Klaus Turowski
(Hrsg.)

Tagungsband

5.Workshop Komponentenorientierte betriebliche Anwendungssysteme

25./26. Februar 2003

Augsburg

Gesellschaft für Informatik
Arbeitskreis 5.10.3
Komponentenorientierte betriebliche Anwendungssysteme

Tagungsleitung

Prof. Dr. Klaus Turowski
Lehrstuhl für Betriebswirtschaftslehre, insbesondere Wirtschaftsinformatik II
Universität Augsburg
Universitätsstraße 16, 86135 Augsburg
Phone: +49(821)598-4431; Fax : -4432
E-Mail: klaus.turowski@wiwi.uni-augsburg.de
URL: http://wi2.wiwi.uni-augsburg.de

Programmkomitee

Prof. Dr. S. Conrad
Heinrich-Heine-Universität Düsseldorf

Prof. Dr. R. Flatscher
Universität Augsburg

Prof. Dr. U. Frank
Universität Koblenz-Landau

Andreas Krammer
accenture

Prof. Dr. P. Loos
Universität Mainz

Prof. Dr. E. Ortner
Universität Darmstadt

Prof. Dr. C. Rautenstrauch
Otto-von-Guericke-Universität Magdeburg

Stephan Sahm
itelligence

Dr. Benno Schmitzer
100world AG

Prof. Dr. E. Sinz
Universität Bamberg

Prof. Dr. K.Turowski (Vorsitz)
Universität Augsburg

ISSN 1619-9006

Vorwort

Dieser Tagungsband enthält die schriftlichen Beiträge zum *fünften Workshop komponentenorientierte betriebliche Anwendungssysteme* (WKBA 5), der vom GI-Arbeitskreis 5.10.3 „Komponentenorientierte betriebliche Anwendungssysteme“ und dem Lehrstuhl für Betriebswirtschaftslehre, insbesondere Wirtschaftsinformatik II der Universität Augsburg am 25./26. Februar 2003 in Augsburg veranstaltet wurde.

Das Thema des Workshops war die komponentenbasierte Gestaltung betrieblicher Anwendungssysteme. Ausgehend von dieser Themenstellung, wurde um die Einreichung von Beiträgen gebeten, die insbesondere folgende Themen aufgreifen:

- Standardisierung und Spezifikation von Fachkomponenten
- Komposition, Configuration Management
- Komponenten-Anwendungs-Frameworks, Business Objects
- Kopplungstechniken und (fachliche) Konfliktbehandlung
- Architekturen, Komponenten-System-Frameworks, Middleware
- Komponentenmärkte
- Domänenanalyse und Komponentenrepositories
- Spezifische Fragestellungen des Information Managements
- Praktische Erfahrungen

Aus den Einreichungen wurden sechs Beiträge ausgewählt, die aktuelle Forschungsergebnisse aus dem Bereich der Wirtschaftsinformatik darstellen oder Erfahrungen aus der betrieblichen Praxis dokumentieren. Jeder der eingereichten Beiträge wurde in einem anonymen Begutachtungsverfahren (double blind) von mindestens zwei Mitgliedern des Programmkomitees begutachtet.

Dieser Tagungsband enthält die schriftliche Fassung der zu Vortrag und Veröffentlichung angenommen Workshopbeiträge.

Abschließend sei allen gedankt, die durch die Einreichung eines Beitrags zum Gelingen des Workshops beigetragen haben. Besonderer Dank gebührt den Mitgliedern des Programmkomitees für die Begutachtung der eingegangenen Beiträge und Herrn Johannes Maria Zaha für die Mitwirkung bei der Organisation des Workshops.

Augsburg, im Februar 2003

Klaus Turowski

Inhaltsverzeichnis

Komponentenmarktplätze – Bestandsaufnahme und Typologie

Peter Fettke[1], Peter Loos[1], Björn Viehweger[2]

[1] *Johannes Gutenberg-Universität Mainz, Lehrstuhl für Wirtschaftsinformatik und BWL, ISYM - Information Systems & Management, D-55099 Mainz, Germany, Tel.: +49 6131 39-22018, Fax: -22185, E-Mail: {fettke|loos}@isym.bwl.uni-mainz.de, WWW: http://www.isym.bwl.uni-mainz.de*

[2] *Technische Universität Chemnitz, Fakultät für Wirtschaftswissenschaften, Professur Wirtschaftsinformatik II, D-09107 Chemnitz, Germany, Tel.: +49 371 531-4375, Fax: -4376, E-Mail: bjoern.viehweger@isym.tu-chemnitz.de, WWW: http://www.isym.tu-chemnitz.de/*

Zusammenfassung. Komponentenmarktplätze sind ein wichtiger Bestandteil zur Verwirklichung der Leitidee der komponentenorientierten Softwareentwicklung. Zur Zeit existiert im Internet eine Reihe von Komponentenmärkten, auf denen Komponenten angeboten und nachgefragt werden. In diesem Beitrag wird zunächst eine Typologie zur Charakterisierung von Komponentenmarktplätzen eingeführt. Auf Basis der eingeführten Typologie werden acht Komponentenmarktplätze näher untersucht und beschrieben.

Schlüsselworte: Komponenten-Repositorium, Komponenten Retrieval, Bibliothek, Fachkomponente, Taxonomie, Klassifikation

1 Ausgangssituation und Problemstellung

Seit geraumer Zeit verfolgt die (Wirtschafts-)Informatik das Leitmotiv der Wiederverwendung von Software-Bestandteilen, um den Prozess der Entwicklung von Anwendungssystemen effektiv und effizient zu gestalten. Ein spezieller Ansatz zur Wiederverwendung sind komponentenorientierte Anwendungssysteme. Der Ansatz der komponentenorientierten Anwendungsentwicklung [Grif98; HeSi99; Same97; Szyp99; Turo01] beruht u. a. auf dem Leitbild, dass die zur Wiederverwendung vorgesehenen Fachkomponenten auf einem Marktplatz gehandelt werden. Ein solcher Marktplatz hat die ökonomische Funktion, das Angebot und die Nachfrage von Fachkomponenten zusammenzuführen.

Aufgrund der Verfügbarkeit und Nutzung des Internet ist es möglich und sinnvoll, Marktplätze für Fachkomponenten prinzipiell als elektronische Marktplätze zu entwerfen und zu betreiben. Derartige Systeme ermöglichen es, Fachkomponenten über das Internet zu offerieren und auszutauchen. Aufgrund der Immaterialität von Fachkomponenten sind elektronische Marktplätze grundsätzlich in der Lage, sämtliche Phasen einer Transaktion zwischen Komponentenanbietern und –nachfragern abzuwickeln. Ein Blick in die wirtschaftliche Realität zeigt, dass sich inzwischen eine Reihe von Software-Marktplätzen im Allgemeinen und Marktplätzen für Fachkomponenten im Besonderen herausgebildet haben.

Zielstellung des vorliegenden Beitrages ist es, auf Basis einer Typologie für Komponentenmarktplätze den Bestand an Komponentenmarktplätzen im Internet darzulegen. Die Untersuchung ist wie folgt gegliedert: Nach diesem einleitenden Abschnitt werden in Abschnitt 2 die wesentlichen Begriffe der Untersuchung eingeführt und die entwickelte Typologie vorgestellt.

In Abschnitt 3 werden die betrachteten Marktplätze im Einzelnen beschrieben. Einen Gesamtüberblick über alle betrachteten Marktplätze wird in Abschnitt 4 gegeben. Die Arbeit schließt mit einer knappen Zusammenfassung und einem Ausblick auf weitere Forschungsaktivitäten und Entwicklungen.

2 Methodik

2.1 Begriffliche Grundlagen

Die Begriffe Software-Komponente (kurz: Komponente) und Fachkomponente werden gemäß [Acke+02, S. 1] eingeführt: Eine Komponente ist ein wiederverwendbarer, abgeschlossener und vermarktbarer Softwarebaustein, der seine Realisierung verbirgt, in Kombination mit anderen Komponenten eingesetzt werden kann und Dienste über wohldefinierte Schnittstellen zur Verfügung stellt. Eine besondere Klasse von Komponenten sind Fachkomponenten: Fachkomponenten offerieren Dienste einer betrieblichen Anwendungsdomäne.

Traditioneller Weise ist ein Marktplatz ein Ort, an dem Güter zwischen verschiedenen Wirtschaftseinheiten ausgetauscht werden [Gabl01, Stichwort: Markt]. Ein elektronischer Marktplatz basiert ausschließlich auf den Möglichkeiten der Informationstechnologie und unterliegt demnach keinen zeitlichen und örtlichen Restriktionen beim Austausch der Güter [MYB87; Schm93; SeES02, S. 946; Timm98]. Auf elektronischen Marktplätzen können im Idealfall sämtliche Phasen einer Transaktion elektronisch abgewickelt werden.

Eine spezielle Klasse von elektronischen Marktplätzen bilden Komponentenmarktplätze. Auf einem Komponentenmarktplatz werden Komponenten zwischen Anbietern und Nachfragern ausgetauscht. Theoretische Anforderungen an einen Komponentenmarktplatz werden in [Kauf00, S. 55-108] behandelt. Spezielle Unterschiede zwischen Komponentenmarktplätzen und Komponentenrepositorien werden in [FLT02, S. 88-91] beschrieben. Die hier eingeführte Auffassung umfasst auch Marktplätze, auf denen ausschließlich Informationen über Komponenten ausgetauscht werden können. Auf solchen Marktplätzen im weiteren Sinne wird die eigentliche Transaktion nicht mehr über dem Marktplatz abgewickelt. Diese breite Auffassung wurde gewählt, um ein möglichst weites Spektrum der wirtschaftlichen Realität abzudecken. Der folgende Abschnitt beleuchtet die unterschiedlichen Facetten des Phänomens eines Komponentenmarktplatzes.

2.2 Typologie für Komponentenmarktplätze

Aufbauend auf den Ansätzen von [Behl00, S. 38-42; MMM98] wird im Folgenden eine Typologie für Komponentenmarktplätze vorgestellt. Die eingeführte Typologie basiert auf folgenden Merkmalen:

- Güter,
- Framework,
- Transaktionsphase,
- Betreiberrolle,
- Zugang,
- Angebot,

- Repräsentation,
- Datenerfassung und
- Suchmöglichkeiten.

Für die vorliegende Untersuchung erscheint es nicht sinnvoll, den Blickwinkel ausschließlich auf den Handel von Fachkomponenten zu beschränken. Um einen guten Überblick über den Bestand an Marktplätzen zu erhalten, ist es notwendig, den Blickwinkel zu erweitern und ein breites Spektrum von Gütern, die auf Marktplätzen gehandelt werden können, zu untersuchen. Diese Vorgehensweise erscheint aus drei Gründen angebracht. Erstens können auf einem Marktplatz neben dem Idealbild von Fachkomponenten auch andere Güter gehandelt werden. Diese Vielfalt sollte beschreibbar sein. Zweitens ist auf dem ersten Blick nicht unmittelbar erkennbar, ob auf einem Marktplatz ausschließlich Fachkomponenten gehandelt werden. Eine abschließende Beurteilung ist letztlich nur dann möglich, wenn sämtliche Komponenten betrachtetet und beurteilt werden. Diese Menge ist allerdings in ihrem Umfang nicht fixiert, sondern kann sich im Zeitablauf verändern. Drittens handelt es sich bei Komponentenmarktplätzen um ein neues Phänomen, dass zur Zeit noch weitgehenden Veränderungen unterworfen ist, so dass eine Einschränkung zum aktuellen Zeitpunkt nicht angebracht erscheint. Vor diesem Hintergrund werden unter dem Aspekt „Güter“ neben Fachkomponenten ebenso allgemeine Komponenten und sonstige Software unterschieden.

