Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Jochen Ludewig
Betreuer:	Dipl.-Ing. Rainer Schmidberger
begonnen am:	21. Oktober 2002
beendet am:	17. April 2003
CR-Klassifikation:	D.1.5, D.2.2, D.2.11, D.2.13, H.3.5, J.1

Diplomarbeit Nr. 2053

Entwurf einer web-basierten Komponentenarchitektur für universitäre Geschäftsprozesse am Beispiel des Software-Praktikums

Stefan Bindel

Institut für Softwaretechnologie
Universitätsstraße 38

70569 Stuttgart

Diese Arbeit beschäftigt sich damit eine methodische und kontinuierliche Vorgehensweise bei der Entwicklung von Softwarekomponenten abzuleiten und an einem Beispiel praktisch zu erläutern. Hierfür werden im ersten Teil die Grundlagen durch Begriffsdefinitionen und die Beschreibung verschiedener Komponententechnologien gelegt.

Im zweiten Teil wird die Findung und Entwicklung von Komponenten entlang eines Softwareentwicklungsprozesses beschrieben. Als Beispiel dient ein Komponentenbaukasten zur Unterstützung universitärer Geschäftsprozesse, speziell zur Verwaltung von Softwarepraktika. Gleichzeitig werden aktuelle Technologien wie die *Model Driven Architecture* und Web-Services verwendet.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Aufgabenstellung

Im Folgenden wird der Wortlaut der Aufgabenstellung zitiert:

Hintergrund

Das Software-Praktikum (SoPra) ist für Studenten der Softwaretechnik eine Pflichtveranstaltung und erstreckt sich über mehrere Monate mit dem Ziel ein vollständiges Projekt in einer kleinen Gruppe zu bearbeiten. Der Verwaltungsaufwand seitens der Organisatoren ist dafür aber erheblich: es müssen nach der Anmeldung der Teilnehmer Gruppen gebildet, Zeitpläne verwaltet, Reviews organisiert und mehrere Besprechungstermine geplant werden. Abschließend erhalten die Studenten einen Schein. Eine web-basierte Lösung würde für Betreuer und Teilnehmer eine erhebliche Vereinfachung erbringen.

Der oben beschriebene Geschäftsprozess findet sich in Teilen in vielen anderen Prozessen der Verwaltung des Lehrbetriebs wieder. Ziel der Aufgabe soll damit ein grundsätzlicher Architekturentwurf, basierend auf dem Konzept der Web Services sein, der als Komponenten-Baukasten auch für weitere Geschäftsprozesse verwendet werden kann.

Aufgabenbeschreibung

Aus der Literatur sollen Verfahren abgeleitet werden, wie aus einer gegebenen Spezifikation sinnvoll Komponenten gebildet werden können. Hierfür sind insbesondere fachliche und technische Komponenten zu unterscheiden. Die daraus entstehenden unterschiedlichen Abstraktionsebenen sollen nach der MDA (Model driven architecture der OMG) entwickelt werden.

Für den Fall des Software-Praktikums soll dann eine solche Komponentenbildung vorgenommen werden. Die Gemeinsamkeiten der Komponenten sollen auf generische Klassen abgebildet werden.

Wichtig ist die universelle Einsatzmöglichkeit der Komponenten im Sinne einer Komponentenbibliothek. Zu jeder Komponente gehört eine präzise Dokumentation. Die Komponenten sind als Web-Service anschließend zu implementieren.

Der spezielle Anwendungsfall des Software-Praktikums wird dann auf Basis der erstellten Komponenten implementiert.

1.2 Motivation

Das Ziel dieser Diplomarbeit besteht aus zwei Teilzielen: Einerseits soll eine Komponentenarchitektur für universitäre Geschäftsprozesse entworfen und an einem kleinen Beispiel implementiert werden, andererseits soll eine allgemeine Methodik zur Findung von Komponenten entwickelt werden. Im Folgenden werden diese Ziele näher ausgeführt.

1.2.1 Entwurf des Komponentenbaukastens

Mit der Verwaltung von Lehrveranstaltungen ist mitunter ein großer Aufwand für Dozenten und Mitarbeiter verbunden. An den Informatik-Instituten gibt es daher schon diverse Ansätze diesen Aufwand mit Softwareunterstützung zu reduzieren. Die dabei (in der Regel in Projekten von Studierenden) entwickelten Softwaresysteme sind allerdings voneinander unabhängig und daher auch inkompatibel zueinander, was die Benutzung unbequem macht. Es ist daher wünschenswert einen Baukasten von Standardkomponenten zu entwickeln aus dem Software zur Unsterstützung universitärer Geschäftsprozesse aufgebaut werden kann.

Hierbei sollen bestehende Systeme zur Integration betrachtet werden, besonders die in einer anderen Diplomarbeit entwickelte Raumplanung[Tög02].

1.2.2 Ableitung eines allgemeinen Verfahrens zur Komponentenfindung

Die auf Komponenten basierende Softwareentwicklung hat im Software Engineering zurzeit eine große Bedeutung. Der Umgang mit dem Begriff Komponente fällt jedoch schwer, da es in der Literatur keine standardisierte Definition dieses Begriffs gibt. Als Folge davon ist auch die Identifikation (im Sinn von Erkennung) von Komponenten schwierig. Es gibt einige Publikationen über Verfahren um Softwaresysteme in Komponenten aufzuteilen (beispielsweise [LA02]). Diese basieren jedoch im Kern auf der nötigen Intuition oder Erfahrung eines Experten der Geschäftsdomäne, der die Aufteilung durchführt. Diese Verfahren sind unsicher und können leicht dazu führen, dass die Aufteilung in Komponenten und damit Entwurf und Implementierung korrigiert werden müssen.

Eine allgemein gültige Methode zur Komponentenfindung wäre daher eine große Erleichterung in der Softwareentwicklung.

1.3 Aufbau

Diese Arbeit gliedert sich in zwei Teile. Zuerst werden die theoretischen Grundlagen erläutert, wobei im zweiten Kapitel der Begriff Komponente und die damit in Zusammenhang stehenden Begriffe erklärt werden und Kapitel drei auf die in der Realisierung verwendeten Technologien eingeht. Der zweite Teil ab Kapitel vier befasst sich dann mit der Realisierung des Komponentenbaukastens. Die einzelnen Kapitel behandeln verschiedene Phasen des Prozesses. Nach einigen allgemeinen Vorbemerkungen enthält das fünfte Kapitel die Ergebnisse der Analyse. Kapitel sechs beschreibt den Entwurf und insbesondere die Aufteilung des Systems in Komponenten. In siebten Kapitel werden wichtige Aspekte der

Implementierung und des Tests beschrieben. In Kapitel acht wird schließlich noch der Entwurf der Webapplikation beschrieben. Eine Zusammenfassung und ein Ausblick in Kapitel neun runden die Arbeit ab.

2 Der Komponentenbegriff

Es ist generell schwer unmissverständlich über den Begriff der Softwarekomponente zu schreiben, da es sehr verschiedene Definitionen für diesen Begriff gibt und keine der Definitionen den Status eines offiziellen Standards besitzt. Die meisten Publikationen verwenden Begriffe, die im Zusammenhang mit Komponenten stehen, in einer selbstverständlichen Art und Weise ohne diesen Begriff vorher erklärt zu haben. Aus diesem Grund soll am Anfang dieser Arbeit zuerst der Begriff der Komponente in der Softwareentwicklung untersucht werden. Dabei wird zunächst der Begriff Komponente allgemein betrachtet und dann schrittweise in den Bereich der Software übergegangen.

2.1 Ursprung und allgemeine Definitionen

Der Begriff *Komponente* hat seinen Ursprung im lateinischen und kommt von dem Verb *componere* – zusammensetzen. Eine Komponente bezeichnet einen Teil eines Ganzen. Im technischen Bereich wird Komponente auch als Bauteil oder Baustein bezeichnet.

Der Standard des IEEE zur Definition von Begriffen im Software Engineering[IEE99] sagt in diesem Punkt etwas ähnlich allgemeines aus:

> **component.** One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components.
>
> *Note:* The terms “module”, “component” and “unit” are often used interchangeably or defined to be sub-elements of one another in different ways depending upon the context. The relationship of these terms is not yet standardized.
>
> *Eines der Teile, aus denen ein System aufgebaut ist. Eine Komponente kann Hardware oder Software sein und ihrerseits wieder aus Komponenten bestehen.*
>
> Bemerkung: *Die Begriffe „Modul”, „Komponente” und „unit”* (engl. Baugruppe, Bauteil, Einheit) *werden oft abhängig vom Kontext als Synonym verwendet. Der Zusammenhang dieser Begriffe ist noch nicht standardisiert.*

Diese Definition stellt sich als zu allgemein heraus, insbesondere da unter anderem keine Abgrenzung zum Begriff Modul existiert. Das hier angesprochene Phänomen, dass vor allem die Begriffe Modul und Komponente in verschiedenen Verhältnissen zueinander verwendet werden ist in der Literatur oft zu beobachten.

Im *IEEE Dictionary of Electrical and Electronics Terms*[Jay84] wird der Begriff *component* aus Hardwaresicht besser definiert als

> A piece of equipment, a line, a section of a line or a group of items that is viewed as an entity for purposes of reporting, analyzing and predicting outages.
>
> *Ein Stück, eine Kette oder eine Gruppe von Teilen, die zu Zwecken der Messung, Untersuchung und Vorhersage von Versagen als Einheit betrachtet wird.*

Der Zweck von Komponenten ist hier also die Abstraktion. Überträgt man dieses Prinzip zusammen mit dem Aufbau von Systemen aus Komponenten auf Software, so gelangt man zu einer pragmatischen Definition wie dieser[Gri98]:

> Eine (Software-)Komponente ist ein Stück Software, das klein genug ist um es in einem Stück zu erzeugen und pflegen zu können, groß genug ist um eine sinnvolle Funktionalität zu bieten und eine individuelle Unterstützung zu rechtfertigen sowie mit standardisierten Schnittstellen ausgestattet ist um mit anderen Komponenten zusammen zu arbeiten.

Eine physischere Sicht auf den Begriff Komponente eröffnet die *Encyclopedia of Software Engineering* [Mar02b]. Eine Komponente wird dort als ein untergeordnetes System bezeichnet. Menschen sprechen von einem System um ein Stück der sie umgebenden Welt als eine elementare Einheit anzusprechen. Bestimmte Details der Welt liegen dann innerhalb des Systems, während andere außerhalb des Systems existieren. An der (eventuell imaginären) Grenze des Systems befindet sich eine Schnittstelle, über die die Innenwelt mit der Außenwelt interagieren kann.

Auch hier spielt wieder die Abstraktion eine große Rolle. Die Person außerhalb braucht nichts über das Innenleben zu wissen. Ebenso muss eine Person innerhalb des Systems die Außenwelt nicht kennen. Beide nehmen nur die Schnittstelle wahr.

Ein gutes Beispiel hierfür ist eine Stereoanlage: Der Verbraucher hat eine einfache Sicht auf die Anlage als Gehäuse mit Knöpfen, Schlitzen und Anschlüssen. Er benutzt beispielsweise einen Tonträger aus der Außenwelt und kann durch die Bedienung des Geräts dessen Wiedergabe erreichen.

Der Entwickler der Anlage kennt den inneren Aufbau. Er muss sie so konstruieren, dass ein Druck auf den Wiedergabeknopf die entsprechenden Schaltungen vornimmt, Motoren bewegt und den Ablauf der Wiedergabe steuert. Hierfür ist es unwichtig welche Musik auf dem Tonträger gespeichert ist oder welche Lautsprecher an die Anlage angeschlossen sind.

Zwischen Entwickler und Verbraucher besteht jedoch durch die Schnittstelle die Vereinbarung standardisierte Tonträger und Lautsprecher zu verwenden.

2.2 Die Softwarekomponente

Im Softwarebereich ist die Komponente eine Metapher. Auch hier wird zwischen „innen” und „außen” unterschieden und es gibt Schnittstellen, über die die Komponente mit der Außenwelt interagiert. Es gibt jedoch einige Unterschiede zu den eben betrachteten physischen Komponenten:

- Softwarekomponenten sind wie Software allgemein immateriell. Der Prozess der Entwicklung und der Verwendung von Softwarekomponenten folgt anderen Gesetzen als der physischer Komponenten. Letztere werden als Prototyp gefertigt und anschließend in Serie hergestellt und in großen Stückzahlen verwendet. Softwarekomponenten jedoch können, einmal hergestellt, beliebig oft wiederverwendet werden. Es reicht bereits aus eine Komponente drei Mal wiederzuverwenden, damit sich die Entwicklung lohnt[Tra95]. In der Praxis werden sie jedoch weniger oft wiederverwendet, als seriengefertigte physische Komponenten, deren Produktion sich erst bei größeren Stückzahlen lohnt.

- Es gibt bisher relativ wenig standardisierte Schnittstellen. Im Vergleich zur Elektrotechnik, wo sehr viele Schnittstellen standardisiert sind und es sehr viele Standardkomponenten (beispielsweise integrierte Schaltkreise) gibt, sind in der Softwareentwicklung standardisierte Schnittstellen selten. In der Regel sind nur die Schnittstellen zwischen Betriebssystemen und Anwendungsprogrammen standardisiert, unter Unix beispielsweise durch die POSIX-Spezifikation. Es gibt jedoch zum Beispiel keine standardisierten Schnittstellen im Bereich der Finanzdienstleistungen. Daher ist es schwierig eine Komponente zu entwickeln, die eine bestimmte Aufgabe für Banken erfüllt und die mit den Softwaresystemen mehrerer Banken kompatibel ist.

- Die Verwendung von Komponenten, die von Drittanbietern eingekauft werden, unterscheidet sich von der Verwendung von Hardwarekomponenten. Zum einen kann eine Komponente beliebig oft kopiert werden, was aus der bereits genannten Immaterialität der Software folgt. Weiterhin kann eine fehlerhafte Komponente mit deutlich weniger Aufwand ersetzt werden. Normalerweise reicht es die Dateien der Komponente durch neuere Versionen zu ersetzen. Andererseits können eingekaufte Softwarekomponenten jedoch unerwünschte Seiteneffekte haben, die weit über die Komponentengrenzen hinausgehen. Fehlerhafte Hardwarekomponenten erfüllen ihre Funktion nicht oder nicht richtig, im schlimmsten Fall verursachen sie einen Kurzschluss und zerstören das Gerät oder Teile davon. Eine Softwarekomponente im Bankwesen, die von einem Drittanbieter gekauft wurde, kann vorsätzlich oder unbeabsichtigt Code enthalten, der weitaus größeren wirtschaftlichen Schaden anrichten kann. Das Problem der Feststellung der Schuld und der Haftung führt dazu, dass das Einkaufen einer solchen Komponente viel Vertrauen in den Hersteller erfordert(vgl. [HC01]).

- Genau dieses nötige Vertrauen führt zu einem weiteren Unterschied, der vor allen Dingen im psychologischen Bereich liegt: Während im Hardwarebereich die Verwendung von eingekauften Komponenten tagtäglich praktiziert und respektiert wird, herrscht im Softwarebereich immer noch ein generelles Misstrauen gegen jedes Stück fremden Codes vor. Dies wird als *Not-Invented-Here*-Prinzip bezeichnet.

Die Entwicklung der komponentenbasierenden Softwareentwicklung hängt stark mit der Entwicklung der objektorientierten Softwareentwicklung zusammen. Obwohl Softwarekomponenten nach den bisherigen Definitionen nicht zwingend auf objektorientierter Tech-

nologie basieren müssen, ist diese trotzdem in der Regel die Grundlage für komponentenbasierende Systeme. Der Hauptgrund hierfür liegt in den zur Verfügung stehenden Entwicklungsumgebungen, die auf Objektorientierung beruhen.

Die Idee der Aufteilung eines Systems in Unterprogramme, Module, Einheiten, Klassen, Komponenten oder Ähnliches existiert schon sehr lange. Zuerst wurde wiederkehrender Code in Makros gepackt um Arbeit zu sparen. Hieraus entwickelte sich das Prinzip des Unterprogramms. Wiederverwendbare Unterprogramme wurden zu Funktionsbibliotheken zusammengefasst. Ein Programm wurde bis dahin nur auf Grund des Programmflusses in Unterprogramme aufgeteilt. 1972 wurde von Dave Parnas das Prinzip der Kapselung von Entwurfsentscheidungen[Par72] und damit das *Information Hiding* (das Verstecken von Implementierungsdetails) vorgestellt.

Hieraus entstand später die objektorientierte Programmierung: Daten und die auf ihnen operierenden Prozeduren wurden zusammen in eine Klasse gekapselt. Gleichzeitig sollte eine Klasse einen Teil der physischen Welt modellieren.

Es stellte sich jedoch bald heraus, dass eine Klasse nicht isoliert betrachtet und wiederverwendet werden kann. Häufig arbeiten mehrere Klassen zusammen um ihre Funktion zu erfüllen. Um diese Kollaboration zu dokumentieren und wiederverwendbar zu machen wurden Entwurfsmuster eingeführt. Diese sollten häufig auftretende oder sich wiederholende Strukturen der Zusammenarbeit zwischen Klassen wiederverwendbar dokumentieren. Entwurfsmuster sind jedoch nur wiederverwendbare Strukturen, auf die im Entwurf zurückgegriffen werden kann um sich wiederholende Probleme zu lösen. Die eigentliche Implementierung muss für jeden Fall neu geschrieben werden und ist in der Regel auch nicht wiederverwendbar.

Um Funktionalität, die durch die Zusammenarbeit mehrerer Klassen realisiert wird, auf Implementierungsebene wiederverwendbar zu machen wurde die Idee der Komponente entwickelt. Eine Komponente kapselt mehrere Klassen, deren Struktur und deren Zusammenarbeit. Im Vordergrund steht hier im Gegensatz zu den Entwurfsmustern die Funktionalität. Eine Komponente, die sich nach außen hin genauso verhält wie eine andere, kann von außen als dieselbe Komponente betrachtet werden, auch wenn die inneren Strukturen oder die Realisierungen voneinander verschieden sind.

Eine der besten Definitionen für den Begriff Softwarekomponente stammt von Clemens Szyperski[Szy99]. Seine Definition basiert auf der IEEE-Definition von *System*, die besagt, dass ein System aus Komponenten aufgebaut ist.

> Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system.
>
> *Softwarekomponenten sind binäre Einheiten, die unabhängig voneinander produziert, erworben und entwickelt werden. Softwarekomponenten arbeiten zusammen um ein funktionierendes System zu bilden.*

Diese Definition bezeichnet Softwarekomponenten als Einheiten, die prinzipiell unabhängig voneinander existieren. Um ein Softwaresystem zu erstellen werden verschiedene Komponenten erworben oder wiederverwendet und zu diesem System zusammengesetzt.

Im Prinzip wird also davon ausgegangen, dass die für ein System benötigten Komponenten bereits existieren. Wird eine Komponente entwickelt oder produziert, so geschieht dies unabhängig von allen anderen.

Softwarekomponenten liegen in binärer Form vor. Das bedeutet, sie müssen zum Zusammenbau eines Systems nicht übersetzt werden. Wird eine Softwarekomponente eingekauft, so kann der Quellcode zwar mitgeliefert werden, dies ist jedoch nicht unbedingt notwendig.

Damit eine Softwarekomponente eingekauft und sinnvoll verwendet werden kann, muss sie jedoch bestimmte Voraussetzungen erfüllen. Zum einen müssen die Schnittstellen genau spezifiziert sein, zum anderen müssen Abhängigkeiten von anderen Komponenten oder Anforderungen an die Umgebung explizit dokumentiert sein. Weiterhin sollte die Komponente in ihrer binären Form an das zu erstellende System und die Umgebung anpassbar sein. Diese weiteren Voraussetzungen beschreibt Szyperski in einer zweiten, konkreteren Definition (ebefalls in [Szy99]):

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.
>
> *Eine Softwarekomponente ist ein Baustein mit formal spezifizierten Schnittstellen und nur explizit angegebenen Abhängigkeiten in Bezug auf die Umgebung. Eine Softwarekomponente kann unabhängig von anderen Komponenten ausgeliefert und von Dritten verwendet werden.*

Um den Komponentenbegriff noch besser verstehen zu können sollen hier noch einige verwandte Begriffe gegen Komponenten abgegrenzt werden.

Eine *Klasse* ist keine Komponente, da sie zum einen nicht generell in binärer Form und unabhängig von anderen Klassen vorliegt (dies ist in Java zwar der Fall, in C++ oder Ada beispielsweise aber nicht). Zum anderen sind bei Klassen externe Abhängigkeiten in der Regel nicht explizit dokumentiert. Klassen können jedoch als Basis für Komponenten verwendet werden. Dies wird bei vielen Komponentenmodellen sogar oft praktiziert (s. Abschnitt 2.4)

Eine *Bibliothek* ist nach IEEE-Definition[IEE99] eine Sammlung von Software inklusive Dokumentation, die bei der Entwicklung, Benutzung oder Wartung hilft. In der Regel enthält eine Bibliothek eine mehr oder minder lose Sammlung von Funktionen ähnlicher Art. Ein System wird durch die Unterstützung von Bibliotheken gebaut, nicht aber aus ihnen zusammengesetzt. Daher ist eine Bibliothek keine Komponente.

2.3 Motivation zur Komponentenbildung

Nachdem nun der Begriff der Softwarekomponente umrissen ist, soll die Frage geklärt werden, zu welchem Zweck Komponenten entwickelt werden.

Die Bildung von Softwarekomponenten geschieht aus verschiedenen Motivationen heraus. Diese haben wiederum Einfluss auf die Grenzen zwischen den Komponenten und

dadurch auf die Architektur. Im Folgenden wird auf die einzelnen Beweggründe und ihre Auswirkung näher eingegangen.

2.3.1 Verbergen von Komplexität

Die auf Komponenten basierende Softwareentwicklung word häufig als logische Fortsetzung der objektorientierten Softwareentwicklung gesehen. Durch die Gruppierung von Daten und Unterprogrammen in Klassen werden Programme übersichtlicher gegliedert und durch Information Hiding werden Abhängigkeiten vermindert. Für große Systeme ist die Bildung von Klassen jedoch zu feingranular. Ein großes System kann aus hunderten oder tausenden von Klassen bestehen. Die Einführung von Komponenten als weiterer Abstraktionsstufe verringert die Anzahl der Einheiten, aus denen das System aufgebaut ist, auf zehn bis 20. Komponenten werden dabei vor allem nach den Gesichtspunkten Kopplung und Zusammenhalt gebildet. Dieses Vorgehen hat vor allem zwei Vorteile: zum einen können Teile des Systems unabhängig von anderen entwickelt werden, sofern die Schnittstellen zwischen den Komponenten frühzeitig entworfen werden. Zum anderen kann schnell ein Überblick über das System gewonnen werden, da die Anzahl der Komponenten überschaubar bleibt.

Ein weiterer Vorteil ergibt sich für iterative Entwicklungsprozesse. Es kann nicht nur die Entwicklung verschiedener Komponenten parallelisiert werden, sondern einzelne Komponenten können auch vorerst nur prototypisch implementiert werden um die Schnittstelle zu erfüllen. Zu einem späteren Zeitpunkt können sie dann durch die endgültige Implementierung ersetzt werden. Beispielsweise könnte eine Komponente, die effiziente komplexe Berechnungen durchführt, in einer frühen Iteration mit einem einfachen Algorithmus implementiert werden, der später im Entwicklungsprozess durch eine optimierte Lösung ersetzt wird.

Durch die Gliederung eines Systems in einfachere Komponenten wird weiterhin die Softwarequalität, insbesondere die Wartbarkeit und die Änderbarkeit gesteigert. Durch die fest geschriebenen Schnittstellen haben kleinere Änderungen innerhalb einer Komponente keine Auswirkungen auf die anderen Teile des Systems. Die Voraussetzung dafür ist natürlich, dass die Komponenten keine Seiteneffekte haben, die über die dokumentierten Schnittstellen hinausgehen. Größere Änderungen, bei denen Schnittstellen verändert werden müssen, sind zwar aufwändig und in einigen Fällen sogar aufwändiger als bei Systemen ohne Komponenten, jedoch sind die Änderungen leichter dokumentierbar und daher die Gefahr der wachsenden Inkonsistenz zwischen Dokumentation und System geringer. Generell ist durch die zusätzliche Abstraktionsstufe die Dokumentation einfacher, da die Anzahl der Abhängigkeiten, die beachtet werden müssen, geringer ist.

2.3.2 Austauschbarkeit

Oft sollen einzelne Elemente eines Systems unter Beibehaltung des Restsystems ersetzt werden, beispielsweise eine Benutzungsschnittstelle oder die Einbindung eines externen Dienstes. Um dies zu ermöglichen und eine solche Änderung möglichst einfach zu gestalten werden Teile, die mit einer gewissen Wahrscheinlichkeit ersetzt werden, in eigene

Komponenten gekapselt. Eine neue Komponente, die diese alte Komponente ersetzen soll, muss nur die Schnittstelle der alten Komponente implementieren.

Gegebenenfalls kann eine abweichende Schnittstelle durch eine Adapterkomponente angepasst werden. Das Prinzip der Adapterkomponente ist ein Muster zur Verwendung einer Komponente mit einer syntaktisch inkompatiblen, jedoch semantisch ähnlichen Schnittstelle. Um die gewünschte Komponente verwenden zu können muss eine neue Komponente erstellt werden, die der geforderten Schnittstelle genügt und alle Aufrufe der Schnittstelle (gegebenenfalls mit Konvertierungen) an die gewünschte Komponente delegiert.

Wird ein System mit austauschbaren Komponenten entwickelt, so wird das System von Grund auf für eine bestimmte Art von Änderungen vorbereitet. Dementsprechend wird die Änderbarkeit des Systems erhöht. Austauschbare Komponenten werden vor allem im Bereich der Geschäftsanwendungen verwendet um technische Abhängigkeiten zu kapseln. Beispielsweise sind Geschäftsanwendungen in der Regel abhängig von einer Persistenzschicht, die Daten in eine Datenbank sichert und diese wieder der Anwendung zur Verfügung stellt. Um nicht von einem bestimmten Datenbankanbieter oder einer bestimmten Version der Datenbank abhängig zu sein wird diese Persistenzschicht in der Regel austauschbar entwickelt, so dass bei einem Wechsel der Datenbank eine entsprechend auf diese abgestimmte Komponente verwendet werden kann.

Eine besondere Verwendung findet diese Strategie im Bereich der Produktlinien. Produktlinien sind mehrere verwandte Softwaresysteme, die einander so ähnlich sind, dass sie wirtschaftlich gemeinsam entwickelt werden können. Dies trifft beispielsweise auf verschiedene Varianten eines Betriebs- oder Anwendungssystems für verschiedene Plattformen zu. [Par01] schlägt für diese Softwarefamilien unter anderem vor, die Entwurfsentscheidungen in denen sich die einzelnen Systeme unterscheiden in eigene Komponenten auszugliedern. Der gemeinsame Teil der Funktionalität kann dann für alle Systeme gleichzeitig entwickelt werden, während die Komponenten zur Variierung systemspezifisch entwickelt werden.

2.3.3 Erweiterbarkeit und Anpassbarkeit

Ein System kann durch Komponenten erweiterbar sein. Die Komponenten werden manchmal als Add-Ons (*to add on*, *engl.* anbauen) oder Plug-Ins (*to plug in*, *engl.* einstecken) bezeichnet. Ein Anwendungsgebiet ist die Möglichkeit ein Softwaresystem flexibel an seine Umgebung anpassen zu können. Diese Möglichkeit ist vor allem im Bereich der eingebetteten Systeme wichtig, da hier der verwendete Speicherplatz eine große Rolle spielt. Programmfunktionen, die nicht unbedingt benötigt werden, brauchen so nicht unbedingt installiert zu werden und verbrauchen keinen Speicherplatz.

