocoma/

[PE31] C.S. Peirce, *Collected Papers of Charles Sanders Peirce*, Harvard University Press, 1931

[SDFE04] Systinet, *Systinet Developer for Eclipse 5.0*, Systinet Corporation, 2004

[SYS05] Systinet, abgerufen März 2005
http://www.systinet.com

[SR02] Christiane Schmitz-Rigal, *Die Kunst offenen Wissens, Ernst Cassirers Epistemologie und Deutung der modernen Physik*, Band 7 der Reihe Cassirer-Forschungen, Ernst Meiner Verlag, Hamburg, 2002.

[W3C04] World Wide Web Consortium
http://www.w3c.org

[WIKI05] Wikipedia, die freie Enzyklopädie, abgerufen Mai 2005
http://de.wikipedia.org