(Fach-)Komponenten können zum Betrieb bestimmte Dienste eines Komponenten-System-Frameworks bzw. eines Komponenten-Anwendungs-Frameworks [Turo01, S. 37-39] benötigen. Das Merkmal „Framework“ beschreibt, welche Anforderungen von den auf dem Marktplatz gehandelten Komponenten an das zugrundliegende Framework gestellt werden. Es werden folgende Ausprägungen unterschieden: „(Distributed) Component Object Model“ ((D)COM), .NET, Java Beans, Enterprise Java Beans und sonstige Komponenten-System- bzw. –Anwendungs-Frameworks.

Unter dem Aspekt „Transaktionsphasen“ wird unterschieden, welche Phasen der Transaktion bei einem Komponentenhandel unterstützt werden. Im Wesentlichen wird unterschieden zwischen den Phasen „Information“, „Kontakt“, „Vereinbarung“ und „Abwicklung“ [SeES02, S. 946]. Marktplätze, die ausschließlich der Informationsbeschaffung dienen, können als Informationsportale verstanden werden. Der Kauf einer Komponente kann in einem solchen Fall nicht über den Marktplatz abgewickelt werden, sondern muss über den jeweiligen Produzenten der Komponente erfolgen. Unterstützt ein Marktplatz sämtliche Transaktionsphasen wird er im Folgenden als Marktplatz im engeren Sinne verstanden. Ein Marktplatz im weiteren Sinne unterstützt nur einzelne ausgewählte Phasen eines Transaktionsprozesses.

Hinsichtlich der „Rolle des Betreibers“ wird differenziert, welchen Einfluss der Marktplatzbetreiber auf die angebotenen Komponenten ausübt. Hier wird unterschieden, ob es sich bei den Komponenten primär um die Komponenten eines Herstellers handelt. Es ist bspw. denkbar, dass die auf dem Marktplatz gehandelten Güter primär von *einem* Produzenten stammen. Andererseits können die Güter primär von *einem* Konsumenten nachgefragt werden. Gleichzeitig ist ebenso eine neutrale Rolle des Markplatzbetreibers denkbar.

Als ein weiteres Kriterium wird unterschieden, ob der Zugang zum Marktplatz für alle Anbieter und Nachfrager offen bzw. geschlossen ist. Bei einem geschlossenen Marktplatz kann es sich bspw. um ein unternehmensinternes Komponentenrepositorium handeln, das ausschließlich bestimmten Kundengruppen zugänglich gemacht wird. In der vorliegenden Untersuchung wurden ausschließlich offene Marktplätze betrachtet.

Ferner wird ebenso ermittelt, wie viele Güter auf dem Markplatz angeboten werden. Diese prinzipiell metrische Größe wird in der hier vorgestellten Typologie mit drei ordinalen Ausprägungen bewertet. Das Angebot wird als gering eingestuft, wenn weniger als 100 Güter angeboten werden, als mittel, wenn mehr als 100 und weniger als 1000 Güter angeboten werden und als umfangreich, wenn mehr als 1000 Güter offeriert werden.

Der Aspekt „Repräsentation“ beschreibt die Art und Weise, wie eine Komponente in einem Komponentenmarkt abgebildet und beschrieben wird. Bei einer Klassifikation wird eine Komponente in eine bestimmte Klasse eingeordnet [FeLo03]. Zwischen den Klassen können hierarchische Beziehungen definiert werden. Bei einer Indexierung können Komponenten frei wählbare Begriffe zugeordnet werden, die den Zweck der Komponente näher spezifizieren. Eine formale Spezifikation ist dann gegeben, wenn eine Komponente durch ein mathematisch fundiertes Modell beschrieben wird (bspw. in Form einer Algebraischen Spezifikation). Bei wissensbasierten Ansätzen werden Konzepte von wissensbasierten Systemen [Kurb92] zur Beschreibung von Komponenten verwendet. Eine hypertextbasierte Repräsentation ist gegeben, wenn die Komponente durch eine Menge von Textdokumenten beschrieben wird, zwischen denen Navigationsbeziehungen bestehen.

Die Registrierung einer Komponente in einem Komponentenmarkt erfordert es, die notwendigen Daten zu erfassen. Hier können manuelle, semi-automatische und eine voll-automatische Datenerfassung unterschieden werden. Eine voll-automatische Erfassung ist bspw. gegeben, wenn eine Komponente aufgrund ihrer Eigenschaften in bestimmte Klassen automatisiert eingeordnet werden kann.

Ein wichtiger Aspekt bei der komponentenorientierten Softwareentwicklung ist die Wiederauffindung von Komponenten [FeLo01]. Hinsichtlich der Suchmöglichkeiten gibt es einen unterschiedlichen Funktionsumfang. Zunächst ist zwischen einer navigierenden Suche (Navigation) sowie einer spezifizierenden Suche (Spezifikation) zu unterscheiden. Der Benutzer kann bei einer Navigation den Bestand an Komponenten direkt durchsuchen. Bei einer Spezifikation muss der Benutzer bestimmte Merkmale angeben, die die zu suchenden Komponenten zu erfüllen haben. Eine spezifizierende Suche kann weiter untergliedert werden in die Kategorien exakte Spezifikation, Suchbegriff, Ähnlichkeitssuche und natürliche Sprache.

Die eingeführte Typologie wird in Bild 1 zusammenfassend dargestellt.

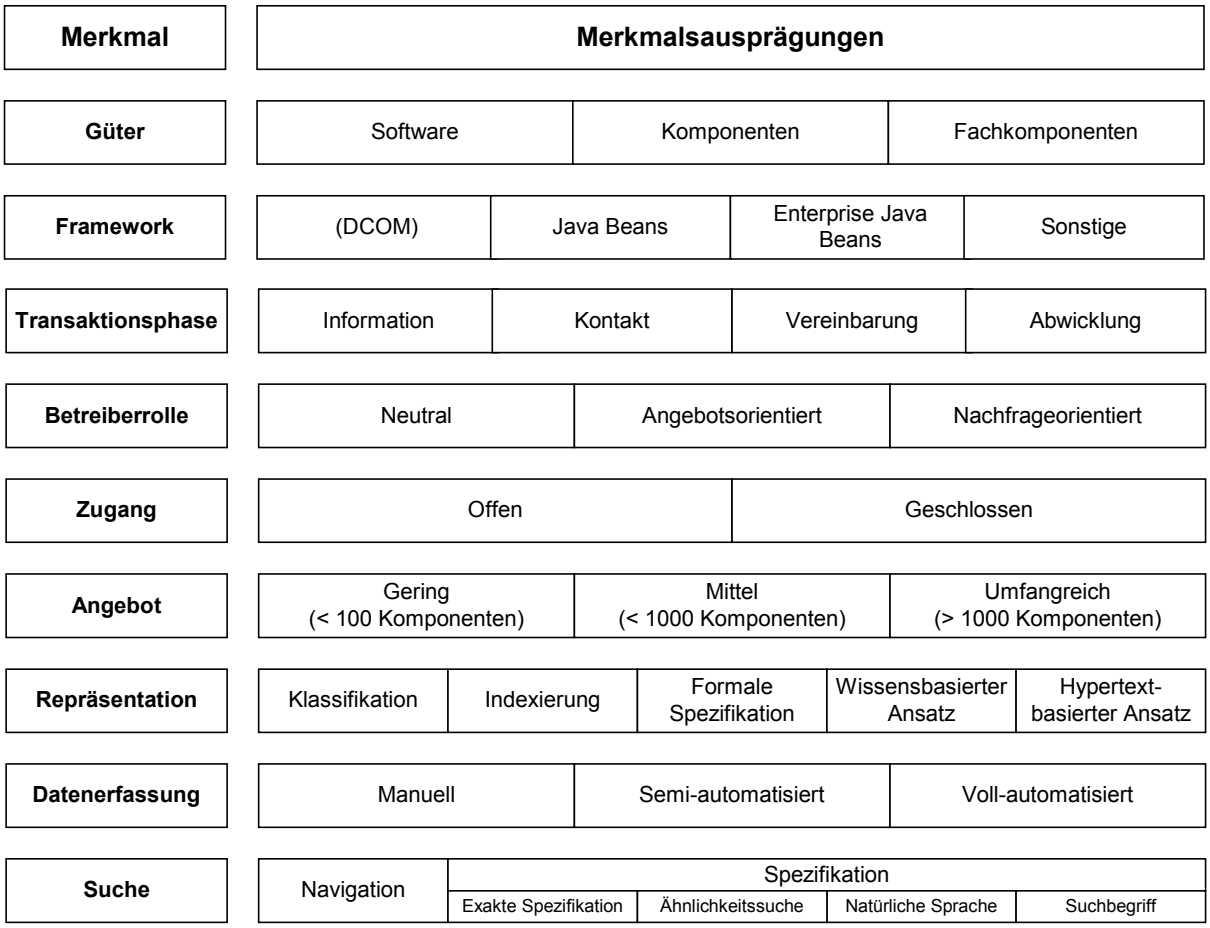

Bild 1: Typologie für Komponentenmarktplätze

2.3 Operative Aspekte

Unter Berücksichtigung von Internet-Katalogen, Suchmaschinen und Hinweisen aus Newsgroups wurden potenzielle Marktplätze ermittelt, die dann auf Ihre Eigenschaften als Marktplatz für Komponenten hin geprüft wurden. Durch die Selektion wurden bspw. Marktplätze für beliebige Programmierressourcen (wie Delphi Super Page (http://delphi.icm.edu.pl), JavaCats (http://www.javacats.com/us/research/) und SourceForge (http://sourceforge.net/)) aus der Untersuchung ausgeschlossen. Ferner wurden ebenso keine Markplätze mit in das Untersuchungsfeld aufgenommen, für die keine Originaldaten erhoben werden konnten, sondern nur Sekundärdaten vorliegen. Deswegen wurden bspw. die Marktplätze CompoNex [OrOv02; Over02, S. 13-16] und das DATEV-Komponenten-Repository [Hau01] aus dem Untersuchungsfeld ausgeschlossen. Ebenso wurden keine Marktplätze mit in das Untersuchungsfeld aufgenommen, die nicht explizit auf den Handel mit Komponenten ausgerichtet sind, sondern ausschließlich allgemeine Softwaresysteme focussieren (bspw. ISIS Software-Marktplatz (www.software-marktplatz.de)).

Die hier eingeführte Typologie für Komponentenmarktplätze wurde zweistufig erstellt. Zu Beginn der Untersuchung wurde eine initiale Version konzipiert, hinsichtlich der sämtliche betrachteten Marktplätze beschrieben wurden. Im Anschluss daran wurden die Ergebnisse der Untersuchung konsolidiert und die Typologie überarbeitet. Die Überarbeitung machte es not-

wendig, die betrachteten Marktplätze erneut aufzusuchen, um sie gemäß der veränderten Typologisierungsmerkmale zu beschreiben.

Die Erfassung aller relevanten Daten erfolgte im Zeitraum von Mai bis Dezember 2001. Der angesprochene Überarbeitungsschritt wurde im Juni 2002 abgeschlossen. Zum Teil ergaben sich bei der Erhebung der Daten Schwierigkeiten, die sich darin äußerten, dass sich die notwendigen Daten über einen Marktplatz von Außenstehenden nicht erheben ließen. Daher wurde versucht, mit den Betreibern der Markplätze in Dialog zu treten, wodurch der Untersuchungszeitraum verlängert wurde. Erschwert wurde die Untersuchung dadurch, dass einige Marktplätze vor dem Abschluss der vorliegenden Untersuchung ihre Internetpräsenz geschlossen haben bzw. nur unzureichend erreichbar waren. Aus diesem Grunde mussten die zunächst in das Untersuchungsfeld aufgenommen Marktplätze http://www.componentplanet.com, http://www.objectools.com, http://www.imagicom.com und http://www.findcomponents.com vom endgültigen Untersuchungsfeld wieder ausgeschlossen werden.