Ein weiteres Anwendungsgebiet ist die Nachrüstung von Funktionalität im laufenden Betrieb, wie dies bei Web-Browsern der Fall ist. Diese können eine gewisse Menge von Dateiformaten anzeigen. Wird ein unbekanntes Dateiformat gefunden, so wird nach Rücksprache mit dem Benutzer ein Plug-In gesucht und installiert, mit dem das neue Dateiformat angezeigt werden kann.

Erweiterungskomponenten zeichnen sich dadurch aus, dass sie gegen eine vom Zielsystem definierte Schnittstelle programmiert sind und über einen Mechanismus verfügen

um dem Zielsystem Informationen über sich selbst mitteilen zu können. Erweiterungskomponenten werden in der Regel nur für ein Zielsystem entwickelt und sind daher im Allgemeinen nicht wiederverwendbar.

2.3.4 Robustheit und Effizienz

Oft geht es Anwendungsentwicklern nicht primär um Prozessqualität wie Wiederverwendbarkeit oder Wartbarkeit, sondern vielmehr um Produktqualität, wie Ausfallsicherheit, Robusheit und Effizienz.

Diese Aspekte lassen sich mit technischen Ansätzen deutlich besser und generischer lösen als die Anforderungen an die Prozessqualität. Daher bieten heutige Komponentenmodelle in erster Linie diese generischen technischen Lösungen.

Speziell Middlewaretechnologien wie die so genannte Verteilungstransparenz machen Systeme robuster und effizienter: Systeme bestehen dabei aus jeweils mehreren Instanzen derselben Komponenten, die über ein Netzwerk auf mehrere Rechner verteilt sind. Ein von einer Komponente angebotener Dienst wird erst durch die Middleware(s.u.) der behandelnden Komponenteninstanz zugeordnet. Hierdurch wird eine Lastverteilung erreicht, die das System effizienter macht. Zusätzlich ist im Falle eines Problems mit einem der Rechner das System trotzdem weiter lauffähig, da eine Komponente auf anderen Rechnern des Systems repliziert ist. Die Komponenteninstanzen auf den anderen Rechnern können die Aufgaben der Instanz auf dem ausgefallenen Rechner mit übernehmen. Abbildung 2.1 stellt dies als UML-Sequenzdiagramm dar.

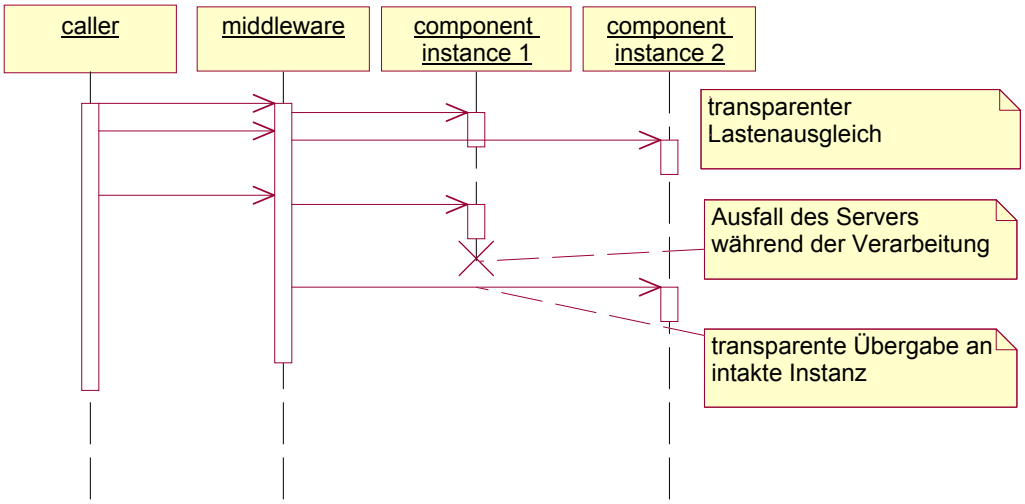


Abbildung 2.1: Effizienz und Ausfallsicherheit durch Middlewareeinsatz

Richtig eingesetzt besitzen diese Technologien deutliche Vorteile, da viele technische Aspekte nicht mehr implementiert werden müssen. Dies senkt die Wahrscheinlichkeit von Defekten.

Andererseits muss bei der Entwicklung jedoch darauf geachtet werden, dass die Prozessqualität nicht vernachlässigt wird. In [Amb02] wird beispielsweise vorgeschlagen Kompo-

nenten anhand der Aufrufhäufigkeiten aufzuteilen um diese Aufrufhäufigkeiten, aus denen Netzzugriffe resultieren, zu minimieren. Dieser Ansatz führt jedoch zu unübersichtlichen Komponenten, die fachliche Aspekte nicht sauber trennen und daher schlecht wartbar und schlecht wiederverwendbar sind.

2.3.5 Wiederverwendbarkeit

Komponenten, die als wiederverwendbare Softwareteile entwickelt werden sollen, sind schwer zu identifizieren. Die Findung von Komponenten geschieht in der Regel als Teil des Entwicklungsprozesses eines ganzen Softwaresystems. Dadurch wird die Komponente primär auf das zu entwickelnde System zugeschnitten. Dies kann die Wiederverwendbarkeit deutlich einschränken. Selbst eine Komponente, die im Hinblick auf Wiederverwendbarkeit entwickelt wurde, kann sich später als nicht wiederverwendbar entpuppen.

Es muss also bereits vorher über die weiteren Einsatzmöglichkeiten der Komponente nachgedacht werden. Aus diesen müssen dann die Anforderungen und Schnittstellen der Komponente abgeleitet werden. Weiterhin müssen konfigurierbare Parameter festgelegt werden. Dies kostet zusätzliche Zeit und Ressourcen. Beides ist jedoch in Industrieprojekten oft knapp, weshalb die Entwicklung von wiederverwendbaren Komponenten erschwert wird.

Ein weiteres Problem bei der Entwicklung wiederverwendbarer Softwarekomponenten im industriellen Umfeld besteht in der Unsicherheit der Investition. Damit sich eine Investition lohnt, muss eine Komponente genügend oft wiederverwendet werden. In *Ted Bigger-man's Rules of Three* [Tra95] wird angegeben, dass Software mindestens drei Mal wiederverwendet werden muss um die Kosten zu decken. Da Komponenten im Softwarebereich jedoch oft sehr technologieabhängig sind, ist sehr unsicher, ob die von der Komponente verwendete Technologie nicht schon veraltet ist, bevor die Komponente oft genug wiederverwendet wurde.

Es ist daher wichtig, bei der Wiederverwendbarkeit einer Softwarekomponente besonders auf die Technologieunabhängigkeit zu achten. In [LM02] wird beispielsweise die Entwicklung eines Frameworks erläutert, mit dem die Verwendung von Java-Klassen unabhängig vom technologischen Umfeld (wie Standardanwendung, Geschäftsanwendung oder Webapplikation) möglich ist.

Bei der Wiederverwendung von Software sind zusätzlich einige Dinge zu beachten. In [Lon01] werden einige Muster zur Wiederverwendung beschrieben, die auf den ersten Blick Erfolg versprechend aussehen, sich jedoch bei genauerer Untersuchung als kontraproduktiv erweisen.

2.4 Komponentenmodelle

Nach [HC01] beschreibt ein Komponentenmodell die Rahmenbedingungen einer Realisierung von Komponenten. Zum einen wird die Struktur jeder Komponente festgelegt, die zu dem Komponentenmodell konform ist. Die Vorgaben dieser Struktur können beispielsweise Implementierungssprache, obligatorische Schnittstellen oder andere Einschränkungen

bei der Nebenläufigkeit oder der Speicherung des Zustandes umfassen. Weiterhin legt das Komponentenmodell die Verknüpfungen von Komponenten fest. Dies umfasst die Deklaration von Abhängigkeiten zwischen Komponenten oder Schnittstellen, das Auffinden von Diensten oder Komponenten, Referenzierung und Aufruf sowie das Zusammenstellen einer Anwendung aus Komponenten.

Es werden hierbei zwei Abstraktionsebenen differenziert: die Realisierungsebene und die Interaktionsebene. Dieses Prinzip taucht bereits in [DK76] auf, wo eine Trennung von Implementierungs- und Interaktionssprache von „Modulen", die in der dortigen Bedeutung mit heutigen Komponenten gleichzusetzen sind, vorgeschlagen wird.

Die Implementierungssprache bezeichnet die Sprache in der die Komponente implementiert ist. In ihr werden Ressourcen wie Datenstrukturen und Operationen ausgedrückt, die die Komponente realisieren. Im Gegensatz dazu bezeichnet der Begriff Interaktionssprache die Sprache, die verwendet wird um die Verbindungen zwischen Komponenten auszudrücken, um beispielsweise Komponenten zu einer Anwendung zusammenzusetzen.

In einigen der unten beschriebenen Komponentenmodelle sind diese Ebenen nicht sehr klar getrennt. Dies führt oft zu weniger Flexibilität in Bezug auf die Implementierungssprache oder die Plattform und zu komplexeren, schwerer verstehbaren Komponentenmodellen.

Ein Komponentenmodell kann schließlich auch Anpassungs- und Konfigurationsmechanismen für Komponenten festlegen. Mit Hilfe dieser Mechanismen kann eine Komponente an die Umgebung angepasst werden. Man spricht auch von Parameterisierung.

Um Komponenten, die einem Komponentenmodell entsprechen, ausführen zu können wird eine *Komponentenmodellimplementierung* benötigt. Diese Software ist vergleichbar mit dem Laufzeitsystem eines Übersetzers. Sie ist dafür zuständig die im Komponentenmodell beschriebenen Mechanismen zu unterstützen und alle mit der Ausführung der Komponente zusammenhängenden Aufgaben zu übernehmen. Hierzu zählt die Ressourcenverwaltung, das Verwalten der Lebenszyklen der Komponenten und weiter gehende Dienste wie Verteilung, Replikation und Transaktionssteuerung. Die Software für diese zuletzt aufgeführten Dienste wird auch als Middleware bezeichnet. Vor allem im komponentenorientierten Umfeld sind die Grenzen zwischen Middleware und Komponentenmodellimplementierung sehr fließend und in der Literatur wird teilweise der Begriff Middleware für die gesamte Implementierung verwendet.

Ein weiteres Missverständnis taucht in den Begriffen *grobkörnig* und *feinkörnig* auf. Feinkörnige Komponenten bezeichnen nur jeweils eine Klasse, die gegebenenfalls durch wenige Hilfsklassen ergänzt sind. Diese Komponenten erfüllen zwar die Kriterien der Definition, sie sind aber in Bezug auf Wiederverwendbarkeit und Komplexitätsreduktion ungeeignet. Trotzdem werden sie in einigen Komponentenmodellen als Komponenten bezeichnet und verwendet.

Die grobkörnigen Komponenten können im Gegensatz dazu wiederverwendbare Einheiten bilden. Sie sind aus feinkörnigen und evtl. anderen grobkörnigen Komponenten zusammengesetzt und bilden größere Einheiten, die jeweils eine bestimmte Funktion erfüllen. Eine aus Komponenten zusammengesetzte Anwendung ist so gesehen ein Spezialfall einer grobkörnigen Komponente. Grobkörnige Komponenten werden in einigen Spezifikationen von Komponentenmodellen als Modul oder als *Assembly* (engl. Baugruppe) bezeichnet.

Im Folgenden sollen kurz die bekanntesten Komponentenmodelle vorgestellt werden. Ein genauerer Vergleich von JavaBeans, Enterprise JavaBeans und der COM-Familie ist in [GT00] enthalten. Ein kürzerer Vergleich, der aber auch das CORBA Component Model und die .NET-Technologie abdeckt findet sich in [CL02].

2.4.1 JavaBeans

JavaBeans[Sun97] basieren auf der Programmiersprache Java. Eine JavaBean ist eine Java-Klasse, die bestimmten Regeln bei der Namensgebung der Methoden und Schnittstellen genügt. Generell kann man hier von ereignisgesteuerten Komponenten sprechen. Die Schnittstellen werden durch das Erzeugen und Verarbeiten von Signalen definiert. Das Zusammenfügen der einzelnen Komponenten zu Anwendungen erfolgt durch ein grafisches Werkzeug, das durch Reflexion auf die Klassen zugreift um deren Struktur festzustellen. JavaBeans erweisen sich generell als zu feingranular und werden eher zu akademischen Betrachtungen herangezogen (vgl. [LV01]).

Ein Vorteil ist die Unabhängigkeit zwischen Implementierungssprache (Java) und Interaktionssprache (grafisch). Im Gegensatz zu den einzelnen Komponentenmodellen, in denen Aufrufe externer Komponenten explizit sind, werden bei JavaBeans nur Quellen und Senken bestimmter Signale definiert. Diese werden dann auf einer höheren Abstraktionsebene miteinander verbunden. Der Vergleich zu Mikrochips und Platinen aus der Hardware liegt nahe.

2.4.2 Enterprise JavaBeans

Das Komponentenmodell der in der Java 2 Enterprise Edition (J2EE)[Sun99] enthaltenen Enterprise JavaBeans (EJB) ist ein auf Geschäftsanwendungen spezialisiertes Komponentenmodell. Es klassifiziert Komponenten nach Prozessen und Daten und gibt so bereits eine bestimmte Struktur für die jeweilige Komponente vor. Im Gegensatz zur JavaBeans werden hier Java-Schnittstellen und die explizite Java-Aufrufsemantik verwendet. Um die Aufgabe der Middleware zu erfüllen werden diese Aufrufe jedoch intern abgefangen und gegebenenfalls umgelenkt. Abhängigkeiten von externen Komponenten sowie weitere Eigenschaften, wie Transaktionsverhalten und konfigurierbare Parameter werden in einem so genannten Deskriptor beschrieben. Durch den Deskriptor wird zwar eine gewisse Trennung von Implementierungs- und Interaktionsebene erreicht, jedoch ist durch die Verwendung der Java Aufrufsemantik die Flexibilität eingeschränkt. Weiterhin darf eine Komponente nur eine einzige Schnittstelle implementieren, so dass verschiedene Sichten auf dieselbe Komponente nur schwer zu realisieren sind.

Eine genaue Beschreibung der Java 2 Enterprise Edition folgt in Abschnitt 3.3.

2.4.3 COM, COM+, DCOM und .NET

All diese Komponentenmodelle sind zwar unabhängig von der Implementierungssprache, sie wurden jedoch von Microsoft entwickelt und sind proprietär, sodass sie nur auf Windows-Plattformen unterstützt werden. COM, COM+ und DCOM sind hierbei verschiede-

ne Entwicklungsstufen derselben Technologie. COM steht für Component Object Model. Das D in DCOM steht zusätzlich für *distributed*, also verteilt. COM[WK94] entstand aus einer weiteren Microsoft-Technologie, dem *Object Linking and Embedding (OLE)*, das in früheren Versionen der Office-Software dazu verwendet wurde beispielsweise Tabellenkalkulationsdaten direkt in Textdokumente einzufügen und auch an Ort und Stelle bearbeiten zu können. Hieraus entwickelte Microsoft die Möglichkeit zum Beispiel die Komponente zur Anzeige von HTML des Internet Explorers in andere Anwendungen zu integrieren. Realisiert wird das Komponentenmodell durch ein rechnerzentrales betriebssystembasiertes Schnittstellen- und Komponentenverzeichnis. Eine Komponente muss sich und ihre Schnittstellen dort registrieren, bevor sie verwendet werden kann. Um die Sprachenunabhängigkeit zu gewährleisten werden die Schnittstellen in einer eigenen Beschreibungssprache (Interface Description Language, IDL) hinterlegt.

Der Aufruf eines Dienstes einer Komponente ist explizit und erfordert relativ viel zusätzlichen Code, was das Komponentenmodell schwer benutzbar und den Quellcode weniger lesbar macht. Aus diesem Grund erweiterte Microsoft für COM+[Kir97] die eigenen Übersetzer derart, dass sie den entsprechend komplexen Code aus einfacherem Quellcode generierten. Hierdurch wurde es zwar deutlich leichter in den jeweiligen Sprachen Komponenten zu entwickeln, die Übersetzer waren jedoch durch zusätzliche Schlüsselworte nicht mehr standardkonform.

Durch die Sprachunabhängigkeit von COM ist es zwar theoretisch möglich Komponenten- und Interaktionscode zu trennen, da jedoch die Mittel zur expliziten Beschreibung von Abhängigkeiten fehlen, ist die Zusammenstellung einer Anwendung auf diesem Weg praktisch nicht möglich. Daher wird dieses Komponentenmodell fast ausschließlich dazu verwendet fremde Dienste aus anderen Anwendungen oder dem Betriebssystem in eigene Anwendungen zu integrieren und einzelne Dienste anderen Anwendungen zur Verfügung zu stellen.

Da die COM-Technologie in Ihrer Entwicklung immer schwieriger zu handhaben war und da sich durch die Web-Services-Technologie neue Anforderungen ergaben, die nur schwer durch die bisherige Technologie realisierbar waren, entwickelte Microsoft in den letzten Jahren die .NET-Technologie (lies: dot-net).

Durch verschiedene an Java angelehnte Merkmale, wie beispielsweise die implizite Freigabe nicht mehr benötigter Objekte, sollten zum einen Anwendungen schneller entwickelt und stabiler eingesetzt werden können, zum anderen wurden die neuen Anforderungen relativ tief in den Betriebssystemkern integriert.

Leider bleibt Microsoft trotzdem bei der etwas eigensinnigen Definition von Komponenten als der Wiederverwendung von bestehenden Klassen in neuen Anwendungen. Generell wurde der Umgang mit diesen zwar erleichtert: die Registrierung im zentralen Verzeichnis entfällt und es ist sogar möglich, Klassen von bestehenden Klassen in anderen Programmiersprachen abzuleiten. Jedoch gibt es auch in der .NET-Technologie keine Möglichkeit auf einer abstrakteren Ebene Anforderungen der Komponente an die Umgebung zu dokumentieren und so Anwendungen aus Komponenten aufzubauen. Ein weiteres Problem ist hier auch die fehlende Möglichkeit Assemblies aus anderen Assemblies zusammensetzen zu können.

2.4.4 CORBA Component Model (CCM)

Das CORBA Component Model (CCM) ist das jüngste der hier vorgestellten Komponentenmodelle. Die Spezifikation[Obj02] wurde erst im Juni 2002 fertig gestellt. Wie CORBA stammt auch das CCM von der Object Management Group (OMG), einer unabhängigen Institution zur Entwicklung technisch hoch stehender und gleichzeitig industriell einsetzbarer Standards im Bereich der objektorientierten Softwareentwicklung.

Das CCM wurde unter mehreren Gesichtspunkten entwickelt: Zum einen sollte die Grundlage ein einfaches Modell sein, das von der Philosophie der Komponenten und nicht von der verfügbaren Technologie abhängen sollte. Zum anderen sollten jedoch zur Realisierung des Modells bestehende Technologien wie CORBA eingesetzt werden können. Ein weiterer Aspekt war die Kompatibilität und Interoperabilität zu anderen Komponentenmodellen, insbesondere zu Enterprise JavaBeans und schließlich die Sprachunabhängigkeit der Realisierung, wie dies auch schon bei CORBA der Fall ist.

Diese Ziele wurden alle erreicht. Besonders das Modell zeichnet sich dadurch aus, dass es den Gedanken der Komponente deutlich widerspiegelt. Eine Komponente wird mit Steckern, Buchsen, Ereignisquellen und -senken ausgestattet, die Verbindungsmöglichkeiten zu anderen Komponenten bieten und durch die Komponenten zusammengesetzt werden können. Weiterhin kann eine Komponente mehrere Facetten besitzen, die verschiedene Sichten oder Rollen widerspiegeln können. All diese Komponenteneigenschaften werden unabhängig von der Implementierungssprache in einer Komponentenbeschreibungssprache festgehalten. Aus dieser kann dann ein Quellcodegerüst in der gewünschten Sprache erzeugt werden. Ähnlich wie CORBA ist es nicht ganz leicht dieses Quellcodegerüst intuitiv zu lesen und zu verstehen.

Da dieses Komponentenmodell sehr jung ist, gibt es bisher nur wenige Implementierungen, die es unterstützen. Da es jedoch Teil der kürzlich fertig gestellten CORBA Version 3 ist, wird sich dies bald ändern. Momentan gibt es aber praktisch keine größeren Projekte, die dieses Modell einsetzen, weshalb eventuelle Schwachstellen im praktischen Einsatz noch nicht bekannt sind.

3 Verwendete Technologien und Methoden

3.1 Unified Modelling Language (UML)

Die *Unified Modelling Language (UML)* ist eine grafische Modellierungssprache. Mit ihr kann Software vor der Implementierung modelliert, entworfen, visualisiert und dokumentiert werden. Die Sprache entstand aus verschiedenen Notationen und Methoden, deren Ziele eben diese Visualisierung und Dokumentation waren. Die Spezifikation und Standardisierung der UML obliegt, wie auch die des im vergangenen Kapitel beschriebenen CORBA Komponentenmodells, der *Object Management Group (OMG)*.

In der aktuellen Version 1.4 stehen zur Modellierung zwölf verschiedene Diagrammtypen zur Verfügung. Diese lassen sich aufteilen in Beschreibung statischer Struktur (beispielsweise durch Klassendiagramme), dynamischen Verhaltens (durch Zustands- und Aktionsdiagramme) und in Diagramme zur Verwaltung des Modells (z.B. Paketdiagramme).

Seit einigen Jahren ist die UML in der Industrie weit verbreitet und findet auch in der Paxis oft Verwendung. Da es sehr viel Literatur über die UML an sich gibt, wird an dieser Stelle auf eine Beschreibung der einzelnen Elemente und deren Verwendung verzichtet. Statt dessen soll hier auf die Besonderheiten in Bezug auf Komponenten und deren Entwicklung eingegangen werden.

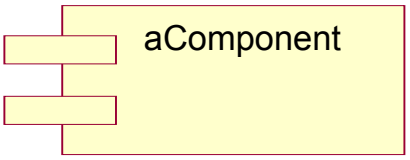
3.1.1 Komponenten in der UML

Da die UML schon seit mehreren Jahren standardisiert ist, der Gedanke der Softwareentwicklung mit Komponenten jedoch erst vor Kurzem seine große Verbreitung gefunden hat, enthält die aktuelle Version der UML[Obj01c] keine vollständigen Mittel zur Modellierung und Dokumentation von Komponenten. Nach [Crn01] fehlen im Standard insbesondere Namenskonventionen und Stereotypdefinitionen. Die grundlegenden Mittel zur Modellierung von Komponenten sind jedoch vorhanden.

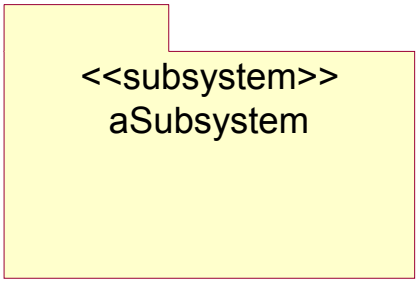
Die UML definiert insbesondere zwei Begriffe, die zur Komponentenmodellierung verwendet werden können: *Komponente* und *Subsystem*.

Eine *Komponente* (siehe Abbildung 3.1(a)) im UML-Sinn ist eine physische Implementierungseinheit eines Systems, die mit wohldefinierten Schnittstellen ausgestattet ist, und die im System ersetzt werden kann. Es gibt einige Stereotypen, die diese Definition konkretisieren, insbesondere für Datenbanktabellen oder ausführbare Dateien.

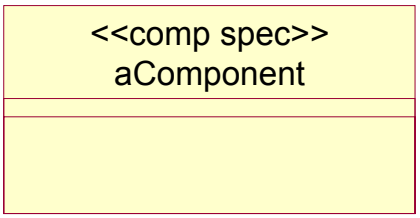
Durch die Stereotypen wird auch deutlich, dass der Sinn von UML-Komponenten nicht primär in der Dokumentation und Modellierung von Komponenten liegt, sondern in einer



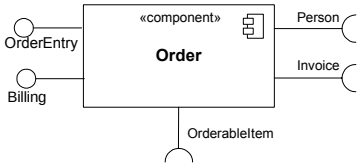
(a) Das Element *Komponente* in UML 1.4



(b) Das *Subsystem*



(c) Alternative Darstellung der Komponente als Klasse



(d) Das Element *Komponente* in UML 2.0

Abbildung 3.1: Verschiedene Darstellungen von Komponenten in UML

physischen Gruppierung von Bestandteilen eines Systems. Ein Komponentendiagramm in der UML zeigt den physischen Aufbau eines Systems aus Dateien und anderen Bausteinen.

Die Modellierung von logischen Komponenten in den frühen Phasen der Entwicklung ist mit diesem Mittel nicht möglich. Die Semantik einer Komponente ist die einer konkreten Implementierung. Eine Komponente aus logischer Sicht ist jedoch von ihrer Implementierung unabhängig. Sie besteht aus definierten Schnittstellen und expliziten Abhängigkeiten, kann jedoch verschieden realisiert werden.

Um den logischen Aspekt der Komponente zu modellieren kann ein UML-Paket mit den Stereotyp *subsystem* versehen werden (Abbildung 3.1(b)). Ein Subsystem modelliert einen Verhaltensaspekt in einem System. Es ist außerdem möglich Klassen zu einem Subsystem hinzuzufügen um die Realisierung zu modellieren. Der Nachteil von Subsystemen ist, dass UML-Paketelemente keine Schnittstellen haben können, weshalb die Spezifikation von Komponenten mit diesem Mittel ebenfalls nicht möglich ist.

Aus diesem Grund wird in dieser Arbeit die in [CD01] vorgestellte Notation verwendet: die Komponenten werden in der Spezifikationssicht als Klassen mit dem Stereotyp *«comp spec»* (kurz für *component specification*) wie in Abbildung 3.1(c) dargestellt. Dies ist im Prinzip jedoch eine Notlösung um die Diagramme mit UML-Werkzeugen erstellen zu können.

Eine endgültige Lösung wird die Ende des Jahres erscheinende Spezifikation der UML

Version 2 bieten (eine Vorabversion ist z.B. in [U2 03] zu finden). In dieser wird das bereits bekannte UML-Element *Komponente* in der Bedeutung und in den Möglichkeiten aufgewertet. In dieser Version können mit Komponenten auch Spezifikationen ausgedrückt werden, die sogar so weit gehen, dass geschachtelte Komponenten, angebotene Schnittstellen sowie Schnittstellen von denen die Komponente abhängig ist, ausgedrückt werden können. In Abbildung 3.1(d) beispielsweise bietet die Komponente zwei Schnittstellen an und hängt von drei weiteren ab.

3.2 Modellgetriebene Architektur

3.2.1 Ziele und Visionen

Die modellgetriebene Architektur (*Model Driven Architecture, MDA)*[Sol00] ist eine relativ junge Technologie, die sich zurzeit noch in einer sehr frühen Phase der Spezifikation befindet. Ähnlich der Webservice-Technologie (s.u.) baut sie jedoch zum Großteil auf bereits bestehenden Standards wie der Unified Modelling Language auf und kombiniert diese nur zu einem größeren Ganzen. Die Vision hinter der Model Driven Architecture besteht aus mehreren Teilen:

Zum einen sollen die Möglichkeiten der automatischen Generierung von Code so gesteigert werden, dass nur noch ein sehr geringer Anteil von unter 10% in der Zielsprache direkt implementiert werden muss. Langfristig wird sogar eine vollständige Codegenerierung angestrebt, so dass die Software zielsprachenunabhängig erstellt werden kann. Zum Vergleich wird oft der Übergang von Assembler zu COBOL und später C angeführt. Dieser Übergang hob die Implementierung auf eine höhere Abstraktionsebene, auf der sich der Entwickler nicht mehr um bestimmte Details (beispielsweise der Zuordnung von Variablen zu Adressen) kümmern musste.

Diese automatische Codegenerierung soll durch die Anreicherung von UML-Modellen mit Semantik, speziell mit Aktionssemantik, erreicht werden[MB02]. Hierbei wird das Verhalten von Objekten durch Zustands- und Aktionsdiagramme modelliert, die durch semantische Ausdrücke zur Formulierung der Zustandsänderungen ergänzt sind.

Ein weiteres Ziel der Model Driven Architecture ist die Trennung der fachlichen und technischen Aspekte im Entwurf. Bei einem Wechsel der verwendeten Technologie, beispielsweise der Implementierungssprache, des Betriebssystems oder der Middleware, kann sich zwar die technische Realisierung und Darstellung verändern, die fachliche Logik ist aber weiterhin dieselbe. Bei der Vermischung von Fachlichkeit und technischer Realisierung im Entwurf, muss dieser bei einem Technologiewechsel komplett überarbeitet oder sogar aus der Spezifikation neu erstellt werden.

Werden diese Aspekte nach der Idee der Model Driven Architecture aufgeteilt, so erhält man ein plattformunabhängiges Modell *(platform-independent model, PIM)*, das aus den fachlichen Anforderungen erstellt wird. Dieses wird dann in einem weiteren Schritt auf eine bestimmte Technologie in ein plattformabhängiges Modell *(platform-specific-model, PSM)* abgebildet. Diese Abbildung geschieht in erster Linie manuell durch einen Entwickler. Jedoch ist vorgesehen hierfür halbautomatische, unterstützende Werkzeuge zu entwickeln, die diesen Prozess z.B. durch Abbildungsvorschläge mit Hilfe von Entwurfsmustern

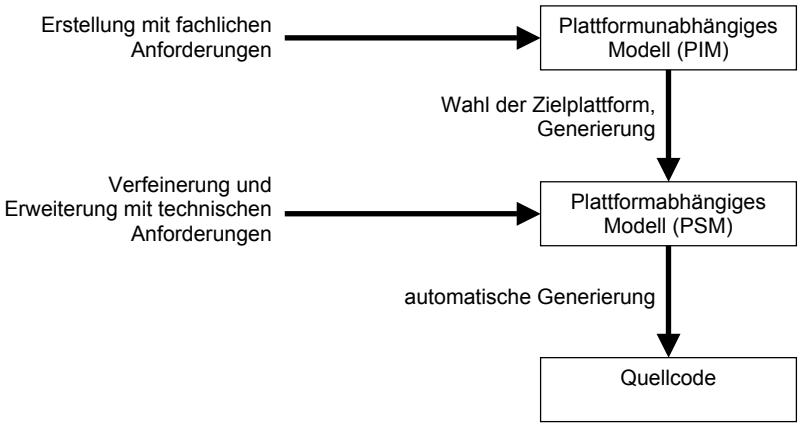


Abbildung 3.2: Das Vorgehen in der modellgetriebenen Architektur

unterstützen. Das PSM wird daraufhin weiter mit Hilfe der technischen Anforderungen verfeinert. Schließlich wird aus diesem Modell Code generiert. Eine Übersicht dieses Prozesses wird in Abbildung 3.2 dargestellt.

Das wohl langfristigste Ziel, das mit der Model Driven Architecture verfolgt wird, ist die Vereinfachung der Integration verschiedener Softwaresysteme. Dies soll auf zweierlei Arten geschehen: Zum einen ermöglicht das so genannte *Common Warehouse Model (CWM)*[Obj01a] die Verbindung verschiedener Datenmengen, die auf diesem Standard beruhen. Erreicht wird dies durch ein standardisiertes Metamodell, in dem die Form, Verarbeitung, Speicherung und andere Eigenschaften über die Daten ausgedrückt werden können. Diese Eigenschaften heißen Metadaten. Durch die einheitliche Beschreibung der Metadaten wird eine gemeinsame Ebene für die eigentlichen Daten erreicht, so dass auf dieser Grundlage die Integration der Daten vollzogen werden kann.

Zum anderen entwickeln bereits verschiedene Gremien der Object Management Group plattformunabhängige Standardmodelle in UML, die Teil der Model Driven Architecture werden sollen. Jedes dieser Standardmodelle ist für eine bestimmte Branche gedacht, beispielsweise Finanzdienstleister. Durch die Plattformunabhängigkeit der Modelle und die Möglichkeit diese per Model Driven Architecture auf eine beliebige Plattform abzubilden, wird erwartet, dass diese Standardmodelle breite Zustimmung und Verwendung finden werden und mittelfristig zu adaptierten Industiestandards werden. Daraus würde sich die Kompatibilität der Softwaresysteme automatisch ergeben.

Generell ist die Philosophie hinter der Model Driven Architecture eine Abstraktion und Zusammenfügung bestehender Standards mit dem Zweck die Entwicklung von Software generell auf eine abstraktere, technologieunabhängigere und dadurch flexiblere Ebene zu heben.

3.2.2 Das Werkzeug ArcStyler

Bisher ist die Werkzeugunterstützung für die modellgetriebene Architektur noch sehr gering. Das am weitesten fortgeschrittene Werkzeug ist der ArcStyler der Firma Interactive Objects Software, die auch in verschiedenen Texten[Wei02, Hub02] beschrieben wird. Die Software besteht aus verschiedenen Teilen, die entweder allein stehen oder sich in weitere Werkzeuge integrieren.

Zum einen ist ein Werkzeug zur Erstellung von Business Object Models enthalten, mit dem CRC-Karten und Anwendungsszenarien entwickelt und dokumentiert werden können. Schon auf dieser hohen Abstraktionsstufe stehen verschiedene Methoden der automatischen Prüfung zur Verfügung. So kann das Modell beispielsweise darauf hin untersucht werden, ob alle Verantwortlichkeiten der einzelnen Objekte auch wirklich zum Einsatz kommen (d.h. ob sie in einem Anwendungsszenario verwendet werden).

Aus dem so gewonnenen Business Object Model kann dann ein UML-Modell für das Werkzeug Rational Rose erzeugt werden. Dies geschieht halbautomatisch. Dem Entwickler wird eine Liste der Verantwortungen, die nicht zugeordnet werden konnten, angezeigt und muss diese nun mit Werkzeugunterstützung auf das gewünschte Modellelement der UML abbilden. Eine Verantwortlichkeit eine bestimmte Information zu kennen wird z.B. in der Regel auf ein Attribut der Klasse abgebildet.

Das so entstandene UML-Modell kann nun mit Rational Rose und der Modellierungssoftware des ArcStyler erweitert werden, so dass nach und nach die Architektur des Systems entsteht. Hierbei werden zusätzlich zu den regulären UML-Modellelementen auch spezielle Stereotypen und von diesen abhängige Eigenschaften angegeben, die Implementierungsstrategien und -details festlegen. Weiterhin können Methoden mit formalen Vor- und Nachbedingungen in der *Object Constraint Language (OCL)*[Obj01b], einer in der UML-Spezifikation enthaltenen formalen Spezifikationssprache, versehen werden. Diese können auf Wunsch automatisch mit in den Implementierungscode übernommen werden und dort die Vor- und Nachbedingungen aktiv prüfen.

Ist der UML-Entwurf vollständig und durch die Validierung überprüft, kann die Generierung des Quellcodes vorgenommen werden. Quellcode kann für verschiedene Zielsysteme generiert werden, vor allem für Java Enterprise Server verschiedener Hersteller. Zusätzlich werden auch alle benötigten weiteren Dateien, wie Projekteinstellungen für Entwicklungsumgebungen, Gerüste für automatische Modultests, Deskriptoren für Komponenten und Skripte zum Anlegen und Löschen der benötigten Datenbanktabellen erzeugt. Die generierten Dateien können nun bearbeitet werden. Um Änderungen an den Dateien verfolgen zu können werden in jeder Datei so genannte gesicherte Blöcke angelegt, innerhalb derer die Änderungen gemacht werden können. Änderungen außerhalb dieser Blöcke gehen bei einer erneuten Generierung verloren.

Insgesamt wird durch die Standardimplementierungen und die Generierung der zusätzlichen Dateien der Implementierungsaufwand um ca. 40% gemindert.

Auch für die Erstellung der web-basierten Benutzungsschnittstelle werden in Rose durch die Modellierungssoftware Hilfen angeboten. So kann die Folge der Webseiten durch Aktionsdiagramme modelliert werden. Formulare werden automatisch erstellt, wobei das Layout der Seiten nachträglich angepasst werden muss.

Auch wenn das Werkzeug ArcStyler die Grundgedanken der moodellgetriebenen Architektur aufgreift und einen Teil der Entwicklung auf eine höhere Abstraktionsebene anhebt, ist es noch weit von den Zielen der MDA entfernt.

Zum einen gibt es momentan kaum andere Zielplattformen außer der Java Enterprise Edition. Ein Modul für Microsofts .NET-Technologie ist zwar vorhanden, jedoch sind Modelle der einen Technologie nicht einfach auf die andere Technologie übertragbar. Der Gedanke, plattformspezifische Modelle aus plattformunabhängigen Modellen zu erzeugen wird hier noch nicht realisiert. Eine solche Modelltransformation muss also noch ohne Unterstützung von Hand erfolgen.

Weiterhin besitzen Codegeneratoren wie der von ArcStyler auch den Nachteil, dass sie auf manche Fehler sensibel reagieren und sich diese Fehler dann nicht zurückverfolgen und korrigieren lassen.

Ein weiterer Nachteil ist die fehlende Dokumentation des erstellten Codes. Es existiert zwar ein Generatorhandbuch, in dem die Abbildungen von Modell zu Code dokumentiert ist, jedoch wird in manchen Punkten der Sinn des generierten Codes, beispielsweise die Benutzung von Hilfsklassen, nicht näher erläutert.

3.2.3 Das Werkzeug XDE

Das Werkzeug *Rational eXtended Development Experience* erweitert eine integrierte Entwicklungsumgebung (wie das in dieser Arbeit verwendete *IBM WebSphere Studio*) um Modellierungs- und Entwurfswerkzeuge für die Unified Modelling Language. Für die Entwicklung von Enterprise Java Anwendungen bietet sie Funktionen an, die in die Richtung der Model Driven Architecture gehen. Die Ansätze sind jedoch nicht so stark ausgeprägt wie bei dem Werkzeug *ArcStyler*.

Es ist jedoch möglich Klassen von Hand plattformunabhängig in UML zu entwerfen und diese danach halbautomatisch mit XDE in Enterprise Java Beans zu transformieren. Diese werden anschließend noch verfeinert, indem einzelne Attribute oder Operationen mit Stereotypen versehen werden oder weitere Eigenschaften im Eigenschaftseditor angepasst werden.

Aus diesem plattformabhängigen Modell kann dann Quellcode generiert werden. Dieser kann auf Wunsch auch weiterhin mit dem Modell synchronisiert werden oder umgekehrt. Darüber hinaus kann aus den angepassten Eigenschaften auch ein für Enterprise Java-Anwendungen notwendiger Deskriptor erstellt werden. Dieser kann jedoch nicht mehr synchronisiert werden, was das Modell in manchen Fällen inkonsistent machen kann.

Generell ist XDE noch nicht sehr weit in Richtung modellgetriebene Architektur entwickelt, bildet jedoch eine gute Grundlage für weitere Schritte in diese Richtung, die Rational sicher spätestens mit UML 2.0 tun wird.

3.3 Java 2 Enterprise Edition

3.3.1 Entstehung und Architektur

Die Java 2 Enterprise Edition (J2EE) ist eine Erweiterung der Standardklassenbibliothek der von Sun Microsystems entwickelten Programmiersprache Java. Diese Erweiterung ist speziell auf die Erstellung verteilter Geschäftsanwendungen ausgerichtet.

Geschäftsanwendungen besitzen einige gemeinsame Merkmale: Sie sind in der Regel Client-Server-Systeme. Das bedeutet, eine größere Anzahl von Arbeitsplatzrechnern greift auf eine geringe Zahl von zentralen Großrechnern zu um gemeinsam eine Aufgabe im Geschäftsumfeld durchzuführen. Hierbei sind die Arbeitsplatzrechner vor allem für den Dialog mit dem Benutzer zuständig, also für die Benutzungsschnittstelle, Abfrage und Eingabe von Informationen. Sinnvolle Vorarbeiten, wie die Überprüfung der Plausibilität der Eingaben, können auch schon von diesem Rechner geleistet werden.

Die Großrechner *(Server)* wiederum sind für die Verarbeitung und Verwaltung der Daten zuständig. Sie besitzen die hierfür notwendigen Ressourcen und greifen auf weitere Ressourcen, beispielsweise eine Datenbank zur permanenten Speicherung von Daten, zu.

In neuerer Zeit wurden, vor allem bedingt durch die Popularität verteilter Datennetze, insbesondere des Internets, Geschäftsanwendungssysteme zunehmend heterogener: Waren die Arbeitsplätze früher örtlich relativ nahe bei den Servern angesiedelt – in der Regel im selben Gebäude – und bestanden in der Regel nur aus ein und derselben Hardwareplattform, so sind die Systeme heute zusammengesetzt aus sehr vielen verschiedenen Plattformen und Betriebssystemen. Weiterhin sind sie durch das Internet praktisch beliebig weit verteilt. Insbesondere im Bereich der Einkaufs- und Bestellmöglichkeiten über das Internet wird der heimische Rechner des Verbrauchers zum Client der Geschäftsanwendung des jeweiligen Internetangebots.

Für genau solche Szenarien wurde die Programmiersprache Java von Sun Microsystems ursprünglich veröffentlicht. Die Plattformunabhängigkeit dieser Sprache wurde speziell für heterogene Systeme entwickelt. Nicht zuletzt aus diesem Grund entwickelte sich Java schnell zu einer allgemein anerkannten Implementierungssprache für Anwendungssysteme.

Leider hatte die Sprache jedoch den Nachteil, dass sie für den Serverbereich nicht geeignet war. Es war relativ umständlich auf der Serverseite effiziente Systeme zu entwickeln. Es entstanden zwar mit der Zeit die entsprechenden Klassenbibliotheken für Serverdienste wie Transaktionen, Datenbankzugriffe, Verteilung usw., jedoch waren dies zum einen nur optionale Erweiterungen zur Standardklassenbibliothek, zum anderen waren verschiedene Versionen der Erweiterungen von verschiedenen Versionen anderer Erweiterungen abhängig, so dass die Vielfalt der Versionen und Erweiterungen mit der Zeit unüberschaubar wurde.

Aus diesem Grund wurde in der Version 2 des Sprachstandards eine neue Variante für diesen Bereich eingeführt, die die Standardbibliothek um eine genau auf einander abgestimmte sinnvolle Menge dieser Zusatzbibliotheken erweiterte: Die Java 2 Enterprise Edition.

Bei der obigen Beschreibung von Geschäftsanwendungen stellt sich als optimale Archi-

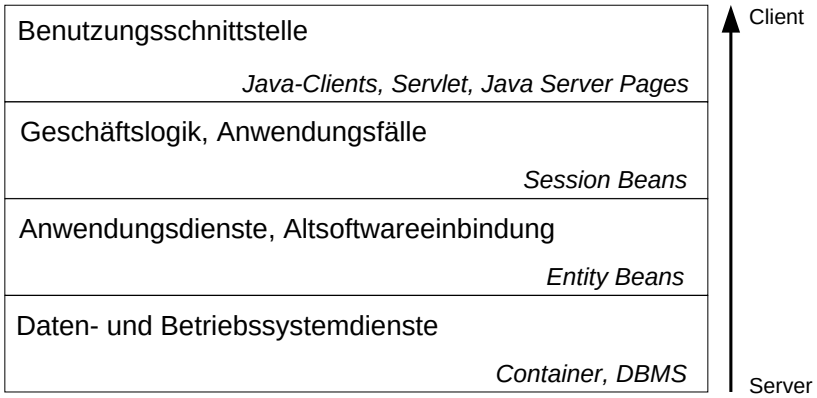


Abbildung 3.3: Schichtenarchitektur von Geschäftsanwendungen

tektur eine Aufteilung in Schichten heraus. Schichtenarchitekturen besitzen einige Eigenschaften, die auch im Bereich der komponentenbasierenden Softwareentwicklung vorteilhaft sind.

Einerseits werden die Abhängigkeiten zwischen einzelnen Komponenten gezielt reduziert, da nur jeweils zwei Schichten direkt voneinander abhängen. Die Realisierung einer Schicht ist austauschbar, ohne dass mehr als die direkt benachbarten Schichten betroffen sind.

Andererseits werden den einzelnen Schichten klare Verantwortlichkeiten zugeordet, so dass auch die Verantwortung von Komponenten in der jeweiligen Schicht klar umrissen sind. Ein Entwurf, der auf einer solchen Architektur beruht ist in der Regel klarer und es entstehen keine Komponenten mit einem unangemessen großen Aufgabenbereich.

In [HC01] wird eine allgemeine Architektur für Geschäftsanwendungssysteme beschrieben, die der J2EE-Architektur sehr ähnlich ist. In Abbildung 3.3 ist diese allgemeine Schichtenarchitektur abgebildet. Unten rechts steht das jeweilige Pendant der Java 2 Enterprise Edition. Die einzelnen Schichten werden nun kurz beschrieben:

Die *Benutzungsschnittstelle* enthält die Systemschnittstelle, die Schnittstelle durch die auf das Gesamtsystem zugegriffen und es gesteuert werden kann. Dies kann entweder durch herkömmliche grafische oder textbasierte Benutzungsschnittstellen erfolgen als auch über Web-Services oder Stapelverarbeitungskommandos. Weiterhin werden hier Dialoge mit dem Benutzer geführt, d.h. der Teil der Anwendungsfälle, in denen der Benutzer involviert ist, wird in dieser Schicht realisiert. Schließlich werden hier die Informationen über die Sitzung des Benutzers gespeichert.

Die *Geschäftslogik* bildet die Geschäftsprozesse im Softwaresystem ab. Ein Geschäftsprozess ist eine Menge von sich gegenseitig bedingenden Aktivitäten im Geschäftsumfeld, die gegebenenfalls auch automatisch ablaufen. Beispielsweise erfordert eine Bestellung unter anderem die Verfügbarkeitsprüfung, Auslieferung und Rechnungsstellung, die ihrerseits wiederum Teilaktivitäten erfordern. Die Abfolge und die gegenseitige Abhängigkeit dieser Aktivitäten muss im Softwaresystem in dieser Schicht abgebildet werden.

Die *Anwendungsdienste* realisieren die oben erwähnten Einzelaktivitäten und die Geschäftsregeln. Dies kann entweder durch entsprechende Geschäftsobjekte geschehen oder durch Delegation an bereits vorhandene Altsysteme, die somit in die neue Geschäftsanwendung integriert werden.

Daten- und Betriebssystemdienste sind technische Dienste, mit deren Hilfe Daten dauerhaft gespeichert werden können. Dies umfasst den Zugriff auf Datenbankmanagementsysteme (DBMS) und Dateisysteme. Es können hier jedoch auch weitere Dienste vorgesehen werden, wie beispielsweise Transaktionsverwaltung.

3.3.2 Technologie

Ein zentraler Begriff in der J2EE-Technologie ist der Begriff *Container* (engl. Behälter). Er bezeichnet eine besondere Ablaufumgebung, in der die verschiedenen Komponenten ausgeführt werden. Nach der Definition in Abschnitt 2.4 ist ein J2EE-Container eine Komponentenmodellimplementierung.

Es gibt drei verschiedene Arten von Containern: Die *Application Client Container* sind Umgebungen für Java Applikationen, die als Client auf eine J2EE-Applikation zugreifen. Die Klassen, die diesen Zugriff ermöglichen, werden einfach in eine beliebige Java Applikation implementiert und zur Laufzeit durch den Container unterstützt.

Die zweite Art Container ist der *Web Container*, ein spezieller Webserver, der zuätzlich zu statischen HTML-Seiten auch dynamische *Java Server Pages* und *Servlets* verarbeiten kann. Ein Servlet ist eine Java-Klasse, die von *javax.servlet.Servlet* abgeleitet ist und zu einer Anfrage mit Parametern eine Ausgabe generiert. In der Regel ist die Anfrage als HTTP-Anfrage formuliert und die Antwort eine HTML-Seite. Das Servlet-Prinzip kann aber auch auf andere Protokolle angewendet werden.

Java Server Pages entstanden aus Servlets um die Trennung von Implementierung und Aussehen der generierten Ergebnisseite zu ermöglichen. Anders als bei Servlets, bei denen der Java-Code im Vordergrund steht und der HTML-Text der Seite per Ausgabemethoden geschrieben wird, steht bei der JSP der HTML-Text im Vordergrund und wird durch spezielle Tags, die Java-Code enthalten, ergänzt. Durch diesen Ansatz können Designer das Aussehen einer Seite ändern und gestalten ohne Java-Code ändern zu müssen.

Zu beachten ist, dass Servlets und JSPs gleich mächtig sind. In der Regel werden JSPs sogar intern im Web Container in Servlets übersetzt und als solche ausgeführt.

Die dritte Art Container ist der *Enterprise Java Bean Container*, oder kurz *EJB Container*. Dieser ist der eigentliche Kern der J2EE-Plattform, denn er bildet die Umgebung für die Komponenten, die Enterprise Java Beans (EJB) genannt werden.

Eine EJB besteht aus einer Implementierungsklasse und zwei Schnittstellen, der so genannten Home-Schnittstelle und der Remote-Schnittstelle. Hierzu können weitere Hilfsklssen kommen.

Die Home-Schnittstelle erlaubt die Steuerung des Lebenszyklus einer Instanz der Implementierungsklasse. Über sie kann eine solche erstellt oder wiedergefunden werden.

Die Remote-Schnittstelle erlaubt den direkten Zugriff auf diese Instanz. Methoden der Implementierungsklasse, die nach außen hin sichtbar sein sollen, werden in der Remote-Schnittstelle definiert und können so aufgerufen werden.

Der Aufrufer greift jedoch nie direkt auf die Klasseninstanz zu, sondern der Container erstellt einen so genannten Wrapper (von *to wrap, engl.* verpacken), auf den zugegriffen wird. Die Aufgabe dieses Wrappers ist es Dienste wie Datenbanksynchronisierung, Transaktionssteuerung, Sicherheit usw. zu implementieren. Dies wird in Abbildung 3.4 veranschaulicht. Die Steuerung der Middleware-Dienste erfolgt hierbei vom Entwickler nicht direkt durch Aufrufe im Code, sondern deklarativ durch Beschreibung im so genannten Deployment Deskriptor (s.u.).

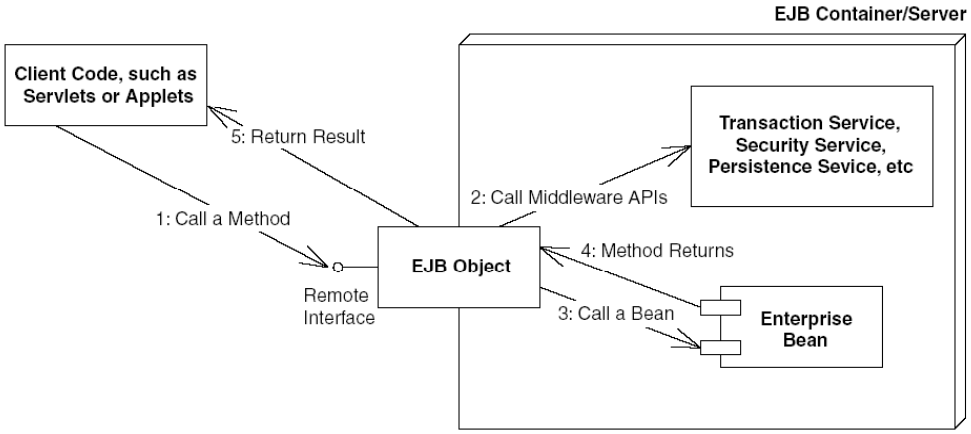


Abbildung 3.4: Funktion des EJB Containers.
Quelle: [RAJ01]

Es gibt verschiedene Arten von Enterprise Java Beans. Eine Taxonomie ist in Abbildung 3.5 dargestellt.

Eine *Session EJB* implementiert in der Regel Geschäftsprozesse oder Teile davon und bietet Dienste an. Man unterscheidet zwischen *Stateless Session EJBs*, die zwischen einzelnen Methodenaufrufen keinerlei Daten speichern und *Stateful Session Beans*, die Daten zwischen den Aufrufen speichern können und daher einen Zustand haben. Der Vorteil der Stateless Session Beans liegt darin, dass sie sehr viel effizienter implementiert und im Container ausgeführt werden können. Es ist fast kein Aufwand zur Verwaltung der Instanzen nötig, da diese wiederverwendet werden können.

Session EJBs werden generell auch dazu verwendet Zugriffe auf Entity Beans zu kapseln. Insbesondere wenn mehrere Aufrufe nötig wären, aber auch um die Implementierung der Entity Beans generell zu verstecken, werden Session EJBs als Fassade verwendet. Man nennt dieses Muster *Session Facade*[Mar02a].

Entity Beans werden dazu benutzt Daten zu verwalten. Sie werden speziell dazu eingesetzt einzelne Datensätze einer Tabelle zu repräsentieren und zu synchronisieren. Eine Entity Bean kann dabei die Logik des Datenbankzugriffs selbst implementieren – man spricht dann von *Bean Managed Persistence (BMP)* – oder die Implementierung dem Container überlassen *(Container Managed Persistence, CMP)*. Es ist außerdem möglich Bean Managed Persistence nicht als Datenbankzugriff, sondern als Zugriff auf die Altsoftware

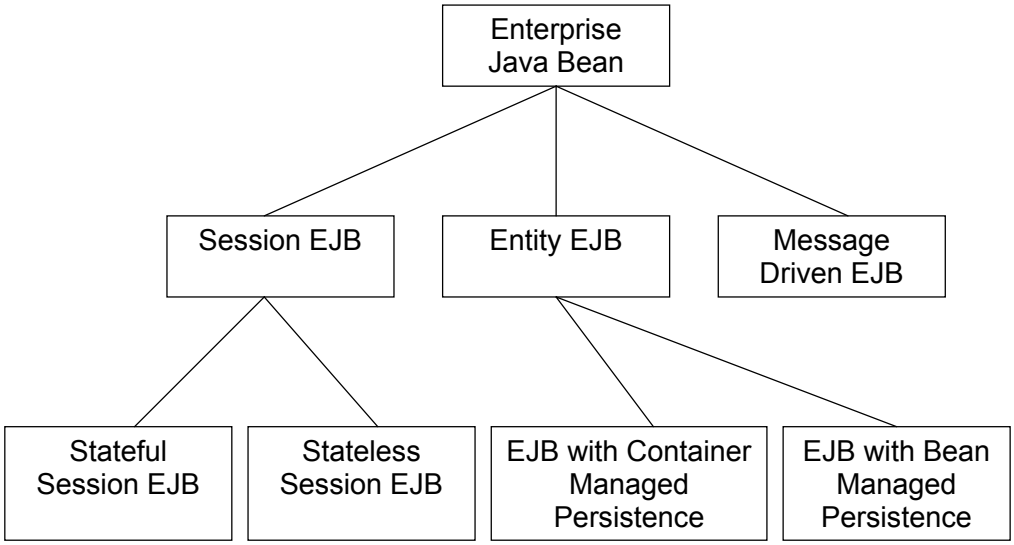


Abbildung 3.5: Taxonomie der Enterprise Java Beans

zu realisieren um das System mit vorhandenen Altsoftwaresystemen zu integrieren.