3 Beschreibung der Markplätze im Einzelnen

3.1 Überblick

Einen Überblick über die in dieser Untersuchung betrachteten Komponentenmärkte gibt Bild 2. Neben dem Namen des Komponentenmarktplatzes ist ebenso in der Darstellung die Uniform Resource Locator (URL) angegeben, unter dem der Marktplatz im Internet aufgerufen werden kann. In den folgenden Abschnitten werden die betrachteten Marktplätze in alphabetischer Reihenfolge vorgestellt.

Marktplatz	URL
ASP Resource Index	http://www.aspin.com
ComponentRegistry	http://www.componentregistry.com/
ComponentSource	http://www.componentsource.com/
EJBProvider	http://www.ejbprovider.com/
Flashline	http://www.flashline.com/
InternetComponent	http://www.internetcomponent.com/
SUN Solutions Marketplace	http://industry.java.sun.com/solutions/
VBXtras	http://www.vbxtras.com/

Bild 2: Betrachtete Komponentenmarktplätze

3.2 ASP Resource Index

Der Komponentenmarkt ASP Resource Index hat sich auf die Skriptsprache Active Server Pages (ASP) von Microsoft spezialisiert. Zusätzlich zum Angebot von Komponenten existieren noch mehrere Bereiche, die weiterführende ASP-Themen behandeln. Hierzu gehören umfangreiche Materialien und Tutorials zur Programmiersprache ASP. Ebenso sind elektronische Diskussionsrunden eingerichtet und nutzbar. Darüber hinaus gibt es eine elektronische Stellenbörse, die auf die Kompetenzen ASP, Visual Basic und verwandte Technologien ausgerichtet ist.

Die angebotenen Komponenten werden jeweils durch eine Kurzbeschreibung charakterisiert. Die Komponenten und weiterführende Informationen finden sich jeweils nicht auf dem Komponentenmarktplatz, sondern auf Internet-Seiten, die vom Hersteller der Komponenten gepflegt werden. Die Erfassung einer Komponente erfolgt manuell.

Die primäre Suche nach Komponenten ist eine Navigation. Auf oberster Ebene wird zwischen den Kategorien „Applications“ und „Components“ unterschieden. Bei näherer Betrachtung stellte sich allerdings heraus, dass beide Kategorien vom Inhalt fast deckungsgleich sind. In der jeweils nächsten Hierarchiestufe der Navigation werden funktionale Kategorien verwendet, die nicht einheitlich systematisiert sind (bspw. „Shopping & Commerce“ und „User Interface“). Neben den Möglichkeiten der Navigation existiert ebenso eine einfache Suchmaschine, die auf Basis von Stichwörtern funktioniert. Weitergehende Filtermöglichkeiten sind nicht einstellbar.

Neben der angesprochenen Kurzbeschreibung des Anwendungszwecks einer Komponente werden zusätzliche Angaben zum Update-Status und zu Kosten der Komponente getroffen. Es besteht die Möglichkeit, Bewertungen zu den Komponenten als Benutzer anzugeben bzw. zu lesen. Fehler von Komponenten können direkt an den Marktplatzbetreiber übermittelt werden. Ferner wird die Wiederauffindung von Komponenten dadurch unterstützt, dass der Benutzer die Möglichkeit hat, die angezeigten Informationen zu personalisieren (bspw. durch Favoritenangaben).

3.3 ComponentRegistry

Der Marktplatz ComponentRegistry bietet Komponenten auf Basis der Technologien Java Beans, Enterprise Java Beans (EJB) und Component Object Model (COM) an. Der Marktplatz basiert auf einer mit der Extensible Markup Language (XML) realisierten Datenbank. Zwischen diesem Marktplatz und dem Marktplatz Flashline besteht eine Kooperation, die sich nach außen allerdings darauf beschränkt, dass gegenseitige Bannerwerbungen eingeblendet werden. Weitere Funktionalitäten, die über den reinen Austausch von Komponenten hinaus gehen, werden nicht angeboten.

Die Komponenten werden auf dem Marktplatz durch XML-Dokumente repräsentiert. Der Marktplatz bietet nur eine Informationsstelle über Komponenten an. Weitere Informationen sind durch entsprechende Verknüpfungen zu den Internet-Seiten des Herstellers einer Komponente abrufbar. Die Erfassung einer Komponente basiert auf einer XML-Beschreibung. Der Marktplatz offeriert eine Document Type Definition (DTD) zur Beschreibung der Komponenten. Das Übersenden der XML-Beschreibung der Komponente stößt ein automatisiertes Katalogisieren der Komponente auf dem Marktplatz an.

Zur Navigation bietet der Marktplatz acht Hauptkategorien, die unterschiedlich systematisiert sind (bspw. Information Management und User Interface). Die Hauptkategorien werden in diverse Unterkategorien aufgeschlüsselt. Eine spezifizierende Suche ist durch selektive Angabe des Namens der Komponente, einer Beschreibung oder des Herstellers möglich. Ebenso kann nach Klassennamen gesucht werden, die zur Realisierung der Komponente implementiert wurden. Ferner ist es möglich, Eindrücke und Erfahrungen mit der Komponente anzugeben, wobei diese Funktionalität bisher nur wenig genutzt wurde.

3.4 ComponentSource

Der Marktplatz ComponentSource bietet ein umfangreiches Angebot an Komponenten an und ist nach eigener Aussage weltweit führend. Auf dem Marktplatz werden neben Möglichkeiten zum Kauf und Verkauf von Komponenten ebenso umfangreiche Informationen zur komponentenorientierten Softwareentwicklung angeboten. Der Marktplatz spricht insbesondere Großunternehmen an, indem für diese Zielgruppe ein eigener Marktbereich zur Verfügung gestellt wird.

Komponenten auf den Marktplatz werden auf Grundlage einer Klassifikation repräsentiert, die mehr als 50 Klassen umfasst. Mögliche Klassen sind fachliche Kategorien wie „Barcodes“, „Business Rules“, „Calendar“, „Manufacturing“ u. ä., aber auch technische Kategorien wie „Button Design“ und „VBA and Scripting“.

Die Erfassung von Komponenten erfolgt auf Basis eines Fragebogens, der vom Hersteller der Komponente auszufüllen ist. Auf Grundlage des Fragebogens wird die Komponente vom Marktplatzbetreiber in eine der Kategorien eingeteilt. Das Wiederauffinden von Komponenten wird durch umfangreiche Funktionalitäten ermöglicht. Zunächst kann auf Basis der definierten Klassifikationshierarchie der Bestand durchstöbert werden. Gleichzeitig können ebenso genaue Komponentenmerkmale als Suchkriterien verwendet werden. Hierbei besteht die Möglichkeit weitere Filterfunktionalitäten auszuführen. Die vorgenommenen Einstellungen können für spätere Suchanfragen gespeichert werden.

Zu jeder Komponente werden umfangreiche Informationen angezeigt. Gleichzeitig können zu jeder Komponente in einem Diskussionsforum Erfahrungen und Einsatzgebiete mit anderen Anwendern diskutiert werden. Der Marktplatz unterstützt eine Reihe verschiedener Lizenzarten (Freeware, Shareware, unternehmensweite Lizenzen etc.).

3.5 EJBProvider

EJBProvider ist ein spezialisierter Komponentenmarktplatz für Komponenten auf Basis der EJB-Technologie. Es werden hauptsächlich Fachkomponenten gehandelt. Der Marktplatz bietet Funktionalität für den Austausch von Komponenten an. Darüber hinaus wird ebenso die Möglichkeit gegeben, Ausschreibungen für bestimmte Komponentennachfragen zu generieren. Weitere Funktionen wie Diskussionsforen etc. werden nicht angeboten.

Komponenten auf dem Marktplatz werden durch eine zwei-stufige Klassifikationshierarchie geordnet. Das Klassifikationssystem umfasst bspw. die Klassen „Business“ mit den Unterklassen „Accounts“, „Fundraising“ u. ä. sowie „Insurance“ mit den Unterklassen „Auto Insurance“ und „Health Insurance“. Ausführliche Informationen können durch Verfolgung von Verweisen auf Seiten des Herstellers einer Komponente abgerufen werden. Die Registrierung der Komponente auf dem Marktplatz erfolgt manuell, indem per E-Mail eine entsprechende Meldung an den Marktplatzbetreiber übersandt wird.

Die möglichen Suchfunktionen basieren auf der oben genannten Klassifikationshierarchie (Navigation). Darüber hinaus besteht die Möglichkeit einer einfachen Stichwortsuche im hinterlegten Anwendungszweck der Komponente. Bei der Recherche fällt negativ auf, dass häufig sehr viele Werbeaussagen ohne konkrete Angebote bei den Einträgen hinterlegt sind.

3.6 Flashline

Flashline ist nach eigenen Angaben Industrieführer bei dem Vertrieb von Komponenten. Neben Funktionen zum Handel vom Komponenten bietet der Markplatz ebenso vielfältige Informationen zum Thema Wiederverwendung. Auf dem Marktplatz wird eine Reihe von Komponenten offeriert, die vom Betreiber selbst entwickelt worden sind.

Die Komponenten auf dem Marktplatz werden durch eine Klassifikation hinsichtlich technologischer Kriterien wie EJB oder COM dargestellt. Die vorgefundenen technologischen Kriterien werden auf einer zweiten Stufe in funktionale und anwendungsspezifische Klassen weiter untergliedert (bspw. Electronic Commerce und Data Management). Aussagen bezüglich des Automatisierungsgrades der Datenerfassung sind auf dem Marktplatz nicht ersichtlich, entsprechende Nachfragen seitens der Autoren dieser Untersuchung wurden vom Marktplatzbetreiber nicht beantwortet.

Der Marktplatz bietet primär eine Navigation zur Wiederauffindung von Komponenten an. Diese wird einerseits durch die oben genannte Klassifikation ermöglicht. Darüber hinaus werden zwischen den aufgenommenen Komponenten ebenso weitere Verweise definiert. Ebenso ist es möglich, alle Komponenten eines Herstellers anzeigen zu lassen. Die angebotene Suche nach Komponentenmerkmalen ist nicht benutzbar, da die gewählten Kategorien nicht gepflegt waren.

Als zusätzliche Funktionen bietet der Marktplatz einerseits die Möglichkeit, entsprechende Wünsche bzw. Bedarfe an Komponenten zu spezifizieren. Diese können von Komponentenherstellern ausgewertet werden. Andererseits ist es möglich, Erfahrungsberichte bei dem Einsatz von Komponenten abzugeben und abzurufen.

3.7 InternetComponent

Der Marktplatz InternetComponent ist eine Ausgliederung der amerikanischen Firma netesolutions, die sich auf die Herstellung von komponentenbasierten Internetanwendungen spezialisiert hat. Auf dem Markplatz werden ausschließlich Komponenten der Firma netesolutions gehandelt. Der primäre Anwendungszweck der Komponenten sind Internet-Anwendungen.

Die Komponenten werden auf Basis von Klassen repräsentiert, die eine technologische Ausrichtung haben (bspw. ActiveX, COM etc.). Darüber hinaus besteht die besondere Möglichkeit, mehrere elementare Komponenten zu einer zusammengesetzten Komponente zu aggregieren und als eigenständiges Produkt (der Marktplatzbetreiber spricht in diesem Zusammenhang von einer Suite) anzubieten. Eine zusammengesetzte Komponente deckt die Anforderungen eines größeren Anwendungsgebietes ab. Es konnten keine Informationen ermittelt werden, nach welchem Kriterien die Datenerfassung auf dem Marktplatz durchgeführt wird.