Die Dritte Art von EJBs sind *Message Driven Beans*. Diese funktionieren ähnlich wie Session Beans, jedoch werden Prozesse hier asynchron im Stapelverarbeitungsmodus ausgeführt. Dies ist bei langwierigen Operationen, die keine weitere Interaktion benötigen, sinnvoll.

Um Komponenten, Anwendungsclients und Webprojekte in die entsprechenden Container installieren zu können müssen diese in Archive gepackt werden. Es gibt für jede Containerart ein passendes Archiv. Im Archiv sind alle benötigten Klassen und Ressourcen enthalten. Zusätzlich enthält jedes Archiv einen so genannten Deployment-Deskriptor, der den Inhalt des Archivs in einem XML-Format beschreibt. Für ein Web-Archiv werden hier Parameter wie etwa das Wurzelverzeichnis, die URL der Servlets usw. angegeben.

Für ein EJB-Archiv werden die einzelnen enthaltenen Beans beschrieben, d.h. die Art des EJBs und bei Entity Beans beispielsweise die persistenten Attribute, Schlüsselfelder und Anweisungen zur Abbildung dieser in die Datenbank.

Der Deskriptor wird vom Bean-Entwickler erstellt und mit Standardwerten gefüllt. Soll jedoch zu einem späteren Zeitpunkt aus den Beans eine Anwendung zusammengesetzt werden, so passt der Anwendungsentwickler die Deployment-Deskriptoren der verwendeten Beans an und stimmt sie aufeinander ab. Durch die Verwendung von Deskriptoren wird also erreicht, dass eine Komponente in ihre Umgebung eingepasst werden kann ohne den eigentlichen Code zu verändern.

Die Technologie von J2EE bietet einige Vorteile. Unter anderem wird die Entwicklung von Komponenten dadurch vereinfacht, dass häufig vorkommende technische Aspekte deklarativ ausgedrückt werden können und nicht im Anwendungscode implementiert werden müssen. Dies führt zum einen zu verkürzten Entwicklungszeiten, zum anderen zu einer höheren Lesbarkeit des Quellcodes.

Es gibt jedoch auch Nachteile der J2EE-Technologie. Beispielsweise ist die Granularität der eigentlichen Komponente – der Enterprise Bean – zu fein, da sie in der Regel aus nur einer Klasse besteht. Große Systeme können so schnell unübersichtlich werden. Eine Möglichkeit dies zu umgehen besteht darin zusammengehörende Beans zusammen in ein EJB-Archiv zu packen und eine Anwendung später aus EJB-Archiven zusammenzusetzen. Dies ist der Weg, der zur Realisierung des Komponentenbaukastens verwendet wird. Die Praxis zeigt, dass dieses Vorgehen möglich ist, jedoch wird es in der Literatur kaum beachtet. Hier wird in der Regel der Schwerpunkt auf die deklarativen technischen Möglichkeiten und dem damit verbundenen Rapid Application Development gelegt (z.B. in [RAJ01]).

Weiterhin werden durch das Komponentenmodell technische Einschränkungen impliziert. Beispielsweise ist es nicht ohne Weiteres möglich eine Bean mit zwei verschiedenen Remote-Schnittstellen zu versehen, da der Deskriptor nur eine Schnittstelle zulässt. Eine saubere Lösung lässt sich nur durch genaue Kenntnis der Spezifikation ableiten.

3.4 Web-Services

Der Begriff *Web-Service* ist sehr neu und einer der zentralen Begriffe in der jüngeren Entwicklung. Die dahinterstehende Technologie ist jedoch nur eine Kombination bereits bekannter einzelner Technologien, die zu einem sinnvollen Gesamtkonzept kombiniert wurden.

Motiviert wurden Web-Services durch die zunehmenden Anforderungen der Integration von heterogenen und stark verteilten Systemen. Ein oft angeführtes Beispielszenario ist die automatische Bestellung bei Zulieferern: Ein Kunde möchte bei einem seiner Zulieferer Bauteile nachbestellen. Hierbei soll das günstigste Angebot bei ausreichender Verfügbarkeit berücksichtigt werden.

Soll die Abwicklung dieses Szenarios automatisch erfolgen, so benötigt der Kunde zu jedem seiner Zulieferer eine Schnittstelle. Da die Schnittstellen in der Regel nicht standardisiert und oft sogar plattformabhängig sind und die Systeme sich in der Regel deutlich unterscheiden, ist die Integration kompliziert und erfordert gegebenenfalls komplexe Datenkonvertierungen.

Die technische Grundlage für Web-Services bildet die *Extensible Markup Language (XML)*, die seit einiger Zeit dazu verwendet wird Daten plattformunabhängig darzustellen und zu übertragen.

Web-Services sollen für Szenarien wie das oben beschriebene einfache Lösungen ermöglichen, da die Schnittstelle sowie die Daten in einer standardisierten Form, dem XML-Format WSDL (Web Service Description Language, [C$^+$01]), beschrieben werden. Darüber hinaus erfolgt auch die Anfrage und die Antwort durch die Übertragung von XML-Dokumenten, die gemäß dem Simple Object Access Protocol (SOAP, [B$^+$00]) formatiert sind.

Als Transportmedium wird in der Regel das aus dem Internet bekannte *Hypertext Transport Protocol (HTTP)* verwendet. Es können theoretisch jedoch auch andere Transportprotokolle wie beispielsweise das Email-Protokoll SMTP verwendet werden.

Die Benutzung von Web-Services erfolgt als entfernter Funktionsaufruf. Nach dem Standard speichert ein Web-Service keine Zustandsinformationen und besitzt daher auch nicht die Semantik eines Objekts im Sinn der Objektorientierung. Diese Tatsache und die momentan noch fehlende Möglichkeit der Authentifizierung sind die Schwachpunkte der aktuellen Spezifikation und eine Hemmschwelle für die Verbreitung von Web-Services.

Inzwischen haben die meisten Hersteller von Entwicklungsumgebungen für Web-Services proprietäre nicht standardkonforme Erweiterungen zur Lösung dieser Probleme erstellt. Die Nutzung dieser Möglichkeiten zieht jedoch wieder Inkompatibilität nach sich.

In der zukünftigen Version des Web-Service-Standards werden die Probleme jedoch gelöst werden und die Kompatibilität dadurch wieder hergestellt werden.

Über die eigentliche Implementierung der Web-Services wird in den Standards nichts gesagt. Eine Möglichkeit hierfür ist beispielsweise ein im Internet übliches CGI-Skript. Dieses nimmt eine HTTP-Anfrage entgegen und generiert als Antwort eine HTML-Seite. XML (bzw. SOAP) als Ein- und Ausgabeformat zu verwenden bedeutet hier nicht viel Aufwand und in der Tat werden von vielen Java-Servern Web-Services intern als Servlets implementiert.

Da die Implementierung von Web-Services nicht standardisiert ist (und auch nicht sein muss), sind Web-Services nicht als Komponenten im eigentlichen Sinn anzusehen. Weiterhin werden Web-Services auch nicht im Komponentensinn ausgeliefert oder für eine Umgebung konfiguriert. Vielmehr sind Web-Services eine Möglichkeit mit entfernten Komponenten zu kommunizieren und so die Integration mehrerer Systeme zu ermöglichen.

4 Realisierung des Komponentenbaukastens

In den folgenden Kapiteln wird nun die Realisierung des Komponentenbaukastens sowie der Beispielanwendung für die Verwaltung von Softwarepraktika beschrieben. Hierbei werden anhand der Phasen eines Softwareprozesses die jeweiligen Tätigkeiten beschrieben.

Der Schwerpunkt liegt hierbei auf der Vorgehensweise und nicht auf der Beschreibung der Ergebnisse. Daher wird zur Veranschaulichung jeweils nur ein kleiner Teil der Anwendung herausgegriffen und an diesem das Vorgehen demonstriert. Beispielsweise werden die meisten Diagramme nur kurz beschrieben und nicht, wie bei der eigentlichen Entwicklung üblich, ausführlich und vollständig dokumentiert. Die eigentlichen Entwicklungsdokumente sind auf der beiliegenden CD in elektronischer Form zu finden.

In Abbildung 4.1 ist eine Übersicht über den Entwicklungsprozess dargestellt. Man sieht, dass die einzelnen Dokumente entlang den Pfeilen voneinander abgeleitet werden. Die waagrechten Unterteilungen entsprechen den einzelnen Phasen. Die Dokumente werden in der Reihenfolge von links nach rechts und von oben nach unten fertig gestellt.

In der Spezifikation werden zuerst aus den Anforderungen Use Cases erstellt, aus denen dann das Business Object Model abgeleitet wird. Beide Dokumente bilden die Basis für das Begriffslexikon.

Im Entwurf wird das Business Obejct Model schrittweise verfeinert und transformiert, so dass eine Aufteilung der fachlichen Aspekte in Komponenten möglich wird. Weiterhin werden technische Komponenten aus den Use Cases abgeleitet.

Um die Schnittstellen der Komponenten zu finden werden die Use Cases durch Komponenteninteraktionen realisiert und daraus dann die Schnittstellen abgeleitet.

Für jede Komponente wird nun ein eigener, unabhängiger Entwurf erstellt, der zunächst wie der Schnittstellenentwurf plattformunabhängig ist. Die plattformunabhängigen Modelle werden dann in ein Modell der Zielplattform überführt.

Der so erstellte plattformabhängige Entwurf wird schließlich in der Java Entwrprise Edition implementiert. Aus der implementierung werden durch ein Werkzeug automatisch die Web-Services generiert.

Das hier geschilderte Vorgehen wurde aus verschiedenen Quellen abgeleitet. Zum einen orientiert sich der Prozess am Rational Unified Process[JBR99], zum anderen an einem speziell für Komponentenerstellung entwickelten Prozess, der in [CD01] beschrieben ist.

Der Komponentenbaukasten wurde in zwei Iterationen erstellt. In der ersten Iteration, die in der folgenden Beschreibung hauptsächlich beschrieben wird, wurde ein lauffähiges Grundsystem zur Verwaltung von Softwarepraktika entwickelt. In der zweiten Iteration wurde dieses Grundsystem um die Funktionalität erweitert Reviews verwalten zu können.

Auf diese zweite Iteration wird in der Beschreibung nicht näher eingegangen.

Das Vorgehen des Rational Unified Process wurde vor allem in den frühen Phasen angewendet um Use Cases zu erstellen und zu verfeinern und daraus die Verantwortlichkeiten der einzelnen Klassen abzuleiten.

Das Vorgehen um die Schnittstellen und Komponenten zu finden und zu entwerfen wurde insbesondere aus [CD01] abgeleitet. Hierbei wurde auch die dortige Notation verwendet um UML Komponenten in Diagrammen darstellen zu können.

Insgesamt wurde im Entwicklungsprozess darauf geachtet eine möglichst methodische Vorgehensweise auf Grund von Transformationen von UML-Modellen zu finden, wie dies auch von der modellgetriebenen Architektur (s. Abschnitt 3.2) propagiert wird.

Da die in der Aufgabenstellung geforderten Web-Services nicht auf einer spezifischen Architektur basieren, wurde als Basis das Komponentenmodell der Java 2 Enterprise Edition verwendet. Dieses bietet viele Vorteile gegenüber einer direkten Implementierung der Web-Services. Einerseits ist die Werkzeugunterstützung für J2EE deutlich besser als für Web-Services, andererseits bietet J2EE viele vorgefertigte Lösungen für technische Problemstellungen wie Sicherheit, Persistenz und Transaktionen an. Des weiteren können aus zustandslosen Session Beans automatisch Web-Services erzeugt werden, so dass dieser Schritt, der in Abschnitt 7.1 beschrieben ist, nur mit sehr geringem Aufwand verbunden ist.

Zur Modellierung wurden die Werkzeuge *ArcStyler* (s. Abschnitt 3.2.2) und *Rational Rose* verwendet.

Für die weitere Implementierung wurde das *IBM WebSphere Studio Application Developer* verwendet, das einige Aufgaben im Bereich der Implementierung von J2EE und Web-Services sehr gut unterstützt.

Für die automatschen Modultests wurde das Framework *JUnit* verwendet, das weiter unten genauer beschrieben wird.

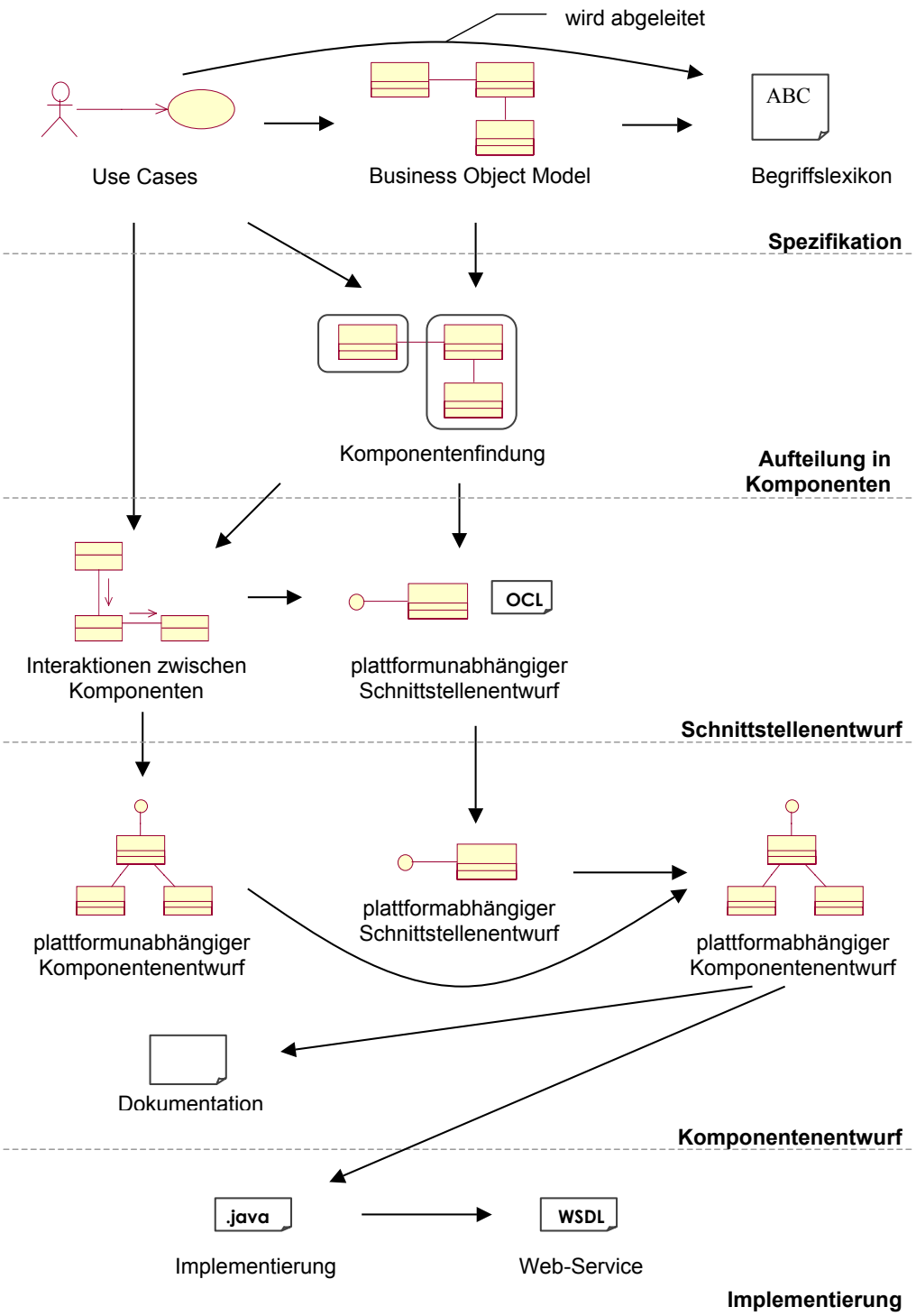


Abbildung 4.1: Überblick über die Realisierung

5 Anforderungen und Spezifikation

Die Anforderungsanalyse erfolgt nach den gängigen Methoden des Software Engineering. Anhand von Befragungen der Betreuer und mit Hilfe von eigenen Erfahrungen, die sowohl durch eigene Teilnahme am Softwarepraktikum als auch durch Hilfskrafttätigkeiten in der Vergangenheit vorhanden waren, wurden Ist- und Soll-Zustand analysiert.

Diese Analyse wird im Folgenden mit Hilfe von Use Cases und Geschäftsvorfällen beschrieben. Die Dokumentation der Use Cases erfolgt nach einem Schema, das an [KG00] angelehnt ist.

5.1 Ist-Zustand

An der Fakultät Informatik, Elektrotechnik und Informationstechnik am Institutsverband Informatik (IVI) wird jeweils im Sommersemester ein Softwarepraktikum für Studierende des Diplomstudiengangs Softwaretechnik angeboten. Zweck dieses Praktikums ist es an einem kleinen, in Gruppen zu je drei Personen durchgeführten Projekt den Softwareentwicklungszyklus praktisch zu üben. Das Softwarepraktikum wird hierbei von einer Abteilung mit einer gemeinsamen Aufgabenstellung für alle Gruppen durchgeführt.

Wichtiger Bestandteil eines Softwarepraktikums ist ein Review, ein Termin, an dem ein Dokument (meistens die Spezifikation) von Teilnehmern anderer Gruppen als Gutachter gelesen und geprüft wird. Die Ergebnisse dieser Prüfung werden an einem gemeinsamen Termin unter Anwesenheit eines Autors und unter Leitung eines Moderators vorgestellt und erörtert.

Während des Praktikums fallen verschiedene Tätigkeiten an, die eine Interaktion mit den Betreuern erfordern. Einige dieser Tätigkeiten werden als Geschäftsvorfälle in dem Use Case Diagramm in Abbildung 5.1 dargestellt und im Folgenden beschrieben.

Geschäftsvorfall:
Zu SoPra anmelden

Kurzbeschreibung:
Ein Student möchte an einem Softwarepraktikum teilnehmen und meldet sich dafür an.

Initiator:
Student

Ablauf:

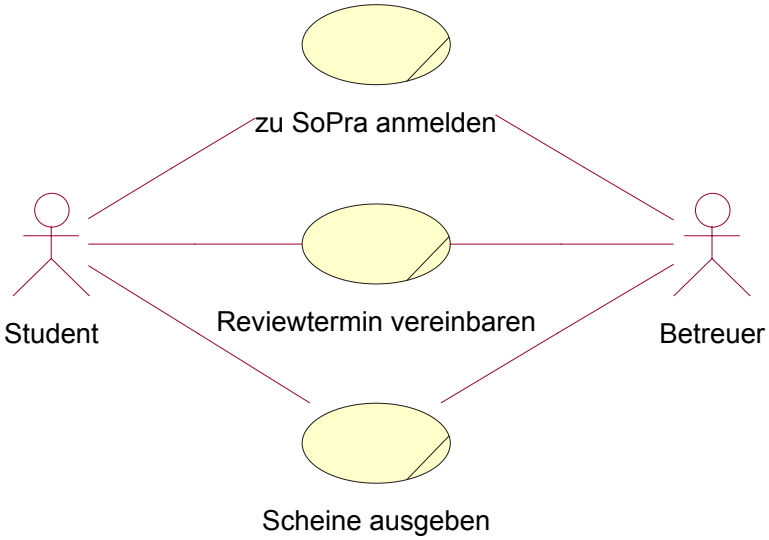


Abbildung 5.1: Use Cases: Ist-Zustand

1. Der Student sucht das Schwarze Brett mit den aushängenden Anmeldelisten und trägt sich in die gewünschte Gruppe ein.
2. Der Betreuer hängt nach Ablauf der Anmeldefrist die Listen ab und erfasst die Daten. Hierbei muss er evtl. an den ersten BEsprechungsterminen die Daten auf Richtigkeit prüfen und beispielsweise um Matrikelnummer und Emailadresse ergänzen lassen. Diese Änderungen muss er in seiner erfassten Liste nachführen.

Alternative Abläufe:

- Möchte sich ein Student anmelden nachdem die Listen abgehängt wurden, muss er direkt zum Betreuer gehen und sich bei diesem anmelden.

Geschäftsvorfall:
Review organisieren

Kurzbeschreibung:
Im laufenden SoPra soll ein Review organisiert werden.

Initiator:
Betreuer

Ablauf:

1. Der oder die Betreuer erstellen aus ihren Gruppenlisten eine Rolleneinteilung für die Reviews und geben diese Einteilung in einem Besprechungstermin bekannt.
2. Die jeweiligen Reviewteilnehmer holen sich bei den Betreuern die Emailadressen der anderen Teilnehmer ab um sich per Email auf einen Termin zu einigen.

3. Die Reviewteilnehmer teilen den Termin den Betreuern mit, die sie in ihrem Kalender festhalten.
4. Die Reviewteilnehmer suchen einen zum Reviewtermin freien Raum und reservieren diesen durch Eintragung auf der Reservierungsliste.

Alternative Abläufe:

- Der Betreuer muss die Reviewteilnehmer bei Verzögerung ggf. ermahnen den Termin fest zu legen.
- Terminänderungen erfordern die erneute Durchführung von 3. und 4.
- Mitteilungen an mehrere Gruppen werden wegen des hohen Aufwands an die Info-Mailingliste des Softwaretechnikstudiengangs verschickt.

Geschäftsvorfall:
Scheine ausgeben

Kurzbeschreibung:
Im abgeschlossenen SoPra sollen Scheine für alle erfolgreichen Teilnehmer erstellt werden.

Initiator:
Betreuer

Ablauf:

1. Die Betreuer überprüfen anhand ihrer Gruppenlisten, ob alle teilnehmenden Studenten die Scheinkriterien erfüllt haben und erstellen eine Scheinvorlage, in die jeweils die Daten der Studenten eingesetzt werden. Die Scheine werden dann ausgedruckt.
2. Die Betreuer geben die Scheine an das Sekretariat der Abteilung und teilen allen erfolgreichen Teilnehmern die Verfügbarkeit der Scheine per Email mit. Da der Aufwand ein Email an alle Empfänger eines Scheins zu hoch ist, wird eine Email an die Info-Mailingliste des Softwaretechnikstudiengangs verschickt.
3. Die teilnehmenden Studierenden holen die Scheine ab.

Es ist offensichtlich, dass bei der bisherigen Durchführung ein beträchtlicher Aufwand für Betreuer entsteht. Besonders die hier aufgeführten Fälle der Anmeldung und der Revieworganisation erfordern viel Kommunikationsaufwand einerseits und Verwaltungsaufwand um die Daten zu sammeln und manuell zu erfassen andererseits. Die Daten werden in der Regel aus den handschriftlich ausgefüllten Übungslisten manuell in eine Datenbank oder ein Tabellenverwaltungsprogramm übertragen und dort weiter verwendet. Es wäre wünschenswert die Datenerfassung direkt von den Studenten vornehmen zu lassen.

Es gab vor allem in jüngerer Zeit verschiedene Ansätze am IVI ähnliche Verwaltungsaufgaben für diverse Lehrveranstaltungen teilweise zu automatisieren. Zwei der im größeren Rahmen eingesetzten Anwendungen sind die elektronische Eintragung in die Übungsgruppen, das von einer studentischen Hilfskraft der Abteilung Betriebssysteme entwickelt wurde und das von der Abteilung Formale Konzepte in einem Studienprojekt erstellte System „Computerbasierter Übungsbetrieb (ECÜ)".

Diese Systeme sind jedoch Einzellösungen, die für eine bestimmte Aufgabe (z.B. die Eintragung in Übungsgruppen) oder eine bestimmte Lehrveranstaltung (bei ECÜ die Einführung in die Informatik) zugeschnitten und nicht in erster Linie für andere Zwecke verwendbar oder anpassbar sind.

5.2 Soll-Zustand

Es soll nun ein System entwickelt werden, das den Verwaltungsaufwand für Betreuer von Softwarepraktika reduziert. Dieses System soll insbesondere die Verwaltung der teilnehmenden Studierenden und die damit verbundene Kommunikation zwischen Studierenden und Betreuern übernehmen.

Weiterhin soll das System nicht als Einzellösung konzipiert sein, sondern die Basis für eine baukastenähnliche Komponentenarchitektur werden, aus der weitere Systeme zur Unterstützung von Geschäftsprozessen im universitären Bereich erstellt werden können. Durch die gemeinsame Basis solcher Anwendungen soll auch eine etwaige Integration erleichtert werden.

Die Architektur soll nach außen hin ein Web-Interface besitzen, über das Studierende und Betreuer auf das System zugreifen können. Zum anderen soll es mit Web-Service-Schnittstellen ausgestattet sein um auch mit weiteren Anwendungen relativ einfach integriert werden zu können. Gleichzeitig soll sie auch mit bestehenden Systemen, insbesondere mit dem Raumplanungssystem „Rabe"[Tög02], das um entsprechende Schnittstellen erweitert wird[Küp03], integriert werden können.

Schließlich soll das System mit IBM-Produkten entwickelt und auf einem UNIX-Server von Sun ausgeführt werden. Hierbei kommen insbesondere das *WebSphere Studio Application Developer* als Entwicklungswerkzeug und der *WebSphere Applikationsserver* zum Einsatz um eine Komponentenarchitektur auf der Basis der Java 2 Enterprise Edition zu erstellen und auszuführen.

Mit diesen Rahmenbedingungen und den analysierten aktuellen Prozessen werden die oben beschriebenen Geschäftsvorfälle in Use Cases überführt um die funktionalen Anforderungen zu identifizieren. Im Folgenden ist der Use Case „Zu SoPra anmelden" stellvertretend beschrieben:

Use Case:
Zu SoPra anmelden

Kurzbeschreibung:
Ein Student möchte an einem SoPra teilnehmen und meldet sich dafür an.

Initiator:
Student

Ablauf:

1. Der Student wählt die Veranstaltung zu der er sich anmelden möchte.
2. Das System zeigt Gruppennummern in der Veranstaltung an in denen noch Plätze frei sind und fragt den Benutzer nach der gewünschten Gruppennummer.
3. Der Student gibt die gewünschte Gruppennummer ein.
4. Das System fordert eine Bestätigung vom ersten Mitglied der Gruppe an.
5. Das System informiert den Studenten, dass die Gruppe noch bestätigt werden muss und er auf eine Bestätigungsmail warten soll.
6. Das erste Mitglied der Gruppe bestätigt die Aufnahme.
7. Das System bestätigt dem Student als neuem Teilnehmer die Gruppenmitgliedschaft per Mail.

Format der Eingabedaten:
Keine direkte Eingabe, nur Auswahl aus einer Liste

Alternative Abläufe:

- Der Student kann sich auch anmelden, indem er einen externen Link verfolgt, der die Anmeldung zu einer bestimmten Veranstaltung einleitet. Die Auswahl der Veranstaltung (1.) entfällt dann.
- Der Student kann bei 3. anstatt eine vorhandene Gruppe auszuwählen, eine neue Gruppe eröffnen. Er wird dann sofort in eine neue Gruppe angemeldet und erhält sofort eine Bestätigung (siehe 7.).
- Das erste Gruppenmitglied kann bei 6. die Aufnahme des Studenten ablehnen. In diesem Fall bekommt der Student eine Mail, die ihn über die negative Antwort informiert und ihn auffordert, sich noch einmal in eine andere Gruppe anzumelden.