Eine Suche nach Komponenten erfolgt auf Basis der eingeführten technologischen Klassen. Als weitere Suchparameter können der Produktname und Stichwörter einer Produktbeschreibung gewählt werden. Der Marktplatz bietet zu jeder Komponente umfangreiche Informationen an, die sowohl das fachliche Anwendungsgebiet als auch die programmiertechnische Schnittstelle umfassen. Diese Beschreibung erfolgt indes durchweg natürlichsprachig. Als Besonderheit ist noch anzuführen, dass fast jede Komponente im Internet testbar ist.

3.8 SUN Solutions Marketplace

Die Firma SUN bietet mit ihrem Internet-Auftritt ein umfassendes Informationsangebot zum Thema Java (Standardspezifikationen, Referenzimplementierungen etc.). Ein Teil dieses umfassenden Angebots ist ein Marktplatz für Komponenten, die auf den Java-Technologien basieren (Java Beans und EJB). Darüber hinaus wird in einem speziellen Bereich eine weltweite Übersicht über Dienstleistungen im Java-Umfeld gepflegt, die regional gegliedert ist.

Auf dem Marktplatz werden nur Informationen über Komponentenanbieter verzeichnet; der Kauf von Komponenten ist nicht möglich. Hierzu müssen die Verweise auf die angebotenen Verknüpfungen verfolgt werden. Die Erfassung der Komponentenanbieter erfolgt manuell.

Die Wiederauffindung von Komponenten wird durch den Marktplatz mit Hilfe von Kategorien unterstützt, die sowohl fachlich (bspw. „Financial Services“ oder „Retail“) als auch technologisch (bspw. „J2EE“ oder „UI Elements“) ausgerichtet sind. Zu jedem Eintrag in der Übersicht werden Angaben zu aktuellen Neuerungen oder Veränderungen vorgestellt. Ebenso ist ein alphabetischer und regionaler Einstieg auf die Einträge möglich. Ferner besteht die Möglichkeit einer stichwortbasierten Suche, die die Felder Herstellername, Produktbeschreibung und Dienstleistung durchsucht.

3.9 VBXtras

Der Marktplatz VBXtras ist ausgerichtet auf Komponenten, die mit der Programmiersprache Visual Basic entwickelt worden sind. Neben den angebotenen Komponenten findet sich im Angebot ebenso eine Reihe von weiteren Informationen, die nützliche Werkzeuge, Tutorials, (elektronische) Bücher u. ä. betreffen.

Die Repräsentation von Komponenten erfolgt auf dem Marktplatz auf Basis einer Klassifikation, die an funktionalen Kriterien ausgerichtet ist. Die gebildeten Klassen werden wiederum mehrstufig verfeinert. Die Daten zur Erfassung einer Komponente müssen manuell gepflegt.

Eine Suche nach Komponenten erfolgt einerseits über eine Navigation. Als Einstiegspunkte dienen hier die Kategorien Produkte, Hersteller und Produktkategorien. Die Produktkategorien sind weiter untergliedert und so umfangreich, dass sie unübersichtlich werden. Die registrierten Komponenten sind über Verknüpfungen verbunden, so dass einerseits die Komponenten eines Herstellers aufgefunden werden können und andererseits Komponenten, die gut miteinander zu integrieren sind, angezeigt werden können. Ferner können Komponenten durch eine Stichwortsuche hinsichtlich der Aspekte Name, Hersteller, Kategorie und Beschreibung wieder aufgefunden werden. Abgerundet werden die Suchmöglichkeiten durch eine Anzeige der 40 meistverkauften Komponenten.

Der Marktplatz bietet ebenso die Möglichkeit, Erfahrungsberichte bei der Nutzung von Komponenten zu hinterlegen. Allerdings wurde von dieser Möglichkeit bisher nur wenig Gebrauch gemacht.

4 Vergleichender Gesamtüberblick

Einen vergleichenden Gesamtüberblick über die betrachteten Komponentenmarktplätze gibt Bild 3. Im Folgenden wird eine qualitative Einschätzung der ermittelten Komponentenmerkmale vorgenommen.

Bis auf den Marktplatz ComponentRegistry sind die betrachteten Marktplätze nicht ausschließlich auf den Handel mit Komponenten ausgerichtet. Vielmehr werden auf den Marktplätzen ebenso Fachkomponenten sowie Software offeriert, die nicht unter dem Komponentenbegriff fällt.

Auch wenn sich teilweise Marktplätze herausgebildet haben, die ausschließlich mit Komponenten handeln, die die Dienste eines bestimmten Frameworks benötigen (bspw. EJBProvider oder ASP Resource Index), so ist doch festzustellen, dass die betrachteten Markplätze sich i. d. R. nicht auf ein spezielles Framework konzentriert haben, sondern allgemeiner ausgerichtet sind. Dies ist ein Indiz dafür, dass sich noch keine Komponenten-System- bzw. -Anwendungs-Frameworks am Markt durchgesetzt haben.

Es existieren sowohl Märkte, die alle Transaktionsphasen unterstützen, als auch Märkte, die ausschließlich der Information bzw. der Kontaktvermittlung zwischen Anbietern und Nachfragern dienen. Märkte, die alle Transaktionsphasen unterstützen, bieten für den Nachfrager eine wesentlich komfortablere Recherchemöglichkeit, da der gesamte Beschaffungsprozess unter einem einheitlichen „Look and Feel“ realisiert werden kann. Indes können direkte Verweise auf Seiten des Herstellers zusätzliche Informationen bieten, die aufgrund des hierzu notwendigen Aufwandes vom Betreiber des Marktplatzes (im engen Sinne) nicht erhoben worden sind.

Die Betreiber der betrachteten Marktplätze sind weder als angebots- noch als nachfrageorientiert einzustufen. Vielmehr nehmen die Betreiber der Marktplätze eine neutrale Rolle ein. Eine Ausnahme bildet der Marktplatz InternetComponent, der als angebotsorientiert zu bezeichnen ist. Der Zugang zu den betrachteten Marktplätzen ist aufgrund der gewählten Methodik der Untersuchung (vgl. Abschnitt 2) stets offen.

Die Anzahl der angebotenen Komponenten auf den Marktplätzen schwankt erheblich. Auf dem Marktplatz EJBProvider werden weniger als 100 Komponenten, auf dem Marktplatz ComponentSource ca. 8000 Komponenten angeboten.

Zur Repräsentation von Komponenten verwenden die betrachteten Marktplätze primär eine Klassifikation sowie einen hypertextbasierten Ansatz. Wissensbasierte Ansätze oder Ansätze einer formalen Spezifikation kommen nicht zur Anwendung.

Die für die Erfassung einer Komponente notwendigen Daten werden i. d. R. manuell erhoben. Ansätze einer voll-automatisierten Erfassung sind nicht vorgesehen. Zum Teil sind allerdings entsprechende Erfassungsprozesse relativ weitreichend automatisiert.

Die betrachteten Marktplätze bieten durchweg eine Möglichkeit, durch den Bestand der Komponenten zu navigieren. Trotzdem unterscheidet sich die Mächtigkeit der Navigation im Einzelnen erheblich. Einige Marktplätze bieten neben der Navigation zwischen Klassen ebenso eine Möglichkeit an, Verknüpfungen zwischen Komponenten zu folgen, um häufig miteinander kombinierte Komponenten oder Komponenten vom selben Hersteller zu ermitteln. Ebenso können auf den Marktplätzen Komponenten durch eine spezifizierende Suche wieder aufgefunden werden. Auch hier gibt es im Detail erhebliche Funktionsunterschiede hinsichtlich der Granularität der zu spezifizierenden Merkmale, der Möglichkeiten der Filterung und Verfeinerung der gestellten Suchanfrage. Da keiner der betrachteten Markplätze eine formale Spezifikation zur Repräsentation von Komponenten unterstützt, verwundert es nicht, dass ebenso keiner der Marktplätze eine exakte Spezifikation als Suche bereitstellt. Darüber hinaus bieten die betrachteten Marktplätze durchweg keine Funktionalitäten an, um ähnliche Komponenten wieder aufzufinden.

Typologie			Komponentenmarktplätze							
Merkmal	**Merkmalsausprägungen**		**ASP Resource Index**	**ComponentRegistry**	**ComponentSource**	**EJBProvider**	**Flashline**	**InternetComponent**	**SUN Solutions Marketplace**	**VBXtras**
Güter	Software		●	–	–	●	●	–	●	●
	Komponenten		●	●	●	●	●	●	●	●
	Fachkomponenten		●	–	●	●	●	●	●	–
Framework	(D)COM		–	●	●	–	●	●	–	–
	.NET		–	–	●	–	●	●	–	–
	Java Beans		–	●	●	–	●	●	●	–
	Enterprise Java Beans		–	●	●	●	●	●	●	–
	Sonstige		●	–	●	–	●	●	–	●
Transaktions-phase	Information		●	●	●	●	●	●	●	●
	Kontakt		●	●	●	●	●	●	●	●
	Vereinbarung		–	–	●	–	●	●	–	●
	Abwicklung		–	–	●	–	●	●	–	●
Betreiberrolle	Neutral		●	●	●	●	●	–	●	●
	Angebotsorientiert		–	–	–	–	–	●	–	–
	Nachfrageorientiert		–	–	–	–	–	–	–	–
Zugang	Offen		●	●	●	●	●	●	●	●
	Geschlossen		–	–	–	–	–	–	–	–
Angebot	Gering		–	–	–	●	–	–	–	–
	Mittel		●	●	–	–	●	●	–	●
	Umfangreich		–	–	●	–	–	–	●	–
Repräsentation	Klassifikation		●	●	●	●	●	●	●	●
	Indexierung		–	–	–	–	–	–	–	–
	Formale Spezifikation		–	–	–	–	–	–	–	–
	Wissensbasierter Ansatz		–	–	–	–	–	–	–	–
	Hypertextbasierter Ansatz		●	●	●	–	–	–	–	–
Datenerfassung	Manuell		●	–	●	●	?	?	●	●
	Semi-automatisiert		–	●	–	–	?	?	–	–
	Voll-automatisiert		–	–	–	–	?	?	–	–
Suche	Navigation		●	●	●	●	●	●	●	●
	Spezifikation	Exakte Spezifikation	–	–	–	–	–	–	–	–
		Suchbegriff	●	●	●	●	●	●	●	●
		Ähnlichkeitssuche	–	–	–	–	–	–	–	–
		Natürliche Sprache	–	–	–	–	–	–	–	–

Legende: ●/–: Merkmalsausprägung erfüllt / nicht erfüllt, ?: keine Aussagen möglich

Bild 3: Gesamtüberblick über die betrachteten Komponentenmarktplätze

5 Zusammenfassung und Ausblick

In der vorliegenden Untersuchung wurde ein Überblick über aktuelle Komponentenmärkte geben. Die betrachteten Komponentenmärkte wurden auf Grundlage einer Typologie näher charakterisiert. Auch wenn auf den betrachteten Komponentenmärkte z. T. bereits beachtliche Mengen an (Fach-)Komponenten angeboten werden, ist eine ausschließliche Entwicklung von Anwendungssystemen aus diesem Angebot gemäß des komponentenorientierten Leitbildes noch nicht realistisch. Einerseits lässt die Abdeckung möglicher Anwendungsbereiche zu wünschen übrig. Andererseits werden Komponenten auf Marktplätzen zu unterschiedlich beschrieben, so dass das Aufsuchen von Komponenten und die Beurteilung, ob Komponenten für den vorliegenden Anwendungsfall geeignet sind, mit einem erheblichen Aufwand verbunden sind.

Eine Abhilfe versprechen standardisierte Komponentenkataloge, die sich in ihrer Struktur an Referenzmodellkataloge [FeLo02] anlehnen könnten. Komponentenkataloge wurden bereits in ihren Grundzügen von einigen Autoren [Broe94; Lang98; OrLK99] charakterisiert. Es erscheint notwendig, derartige Ansätze weiter auszubauen, um die Vergleichbarkeit und Wiederauffindbarkeit von vorhanden Komponenten sicherzustellen.

Vor dem Hintergrund, dass erfahrene Praktiker [PSWM00, S. 136-149] sich kritisch zu einem standardisierten Komponentenmarkt äußern, ist weiterhin zu überlegen, ob zwischen Anbietern und Nachfragern andere Koordinationsmechanismen als eine idealtypische Marktform zu etablieren sind. Hier ist bspw. in Anlehnung an Ansätze der verteilten Referenzmodellierung [Broc02] an virtuelle Engineering-Communities für Komponenten zu denken, die eine Plattform für die arbeitsteilige Entwicklung von Komponenten bieten, die über den reinen marktlichen Austausch von Komponenten hinausgehen. In einer solchen Engineering-Community könnten sich Entwickler und Nachfrager von Komponenten für einzelne Projekte zusammenschließen, um gemeinsam an einer adäquaten Zerlegung eines Gesamtsystems zu arbeiten und eine arbeitsteilige Entwicklung der einzelnen Komponenten vorzunehmen. Ein Akteur in diesem Netzwerk übernimmt die Verantwortung für die Komposition der Komponenten zu einem Gesamtsystem. Erste Ansätze in Richtung solcher Engineering-Communities werden bereits von heutigen Marktplätzen eingeschlagen, indem Funktionen für die Beurteilung von Komponenten, Mechanismen für die Formulierung von Komponentenbedarfen und Personalisierungsmöglichkeiten des Marktplatzauftrittes bereitgestellt werden. Derartige Funktionen sollten in Zukunft weiter ausgebaut werden.

Literatur

[Acke+02] *Ackermann, J.; Brinkop, F.; Conrad, S.; Fettke, P.; Frick, A.; Glistau, E.; Jaekel, H.; Kotlar, O.; Loos, P.; Mrech, H.; Raape, U.; Ortner, E.; Overhage, S.; Sahm, S.; Schmietendorf, A.; Teschke, T.; Turowski, K.*: Vereinheitlichte Spezifikation von Fachkomponenten - Memorandum des Arbeitskreises 5.10.3 Komponentenorientierte betriebliche Anwendungssysteme. http://wi2.wiso.uni-augsburg.de/gi-memorandum.php.htm, access date: 2002-05-10. Augsburg 2002.

[Behl00] *Behle, A.*: Wiederverwendung von Softwarekomponenten im Internet. Wiesbaden 2000.

[Broc02] *Brocke vom, J.*: Referenzmodellierung - Gestaltung und Verteilung von Konstruktionsprozessen. Diss., Universität Münster. Münster 2002.

[Broe94] *Broer, H.*: Software-Montagetechnik mit Software-Komponenten aus dem Katalog. In: OBJEKTspektrum (1994) 5, S. 68-77.

[FeLo01] *Fettke, P.; Loos, P.:* Fachkonzeptionelle Standardisierung von Fachkomponenten mit Ordnungssystemen - Ein Beitrag zur Lösung der Problematik der Wiederauffindbarkeit von Fachkomponenten. Working Papers of the Research Group Information Systems & Management, Paper 3. Chemnitz 2001.

[FeLo02] *Fettke, P.; Loos, P.*: Der Referenzmodellkatalog als Instrument des Wissensmanagements - Methodik und Anwendung. In: *J. Becker; R. Knackstedt (Hrsg.)*: Wissensmanagement mit Referenzmodellen. Konzepte für die Anwendungssystem- und Organisationsgestaltung. Berlin et al. 2002, S. 3-24.

[FeLo03] *Fettke, P.; Loos, P.*: Classification of reference models - a methodology and its application. In: Information Systems and e-Business Management 1 (2003) 1, S. 35-53.

[FLT02] *Fettke, P.; Loos, P.; Tann von der, M.*: Entwicklung eines Repositoriums für Fachkomponenten auf Grundlage des Vorschlages zur Vereinheitlichung der Spezifikation von Fachkomponenten - Analyse von Problemen und Diskussion von Lösungsalternativen. In: *K. Turowski (Hrsg.)*: Modellierung und Spezifikation von Fachkomponenten: 3. Workshop im Rahmen der MKWI (Multi-Konferenz Wirtschaftsinformatik) 2002, Nürnberg, Deutschland, 11. September 2002. Nürnberg 2002, S. 87-117.

[Gabl01] *Gabler (Hrsg.)*: Gabler Wirtschaftslexikon - CD-ROM. 15. Aufl., Wiesbaden 2001.

[Grif98] *Griffel, F.*: Componentware - Konzepte und Techniken eines Softwareparadigmas. Heidelberg 1998.

[Hau01] *Hau, M.:* Das DATEV-Komponenten-Repository - Ein Beitrag zu Marktplätzen für betriebswirtschaftliche Software-Bausteine. FORWIN-Bericht-Nr.: FWN-2001-003. Bamberg et al. 2001.

[HeSi99] *Herzum, P.; Sims, O.*: Business Component Factory - A Comprehensive Overview of Component-Based Development for the Enterprise. New York et al. 1999.

[Kauf00] *Kaufmann, T.*: Entwurf eines Marktplatzes für heterogene Komponenten betrieblicher Anwendungssysteme. Berlin 2000.

[Kurb92] *Kurbel, K.*: Entwicklung und Einsatz von Expertensystemen - Eine anwendungsorientierte Einführung in wissensbasierte Systeme. 2. Aufl., Berlin et al. 1992.

[Lang98] *Lang, K.-P.:* Variantenkonstruktion betriebswirtschaftlicher Anwendungssoftware. Bericht 98/02 der FG Wirtschaftsinformatik I der Technischen Universität Darmstadt. Darmstadt 1998.

[MYB87] *Malone, T. W.; Yates, J.; Benjamin, R. I.*: Electronic Markets and Electronic Hierarchies. In: Communications of the ACM 30 (1987) 6, S. 484-497.

[MMM98] *Mili, A.; Mili, R.; Mittermeier, R. T.*: A survey of software reuse libraries. In: Annals of Software Engineering 5 (1998), S. 349-414.

[OrLK99] *Ortner, E.; Lang, K.-P.; Kalkmann, J.*: Anwendungssytementwicklung mit Komponenten. In: Information Management & Consulting 14 (1999) 2, S. 35-45.

[OrOv02] *Ortner, E.; Overhage, S.*: CompoNex: Ein elektronischer Marktplatz für den Handel mit Software-Komponenten über das Internet. In: *S. Schubert; B. Reusch; N. Jessen (Hrsg.)*: Informatik bewegt - Informatik 2002, 32. Jahrestagung der Gesellschaft für Informatik e.V. (GI). Lecture Notes in Informatics (LNI P-19). Bonn 2002, S. 625-631.

[Over02] *Overhage, S.*: Die Spezifikation - kritischer Erfolgsfaktor der Komponentenorientierung. In: *K. Turowski (Hrsg.)*: 4. Workshop Komponentenorientierte betriebliche Anwendungssysteme, Augsburg, Deutschland, 11. Juni 2002, Tagungsband. Augsburg 2002, S. 1-17.

[PSWM00] *Plattner, H.; Scheer, A.-W.; Wendt, S.; Morrow, D. S.*: Dem Wandel voraus - Hasso Plattner im Gespräch mit August-Wilhelm Scheer, Sigfried Wendt und Daniel S. Morrow. o. O. 2000.

[Same97] *Sametinger, J.*: Software Engineering with Reusable Components. Berlin et al. 1997.

[SeES02] *Scheer, A.-W.; Erbach, F.; Schneider, K.*: Elektronische Marktplätze in Deutschland: Status quo und Perspektiven. In: WISU 31 (2002) 7, S. 946-950.

[Schm93] *Schmid, B.*: Elektronische Märkte. In: Wirtschaftsinformatik 35 (1993) 5, S. 365-480.

[Szyp99] *Szyperski, C.*: Component Software - Beyond Object-Oriented Programming. Harlow, England, et al. 1999.

[Timm98] *Timmers, P.*: Business Models for Electronic Markets. In: EM - Electronic Markets 8 (1998) 2, S. 3-8.

[Turo01] *Turowski, K.*: Fachkomponenten - Komponentenbasierte betriebliche Anwendungssysteme. Habil.-Schr. Magdeburg 2001.

Stücklistenbasiertes Komponenten-Konfigurationsmanagement

Steffen Becker[+], Sven Overhage[*]

+ *Georg-Spengler-Straße 20b, D-64291 Darmstadt, E-Mail: stbecker@rbg.informatik.tu-darmstadt.de*

* *An der Steinmauer 2, D-61191 Rosbach, E-Mail: sven@overhage.org*

Zusammenfassung. In diesem Beitrag wird ein Konzept für das Konfigurationsmanagement komponentenorientierter Anwendungen dargestellt. Dabei wird zunächst der Begriff „Konfigurationsmanagement" näher erläutert und anschließend die Stücklistenorganisation als eine geeignete Methode für das Konfigurationsmanagement beschrieben. Der Beitrag konzentriert sich auf die Entwicklung einer Vorgehensweise zur (automatisierten) Unterstützung der Komponentenauswahl, die auf Stücklisten, einem einheitlichen Spezifikationsrahmen und einer multiattributiven Entscheidungsfindung basiert. Abschließend wird das Änderungsmanagement beschrieben, das ebenfalls zum Konfigurationsmanagement zu zählen ist.

Schlüsselworte: Konfigurationsmanagement, Auswahlmanagement, Änderungsmanagement, Stückliste, Variantenstückliste, Spezifikation, Multiattributive Entscheidungsfindung

1 Einleitung

Die Einführung der Komponentenorientierung in die betriebliche Anwendungsentwicklung verspricht eine Reihe von Vorteilen, darunter neben einer kürzeren Entwicklungszeit der zu entwickelnden Anwendungen vor allem eine verbesserte Wartbarkeit, Skalierbarkeit und Qualität [Szyp1998:3ff]. Durch den Einsatz weitgehend vorgefertigter Komponenten reduziert sich zunächst vor allem der Aufwand während der Implementierung, die nunmehr lediglich die Neuentwicklung einiger weniger Komponenten sowie den Zusammenbau (die Konfiguration) von Komponenten zu Anwendungen umfasst. Darüber hinaus konzentriert sich die Wartung bzw. Skalierung einer komponentenorientierten Anwendung nur noch auf die jeweils von Änderungen betroffenen wenigen Komponenten, die noch dazu in der Regel von fachkundigen Herstellern realisiert wurden und durch zahlreiche Wiederverwendungen auch bereits entsprechend ausgereift sein dürften.