Fehler und Ausnahmen:

- Wenn die Anmeldefrist zu einer Veranstaltung abgelaufen ist, so erhält der Student eine Meldung, die besagt, dass die Frist abgelaufen ist mit dem Hinweis, dass er sich an die Betreuer wenden soll.
- Wenn der Student bereits an dieser Veranstaltung als Student teilnimmt, so wird eine entsprechende Fehlermeldung ausgegeben.

Annahmen:

- Es wird davon ausgegangen, dass Studenten nicht unnötigerweise neue Gruppen anlegen.

Vorbedingungen:

- Der Student ist am System angemeldet und authentifiziert.
- Der Student ist noch nicht als Teilnehmer an der ausgewählten Veranstaltung angemeldet.

Nachbedingungen:

- Der Student hat eine Email erhalten, in der entweder bestätigt wird, dass er an der Veranstaltung in der gewählten Gruppe angemeldet ist oder dass die Anmeldung nicht möglich war, weil er von der Gruppe abgelehnt wurde.
- Die Teilnahme des Studenten an der Veranstaltung wurde im System festgehalten.

Änderungen:

- 24.9.2002: Use Case erstellt
- 24.2.2003: Use Case überarbeitet

Realisiert durch Iteration:
1

Offene Fragen:
–

Aus den Use Cases lässt sich die technische Notwendigkeit weiterer Anforderungen ableiten, beispielsweise Use Cases zur Verwaltung von Benutzern und deren Eigenschaften, wie Emailadressen und Matrikelnummern. Diese Use Cases werden hier jedoch nicht näher beschrieben.

Abbildung 5.2 zeigt das vollständige Use Case-Diagramm der SoPra-Verwaltung.

Ergänzend zu den Use Cases wird ein Oberflächenprototyp erstellt, der gleichzeitig die Funktion des Storyboards übernimmt, also Aussehen, Struktur und Abfolge der Benutzungsschnittstelle und Benutzerinteraktionen werden hiermit festlegt. Da die Benutzungsschnittstelle als Web-Interface realisiert werden soll, eignen sich für den Oberflächenprototyp einfache statische HTML-Seiten, die durch JavaScript mit grundlegender Funktionalität versehen werden.

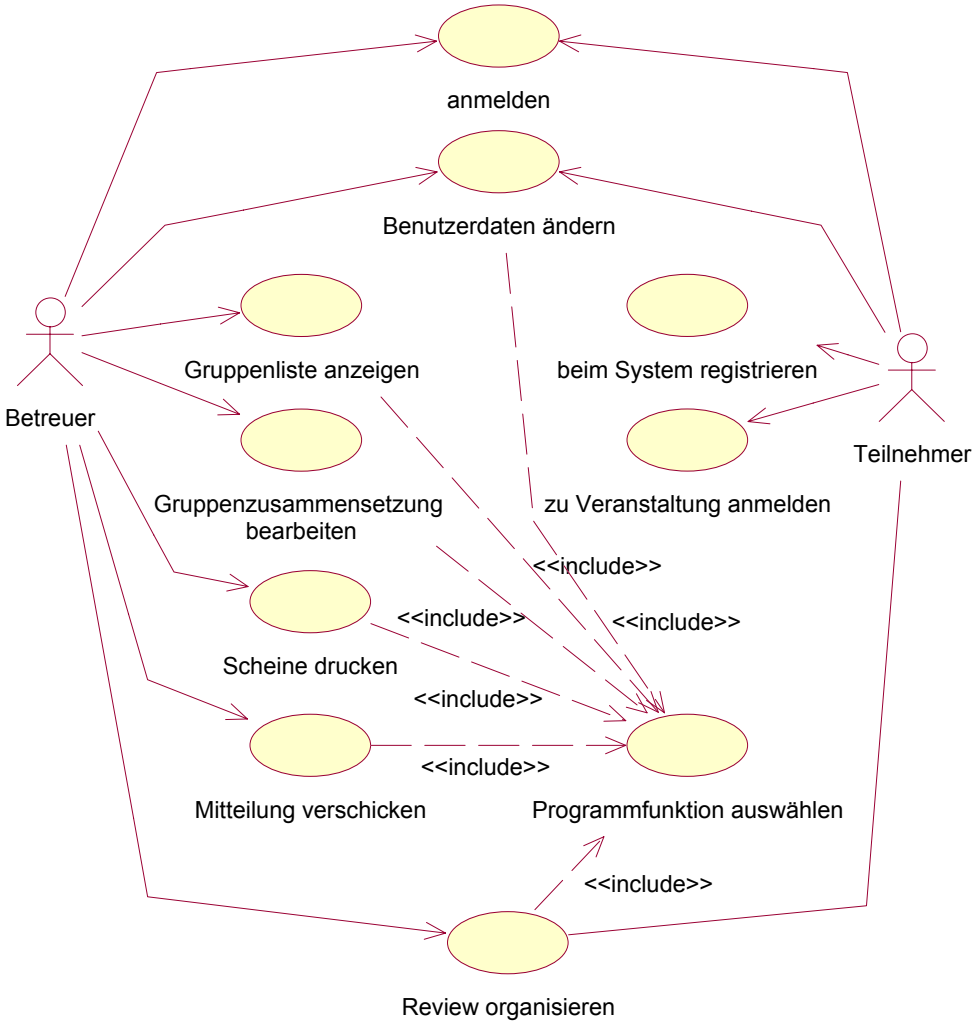


Abbildung 5.2: Use Cases: Soll-Zustand

Schließlich muss noch ein *Business Object Model* erstellt werden. In diesem werden Dinge und Ereignisse aus der Umgebung des Systems dargestellt und zueinander in Beziehung gesetzt. Hierzu werden die Use Cases auf mögliche Begriffe hin untersucht und diese in Beziehung zueinander gesetzt. Das Business Object Model wird in Abbildung 5.3 dargestellt.

Das Diagramm zeigt das Softwarepraktikum (SoPra) und ein Review oben, sowie einen Betreuer und einen Studenten unten. Ein Betreuer betreut Softwarepraktika, an denen Studenten in Gruppen teilnehmen. In einem Softwarepraktikum kann es Reviews geben, an denen die Studenten in einer Rolle als Autor, Moderator oder Gutachter teilnehmen. Das hier dargestellte Modell kann noch weiter verfeinert und durch Attribute ergänzt werden, wie beispielsweise der Matrikelnummer als Attribut des Studenten. Dies sollte jedoch nur an den Stellen geschehen, an denen die Existenz der Attribute eindeutig und intuitiv klar

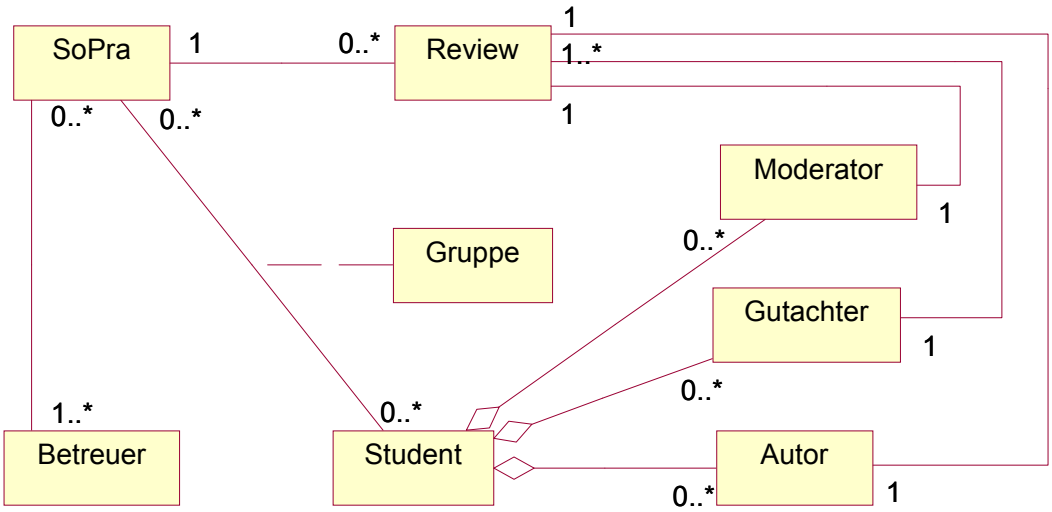


Abbildung 5.3: Business Object Model

ist. Der Datentyp des Attributs ist in dieser Phase ebenfalls irrelevant und kann daher weggelassen werden. Die Verfeinerung des Modells mit Attributen und Regeln geschieht erst in der Entwurfsphase.

Aus den Use Cases und dem Business Object Model wird zusätzlich noch ein Begriffslexikon erstellt, das mindestens alle Begriffe aus dem Business Object Model enthält, jedoch sinnvoll durch Begriffe, die in den Use Cases verwendet werden, ergänzt wird.

6 Entwurf

In diesem Kapitel werden nun aus den in der Anforderungsanalyse entstandenen Dokumenten Schritt für Schritt Komponenten, Schnittstellen und die Architektur entwickelt.

6.1 Verfeinerung des Business Object Model

In diesem ersten Entwurfsschritt wird das rudimentäre Business Object Model der Analyse durch Attribute und Datentypen verfeinert. Abbildung 6.1 zeigt exemplarisch die Verfeinerung von *Instructor* und *Student*. Da die Entwicklungssprache Englisch ist, wurde mit Übergang zum Entwurf auch Englisch als Sprache für die Namensgebung von Modellelementen gewählt.

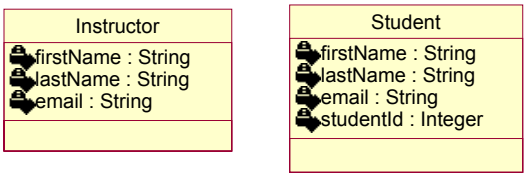


Abbildung 6.1: Verfeinerung der Klassen im Business Object Model

Weiterhin können Regeln und Einschränkungen, die nicht mit grafischen Mitteln in UML ausgedrückt werden können mit der Object Constraint Language[Obj01b] formal spezifiziert werden:

```
context Student
inv: studentId >= 1000000 and studentId <=9999999
```

Dieser Ausdruck legt fest, dass die Matrikelnummer des Studenten in jedem Fall aus sieben Ziffern bestehen muss.

Aus Gründen der Übersichtlichkeit werden in einigen der folgenden Diagramme die Attribute nicht dargestellt.

6.2 Aufteilung des Systems in Komponenten

Die Aufteilung des Gesamtsystems in Komponenten erfolgt bei der Erstellung des Komponentenbaukastens vor allem aus zwei Motiven heraus. Zum einen sollen die entstehenden

Komponenten wiederverwendbar sein. Es sollen zukünftig weitere Anwendungen auf den bestehenden Komponenten aufbauen können. Zum anderen sollen insbesondere technische Entwurfsentscheidungen so in Komponenten gekapselt werden, dass sie ersetzbar sind. Systeme, die auf diesen Komponenten aufbauen, werden damit technologieunabhängiger und flexibler.

Somit werden bei der Identifikation der Komponenten also zwei verschiedene Kriterien verwendet: die Kapselung von relativ unabhängigen fachlichen Konzepten zur unabhängigen Wiederverwendung einerseits und die Kapselung von technischen Entwurfsentscheidungen und Technologieabhängigkeit andererseits.

Beide werden in den folgenden Abschnitten genauer betrachtet.

6.2.1 Kapselung technischer Aspekte

Bevor die technischen Aspekte identifiziert werden können, müssen die Rahmenbedingungen geklärt werden. Hierzu müssen die Anforderungen auf technische Bedingungen hin untersucht werden und ein Konzept zur technischen Umsetzung erstellt werden. Insbesondere müssen die Bedingungen konkretisiert und geklärt werden, beispielsweise welche Fremdprodukte zum Einsatz kommen sollen. Im Fall der hier zu erstellenden Architektur ist die Bindung an den WebSphere Applikationsserver von IBM bereits vorgegeben.

Die zu kapselnden technischen Aspekte sind im Fall des zu erstellenden SoPra-Verwaltungssystems die Möglichkeit des Emailversands und die Authentifizierung, sowie eines Dienstes zur Abwicklung von Bestätigungen per Email.

Die so identifizierten zukünftigen Komponenten werden dann kurz dokumentiert, hier am Beispiel der Bestätigungsabwicklung.

Name der Komponente:

Confirmation

Kurzbeschreibung:

Abwicklung von asynchronen Bestätigungen, die nicht direkt interaktiv über die Benutzungsschnittstelle durchgeführt werden können. Hierbei wird eine Person dazu aufgefordert binnen eines bestimmten Zeitraums (in der Größenordnung von einigen Tagen) die Anfrage zu beantworten. Für Anfrage und Antwort wird ein asynchrones Medium (z.B. Email) verwendet.

Realisierungsansatz:

Die Bestätigung wird über die Messaging-Komponente z.B. per Email angefordert. Die Antwort wird über einen in der Email mitgeschickten Link eingegeben und verarbeitet.

Alternative Realisierungsmöglichkeiten:

Nutzung einer anderen Kommunikationsform (z.B. SMS) sowohl für die Aufforderung als auch für die Antwort.

Die Aufgabe dieser kurzen Dokumentation ist es zum einen, die Grenzen und die Verantwortlichkeiten der Komponente festzulegen, zum anderen eine Grundlage für die Schnittstellenspezifikation zu bilden. Insbesondere die beiden Punkte *Realisierungsansatz* und *Alternative Realisierungsmöglichkeiten* dienen dazu, dass beim Spezifizieren der Schnittstellen darauf geachtet werden kann, dass diese nicht zu speziell auf die beabsichtigte Realisierung abgestimmt werden. Statt dessen sollten auch die alternativen Möglichkeiten mit der Schnittstelle realisierbar sein. Der hierbei erreichte Abstraktionsgrad verbessert die Ersetzbarkeit.

6.2.2 Kapselung fachlicher Aspekte

Bevor die fachlichen Komponenten identifiziert werden, wird zuerst versucht eine allgemeinere und damit breiter ausgelegtere Basis der fachlichen Funktionalität zu finden. Hierzu wird das in der Analysephase erstellte Business Object Model untersucht und unter Anwendung einiger Entwurfsmuster transformiert.

In einem ersten Schritt werden Gemeinsamkeiten der dargestellten Objekte identifiziert. Objekte, die mit „ist-ein(e)" zu Oberbegriffen in Beziehung gesetzt werden können, können so mit Generalisierungen dargestellt werden.

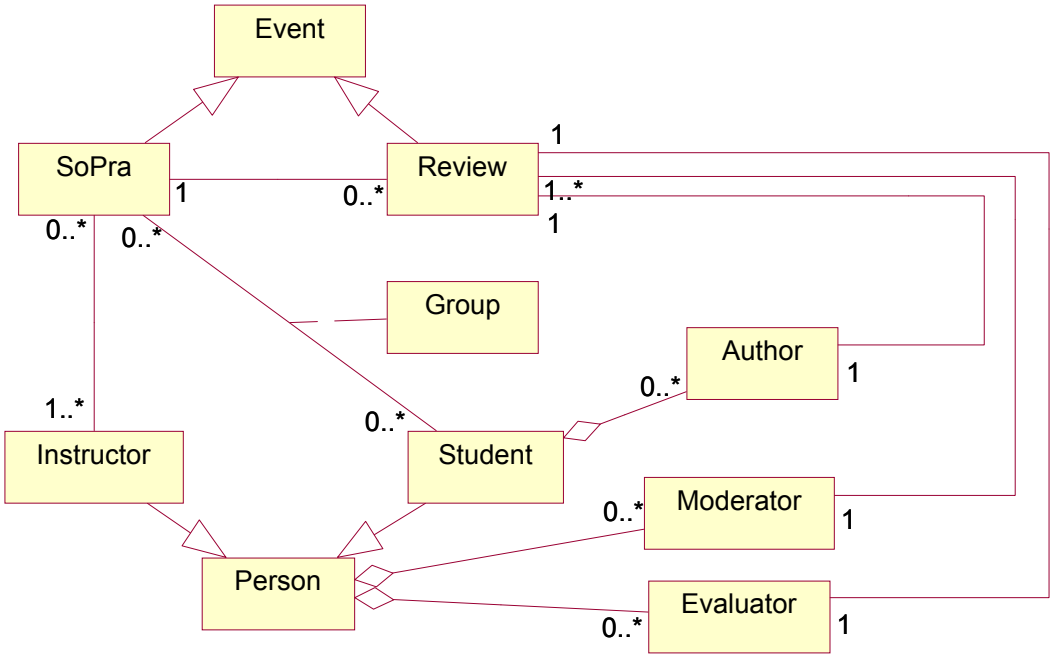


Abbildung 6.2: Transformation des Business Object Model – Schritt 1

In Abbildung 6.2 wurden gegenüber dem Business Object Model der Analyse *Person* als Oberbegriff von *Student* und *Instructor*(Betreuer), sowie *Event*(Veranstaltung) als Oberbegriff von *SoPra* und *Review* identifiziert.

Weiterhin können im Business Object Model dargestellte Attribute und Rollen ggf. in

der Hierarchie verschoben werden um die Flexibilität weiter zu erhöhen. Im obigen Diagramm wurden beispielsweise die Rollen *Gutachter* und *Moderator* der *Person* zugeordnet. Hierdurch entsteht nun die Möglichkeit auch einen Betreuer in einem Review als Moderator oder Gutachter einzusetzen.

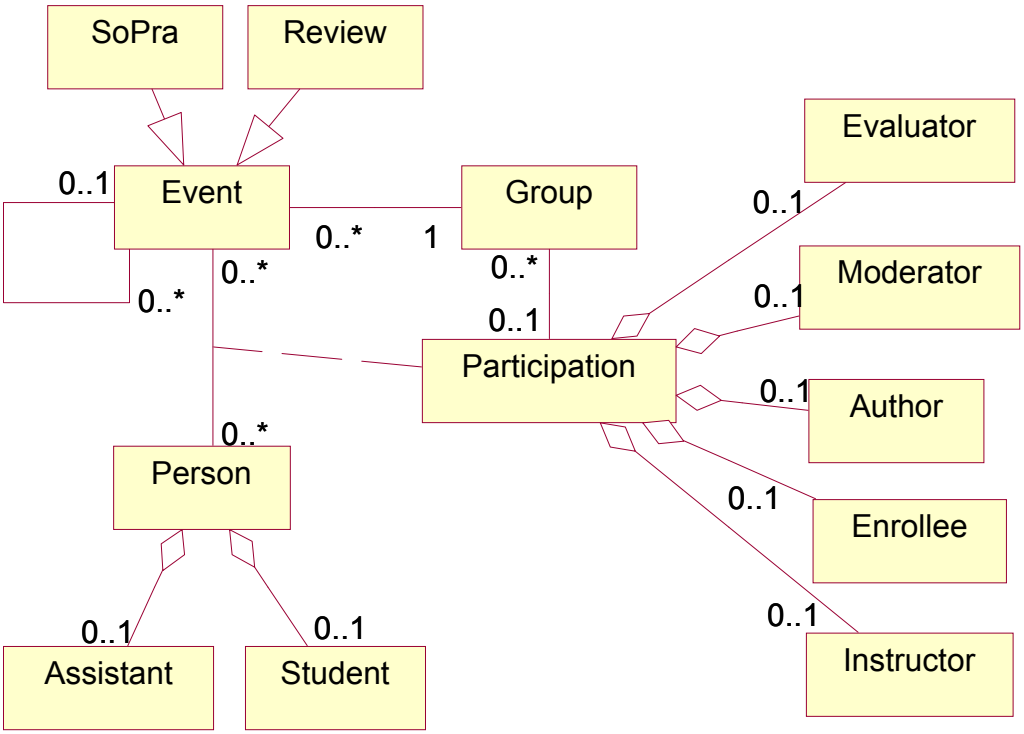


Abbildung 6.3: Transformation des Business Object Model – Schritt 2

In einem weiteren Schritt, der in Abbildung 6.3 dargestellt ist, werden die Konzepte, die als Oberbegriffe identifiziert wurden, vereinheitlicht. Im Kern dieser abstrakteren Sicht steht eine Person, die an einer Veranstaltung teilnimmt. Diese Teilnahme wird weiter spezifiziert durch eine Rolle (Hörer einer Vorlesung, Moderator eines Reviews, etc.) und einer eventuellen Zugehörigkeit zu einer Gruppe innerhalb einer Veranstaltung.

Einige Einschränkungen, die in früheren Schritten durch die Syntax ausgedrückt wurden, sind nun jedoch weggefallen. Beispielsweise kann ein Betreuer kein Gruppenmitglied sein, oder ein Review muss genau einen Teilnehmer als Moderator haben. Solche Einschränkungen können wie die bereits in der Verfeinerung erstellten Regeln mit Hilfe von Ausdrücken in der Object Constraint Language formuliert werden:

```
context Participation
inv: instructor <> null implies group = null
```

```
context Event
inv: self instanceof Review implies
  participations->one(p | p.moderator <> null)
```

Eine weitere Änderung in diesem Schritt ist der Wegfall der Generalisierungen bei der Person. Diese wurden durch Aggregationen ersetzt. Dies hat zur Folge, dass eine Person nun sowohl Student (mit Matrikelnummer) als auch Dozent (z.B. mit Personalnummer o.ä.) sein kann. Dies ist z.B. der Fall, wenn Studenten im Rahmen von wissenschaftlichen Hilfskrafttätigkeiten universitäre Veranstaltungen betreuen.

Bei der Veranstaltung ist die Umformung der Generalisierung dagegen nur bedingt sinnvoll, da ein SoPra beispielsweise nicht gleichzeitig ein Review sein kann.

Die Zugehörigkeitsbeziehung zwischen SoPra und Review wurde zu einer allgemeinen Beziehung zwischen Veranstaltungen umgeformt. Durch diese Beziehung können Veranstaltungen hierarchisch beliebig gegliedert werden, so dass auch hier Flexibilität gewonnen wurde.

Aus dem ursprünglichen Business Object Model ist nun durch Umformungen ein sehr viel flexibleres und vielseitiger einsetzbares Modell entstanden, das nun als Grundlage für die weitere Komponentenaufteilung dient.

Diese Aufteilung geschieht im nächsten Schritt, indem zuerst Kernklassen identifiziert werden. Kernklassen sind Klassen im Modell, die (auch und gerade in der Realität) unabhängige Einheiten bilden, die höchstens von klassifizierenden oder näher bestimmenden Klassen abhängen. Diese Kernklassen sind in unserer Anwendung konkret *Person* und *Event*. Die Rollen und Unterklassen sind hier nur näher bestimmend. Die Assoziationsklasse *Participation* ist keine unabhängige Einheit und *Group* ebenfalls nicht – sie bestimmt die Teilnahme näher.

Nachdem die beiden Kernklassen identifiziert wurden, müssen nun die Verantwortlichkeiten bestimmt werden. Jede Nicht-Kernklasse muss einer für sie verantwortlichen Kernklasse zugeordnet sein. Für die Rollen und Unterklassen ist diese Zuordnung trivial. Die Klasse *Participation* könnte sowohl *Person* als auch *Event* zugeordnet werden. Da sie durch die bestimmende Klasse *Group* jedoch „näher" an *Event* gebunden ist, bietet sich die letztere Zuordnung an. Die vollzogene Trennung der Komponenten ist in Abbildung 6.4 dargestellt.

Der letzte Teilschritt der Komponentenbestimmung betrifft die Assoziation zwischen *Person* und *Event*. Diese ist auf logischer Ebene bidirektional – eine Person nimmt an Veranstaltungen teil und Veranstaltungen haben teilnehmende Personen. Auf formeller Ebene muss die Assoziation jedoch in eine einseitige umgewandelt werden, da eine bidirektionale Assoziation auch eine bidirektionale und damit zyklische Abhängigkeit bedeutet.

Zur Auflösung der Assoziation werden die beteiligten Klassen untersucht und festgestellt, welche der beiden Klassen in der Realität ohne die jeweils andere „besser" existieren kann. Im SoPra-Beispiel können Personen ohne Veranstaltungen durchaus existieren, jedoch sind Veranstaltungen ohne Personen schwer vorstellbar. Der Fall muss nicht so klar aussehen wie im Beispiel. In solchen Fällen muss die Auflösung der Bidirektionalität subjektiv oder durch Betrachten der technischen Konsequenzen geschehen.

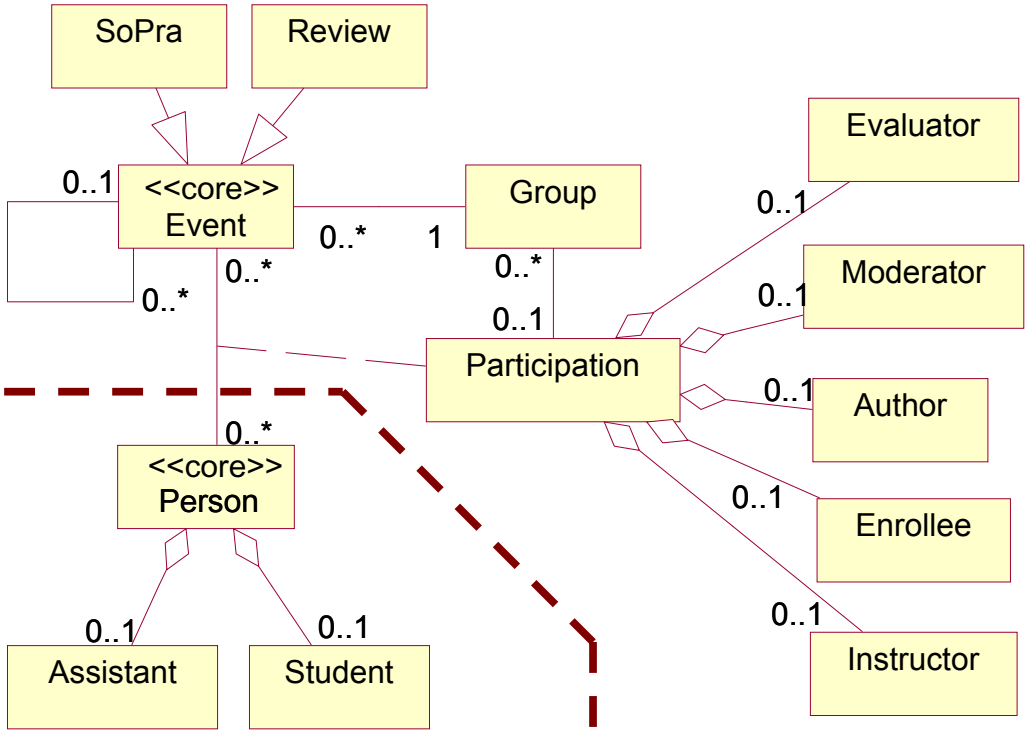


Abbildung 6.4: Trennung der fachlichen Aspekte in Komponenten

Die im letzten Abschnitt begonnene Liste der Komponenten kann nun durch die beiden neuen Komponenten ergänzt werden, wie in der folgenden Beschreibung beispielhaft für *PersonManagement* dargestellt:

Name der Komponente:
PersonManagement

Kurzbeschreibung:
Verwaltung von Personen, die in einem System im universitären Bereich erfasst sind, inklusive ihrer Daten. Die Verwaltung umfasst die Erfassung, dauerhafte Speicherung, Möglichkeit der Änderung, Löschung sowie den Zugriff auf Eigenschaften wie Name usw. von Personen.

6.3 Die Architektur

Nachdem alle nötigen Komponenten identifiziert wurden, ist es möglich eine Architektur aus den Komponenten aufzubauen. Hierzu wird für jede Komponente vorerst eine leere

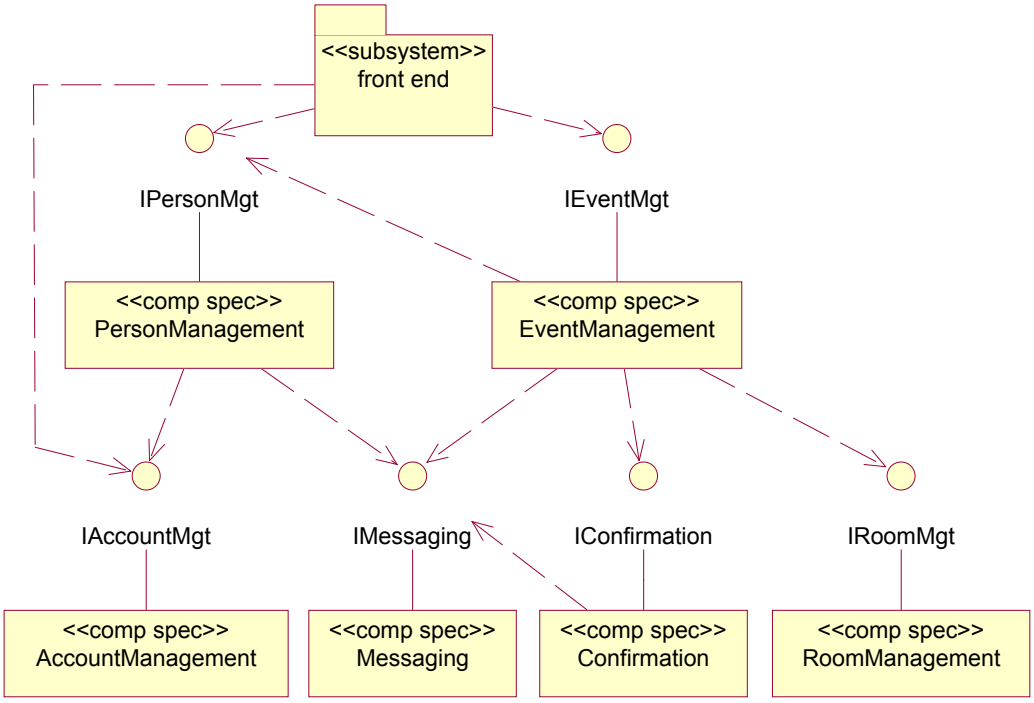


Abbildung 6.5: Die Architektur

Schnittstelle erstellt, durch die die Komponenten in Abhängigkeit zueinander gesetzt werden. Das Ergebnis ist in Abbildung 6.5 dargestellt.

Die Benutzungsschnittstelle (*front end*) greift auf die Komponenten *PersonManagement*, *EventManagement* und *AccountManagement* zu. Dies geht aus den Use Cases hervor, dient jedoch nur der Vervollständigung. Die verfeinerte Interaktion zwischen Benutzungsschnittstelle und zwischen den einzelnen Komponenten wird später entwickelt.

Die Komponenten sind hier als Klassen mit dem Stereotyp *«comp spec»* dargestellt. Die Gründe für diese Darstellung werden in Abschnitt 3.1.1 beschrieben.

6.4 Entwurf der Schnittstellen

Es hängt von den Operationen ab, die von anderen Komponenten und von der Benutzungsschnittstelle benötigt werden, welche Operationen von den Komponentenschnittstellen bereit gestellt werden sollen. Um diese Operationen zu finden werden die Use Cases einzeln betrachtet und eine Realisierung wird durch Interaktionsdiagramme entworfen. Hierbei können auch für einen Use Case mehrere Interaktionsdiagramme modelliert werden, die verschiedene Szenarien oder Abläufe darstellen. Dies hängt von der Komplexität der einzelnen Use Cases ab.

Hierbei wird gleich das durch die Java Enterprise Edition vorgegebene Schichtenmodell mit verwendet: Die Benutzer-System-Interaktionen werden mit so genannten Boundary-Klassen realisiert, die dann auf die einzelnen Komponenten zugreifen. Diese Zugriffe er-

folgen auf Control-Klassen, die Geschäftsprozesslogik kapseln. Als Basis der Datenschicht dienen die zuvor identifizierten Klassen des Business Object Model in Form von Entity-Klassen. Weiterhin wird die zuvor erstellte Architektur als Basis für die Klassenaufteilung und Interaktion verwendet.

Abbildung 6.6 zeigt eine Verfeinerung des Use Case „Zu Veranstaltung anmelden". Die einzelnen Schritte im Diagramm sind im Folgenden dokumentiert. In Klammern wird jeweils der entsprechende Interaktionsschritt angegeben.

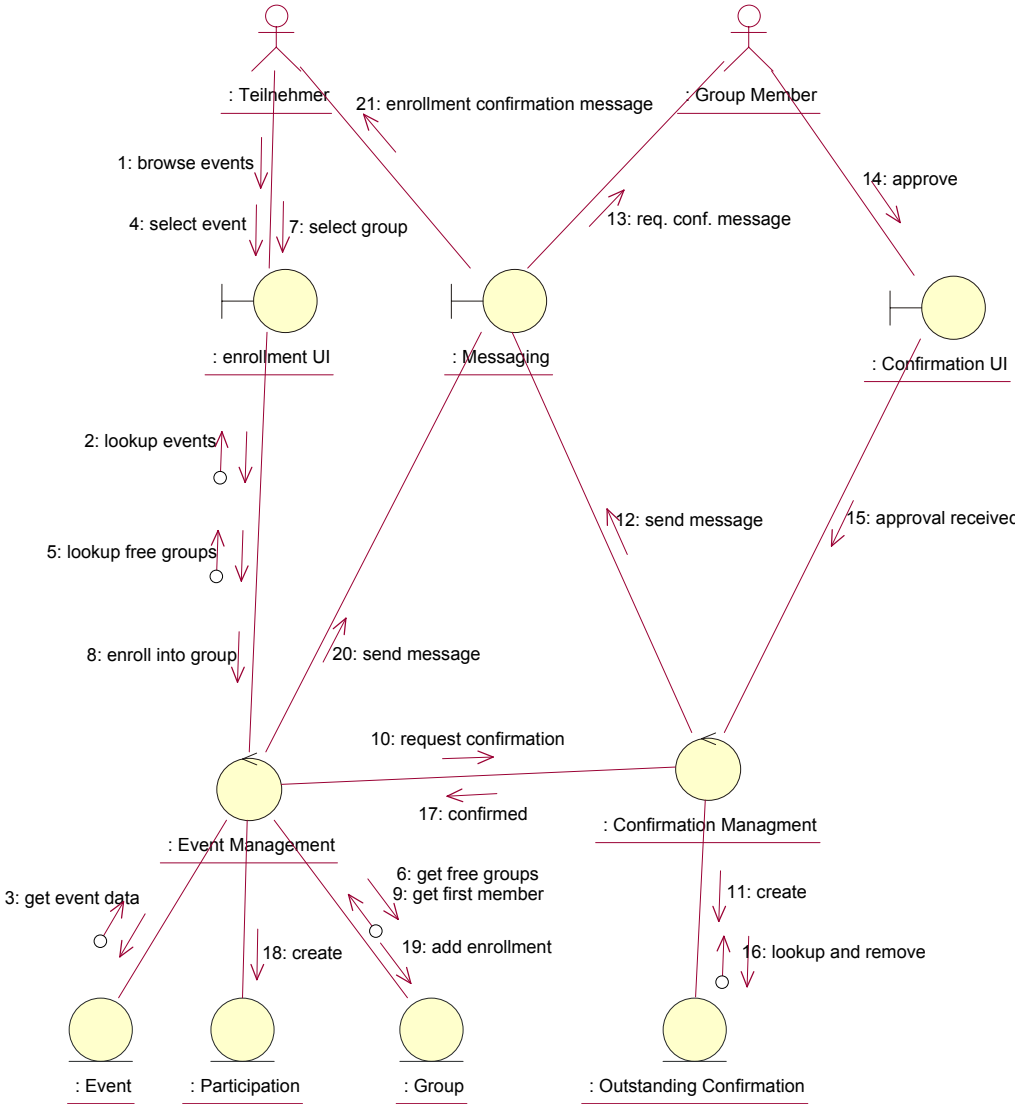


Abbildung 6.6: Verfeinerung des Use Case „Zu Veranstaltung anmelden"

Use Case Realisierung:
enroll

realisierter Use Case:
Zu Veranstaltung anmelden

Beschreibung:
Die Schritte 1 bis 4 sind optional - eine Vorlesung könnte auch von der Vorlesungshomepage aus vorausgewählt worden sein.

- Ein Benutzer fordert die Liste der Veranstaltungen an (1),
- Die Benutzungsschnittstelle holt sich die Veranstaltungen (2, 3).
- Der Benutzer wählt aus der Liste die gewünschte Vorlesung aus (4).
- Die Benutzungsschnittstelle gibt die Auswahl an den EventManager weiter (5).
- Dieser sucht alle Gruppen der Veranstaltung, die noch einen freien Platz haben und übermittelt sie an die Benutzungsschnittstelle zur Auswahl (6).
- Der Benutzer wählt eine Gruppe aus (7).
- Alternativ kann er auch eine neue Gruppe erstellen – in diesem Fall wird eine neue Gruppe erstellt und er wird vom EventManager an diese angemeldet (17-20).
- Die getroffene Gruppenauswahl wird an den EventManager übermittelt (8).
- Dieser fordert nun über das Confirmation Management eine Bestätigung an (9).
- Der Confirmation Manager erstellt eine offene Bestätigung (10) und sendet über das Messaging (11) eine Aufforderungsnachricht (12).
- Das Gruppenmitglied bestätigt die Aufnahme in die Gruppe über das Confirmation UI (13, 14).
- Die Daten der Bestätigung werden überprüft und die offene Bestätigung wird gelöscht (15).
- Der Confirmation Manager meldet die Bestätigung an den EventManager (16). (Im Falle einer Ablehnung benachrichtigt dieser den Teilnehmer, dass er sich eine andere Gruppe suchen muss.)
- Die Teilnahme wird nun erzeugt (17), der Teilnehmer in die Gruppe angemeldet (18) und die Bestätigung über die erfolgreiche Anmeldung an den Teilnehmer verschickt. (19,20)

Aus den dokumentierten Interaktionsdiagrammen können nun die Verantwortlichkeiten der einzelnen Klassen zusammengestellt werden, indem jede eingehende Interaktion in der Dokumentation gesucht und mit der Referenz auf diese in die Liste der Klassenverantwortlichkeiten übertragen wird. Diese Liste wird außerdem durch eventuelle nichtfunktionale Anforderungen aus der Spezifikation und durch alternative Interaktionsschritte ergänzt. Im folgenden Beispiel wurde dies für die Klassen EventManagement und Confirmation-Management getan. In Klammern wird die Referenz zum jeweiligen Interaktionsschritt aufgeführt.

Klasse:
EventManagement

Beschreibung:
Die Fassade der Veranstaltungsverwaltung. Die Veranstaltungsverwaltung verwaltet Veranstaltungen und Anmeldungen zu den Veranstaltungen.

Zuständigkeiten:

- Veranstaltung erstellen und Betreuer anmelden (createEvent:9)
- Liste aller Veranstaltungen erstellen und zurückgeben (select function, enroll:2)
- Liste aller Veranstaltungen, zu denen eine Person angemeldet ist, erstellen und zurückgeben (select function:1)
- Liste aller Gruppen zu einer Veranstaltung zurückgeben, in denen noch mindestens ein Platz frei ist (enroll:5)
- An ein Softwarepraktikum in eine bestimmte Gruppe anmelden (enroll:8)

Klasse:
ConfirmationManagment

Beschreibung:
Fassade der Bestätigungsverwaltung. Die Bestätigungsverwaltung fordert Bestätigungen über die Benachrichtigungskomponente an und verarbeitet die Antworten.

Zuständigkeiten:

- Bestätigung anfordern (enroll:10)
- Antwort entgegennehmen (enroll:15)
- auf ausbleibende Antwort reagieren (nichtfunktionale Anforderungen)

Aus den Verantwortlichkeiten der einzelnen Klassen, insbesondere der Fassadeklassen der Komponenten, können nun die Operationen der Schnittstellen abgeleitet werden. Fassadeklassen sind die Klassen, die die Prozesslogik der Komponenten realisieren und damit die Komponentenschnittstellen implementieren. Hierzu wird jede Verantwortlichkeit auf benötigte Parameter untersucht und damit die Signatur der Operation gebildet. Wie bereits in Abschnitt 6.2.1 beschrieben wurde, ist hierbei gegebenenfalls noch darauf zu achten, dass bei Schnittstellen technischer Komponenten alternative Realisierungen insbesondere bei der Wahl der Parametertypen zu berücksichtigen sind, damit die Schnittstellen abstrakt in Bezug auf die verwendete Technologie sind.

Um Daten zu übertragen, die über einfache Datentypen wie Zeichenketten, Zahlen oder Listen hinaus gehen müssen zusätzlich so genannte *Value Objects* (engl. Wertobjekte) entworfen werden. Diese Objekte sind in verschiedenen Entwurfsmustern (siehe beispielsweise [Mar02a]) zum Transport von Daten enthalten. Sollen z.B. die Daten einer Person (Name, Emailadresse, Matrikelnummer etc.) durch eine Operation abgefragt werden können, so ist es sinnvoll ein solches *Value Object* zu erstellen, das alle diese Daten auf einmal aufnehmen und transportieren kann. Eine getrennte Abfrage der einzelnen Daten über verschiedene Operationen ist im Hinblick auf eine spätere Implementierung zu ineffizient.

Da später Web-Services erstellt werden sollen, ist es auch wichtig keine Datentypen zu verwenden, die nicht in SOAP abgebildet werden können. Die verwendeten Datentypen sollten also entweder einfache Datentypen (wie Integer, String, usw.), Mengentypen oder zusammengesetzte Typen wie die eben vorgestellten Value Objects sein. Letztere sollten wiederum nur aus einfachen Datentypen bestehen.

Die Menge der entworfenen Operationen bildet dann jeweils die Schnittstelle, wie sie als Beispiel anhand der Komponente zur Veranstaltungsverwaltung in Abbildung 6.7 dargestellt ist.

Der syntaktische Entwurf der Schnittstellen reicht für Komponenten jedoch nicht aus. Komponenten müssen wohldefinierte, das heißt auch semantisch spezifizierte Schnittstellen besitzen. Als Spezifikationssprache kann hier wiederum die bereits erwähnte Object Constraint Language dienen um Vor- und Nachbedingungen der einzelnen Operationen formal zu dokumentieren.

Um in der OCL Vor- und Nachbedingungen sinnvoll ausdrücken zu können ist es zuvor jedoch noch nötig für jede Schnittstelle ein Datenmodell anzugeben. Dieses Datenmodell stellt keine später implementierten Datenstrukturen dar, sondern ist ein Modell für eine spätere Realisierung. Diese abstrakten Form wird verwendet um Auswirkungen der Schnittstellenoperationen auf Daten formal darstellen zu können. Somit ist die Repräsentation der Daten prinzipiell beliebig. Um ein Datenmodell ohne großen Aufwand erstellen zu können ist es sinnvoll es direkt aus dem in Abschnitt 6.2 transformierten Business Object Model zu entwickeln. Da dieses die Grundlage der Architektur ist und der Prozess der Schnittstellenfindung auf dieser aufbaut, passen beide gut zusammen. Aus dem transformierten Business Object Model lässt sich das benötigte Datenmodell entweder durch einfaches Kopieren oder durch kleine Anpassungen direkt erzeugen.

Anhand dieses Datenmodells, das ebenfalls in Abbildung 6.7 durch Aggregatsbeziehungen zur Schnittstelle dargestellt ist, kann nun die Schnittstelle formal spezifiziert werden, wie in folgendem Ausschnitt zu sehen ist:

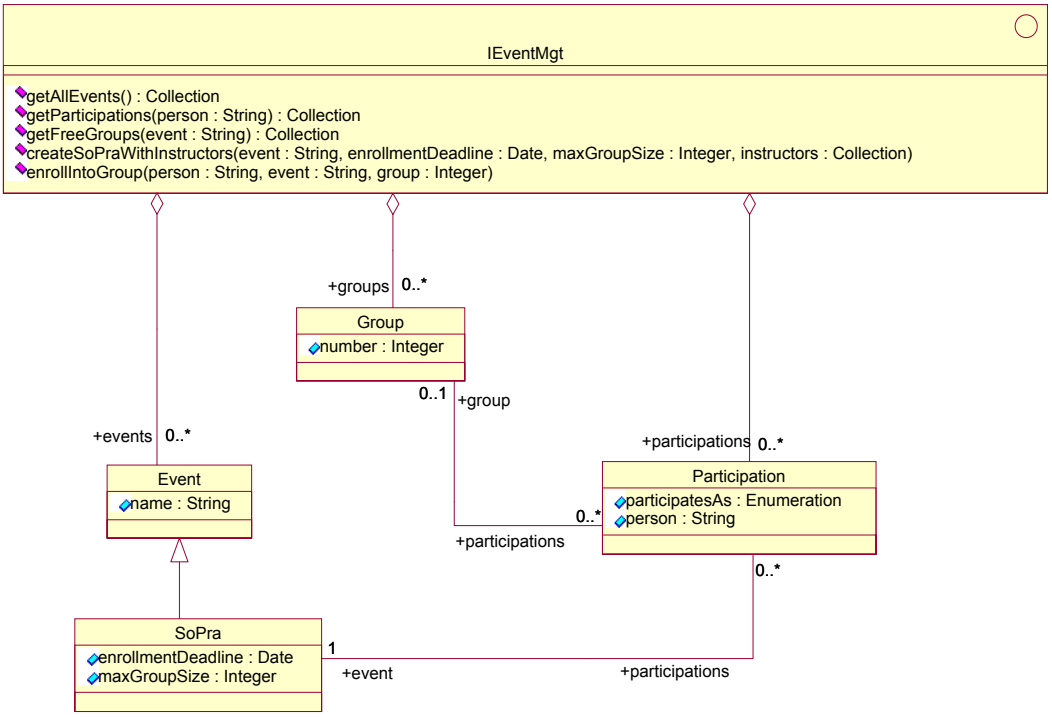


Abbildung 6.7: Die Schnittstelle *IEventMgt* und ihr Datenmodell

---

```
context IEventMgt::getAllEvents() : Collection
-- liefert alle erfassten Veranstaltungen zurück
post: result = events->collect(e.name)


context IEventMgt::getParticipatedEvents(person : String)
            : Collection
-- liefert alle Veranstaltungen zurück, die als
-- Teilnehmer besucht werden
post: let selectedParticipations = participations->select(
        e | e.participatesAs = PARTICIPANT and
            e.person = person)
      in result = selectedParticipations->collect(event)


context IEventMgt::getFreeGroups(event : String)
            : Collection
-- liefert die Nummern aller Gruppen einer Veranstaltung
-- zurück, die noch mindestens einen freien Platz besitzen
```

```
post: let selectedGroups = groups->select( g |
            g.event = event and
            g.participations->size() < g.event.maxGroupSize)
      in result = selectedGroups->collect(number)


context IEventMgt::createSoPraWithInstructors(
    event              : String,
    enrollmentDeadline : Date,
    maxGroupSize       : Integer,
    instructors        : Collection)
-- es darf noch keine Veranstaltung dieses Namens geben
pre: not events->exists( e | e.name = event)
-- Die neue Veranstaltung wurde mit den gegebenen
-- Daten erstellt
post: events->one(ev | ev.oclIsNew()) and
      events->exists(ev : SoPra |
        ev.name = event and
        ev.enrollmentDeadline = enrollmentDeadline and
        ev.maxGroupSize = maxGroupSize) and
-- Alle angegebenen Betreuer betreuen die neue Veranstaltung
      instructors->forAll( instructor |
        participations->exists( participation |
          participation.person = instructor and
          participation.event = event and
          participation.participatesAs = INSTRUCTOR))
```

In einigen Fällen sollten Schnittstellen in zwei oder mehr einzelne Schnittstellen aufgeteilt werden um schmalere Schnittstellen und damit geringere Abhängigkeiten zu schaffen. Diese Aufteilung ist unter anderem sinnvoll, wenn sich herausstellt, dass andere Komponenten nur einen Teil der jeweiligen Schnittstelle verwenden oder wenn mehrere unabhängige Aspekte in der Schnittstelle behandelt werden.

Beispielsweise sind die Anforderung einer Bestätigung über die Komponente *ConfirmationManagement* und die Beantwortung der Bestätigung zwei verschiedene Aspekte, die auch von verschiedenen Komponenten genutzt werden. Die Anforderung wird wie im Interaktionsdiagramm (Abbildung 6.6) gezeigt von der Veranstaltungsverwaltung (Komponente *EventManagement*) verwendet. Die Beantwortung erfolgt jedoch über ein Web-Interface. Es bietet sich also an, die Schnittstelle *IConfirmationManager* in zwei Schnittstellen aufzuteilen: *IConfirmationRequest* zur Anforderung der Bestätigung und *IConfirmationReply* zur Verarbeitung eintreffender Antworten.

Durch die sinnvolle Aufteilung von Schnittstellen kann die Flexibilität und Austauschbarkeit der Komponenten erhöht werden. Die beschriebene Aufteilung der Schnittstelle *IConfirmation* ermöglicht beispielsweise die Anpassung der Antwortmöglichkeiten an

neue Kommunikationswege und eventuell daraus resultierende Änderungen an der Schnittstelle *IConfirmationReply*. Die Schnittstelle *IConfirmationRequest* und damit die Implementierung davon abhängiger Komponenten sind davon jedoch nicht betroffen.

6.5 Entwurf der Komponenten

Ab diesem Punkt ist es möglich die Entwicklung der Komponenten zu parallelisieren. Da nun die Schnittstellen spezifiziert und festgeschrieben sind, ist eine unabhängige Entwicklung möglich.

6.5.1 Plattformunabhängiger Entwurf

Mit den spezifizierten Schnittstellen, der von J2EE vorgegebenen Schichtenarchitektur sowie den durch Interaktionsdiagramme verfeinerten Use Cases und den daraus abgeleiteten Klassen und Verantwortlichkeiten werden nun die Komponenten entworfen. Dieser Schritt ist wie die vorangegangenen Schritte auch noch weit gehend technologie- und plattformunabhängig. Es werden noch immer plattformunabhängige Modelle (PIM) im Sinn der modellgetriebenen Architektur erstellt. Die Datentypen sind abstrakt: *String*, *Integer*, *Collection* etc. sind auf viele Plattformen übertragbar und auch die Schichtenarchitektur ist auf andere Komponententechnologien wie beispielsweise das CORBA Component Model abbildbar.

Um die Plattformunabhängigkeit weiter zu garantieren sollten Implementierungsdetails der Komponenten vorerst nicht entworfen, sondern durch UML-Stereotypen markiert werden. Die Version 2 der UML wird hierfür speziell abgestimmte Profile anbieten, beispielsweise Stereotypen für Erstellungsmethoden (*«create»*) oder für Attribute, die den Primärschlüssel bilden (*«PrimaryKey»*).

In den Abschnitten 3.2.2 und 3.2.3 werden zwei Werkzeuge vorgestellt, die bereits mit solchen Stereotypen arbeiten. Auf UML 2.0 ausgerichtete zukünftige Werkzeuge werden auch die halbautomatische Generierung des plattformabhängigen Modells aus solchen Modellen unterstützen.

Komponenten können aus drei verschiedenen Arten von Klassen bestehen. Zum einen gibt es Klassen, die Daten speichern und verwalten. Diese entstehen aus den Geschäftsklassen, indem deren Attribute kopiert werden und durch Operationen zur Erstellung, Auffindung und Löschung ergänzt werden. Weiterhin werden Verantwortlichkeiten, die bei der Verfeinerung der Use Cases gefunden wurden als Operationen entworfen. Schließlich werden noch Operationen zum Zugriff auf die Attribute (*Getter* und *Setter*) benötigt, sofern das verwendete Werkzeug diese nicht automatisch generiert. Im unten folgenden Diagramm werden diese Datenklassen mit dem Stereotyp *«data»* markiert.

Die zweite Art von Klassen implementiert die Logik der Komponenten und realisieren damit die im vorigen Abschnitt identifizierten Schnittstellen und deren Operationen. Diese Klassen werden mit *«logic»* markiert. Es kann hierbei pro Komponente eine solche Klasse geben, die alle Schnittstellen der Komponente implementiert oder mehrere Klassen, die jeweils einen Teil der Schnittstellen implementieren. Die Aufteilung ist beliebig

und sollte von Fall zu Fall anhand der Abhängigkeiten und weiterer Folgen untersucht und entschieden werden. Generell muss in jeder Komponente jede für sie spezifizierte Schnittstelle genau einmal implementiert werden. Es kann notwendig sein für die Kommunikation zwischen mehreren Klassen weitere Operationen zu entwerfen.

Die Klassen der dritten Art sind Hilfs- und Ressourcenklassen (markiert mit *«helper»*). Diese enthalten gemeinsam genutzte Methoden und erlauben speziell den Zugriff auf Konfigurationsparameter der Komponente. Dafür sind jeweils Attribute und Zugriffsoperationen in dieser Klasse vorzusehen. Parameter realisieren alle zur Installation veränderbaren Eigenschaften der Komponente, z.B. Standardwerte, Schablonen oder Ressourcen.

Je nach Komponente gibt es auch nur Klassen zur Implementierung der Logik. Insbesondere technische Komponenten, wie beispielsweise die Komponente zum Versand von Emails, benötigen keine Datenklassen. Außerdem bietet es sich an die Funktionalität von Ressourcenklassen bei nur wenigen Parametern direkt in die Logikklassen zu übernehmen.

In Abbildung 6.8 wird die entworfene Komponente *EventManagement* dargestellt. Das Diagramm stellt einen Auszug aus dem Entwurf dar, der zusätzlich weitere Diagramme und die genaue Dokumentation enthält.

Weiterhin werden im hier abgebildeten Diagramm der Übersichtlichkeit halber die Signaturen der Operationen, sowie die Details der Klasse *ConfirmationHandler* weggelassen. Diese Klasse dient zur Abwicklung der Bestätigungen seitens der Veranstaltungsverwaltung. Insbesondere werden die Prozesse realisiert, die nach dem Eintreffen der Antwort auf der Anforderung der Bestätigung ausgeführt werden müssen. Die in den weiteren Klassen aufgeführten Operationen und Attribute sind ebenfalls nicht vollständig aufgeführt, sondern stehen nur als Vertreter weiterer Elemente.

Wie oben beschrieben sind in den Datenklassen einige Operationen zur Erstellung und Auffindung identifiziert worden. Die Klasse *EventManager* implementiert die Prozesslogik und realisiert die Schnittstelle *IEventMgt*. Darüber hinaus wurde die Funktionalität des Emailversands im Falle der erfolgreichen Anmeldung in eine eigene Methode *sendSuccessMessage()* ausgelagert.

Die Hilfsklasse *EventMgrHelper* enthält hier exemplarisch den Komponentenparameter *confirmationTimeout*, der bestimmt, wie lange auf eine angeforderte Bestätigung zur Eintragung in eine bestehende Gruppe gewartet werden soll. Weiterhin ermöglicht die Methode *getMessaging()* die Verbindung zur Email-Komponente, über die dann Nachrichten versendet werden können.

Auch die neuen Operationen werden wieder mit OCL oder sogar einer in UML Version 1.5 oder 2.0 vorgesehenen Aktionssemantik spezifiziert. Letztere hat den Vorteil, dass Werkzeuge, die mittelfristig auf den Markt kommen werden, die Aktionssemantik auf die Zielsprache abbilden und so umfassend Quellcode generieren können.