Trotz dieser vielen Vorteile konnte sich die seit langem in der Literatur empfohlene komponentenorientierte Anwendungsentwicklung [Mcil1968] in der Praxis bislang jedoch nicht auf breiter Front durchsetzen. Als Gründe für diese Entwicklung werden häufig fehlende fachliche und technische Standards für die Entwicklung und Spezifikation von Komponenten genannt, wodurch das Auftreten von Inkompatibilitäten bzw. Heterogenitäten [Over2002b] zwischen diesen begünstigt wird. Des Weiteren werden die bislang mangelhaft entwickelten Komponentenmärkte dafür verantwortlich gemacht, die jedoch eine weitere wichtige Voraussetzung für den Erwerb von Komponenten zur Anwendungsentwicklung darstellen [Hahn2002]. Darüber hinaus ist festzuhalten, dass die Einführung der Komponentenorientie-

rung in die Anwendungsentwicklung nicht automatisch zu besser strukturierten und beherrschbaren Anwendungen führt. So bleibt einerseits die Zerlegung einer zu entwickelnden Anwendung in geeignete Komponenten dem Geschick der jeweiligen Beteiligten überlassen. Andererseits führt die Zerlegung einer Anwendung in eine Vielzahl von Komponenten auch zu einer höheren Komplexität in der Anwendungsentwicklung, die zu beherrschen und durch entsprechende Methoden zu unterstützen ist. Als Beispiele hierfür sind vor allem Methoden für das Auswahlproblem im Hinblick auf einen geeigneten Komponenten-Mix bei der Entwicklung einer Anwendung sowie Methoden für die effiziente Ermittlung der in Anwendungen verbauten Komponenten zur Unterstützung von Wartungsarbeiten bzw. Skalierungsarbeiten an bestehenden Systemen zu nennen.

Hieraus lässt sich entnehmen, dass mit der komponentenorientierten Anwendungsentwicklung auch ein effizientes Komponenten-Konfigurationsmanagement entwickelt werden muss, dass den Entwurf sowie die Verwaltung von Konfigurationen (also Anwendungen) aus Komponenten unterstützt. Im Rahmen dieses Beitrags wird als Grundlage für das Komponenten-Konfigurationsmanagement in der Anwendungsentwicklung die Stücklistenorganisation verwendet die in der betrieblichen Produktionswirtschaft in ähnlicher Funktion bereits seit langem eine zentrale Rolle spielt [OrLK1999], [Ortn2001]. Im folgenden Kapitel wird zunächst der Begriff „Komponenten-Konfigurationsmanagement“ näher definiert und in die Anwendungsentwicklung eingeordnet. Daran anschließend werden verschiedene Arten von Stücklisten vorgestellt und deren Eignung für das Komponenten-Konfigurationsmanagement untersucht. Im vierten Kapitel wird auf Basis einer speziellen Stücklistenform ein möglicher Algorithmus für die Unterstützung bei der Auswahl von Komponenten vorgestellt, der die einzelnen Ebenen des standardisierten Komponentenspezifikationsrahmens des GI-Arbeitskreises Komponentenorientierte betriebliche Anwendungssysteme (im Folgenden als GI-Arbeitskreis bezeichnet) [Turo2002] als Parameter für die Entscheidungsfindung einbezieht. Abschließend werden einige mögliche Einsatzgebiete für Komponenten-Stücklisten im Rahmen der Wartungsarbeiten an bestehenden Anwendungen skizziert und weitere Forschungsaufgaben in diesem Themengebiet diskutiert.

2 Komponenten-Konfigurationsmanagement

Für die Wartung und Weiterentwicklung komponentenorientierter Anwendungen (bzw. Baugruppen) benötigen die Entwickler eine angemessene Dokumentation der Innensicht. Zwar lässt sich nach dem Komponentenmodell von Shaw und Garlan [ShGa1996:196ff], [OvTh2002] und der Komponentendefinition des GI-Arbeitskreises [Turo2002] eine Konfiguration (also eine Baugruppe bzw. eine Anwendung) wiederum als komplexe Komponente nach dem Black-Box-Prinzip (ggf. angereichert um eine Spezifikation) in die weitere Anwendungsentwicklung einbeziehen. Insbesondere für den Fall, dass in der bestehenden Anwendung bzw. Baugruppe einzelne Komponenten (im Falle eines Versions- oder Herstellerwechsels) auszutauschen sind, reicht jedoch diese Sichtweise für den Hersteller nicht aus.

Die Anforderungen, die sich im Hinblick auf die Dokumentation komponentenorientierter Anwendungen und Baugruppen aus Herstellersicht ergeben, werden vom Komponenten-Konfigurationsmanagement [Szyp1998:334], [LaCr2000] aufgegriffen, das hierfür geeignete Methoden und Werkzeuge bereitzustellen versucht. Es unterstützt schwerpunktmäßig den effizienten Zusammenbau, die Spezifikation der Innensicht sowie die strukturierte Verwaltung von Anwendungen, die auf der Basis einzelner Komponenten entwickelt werden.

In den Vorgehensmodellen zur Entwicklung komponentenorientierter Anwendungen spielt es somit vor allem in den Phasen „Systementwurf/Komponentenauswahl“, „Konfigurierung“ und „Betrieb/Wartung“ eine wichtige Rolle und ist der Entwicklung im Großen, also dem Bau von Anwendungen und Baugruppen aus fertigen Komponenten, zuzuordnen (vgl. Abbildung 1)[1].

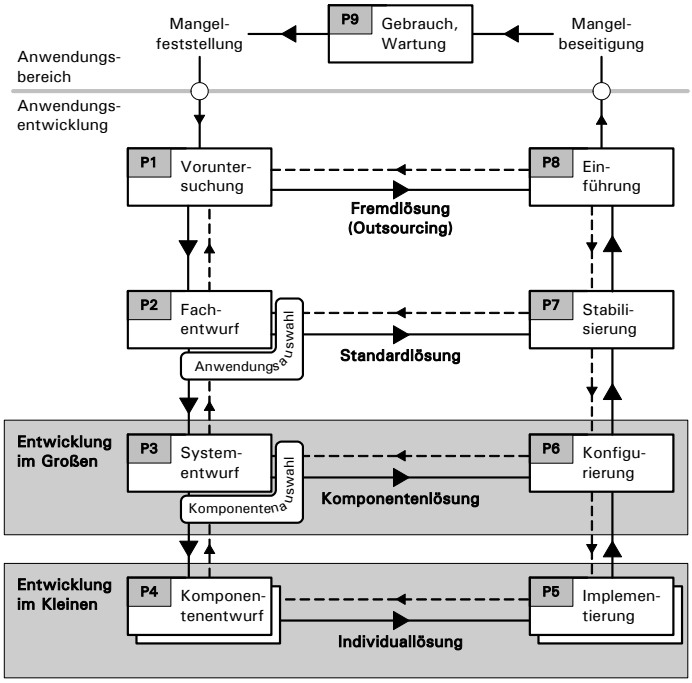

Abbildung 1: Flexibles Multipfad-Vorgehensmodell der komponentenorientierten Anwendungsentwicklung (basierend auf [Ortn1998], [Over2002a]).

Das Komponenten-Konfigurationsmanagement besteht im Wesentlichen aus zwei Anwendungsgebieten, die wie folgt voneinander abzugrenzen sind:

- **Auswahlmanagement:** Bei ausgereiften Komponentenmärkten (vgl. [Hahn2002]) ist allgemein davon auszugehen, dass eine Vielzahl von Bauteilen in einem oder mehreren Teilelagern zur Verfügung stehen. Die Herausforderung der komponentenorientierten Anwendungsentwicklung besteht nun zunächst darin, aus dieser Vielzahl von Teilen durch systematische Auswahl von Komponenten eine Anwendung zu konfigurieren, die die Anforderungen aus der Voruntersuchung bzw. dem Fachentwurf möglichst optimal erfüllt. Dabei ist es ferner wichtig, schon bei der Auswahl eine möglichst gute Abschätzung der einzukaufenden Komponenten und somit der zu erwartenden Einkaufkosten zu liefern, um ggf. andere (günstigere) Komponenten zu suchen und in den Entwicklungsprozess einzubeziehen.
- **Änderungsmanagement/Versionsmanagement:** Wartungsarbeiten an bestehenden komponentenorientierten Anwendungen werden häufig durch die Einführung neuer

[1] Im Gegensatz hierzu befasst sich die Entwicklung im Kleinen mit der Entwicklung einzelner Komponenten. Sie konzentriert sich also auf die Phasen „Komponentenentwurf“ und „Implementierung“ (vgl. Abbildung 1).

Versionen von Komponenten bzw. den Ersatz bereits verwendeter Komponenten verursacht. Mögliche Herausforderungen bei einem solchen Wartungsprozess bestehen einerseits in der effizienten Ermittlung der von Änderungen betroffenen Anwendungen sowie andererseits dem Einbinden der neuen (gegebenenfalls inkompatiblen) Komponente in die bestehenden Anwendungen. Dabei ist zunächst zu ermitteln, in welchen Anwendungen eine auszutauschende Komponente verwendet wird und ob durch die einzubauende Komponente Kompatibilitätsprobleme auftreten werden.

Beide Gebiete sind auf eine effiziente Methode zur Erfassung der mengenmäßigen Verwendung von Komponenten in Anwendungen bzw. Baugruppen angewiesen. Dazu können als Grundlage einerseits die während des Systementwurfs erstellten Baupläne (Systemdiagramme) dienen[2]. Diese haben jedoch den Nachteil, dass sie die mengenmäßige Verwendung nicht explizit ausweisen und man sie daher zunächst jedes Mal umständlich auswerten muss.

In der betrieblichen Produktionswirtschaft werden aus diesem Grund statt den vorhandenen Konstruktionsplänen der zu produzierenden Erzeugnisse vor allem Stücklisten verwendet, die sich durch ihre Spezialisierung auf die mengenmäßige Struktur auszeichnen. Hierdurch ergeben sich einige Vorteile: So ist es durch die Stücklistenauflösung einerseits effizient möglich, sowohl die verschiedenen Arten der Teile zu identifizieren, die jeweils als Bauteile in ein Erzeugnis eingehen, als auch die jeweilige Menge der benötigten Teile zu berechnen (und auf dieser Basis die Einkaufskosten zu berechnen). Andererseits können Stücklisten auch leicht so erweitert werden, dass sie einen sog. Verwendungsnachweis führen. Dieser erfasst, in welchen Erzeugnissen (genauer gesagt Erzeugnisarten) ein Bauteil verwendet wurde und ermöglicht so das Aufspüren der betroffenen Erzeugnisse im Falle der Änderung eines Bauteils.

Auf Grund ihrer Eigenschaften eignen sich Stücklisten auch für die in diesem Beitrag genannten Aufgaben des Komponenten-Konfigurationsmanagements und werden im Folgenden als dessen methodische Grundlage verwendet (zur prinzipiellen Eignung von Stücklisten in der komponentenorientierten Anwendungsentwicklung vergleiche insbesondere auch [OrLK1999], [Ortn2001]). Da Stücklisten bereits seit längerer Zeit für ähnliche Aufgabenstellungen eingesetzt werden, ist die Wahrscheinlichkeit gegeben, somit eine ausgereifte Methode wieder zu verwenden und ggf. auch einige an diese anknüpfende Methoden für das Komponenten-Konfigurationsmanagement anpassen zu können.

3 Stücklisten für das Konfigurationsmanagement

Allgemein betrachtet beschreiben Stücklisten die Zusammensetzung von Erzeugnissen aus Teilen und Baugruppen. Im Rahmen der komponentenorientierten Anwendungsentwicklung lassen sich dabei als mögliche Bauteile zur Entwicklung einer Konfiguration die einzelnen Komponenten betrachten. Solche Konfigurationen, die sich aus mehreren Komponenten zusammensetzen, können mit Baugruppen gleichgesetzt werden. Da Baugruppen wiederum als (komplexe) Komponenten angesehen werden können und als solche in die Anwendungsentwicklung eingehen [ShGa1996:196ff], bilden sie eine weitere Klasse von Teilen. Eine fertige Anwendung (also ein Endprodukt) wird dabei lediglich als eine besondere Art der Baugruppe angesehen.

---

[2] So finden sich beispielsweise in der UML auch speziell auf die Modellierung komponentenorientierter Anwendungen spezialisierte Diagrammsprachen [HoNo2001].