6.5.2 Plattformabhängiger Entwurf

Sind die Komponenten nach den bisher beschriebenen Methoden fertig entworfen, so kann die Transformation des Modells auf die Zielplattform statt finden. Dieser Schritt wird zukünftig durch Werkzeuge unterstützt und damit vereinfacht werden. Im Folgenden wird

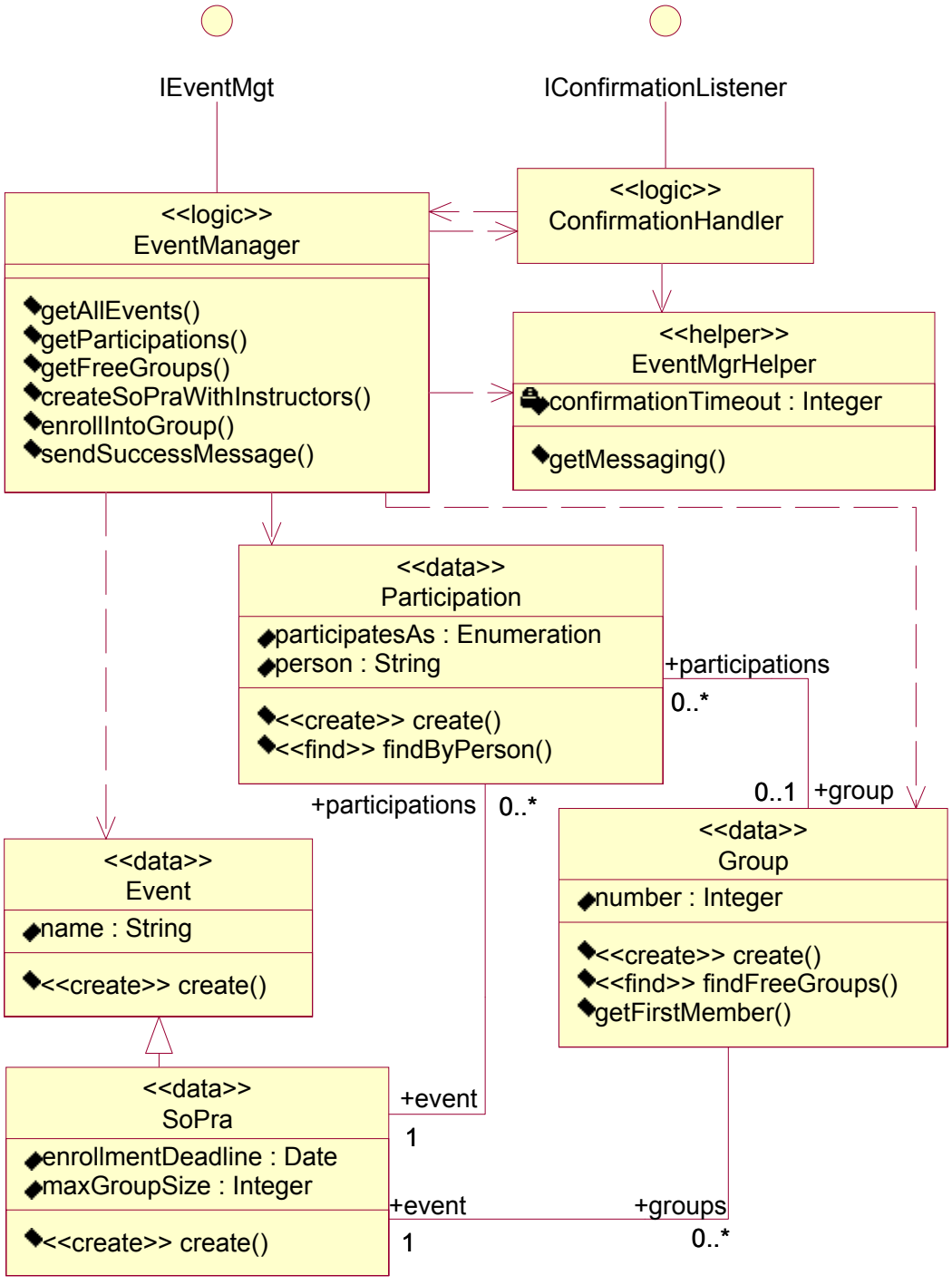


Abbildung 6.8: Entwurf der Komponente *EventManagement*

beschrieben, wie die bisher entworfenen Komponenten manuell auf die J2EE-Plattform abgebildet werden.

Hierbei müssen zunächst die Rahmenbedingungen der Plattform beachtet werden: Die Implementierungssprache für J2EE ist Java. Daher müssen alle Typen auf ihre Java-Pendants abgebildet werden. Die meisten bisher angesprochenen Typen wie String oder Integer sind leicht abzubilden. In der Klasse *Participation* wird jedoch ein Aufzählungstyp verwendet. Da Java keine Aufzählungstypen kennt, müssen hier Integerkonstanten verwendet werden.

Dann müssen die Klassen und Komponentenschnittstellen auf die in Abschnitt 3.3 beschriebenen Enterprise Java Beans abgebildet werden: Die Datenklassen werden auf Entity Beans abgebildet. Entity Beans benötigen wie alle Enterprise Beans eine Home- und eine Remote-Schnittstelle, die ebenfalls erstellt werden müssen. Als Operationen besitzt die Home-Schnittstelle alle im vorangegangenen Schritt mit *«create»* und *«find»* markierten Methoden. Diese Methoden müssen gegebenenfalls umbenannt werden um den Konventionen von J2EE zu genügen. Alle weiteren Methoden wie Zugriffsmethoden für die Attribute werden in die Remote-Schnittstelle übertragen.

Weiterhin wird für eine Entity Bean ein Primärschlüssel, benötigt. Wurden Schlüsselattribute noch nicht identifiziert, so muss dies nun geschehen. Gibt es nur ein Schlüsselattribut, so muss nichts weiter geändert werden. Bei mehreren Schlüsselattributen, die zusammen den Primärschlüssel bilden sollen, muss eine Primärschlüsselklasse aus den Attributen erstellt werden.

Die Zielplattform EJB 1.2 unterstützt noch keine automatische Verwaltung von Assoziationen zwischen Datenklassen, *Container Managed Relationships* genannt. Diese sind erst ab Version 2.0 in der EJB-Spezifikation enthalten. Deshalb müssen die Assoziationen zwischen den Datenklassen aufgelöst werden und mit Hilfe von Fremdschlüsseln und zusätzlichen Zugriffsmethoden nachgebildet werden.

Die Generalisierungsbeziehung zwischen *SoPra* und *Event* wird in J2EE unterstützt wird daher übernommen.

Die mit *«logic»* markierten Klassen sind einfach transformierbar: Sie werden zu zustandslosen *Session Beans*, deren Home-Schnittstellen jeweils nur die Methode *create* enthalten um eine Instanz zu erzeugen.

Die jeweiligen Remote-Schnittstellen werden erzeugt, indem zunächst die Komponentenschnittstellen in Remote-Schnittstellen verwandelt werden. Dies geschieht durch Ererbung der Schnittstelle *javax.ejb.EJBObject*. Für jede erstellte Remote-Schnittstelle wird dann eine entsprechende Home-Schnittstelle mit der passenden *create*-Methode erzeugt.

Alle Remote-Schnittstellen, die die Klasse realisieren soll, werden dann durch Vererbung in die eigentliche Remote-Schnittstelle der Logikklasse übernommen. Schließlich werden die weiteren Operationen, die komponentenintern verwendet werden, noch zur Remote-Schnittstelle hinzugefügt. Abbildung 6.9 zeigt dies wieder beispielhaft an der Klasse *EventManager*.

In der Abbildung werden die Stereotypen *«HomeInterface»* und *«RemoteInterface»* abkürzend für die Art der Schnittstellen verwendet. Die Schnittstelle *IEventMgt* ist die plattformabhängige Abbildung der Komponentenschnittstelle und enthält die Operationen, die die Komponente nach außen hin anbietet. Die komponenteninterne Operation *sendSuccess-*

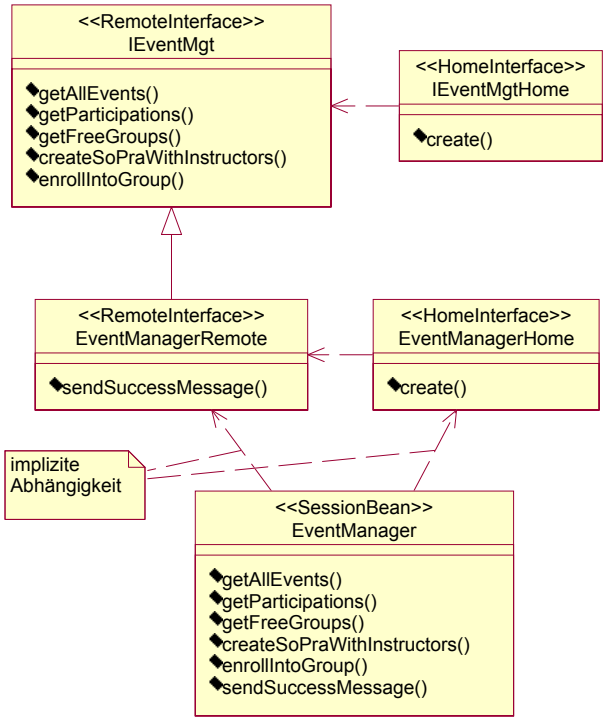


Abbildung 6.9: J2EE-Schnittstellen der Klasse *EventManager*

Message ist in der eigentlichen Remote-Schnittstelle der Klasse *EventManager* deklariert. Zu beiden Remote-Schnittstellen existiert eine Home-Schnittstelle mit der entsprechenden *create*-Methode.

Die Klasse *EventManager* ist darüber hinaus nicht direkt syntaktisch von den Remote-Schnittstellen abhängig. Dies ist eine Folge der J2EE-Konventionen und ermöglicht es dem Container die beschriebenen Wrapper zu erstellen. Die korrekte Übereinstimmung der Operations-Signaturen wird erst bei der Inbetriebnahme der Komponente überprüft.

Hilfsklassen werden ebenfalls als zustandslose *Session Beans* realisiert. Hierbei wird wie oben eine einfache Home-Schnittstelle erstellt. Die Remote-Schnittstelle setzt sich aus allen öffentlichen Operationen zusammen.

Für jedes Attribut, das einen Komponentenparameter repräsentiert, muss zusätzlich eine String-Konstante erzeugt werden, die die Stelle bezeichnet, von der der Parameter im Verzeichnisdienst JNDI eingelesen werden kann. Der Einlesevorgang wird später in der Methode *create* implementiert.

In der J2EE-Plattform bietet es sich weiterhin an Operationen für die Auffindung und Instantiierung von Enterprise Beans zentral in die Hilfsklasse zu übernehmen. Von dort können die Operationen dann überall innerhalb der Komponente verwendet werden. Hierzu muss dann ebenfalls ein Attribut zur Speicherung der Referenz auf die jeweilige Home-Schnittstelle, eine String-Konstante für den JNDI-Namen und eine Zugriffsmethode erstellt werden.

Im Entwurf der Komponente zur Bestätigungsverwaltung gibt es noch eine Problemstelle, deren Abbildung nicht einfach ist: die Komponente soll einen Rückruf unterstützen. Eine Komponente, die eine Bestätigung anfordert, soll über die Schnittstelle *IConfirmationListener* zurückgerufen werden, wenn eine Antwort eingetroffen ist. Im plattformunabhängigen Entwurf wurde in der Schnittstelle der Typ *IConfirmationListener* verwendet. In J2EE ist die Realisierung eines Rückrufs durch die Übergabe einer Referenz aber nicht möglich, da in J2EE wegen der Verteilung schwache Referenzen verwendet werden und die Lebenszeit der Bean-Instanzen vom Container gesteuert wird. Diese Lebenszeit, die auf wenige Minuten oder Stunden beschränkt ist, steht mehreren Tagen Wartezeit für die Antwort der Bestätigung gegenüber. Trifft die Bestätigung dann ein, ist die vorher übergebene Referenz ungültig.

Die Lösung besteht darin statt der Referenz auf eine *IConfirmationListener*-Instanz eine Referenz auf die zugehörige Home-Schnittstelle *IConfirmationListenerHome* zu übergeben. Zum einen sind Home-Schnittstellen dauerhaft und Referenzen darauf verfallen nicht, zum anderen sind sie serialisierbar, können also als so genannter *BLOB* (Binärdaten) in die Datenbank gespeichert werden. Beim Rückruf kann aus der Home-Schnittstelle wieder die eigentliche Instanz zur Behandlung des Rückrufs erstellt werden. Dieser Vorgang wird in Abbildung 6.10 dargestellt.

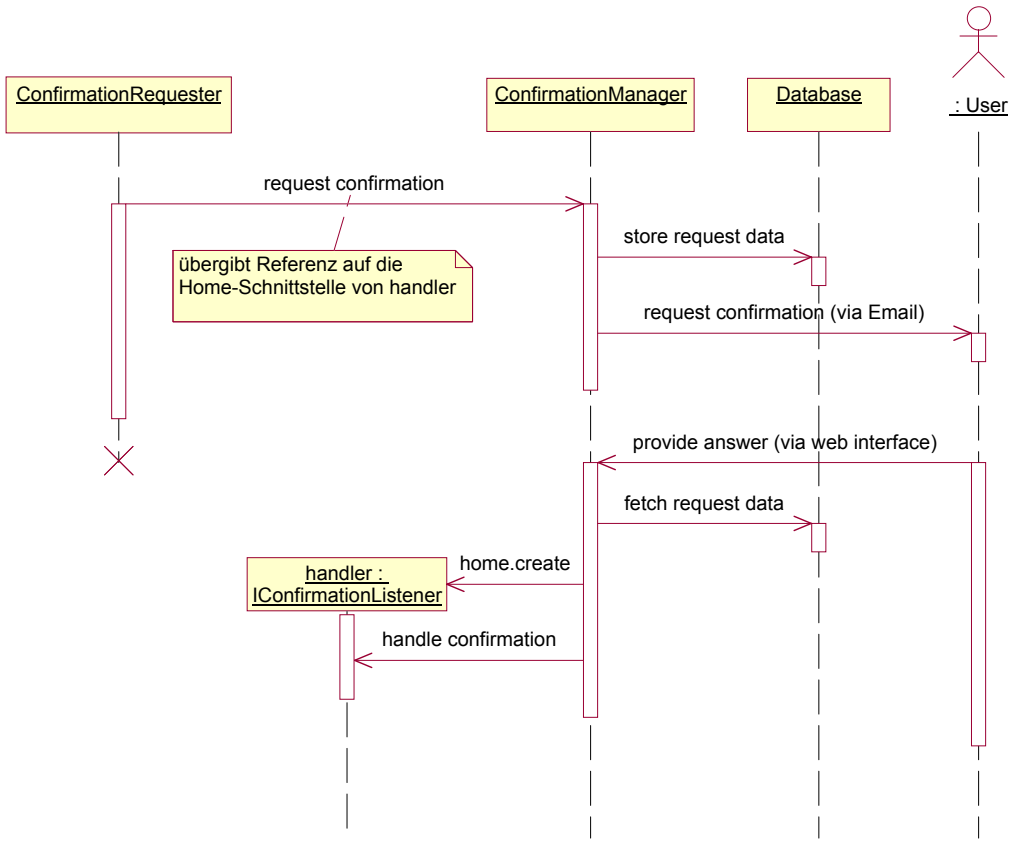


Abbildung 6.10: Rückruf in J2EE

Der Entwurf ist nun soweit verfeinert, dass er auf der gewählten Plattform realisiert werden kann. Bevor die Implementierung beschrieben wird, wird noch die Dokumentation der Komponenten beschrieben.

6.6 Dokumentation von Komponenten

Zu einem vollständigen Softwaresystem gehört ein Handbuch für die Benutzer des Systems. Dies sind in der Regel Anwender, die mit dem System arbeiten sollen. Bei Komponenten verhält sich dies anders. Hier sind die Benutzer Entwickler, die aus bestehenden Komponenten ein Anwendungssystem zusammenbauen. Um diesen Entwicklern eine brauchbare Dokumentation an die Hand zu geben müssen andere Gesichtspunkte als bei der konventionellen Handbucherstellung berücksichtigt werden.

Hierbei muss zwischen offenen und geschlossenen Systemen unterschieden werden. Offene Systeme sind Systeme, bei denen die Implementierung und der Entwurf der Komponenten nach außen hin sichtbar sein darf. Diese Systeme sind im akademischen Umfeld und bei Open Source Software anzutreffen. Die Dokumentation beschränkt sich in diesen Fällen auf die Ergänzung der schon vorhandenen Entwurfsdokumente mit Übersichten und Informationen für die externe Sicht auf die Komponente.

Geschlossene Systeme sind schwieriger zu dokumentieren. Firmen, die Komponenten anbieten möchten ohne die Implementierung offen zu legen, müssen bei der Dokumentation für die Entwickler sehr sorgfältig vorgehen. Auf der einen Seite muss die Dokumentation genügend Informationen bereit stellen um die korrekte Zusammenarbeit der Komponente mit der Umgebung und mit anderen Komponenten zu gewährleisten. Das bedeutet, dass alle Schnittstellen und Seiteneffekte sehr detailliert und genau und sowohl syntaktisch als auch semantisch dokumentiert sein müssen. Andererseits möchten die Firmen aber nicht zu viel über die Realisierung der Komponenten aussagen. Es besteht für sie die Gefahr mit diesen Informationen die angebotene Komponente überflüssig zu machen.

Als Schwerpunkt der folgenden Erläuterungen wird nur die Ergänzung der Entwurfsdokumentation beschrieben, da es sich bei dem hier beschriebenen Komponentenbaukasten um ein offenes System handelt. Mehr über die Dokumentation geschlossener Systeme ist z.B. in [HC01] zu lesen.

Wie bereits gesagt, enthält der Entwurf bereits große Teile der Komponentendokumentation: angefangen vom Business Object Model über die Schnittstellenspezifikation bis hin zum Komponentenentwurf. Für die Verwendung der Komponente zum Einbau in ein System fehlt hier noch ein Bild des Ganzen: die Sicht von außen auf die Komponente.

Dieser externen Sicht muss der Entwickler entnehmen können, welche Schnittstellen die Komponente implementiert, welche Anforderungen sie in Form von Schnittstellenabhängigkeiten an die Umgebung hat und wie sie an die Umgebung anpassbar und konfigurierbar ist. Ein Teil dieser Eigenschaften ist durch die verwendete Plattform festgelegt. Eine Komponente in J2EE ist z.B. von dem Verzeichnisdienst JNDI und einer JDBC-Datenbankanbindung abhängig. Diese Eigenschaften müssen nicht explizit erwähnt werden.

Das Diagramm in Abbildung 6.11 zeigt eine Außensicht der Komponente *EventMana-*

ger. Die Komponente wird hier wieder als Klasse mit dem Stereotyp «*comp spec*» dargestellt. Darüber hinaus werden realisierte Schnittstellen mit durchgezogenen Linien und benötigte Schnittstellen mit gestrichelten Abhängigkeitspfeilen dargestellt. Die konfigurierbaren Parameter der Komponente werden als Attribute dargestellt.

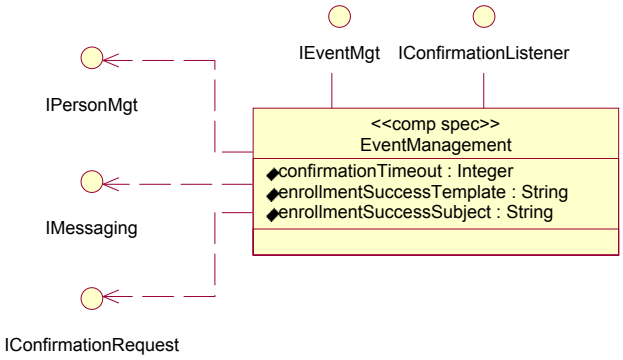


Abbildung 6.11: Außensicht der Komponente *EventManagement*

Die Dokumentation zu diesem Diagramm muss auf die Schnittstellenspezifikationen der verwendeten Schnittstellen verweisen, sowie die einzelnen Konfigurationsparameter dokumentieren.

Eine Ergänzung zu dieser Dokumentation sind Sequenzdiagramme, die die Schnittstellenabhängigkeiten spezifischer darstellen. Bei offenen Systemen sind diese Verfeinerung bereits in den Sequenzdiagrammen des Entwurfs enthalten, dann kann darauf verwiesen werden. In geschlossenen Systemen ist die Angabe diese Verfeinerungen jedoch sehr wichtig, da aus ihnen ableitbar ist, durch welche Operation welche weiteren Operationen aufgerufen werden. Hieraus können beispielsweise Antwortzeiten genauer bestimmt werden. Abbildung 6.12 zeigt am Beispiel der Operation *enrollIntoGroup* ein solches Sequenzdiagramm.

Man sieht darin, dass der Aufruf der Operation im Fall, dass die übergebene Gruppe existiert, die Operation *requestConfirmation* aufruft. Falls eine neue Gruppe erstellt werden soll, wird die Operation *sendMessage* zum Versenden der Erfolgsnachricht aufgerufen. Dabei werden Vorgänge innerhalb der Komponente, z.B. Aufrufe der Entity Beans *Group* oder *Event* bewusst nicht wiedergegeben.

Um dem Entwickler einen Überblick über die Möglichkeiten der Komponente zu vermitteln werden Use Case Diagramme wie das in Abbildung 6.13 benutzt. Der dort dargestellte Aktor ist in diesem Fall keine natürliche Person, sondern ein externes System, das die Komponente verwendet.

Im folgenden Auszug der Use Case Dokumentation wird der Ablauf beschrieben, der zu der Anmeldung einer Person an ein Softwarepraktikum führt. Durch diese Dokumentation werden Abfolgen von Operationsaufrufen beschrieben, die zum Erreichen eines Ziels nötig sind.

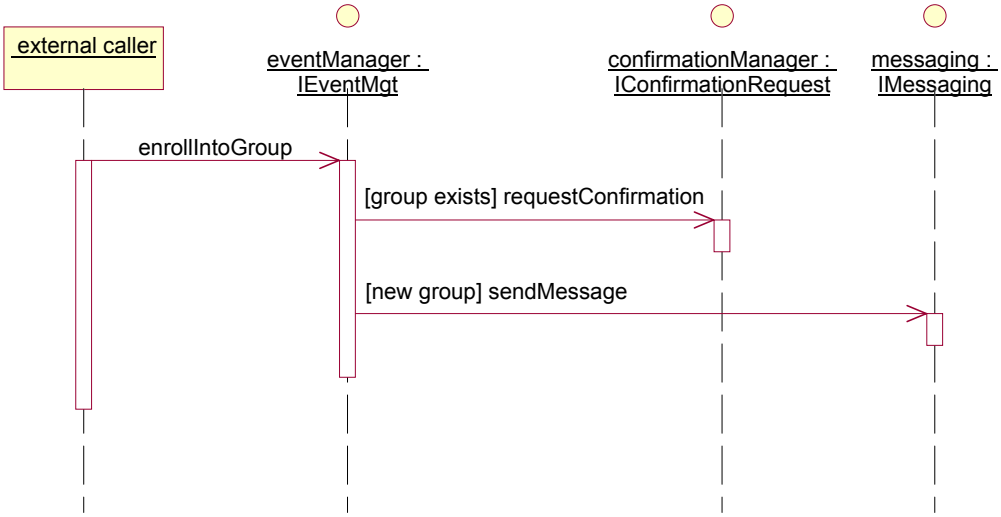


Abbildung 6.12: Verfeinerung der Schnittstellenoperation *enrollIntoGroup*

Abbildung 6.13: Use Case der Komponente *EventManagement*

Use Case:
Zu SoPra anmelden

Kurzbeschreibung:
Ein System möchte eine Person an ein Softwarepraktikum anmelden.

Initiator:
ext. System

Ablauf:

1. Das ext. System ruft die Operation *getSoPras* auf.
2. Die Komponente liefert die Namen der Softwarepraktika in einer Liste zurück.
3. Das ext. System ruft die Operation *getFreeGroups* mit einem der erhaltenen Veranstaltungsnamen als Parameter auf.
4. Die Komponente liefert die Nummern der Gruppen, in denen noch Plätze sind, als Liste zurück.
5. Das ext. System ruft die Operation *enrollIntoGroup* mit der Emailadresse der anzumeldenden Person, dem Namen der Veranstaltung und der Nummer der gewünschten Gruppe auf.
6. Die Komponente fordert eine Bestätigung vom ersten Mitglied der Gruppe an. Sobald die Bestätigung vorliegt, wird die anzumeldende Person per Email informiert und an die Veranstaltung angemeldet.

7 Implementierung und Test

Aus dem plattformabhängigen Entwurf kann nun die Implementierung erzeugt werden. Es ist heute mit geeigneten Werkzeugen schon möglich große Teile der Umgebung und der Skelette zu erstellen (s. Abschnitt 3.2.2). Manche Werkzeuge können aus den OCL-Spezifikationen auch Code zur Überprüfung der Vor- und Nachbedingungen generieren. In Zukunft soll es laut der Vision der modellgetriebenen Architektur möglich sein aus UML-Modellen mit Aktionssemantik auch mehr als nur strukturellen Code zu erzeugen.

Die Abbildung der Klassen und Schnittstellen des plattformabhängigen Entwurfs auf Quellcode ist sehr direkt: aus den UML-Strukturen müssen die entsprechenden Konstrukte in Java generiert werden. Schließlich sind die einzelnen Methoden entsprechend der jeweiligen Spezifikationen zu implementieren.

Ist dies geschehen, müssen die Komponenten aus ihren Klassen zusammengestellt und eingepackt werden. Hierfür wird pro Komponente ein EJB-Archiv erstellt, das alle Klassen der Komponente enthält. Zusätzlich muss jeweils der Deployment-Deskriptor erstellt werden. In diesem muss jede enthaltene Enterprise Bean beschrieben werden. Oft wird auch das von einem Werkzeug erledigt. Eine Besonderheit bei der Erstellung von größeren Komponenten aus einzelnen Enterprise Beans ist die korrekte Konfiguration der Zusammenarbeit bereits in dieser Phase.

Die EJB-Spezifikation sieht vor, dass die Konfiguration der Verbindungen zwischen einzelnen EJBs erst beim Zusammenbau der Anwendung erfolgt. Betrachtet man die EJB-Archive jedoch nicht als Sammlung von Einzelkomponenten, sondern selbst als Komponente, so wird dem Anwendungsentwickler bereits ein fertig vorkonfiguriertes Packet von Beans zur Verfügung gestellt. Der Anwendungsentwickler muss nur noch Feinarbeit leisten um die Komponente einzupasen.

Schließlich müssen im Deployment-Deskriptor noch Einträge für die konfigurierbaren Komponentenparameter erzeugt werden sowie weitere Einstellungen, wie die Abbildung der Entity-Beans auf Datenbanktabellen, getätigt werden. Ist dies geschehen, kann das Archiv erstellt und getestet werden.

7.1 Implementierung der Web-Services

Die Implementierung der Web-Services erfolgt automatisch durch das Werkzeug *IBM WebSphere Studio Application Developer*. Diese Generierung ist möglich, weil Web-Services und zustandslose Session-Beans sehr viel gemeinsam haben. Unter anderem sind sie beide zustandslos und auf ferne Aufrufe einzelner Operationen abgestimmt. Sie unterscheiden sich lediglich im Kommunikationsmittel: Die Basis für EJBs ist CORBA, die der Web-Services SOAP.

Es ist daher kein Problem eine Klasse zu erzeugen, die einen Web-Service implementiert und dann den eingehenden Aufruf über EJB-Mechanismen an eine Session-Bean delegiert. Dies geschieht automatisch mit dem oben erwähnten Werkzeug.

Anhand des folgenden kleinen Beispiels lässt sich die Generierung verdeutlichen. Gegeben sei der folgende Java-Code, der eine Methode einer zustandslosen Session Bean implementiert:

```
public class HelloBean implements javax.ejb.SessionBean {
    /* J2EE-spezifische Funktionen (hier weggelassen)
     * [...]
     */

    /* Gibt einen Begrüßungs-String zurück */
    public String sayHelloTo(String name) {
        return "Hello, " + name + "!";
    }
}
```

Wird nun ein Web-Service aus dieser Session Bean generiert, so wird automatisch eine WSDL-Beschreibung erzeugt, die außer den Standardangaben den folgenden Text enthält:

```
[...]
    <message name="sayHelloToRequest">
        <part name="name" type="xsd:string"/>
    </message>
    <message name="sayHelloToResponse">
        <part name="result" type="xsd:string"/>
    </message>
[...]
   <binding name="HelloBinding" type="tns:Hello">
        <soap:binding
          style="rpc"
          transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="sayHelloTo">
            <soap:operation soapAction="" style="rpc" />
            <input name="sayHelloToRequest" />
            <output name="sayHelloToResponse" />
        </operation>
   </binding>
[...]
```

Man sieht im oberen Teil die Definitionen der Anfrage- und Antwortnachrichten. Es wird hier jeweils ein String mitgegeben. Der untere Teil fügt dann diese beiden Nachrichten zum eigentlichen Aufruf zusammen. In einer weiteren, WebSphere-spezifischen Beschreibungsdatei wird außerdem eine Behandlungsklasse angegeben, die von IBM mitgeliefert wird. Diese erzeugt bei eintreffender SOAP-Anfrage eine Instanz der Session Bean, ruft die jeweilige Methode auf und gibt das Ergebnis wiederum als SOAP zurück.

Hierbei treten jedoch noch zwei Probleme auf: Zum einen müssen nicht-atomare Datentypen (wie die im Entwurf beschriebenen *Value Objects*) auf XML abgebildet werden. Dies beherrscht das Werkzeug zwar, jedoch ist das Ergebnis nicht vollständig kompatibel zum Standard. Zum anderen fehlt in der Spezifikation der Web-Services wie bereits in Abschnitt 3.4 beschrieben ein Sicherheits- und Benutzerauthentifizierungsmechanismus.

Da diese Probleme mit künftigen Versionen der Spezifikation und des Werkzeugs voraussichtlich beseitigt werden, ist es nicht Gegenstand dieser Arbeit für diese temporären Probleme Lösungen oder Umwege zu finden. Die Inkompatibilität sowie die Unsicherheit werden bis zur Verbesserung in Kauf genommen.

7.2 Test

Der Test der Komponenten erfolgt in mehreren Stufen. Da eine Komponente unabhängig von allen anderen entwickelt wird muss sie auch unabhängig testbar sein. Um dies zu erreichen sollte im Entwurf ein zusätzlicher Komponentenparameter berücksichtigt werden. Mit diesem kann die Komponente für einen Modultest konfiguriert werden, so dass sie keine weiteren externen Komponenten aufruft. Zusätzlich können auch weitere ungewollte Effekte wie das Versenden von Emails deaktiviert werden. Der Modultest erfolgt dann als Black-Box-Test gegen die Schnittstellenspezifikation, wobei Ergebnisse, die von anderen Komponenten abhängen, im Testmodus simuliert oder ignoriert werden. Der Black-Box-Test gegen die Schnittstellenspezifikation hat den Vorteil, dass er auch für zukünftige Komponenten, die dieselbe Schnittstelle implementieren, wiederverwendet werden kann.

Weitere Tests können dann unabhängig als Glass-Box-Test ausgeführt werden um die Überdeckung sicher zu stellen.

Sind die Modultests der einzelnen Komponenten abgeschlossen, so können die Komponenten im Verbund noch einmal getestet werden. Hierbei soll vor allem die Zusammenarbeit der Komponenten getestet werden. Auch hier ist wieder ein Black-Box-Test gegen die Schnittstellen vorzuziehen, so dass die Tests wiederverwendet werden können.

Um die Tests zu automatisieren kann ein Werkzeug wie *JUnit* von Erich Gamma[BG98] verwendet werden. Dieses einfache Framework ermöglicht es Klassen zu schreiben, die Testfälle enthalten. Diese Klassen werden mit einem *TestRunner* ausgeführt, der die enthaltenen Testfälle durch Reflexion findet und ausführt. Sind alle Tests abgeschlossen, so wird ein Testbericht und eine Fehlerliste ausgegeben.

Dieses Werkzeug vereinfacht den Testprozess auf zweierlei Weisen: Erstens ermöglicht es die schnelle Ausführung von Tests und ist damit effizient einsetzbar und zweitens werden die Tests immer auf dieselbe Weise ausgeführt. Es kann also nicht zu spontanen Bedienungsfehlern durch die Unachtsamkeit des Testers kommen.

8 Entwicklung der Beispielanwendung

In den bisherigen Kapiteln wurde der Schwerpunkt auf die Entwicklung der Komponenten gelegt. Bis auf die Spezifikation, die auch Grundlage für die Komponenten war, blieb die eigentliche Entwicklung der Beispielanwendung unberücksichtigt.

In diesem Kapitel soll noch kurz der Entwurf der web-basierten Anwendung beschrieben werden. Diese wurde nicht mit Komponenten realisiert. Es ineffektiv Benutzungsschnittstellen als Komponenten zu realisieren, da sie aus ergonomischen Gründen in einer Anwendung einheitlich sein sollten. Der Aufwand, der in eine Komponente investiert werden müsste um ihre Benutzungsschnittstelle so flexibel auf die jeweilige Anwendung anpassbar zu machen, übersteigt den Aufwand der Neuentwicklung der Benutzungsschnittstelle. Dies ist vor allem dann der Fall, wenn Muster oder Frameworks für die Entwicklung verwendet werden. Diese vereinfachen die Entwicklung der Benutzungsschnittstelle stark.

Die in der Beispielanwendung verwendeten Klassen gehen auf Entwurfsmuster von Sun[Mar02a] zurück. Diese bauen auf das Model-View-Controller-Prinzip auf, also die Trennung der Datenaspekte (Model), der Steuerungsaspekte (Control) und der Anzeigeaspekte (View).

Das Datenmodell wird aus *Value Objects* gebildet, die bereits mehrfach erwähnt wurden. Die Steuerung wird durch *Servlets* realisiert. Diese nehmen die Anfragen des Benutzers entgegen, überprüfen die Parameter, verarbeiten sie und kommunizieren mit den Komponenten der J2EE-Anwendung um Daten zu ändern oder abzufragen. Die Verarbeitung wird dann fortgesetzt, indem die nächste anzuzeigende Seite festgestellt wird, die entsprechend benötigten Daten zusammengestellt werden und die Bearbeitung dann an die für die Anzeige zuständige *Java Server Page* weitergegeben wird. Eine genauere Beschreibung der erwähnten Technologien ist in Abschnitt 3.3 zu finden.

Die Steuerung lässt sich durch das Entwurfsmuster *Command*[G$^+$95] noch weiter vereinfachen. Hierfür wird jede Anfrage mit einem Kommandoparameter versehen, der den Namen eines Kommandos enthält. Eine *Factory* (ebenfalls ein Muster aus [G$^+$95]) erzeugt dann aus diesem Kommando die Instanz der Kommandoklasse des selben Namens. Diese Klasse führt dann die weitere Auswertung und Bearbeitung durch. Diese Struktur hat den Vorteil alle Anfragen über eine zentrale Klasse abwickeln zu können. Gemeinsame Aspekte wie Sitzungsverwaltung und Fehlerbehandlung können an einer zentralen Stelle erfolgen. Gleichzeitig ist es möglich, ein neues Kommando nur durch Erstellung der Kommandoklasse und der entsprechenden Anfrage auf der Webseite hinzuzufügen. Es muss kein bestehender Code geändert oder neu übersetzt werden.

Eine weitere Maßnahme zur Erhöhung der Flexibilität ist die Verwendung von *Business Delegates*[Mar02a]. Diese kapseln die Aufrufe der Komponenten. Hierdurch wird die Benutzungsschnittstelle unabhängig vom verwendeten Aufrufprotokoll. Die Komponenten können z.B. entweder per CORBA oder per SOAP aufgerufen werden. Hierzu müssen

jeweils nur die Business Delegates ausgetauscht werden.

Das in Abbildung 8.1 gezeigte Interaktionsdiagramm verdeutlicht den beschriebenen Ablauf.

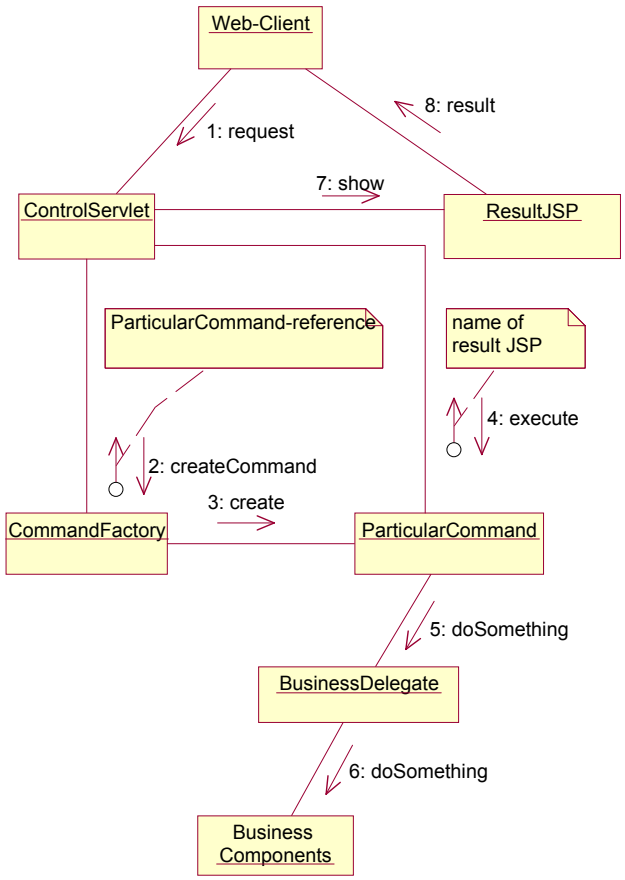
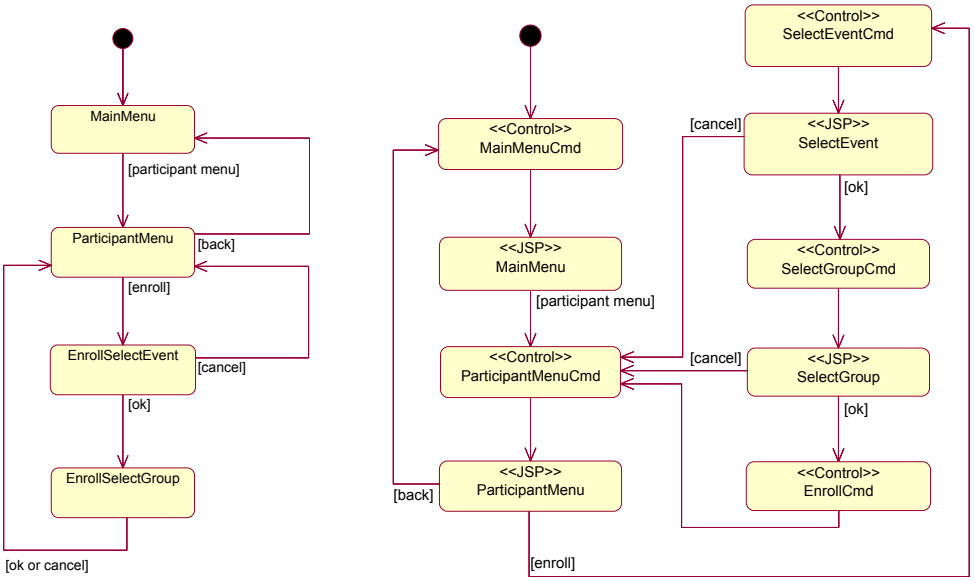


Abbildung 8.1: Der Ablauf der Verarbeitung einer Anfrage an die Webapplikation

Mit diesem beschriebenen Steuerungsschema kann die Spezifikation in den Entwurf umgesetzt werden. Die folgende Methode ist abgeleitet aus der vom Werkzeug *ArcStyler* verwendeten und in [Wei02] beschriebenen Darstellung.

Zunächst wird die Abfolge der anzuzeigenden Seiten aus dem erstellten Oberflächenprototyp in ein Zustandsdiagramm übertragen und gegebenenfalls ergänzt. In Abbildung 8.2(a) ist dies für den Fall der Veranstaltungsanmeldung dargestellt. Die Zustände repräsentieren die Anzeige der jeweiligen HTML- bzw. JSP-Seite. In eckigen Klammern werden Bedingungen für die Transition ausgedrückt, z.B. das Betätigen von Oberflächenelementen.

Nun wird dieses Diagramm verfeinert, indem zu den Anzeigezuständen auch Verarbeitungszustände kommen. Für jede Java Server Page, die Daten anzeigen soll, müssen diese Daten vorher zusammengestellt werden. Dies geschieht in einem vorangestellten Zustand, der wie oben beschrieben als Kommandoklasse realisiert wird. Um die Daten, die vorbereitet werden müssen, zu dokumentieren werden zu den jeweiligen Anzeigezuständen die



(a) Das ursprüngliche Zustandsdiagramm

(b) Das verfeinerte Zustandsdiagramm

Abbildung 8.2: Der Ablauf der Veranstaltungsanmeldung in der Webapplikation

anzuzeigenden dynamischen Daten notiert. Ebenso werden die vom Benutzer einzugebenden Eingabedaten festgehalten. Dies ist im UML-Zustandsdiagramm nicht direkt möglich, so dass hier zusätzlich eine externe Textdokumentation erstellt werden muss.

Die Eingaben des Benutzers in Formularen müssen wiederum in Steuerungszuständen verarbeitet werden.

Gibt es nur einen einzigen linearen Pfad durch mehrere Steuerungszustände, wie bei der Verarbeitung der Veranstaltungsauswahl und dem Vorbereiten der Anzeige der freien Gruppen, so können diese Zustände zu einem zusammengefasst werden. Sie werden später durch nur jeweils eine Kommandoklasse realisiert.

Abbildung 8.2(b) zeigt das verfeinerte Diagramm. Hier werden die Stereotypen *«control»* für Steuerungs- und *«JSP»* für Anzeigezustände verwendet. Wie beschrieben müssen zusätzlich die JSP-Zustände wie in folgendem Ausschnitt dokumentiert werden:

Zustand:
SelectEvent

Auszugebende Daten:

- `events : Collection` – Namen der wählbaren Veranstaltungen als `String`

Anfrageparameter

- `event : String` – Name der ausgewählten Veranstaltung

Zustand:
SelectGroup

Auszugebende Daten:

- `event : String` – Name der ausgewählten Veranstaltung
- `group : Collection` – Nummern der freien Gruppen der Veranstaltung als `Integer`

Anfrageparameter

- `event : String` – Name der ausgewählten Veranstaltung
- `group : String` – Nummer der ausgewählten Gruppe oder "`new`" zum Erstellen und Eintragen in eine neue Gruppe

---

Um die so verfeinerten Abläufe zu implementieren muss zu jedem mit *«control»* markierten Zustand eine Kommandoklasse erstellt werden, die die Parameter der Benutzeranfrage aus dem letzten JSP-Zustand verarbeitet und die Anzeigedaten gemäß dem nächsten JSP-Zustand vorbereitet.

Für jeden mit *«JSP»* markierten Zustand wird eine Java Server Page erstellt, die die bereit gestellten Daten anzeigt und gegebenenfalls ein Formular mit den dokumentierten Eingabemöglichkeiten enthält.

Bei Texteingabefeldern muss das Eingabeformat gemäß der Use Case Dokumentation überprüft werden. Sofern es möglich ist, sollte der Effizienz halber JavaScript innerhalb der JSP verwendet werden. Nur bei weiter gehenden Überprüfungen sollte die Steuerungsklasse das Eingabeformat prüfen.

9 Bewertung und Ausblick

In dieser Arbeit wurde die Entwicklung eines erweiterbaren Komponentenbaukastens für universitäre Geschäftsprozesse beschrieben. Hierzu wurden zunächst Grundbegriffe, insbesondere der Komponentenbegriff, definiert und erläutert. Danach wurde anhand eines kleinen Beispiels zur Verwaltung von Softwarepraktika ein Entwicklungsprojekt für Komponenten in den einzelnen Phasen beschrieben. Die einzelnen Tätigkeiten waren hierbei so aufeinander abgestimmt, dass sie einen kontinuierlichen und methodischen Prozess bilden.

In einer Diplomarbeit von sechs Monaten Dauer ist es praktisch unmöglich wirklich universelle Komponenten zu bauen. Die bereits zitierten *Beggerman's Rules of Three*[Tra95] besagen unter anderem auch, dass Komponenten erst entwickelt werden sollten, wenn das Problem bereits dreimal aufgetreten und gelöst worden ist. Nur in einem solchen Fall ist potenziell genügend Erfahrung vorhanden um sinnvolle Komponenten bauen zu können. Der Autor besaß diese Erfahrung sicherlich nicht in vollem nötigen Umfang.

Des weiteren ist es nach [Hen03] unwirtschaftlich wirklich universelle Komponenten zu entwickeln. Der Aufwand für Spezifikation und Entwurf ist zu hoch und birgt die Gefahr entscheidende Details nicht zu berücksichtigen. Ebenso ist es schwer die Universalität objektiv zu prüfen. Es ist besser möglichst modellgetrieben und erweiterbar vorzugehen und die Erweiterungen dann einzufügen, wenn sie benötigt werden.

Zu außenplanmäßigen Problemen kam es unter anderem beim Einsatz des Werkzeugs *ArcStyler*. Dieses erwies sich in der Code-Generierung als teilweise unausgereift, was sich in unverständlichen Fehlermeldungen bei der Ausführung des generierten Codes äußerte.

Weiterhin stellte sich heraus, dass J2EE bei einigen technischen Problemen, wie der Verwendung einer Komponente mit mehreren Schnittstellen, unflexibel ist. Die Probleme ließen sich durch genaues Lesen der J2EE-Spezifikation oder durch Ausprobieren lösen oder umgehen. Die Recherche in der Literatur und im Internet ergab jedoch keine direkten Lösungshinweise.

Viele der klassischen in [G$^+$95] beschriebenen Entwurfsmuster zur Erhöhung der Erweiterbarkeit sind in J2EE ebenfalls nicht realisierbar. Ein für diese Plattform gültiger Entwurfsmusterkatalog wäre hilfreich.

Dennoch wurde bei der Entwicklung der Komponenten darauf geachtet, dass sie soweit erweiterbar sind, dass sie beim Auftreten von neuen Anforderungen oder bei der Entwicklung eines weiteren Systems wieder einen Schritt in Richtung Universalität entwickelt werden können. Gerade durch die Eigenschaft von Komponenten verschiedene Schnittstellen anbieten zu können und parameterisierbar zu sein wird es möglich eine Komponente stets weiterzuentwickeln und trotzdem kompatibel zu früheren Versionen zu bleiben.

Bei der Entwicklung des Baukastens wurden neue Technologien wie modellgetriebene Architektur und Web-Services verwendet. Diese leiden noch an einigen Kinderkrankheiten und vor allem beim modellgetriebenen Ansatz ist die Werkzeugunterstützung noch

sehr gering. Es gibt jedoch deutliche Anzeichen, dass zukünftige Versionen der Spezifikationen (z.B. UML Version 2) und auch die kommenden Werkzeuge diese Mängel beseitigen und ein weitaus produktiveres Arbeiten ermöglichen werden. Spätestens dann wird die beschriebene Methodik vor allem für Entwicklungsprojekte mit mehreren Beteiligten interessant, da nach dem Schnittstellenentwurf die einzelnen Komponenten unabhängig voneinander entwickelt werden.

Literaturverzeichnis

[Amb02] AMBLER, Scott W. *Deriving Web Services from UML models*. URL: *http://www.ibm.com/developerworks/library/ws-uml1*. März 2002

[B+00] BOX, Don [u. a.]: *Simple Object Access Protocol (SOAP)*. World Wide Web Consortium, Mai 2000. –
URL: *http://www.w3.org/TR/SOAP*

[BG98] BECK, Kent ; GAMMA, Erich. *Test Infected: Programmers Love Writing Tests*. URL: *http://members.pingnet.ch/gamma/junit.htm*. 1998

[C+01] CHRISTENSEN, Erik [u. a.]: *Web Services Description Language (WSDL) 1.1*. World Wide Web Consortium, März 2001. –
URL: *http://www.w3.org/TR/wsdl*

[CD01] CHEESMAN, John ; DANIELS, John: *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001

[CL02] CRNKOVIC, Ivica (Hrsg.) ; LARSSON, Magnus (Hrsg.): *Building Reliable Component-Based Software Systems*. Artech House, 2002

[Crn01] CRNKOVIC, Ivica: *Component-based Software Engineering - New Challenges in Software Development*. 2001. –
URL: *http://www.mrtc.mdh.se/publications/0328.pdf*

[DK76] DEREMER, Frank ; KRON, Hans H.: Programming-in-the-large versus Programming-in-the-small. In: *IEEE Transactions on Software Engineering* 2 (1976), Juni, Nr. 2, S. 80–86

[G+95] GAMMA, Erich [u. a.]: *Design Patterns*. Addison-Wesley, 1995

[Gri98] GRIFFEL, Frank: *Componentware. Konzepte und Techniken eines Softwareparadigmas*. dpunkt-Verlag, Heidelberg, 1998

[GT00] GRUHN, Volker ; THIEL, Andreas: *Komponentenmodelle*. Addison-Wesley, 2000

[HC01] HEINEMAN, George T. ; COUNCILL, William T.: *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001

[Hen03] HENNEY, Kevlin: Programmer's Dozen: 13 Recommendations for Refatoring, Repairing and Regaining Control of Your Code. In: *OOP 2003* Sigs Datacom, 2003

[Hub02] HUBERT, Richard: *Convergent Architecture – Buildung Model-Driven J2EE Systems with UML*. Jon Wiley and Sons, Inc., 2002

[IEE99] Kap. IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology In: IEEE (Hrsg.): *IEEE Software Engineering Standards Collection : spring 1999 edition*. New York : IEEE, 1999

[Jay84] JAY, Frank (Hrsg.): *IEEE Standard Dictionary of Electrical and Electronics Terms*. IEEE Inc., New York, 1984

[JBR99] JACOBSON, Ivar ; BOOCH, Grady ; RUMBAUGH, James: *The Unified Software Development Process*. Addison-Wesley, 1999, S. 191–192

[KG00] KULAK, Daryl ; GUINEY, Eamonn: *Use Cases – Requirements in Context*. ACM Press, 2000

[Kir97] KIRTLAND, Mary: Object-Oriented Software Development Made Simple with COM+ Runtime Services. In: *Microsoft Systems Journal* (1997), November, Nr. 11/97. – URL:*http://www.microsoft.com/msj/1297/complus2/complus2.htm*

[Küp03] KÜPER, Michael: *Implementierung von Web-Services für die Prozesse des Raumplanungssystems „Rabe"*, Institut für Softwaretechnologie, Universität Stuttgart, Studienarbeit, 2003

[LA02] LEVI, Keith ; ARSANJANI, Ali: A goal-driven approach to enterprise component identification and specification. In: *Communications of the ACM* 45 (2002), Nr. 10, S. 45–52

[LM02] LESSNER, Jan ; MÜLLER, Stephan: Ein Code – viele Anwendungen. In: *Javamagazin* (2002), November, Nr. 11/02, S. 42–46

[Lon01] LONG, John: Software reuse antipatterns. In: *ACM SIGSOFT Software Engineering Notes* 26 (2001), Nr. 4, S. 68–76

[LV01] LORENZ, David H. ; VLISSIDES, John: Designing components versus objects: a transformational approach. In: *Proceedings of the 23rd international conference on Software engineering*, 2001

[Mar02a] MARINESCU, Floyd: *EJB Design Patterns*. John Wiley & Sons Inc., 2002

[Mar02b] Kap. Component–based Systems In: MAROINIAK, John J.: *Encyclopedia of Software Engineering*. J. Wiley and Sons Inc., 2002, S. 183–191

[MB02] MILLER, Stephen J. ; BALCER, Marc J.: *Executable UML – A Foundation for Model-Driven Architecture*. Addison-Wesley, 2002

[Obj01a] Object Management Group: *Common Warehouse Model (CWM) Specification, Version 1.0*. Februar 2001. –
URL: *http://cgi.omg.org/docs/formal/01-02-01.pdf*

[Obj01b] Object Management Group: *OMG Unified Modelling Language Specification, Kapitel 6: The Object Constraint Language*. September 2001. –
URL: *http://www.omg.org/cgi-bin/doc?formal/01-09-77*

[Obj01c] Object Management Group: *Unified Modelling Language (UML) Specification, Version 1.4*. September 2001. –
URL: *http://cgi.omg.org/docs/formal/01-09-67.pdf*

[Obj02] Object Management Group: *CORBA Components, Version 3.0*. Juni 2002. –
URL: *http://cgi.omg.org/docs/formal/02-06-65.pdf*

[Par72] PARNAS, David L.: On the criteria to be used in decomposing systems into modules. In: *Communications of the ACM* 15 (1972), Nr. 12

[Par01] PARNAS, David L.: On the design and development of program families. (2001), S. 193–213

[RAJ01] ROMAN, Ed ; AMBLER, Scott W. ; JEWELL, Tyler: *Mastering Enterprise Java Beans*. 2. Auflage. John Wiley & Sons Inc., 2001

[Sol00] SOLEY, Richard: *Model Driven Architecture, Whitepaper, Draft 3.2*. November 2000. –
URL: *ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf*

[Sun97] Sun Microsystems Inc.: *JavaBeans Specification, Version 1.01*. Juli 1997

[Sun99] Sun Microsystems Inc.: *Java 2 Enterprise Edition Specification, Version 1.2*. Dezember 1999

[Szy99] SZYPERSKI, Clemens: *Component software: beyond object-oriented programming*. Addison-Wesley, 1999

[Tög02] TÖGEL, Roland: *Realisierung einer Web-basierten Anwendung zur Abbildung der Geschäftsprozesse zur Raumbelegung an der Fakultät Informatik*, Institut für Informatik, Universität Stuttgart, Diplomarbeit Nr. 2010, 2002

[Tra95] TRACZ, Will: *Confessions of a Used Program Salesman*. Addison-Wesley, 1995, S. 94

[U2 03] U2 Partners: *Unified Modeling Language: Superstructure, Version 2.0 2nd revised submission*. Januar 2003. –
URL: *http://www.u2-partners.org/outgoing/uml2/specs/030106/U2P-UML-Super-v2-PDF-030106.zip*

[Wei02] WEISE, Dirk: Softwareentwicklung mit der Model Driven Architecture. In: *Java Spektrum* (2002), November, S. 44–47

[WK94] WILLLIAMS, Sara ; KINDEL, Charlie: *The Component Object Model: A Technical Overview*. Microsoft Corporation, Oktober 1994. – URL: *microsoft.com/isapi/gomsdn.asp?TARGET=/library/techart/msdn_comppr.htm*