In der allgemeinen Betriebswirtschaftslehre bzw. Produktionswirtschaft werden verschiedene Arten von Stücklisten mit unterschiedlichen Vorzügen unterschieden, insbesondere Mengenstücklisten, Strukturstücklisten, Dispositionsstücklisten, Baukastenstücklisten und Variantenstücklisten [Schn1999:201ff]. Im Folgenden werden diese jeweils kurz charakterisiert, um im Anschluss daran aufzuzeigen, welche Form der Stücklistenorganisation sich für einen Einsatz im Konfigurationsmanagement am Besten eignet (vgl. Abbildung 2):

- **Übersichts- bzw. Mengenstücklisten** geben ausschließlich die aufsummierten (aggregierten) Mengen der jeweiligen Teile wieder, die in das Endprodukt eingehen.
- **Strukturstücklisten** erfassen weiterhin nach Baustufen geordnet, welche Teile bzw. Baugruppen in welchen Mengen zu neuen Baugruppen bzw. dem Endprodukt zusammengesetzt werden. Sie können durch Gozinto-Graphen veranschaulicht werden.
- **Dispositionsstücklisten** bauen auf Strukturstücklisten auf und untergliedern sie nach der Tiefe der Dispositionsstufe (des Gozinto-Graphen). Es werden dabei jeweils alle Teile einer bestimmten Art auf dieselbe Ebene des Graphen gebracht, so dass alle Teile eines Typs während desselben Produktionsschritts disponiert werden können. Sollten also gleiche Teile auf mehreren Dispositionsstufen vorkommen, werden sie einheitlich in die unterst mögliche Dispositionsstufe eingeordnet.
- **Baukastenstücklisten** blenden die Erzeugnisstruktur der einzelnen Bauteile, die in andere Teile eingehen, aus. Sie enthalten somit jeweils nur die Teile und Baugruppen, die direkt in ein Erzeugnis eingehen. Dies entspricht im Prinzip der bei der komponentenorientierten Anwendungsentwicklung üblichen Vorgehensweise, eine Baugruppe (Konfiguration) als neue Black-Box-Komponente anzusehen, deren Aufbau nicht notwendigerweise bekannt ist. Somit zerfällt der Gozinto-Graph, der den Erzeugniszusammenhang darstellt, in mehrere Graphen mit jeweils zwei Stufen.
- **Variantenstücklisten** können für das Konfigurationsmanagement nützlich sein, wenn verschiedene Produktvarianten (die normalerweise in getrennten Stücklisten erfasst werden müssten) gemeinsam in einer Stückliste vereinigt werden. Dieses Konzept ist auf die drei zuvor genannten Stücklisten anwendbar, die somit jeweils zu einer Variantenstückliste erweitert werden können. Um eine effiziente Darstellung für die Darstellung von Varianten zu erhalten, wurde eine Erweiterung der Knotenarten klassischer Stücklisten vorgeschlagen [WeMü1981]. Die verschiedenen Knotentypen sowie ihre Bedeutungen sind Tabelle 1 zu entnehmen. Abbildung 2 zeigt beispielhaft eine Variantenstrukturstückliste. Mit ihnen können beispielsweise Buchhaltungsanwendungen, die als Bilanzierungskomponente eine US-GAAP-Variante (US Generally Accepted Accounting Principles) bzw. eine IAS-Variante (International Accounting Standard) verwenden, gemeinsam erfasst werden. Über die Unterscheidung verschiedener Bilanzierungsarten hinaus wäre es auch denkbar, zu jeweils einer Bilanzierungsart verschiedene Komponenten zu unterscheiden, die diese implementieren – diese würde man ebenfalls als Variante bezeichnen und in der gemeinsamen Variantenstückliste erfassen. In der Literatur finden sich weitere Möglichkeiten Variantenstücklisten darzustellen. Plus-/Minus-Stücklisten [Zimm1988:113ff] bilden beispielsweise die Differenz zwischen einer Grundvariante und den restlichen Varianten dadurch ab, dass Stücklistenpositionen durch Subtraktion von der Grundliste entfernt werden können und durch Addition neue Positionen hinzukommen. Die darüber hinaus bekannten fiktiven bzw. Gleichteilestücklisten bilden dabei eine Sonderform von

Plus-/Minus Stücklisten, in denen nur additive Positionen auftreten, d.h. die Grundvariante wird hierbei immer um geeignete Teile erweitert.

Für den Einsatz im hier betrachteten Komponenten-Konfigurationsmanagement kommen die vergleichsweise einfach aufgebauten Mengen- bzw. Baukastenstücklisten nicht in Betracht, da die Strukturen der zusammengesetzten Teile durch die jeweilige Anordnung in diesen speziellen Stücklisten verloren gehen und somit nicht mehr direkt zu entnehmen sind. Falls nötig, können die komplexeren Formen von Stücklisten jedoch problemlos (d.h. effizient) in diese beiden einfacheren Arten transformiert werden, so dass durch die Nichtbetrachtung kein Nachteil entsteht.

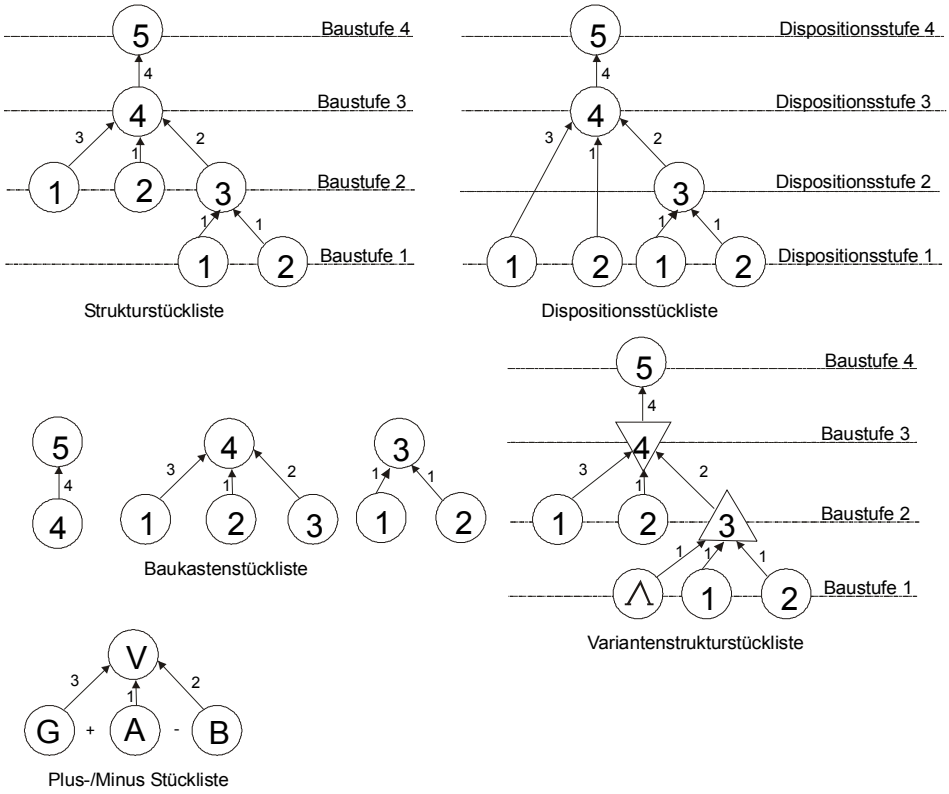

Abbildung 2: Gegenüberstellung verschiedener Stücklistenarten

Strukturstücklisten und Dispositionsstücklisten sind hingegen für eine Verwendung prinzipiell gleichermaßen geeignet. Dispositionsstücklisten können von Vorteil sein, wenn die Teile einer Dispositionsstufe zuerst entwickelt und getestet werden bzw. genauer evaluiert werden müssen. Sie liefern bedingt durch ihre spezielle Struktur die zeitliche Reihenfolge für die Ausführung solcher Tätigkeiten, die jeweils der Reihenfolge der Dispositionsstufen entspricht. Vereinfachend werden im Folgenden jedoch die einfacher strukturierten Strukturstücklisten verwendet, die zur Verdeutlichung des Einsatzgebiets bereits ausreichend sind.

Diese werden für das Komponenten-Konfigurationsmanagement als Variantenstrukturstücklisten verwendet. Dies ermöglicht die Dokumentation von Anwendungen, die sich nur an einigen Stellen durch den Einsatz anderer Komponenten (Varianten) unterscheiden. Dabei können parametrisierte Komponenten als diskrete bzw. stetige Varianten aufgefasst werden [Zimm1988:6ff]. In dieser Arbeit wird dabei die Darstellung gem. Tabelle 1

verwendet. Für eine effiziente Speicherung der Stückliste bietet sich bei der Anwendungsentwicklung Plus-/Minus Stücklisten oder Gleichteilestücklisten eher an, da die zu erwartenden Variationen gegenüber einer Grundvariante gering sein dürften und damit die Differenzinformationen ebenfalls gering ausfallen. Dabei ist eine Umwandlung dieser Stücklisten in die hier verwendete Darstellungsform problemlos möglich.

Knotentyp	Bedeutung	Symbol
Teileknoten	Repräsentiert ein Teil. Mögliche Teile sind Komponenten oder zu einem neuen Bauteil zusammengefasste Komponenten. Die Mengenangabe an der Kante gibt an, wie oft das Teil in die übergeordnete Struktur eingeht.	
Konjunktiv-knoten	Fasst alle Teile an eingehenden Kanten zu einem neuen Bauteil zusammen (entspricht einem logischen Und).	
Alternativ-knoten	Genau eine der eingehenden Kanten wird ausgewählt (entspricht einem logischen exklusiven Oder). Das gewählte Teil rückt de facto an die Stelle des Alternativknotens.	
Leerer Knoten	Wird benötigt, um im Zusammenhang mit Alternativknoten eine optionale Auswahl zu modellieren.	

Tabelle 1: Knotentypen zur Darstellung einer Variantenstückliste

Abbildung 3 zeigt eine Baugruppe, die Dienste zur Erstellung eines Jahresabschlusses im Rahmen der Finanzbuchführung anbietet. Hierbei gibt es mehrere verschiedene Baugruppen, die einen Jahresabschluss nach unterschiedlichen Bilanzrichtlinien ermöglichen (beispielsweise IAS und US-GAAP). Die unterschiedliche Funktionalität wird bedingt durch die Verwendung einer jeweils angepassten Komponente „Bilanzierung“, also den Einsatz von Varianten – eine Variantenstückliste vermag diese Anwendungen zusammengefasst darzustellen.

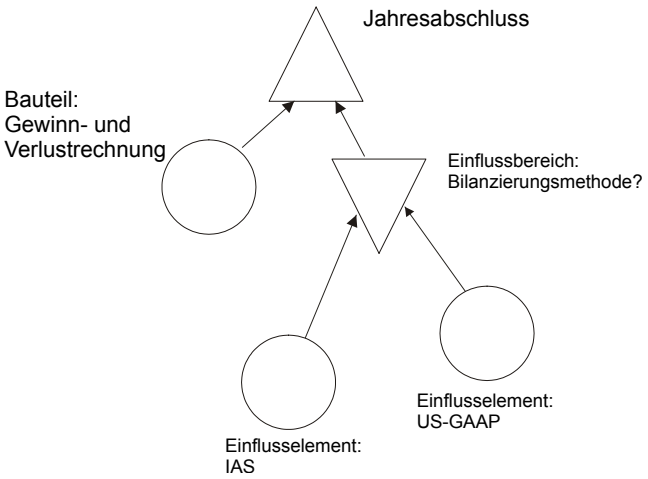

Abbildung 3: Variantenstückliste für einen Jahresabschluss

Kann die Menge der Erzeugnisvarianten bei einer Variantenstückliste explizit, z.B. in Form eines Kataloges, angegeben werden, ergibt sich eine geschlossene Variantenstückliste. Ist diese Menge hingegen nur implizit durch die möglichen Entscheidungen innerhalb der Stückliste definiert, so handelt es sich um eine offene Stückliste. Aufgrund der zu erwartenden ho-

hen Variantenanzahl im betrachteten Anwendungsgebiet werden in diesem Beitrag offene Variantenstücklisten betrachtet.

Zusammengefasst lässt sich festhalten, dass im Rahmen des Komponenten-Konfigurationsmanagements der Einsatz (offener) Strukturvariantenstücklisten als methodische Grundlage zu empfehlen ist. Ein erstes mögliches Einsatzgebiet für solche Stücklisten findet sich im Rahmen der Kalkulation der Einkaufskosten bei der Komponentenauswahl. Alle genannten Stücklisten lassen sich durch die Möglichkeit der Stücklistenauflösung für die Kalkulation dieser Einkaufskosten heranziehen. Bei Variantenstücklisten ist dabei zunächst zu unterscheiden, ob die Entscheidungen für eine Variante jeweils bereits getroffen wurden oder nicht. Falls die Entscheidung nicht getroffen wurde, kann die Kostenkalkulation entweder für jede erlaubte Variantenkombination separat vorgenommen werden (wobei sich dann eine Vielzahl von möglichen Kosten ergibt) oder für jeden Variantenknoten ein Kostenminimum und ein Kostenmaximum bestimmt werden. Durch Aggregation entlang der Stückliste erhält man auf diese Weise eine untere und obere Schranke der anfallenden Kosten. Ist die Entscheidung für eine Variante bereits bei allen Variantenknoten gefallen, so wird aus der Variantenstückliste eine „klassische“ Stückliste, die mit den entsprechenden Verfahren aufgelöst werden kann.

Bei einer „klassischen“ Stücklistenauflösung werden zunächst die verschiedenen Teilearten, die in das Erzeugnis eingehen, und deren jeweilige Mengen ermittelt. Anhand der Summe der mit ihren jeweiligen Einkaufskosten multiplizierten Teile kann daran anschließend eine Abschätzung der gesamten Einkaufskosten vorgenommen werden. Besonderheiten ergeben sich hierbei eventuell durch nicht konstante Einkaufskosten pro Teil. Übertragen auf die komponentenorientierte Anwendungsentwicklung bedeutet dies beispielsweise, dass unterschiedliche Lizenzmodelle mit Auswirkung auf den Einkaufspreis bestehen können. So ist denkbar, dass jede in der Anwendungsentwicklung verwendete Komponente mit der gleichen Lizenzgebühr erworben werden muss oder durch eine einmalig vergütete Lizenzgebühr beliebig viele Komponenten dieser Art verwendet werden dürfen. In letzter Zeit etablieren sich, bedingt durch neue Vertriebsarten wie ASP (Application Service Providing) oder XML Web-Services, auch Lizenzmodelle, die eine Gebühr erst bei der Benutzung der Komponente durch einen Anwender zur Laufzeit vereinbaren. In diesem Fall ist die Verwendung im Rahmen der Entwicklung sogar (bezogen auf die Einkaufskosten) kostenfrei. Daher geht in die Preiskalkulation stets noch eine weitere Funktion ein, die die verwendeten Teile im Hinblick auf die jeweils vereinbarten Lizenzbedingungen gewichtet.

4 Stücklistenbasiertes Auswahlmanagement

Der Einsatz von Variantenstücklisten unterstützt über die Kostenkalkulation hinaus insbesondere auch die Phase der Komponentenauswahl während der komponentenorientierten Anwendungsentwicklung. Angenommen, es existiert für die zu erstellende Anwendung bereits eine Variantenstückliste, die während des Anwendungsentwurfs (z.B. auf Basis des Fachentwurfs) bereits hergeleitet wurde. Im Rahmen des Anwendungsentwurfs fällt dann zunächst für jeden Variantenknoten der Stückliste die Entscheidung für eine bestimmte Variante. Hierdurch wird der exakte Typ der zu erstellenden Anwendung festgelegt und die Variantenstückliste somit zunächst zu einer „klassischen“ Stückliste ohne Alternativknoten. Bezogen auf das in Kapitel 3 eingeführte Beispiel der Baugruppe „Jahresabschluss“ bedeutet dies, dass sich die Entwickler nunmehr auf die Erstellung einer Baugruppe festlegen, die nach den Vorschriften des IAS bilanziert.

Im Anschluss an diese Entscheidung stellt sich im Rahmen der Komponentenauswahl nun das Problem der Beschaffung der einzelnen in der Stückliste auftretenden Teile. Zu Beginn der Komponentenauswahl ist die Menge an existierenden Komponenten, die anstelle eines in der Stückliste verzeichneten Teils verwendet werden können, noch unbekannt. Um diesen Status klar hervorzuheben, wird die Einführung einer sog. Entscheidungsstückliste vorgeschlagen.

Dabei wird ein neuer Knotentyp eingeführt, der mit einem Quadrat dargestellt werden soll und angibt, dass für den entsprechenden Knoten noch eine Entscheidung für ein konkretes Teil zu treffen ist. In einem Entscheidungsknoten werden Informationen über das benötigte Teil vermerkt, die direkt aus den Anforderungen des Anwendungsentwurfs abgeleitet wurden. Dies geschieht durch die Spezifikation einer fiktiven Referenzkomponente, die auf der Basis eines einheitlichen Spezifikationsrahmens erstellt wird. Es handelt sich hierbei um ein Muster, dem die Spezifikationen der Komponenten möglichst entsprechen müssen.

Bezug nehmend auf das in Kapitel 3 gegebene Beispiel bedeutet dies zunächst, dass die in Abbildung 3 gezeigte Variantenstückliste zu einer Entscheidungsstückliste wird, die in Abbildung 4 dargestellt ist. Der vorhandene Variantenknoten „Bilanzierungsmethode“ ist durch das zusätzliche Erfassen von „IAS“ eliminiert worden und existiert nur noch scheinbar. Aus dem Bauteil „IAS“ ist nunmehr ein Entscheidungsknoten geworden, der für eine (derzeit unbekannte) Menge von (unterschiedlich geeigneten) Komponenten steht, die als Kandidaten für die Anwendungsentwicklung in Frage kommen.

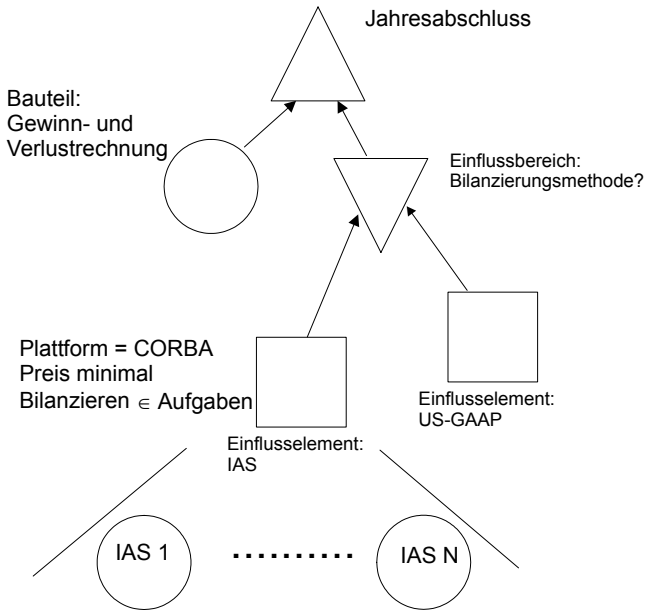

Abbildung 4: Entscheidungsstückliste für die Komponentenauswahl

Da es von zentraler Bedeutung für den Auswahlprozess ist, dass sowohl die Komponenten als auch die Referenzspezifikation einheitlich sind, wird als Spezifikationsrahmen der GI-Spezifikationsrahmen [Turo2002] empfohlen. Dabei ist es jedoch wichtig, den Spezifikationsrahmen als Ganzes in ein maschinenlesbares Format zu bringen, wobei sich hier die Entwicklung einer XML basierten Notation anbietet [OvTh2002], damit die Daten mit möglichst geringem Zusatzaufwand von den entsprechenden CASE-Tools ausgetauscht, eingelesen und

ausgewertet werden können. Auf der Basis einer solchen Notation können entscheidungsunterstützende Systeme arbeiten, die die Auswahl von Komponenten vereinfachen sollen. Die Referenzspezifikation, die in den Entscheidungsknoten beinhaltet ist, muss darüber hinaus nicht vollständig im Sinne des Spezifikationsrahmens sein. In ihr sind lediglich diejenigen Anforderungen zu speichern, die die zur Auswahl stehenden Alternativen (möglichst gut) erfüllen sollen. Wird ein Datensatz oder gar eine ganze Ebene der Spezifikation ausgelassen, so bedeutet dies, dass diese Informationen nicht relevant für die zu treffende Entscheidung sind. Formal ausgedrückt heißt dies, dass die möglichen Alternativen in den ausgelassenen Eigenschaften ihrer Spezifikation nicht gebunden sind.

Die Referenzspezifikation enthält unterschiedliche Angaben. Darunter sind Satisfizierungsziele und Extremalziele zu unterscheiden. Satisfizierungsziele sind Referenzspezifikationen, bei denen es nicht wichtig ist, wie gut sie erfüllt werden, solange sie in irgendeiner Form erfüllt werden. Sie stellen also etwas salopp formuliert K.O.-Kriterien für den Auswahlprozess dar. Bei Extremalzielen wird hingegen gefordert, dass die Referenzspezifikation möglichst gut erreicht wird. In der Entscheidungstheorie werden Referenzspezifikationen der letztgenannten Art über zu optimierende Zielfunktionen abgebildet.

Im Beispiel aus Abbildung 4 fordert eine Referenzspezifikation zum Beispiel als Satisfizierungsziel ein bestimmtes Komponentenframework, die entsprechende Formulierung lautet entsprechend Plattform='CORBA'. Hierbei wird ein Attribut der Vermarktungsebene des Spezifikationsrahmens verwendet, um eine Anforderung zu formulieren. Die zur Auswahl stehenden Komponenten sollen darüber hinaus die geforderte fachliche Aufgabe durchführen können, d.h. es ist zu verlangen, dass „Bilanzieren" in der Menge der von den Komponenten angebotenen Dienste enthalten sein muss. Auch dies ist ein Satisfizierungsziel, das sich diesmal jedoch auf die Aufgabenebene der Spezifikation bezieht. Da das betrachtete Unternehmen keinen großen finanziellen Spielraum besitzt, fordert das Management, dass die einzukaufende Komponente möglichst billig sein soll, d.h. es wird verlangt, dass der Preis, ein weiteres Attribut der Vermarktungsebene, minimal wird. Hierbei handelt es sich offensichtlich um ein Extremalziel. Dieses Beispiel ließe sich beliebig auf die verbleibenden Ebenen des Spezifikationsrahmens ausdehnen.

Im Folgenden soll der Prozess der Komponentenauswahl auf Basis einer Entscheidungsstückliste genauer beschrieben werden. An dessen Ende sind schließlich alle Teile ausgewählt (und alle Variantenentscheidungen getroffen). Das bedeutet, dass als Resultat wieder eine „klassische" Stückliste ohne Entscheidungs- bzw. Variantenknoten entsteht. Um dieses Ziel zu erreichen, wird (basierend auf [Kont1995], [KoHe1996]) eine methodische Vorgehensweise in insgesamt sieben Schritten empfohlen. Diese kann zunächst teilautomatisiert werden und durch entsprechende Systeme unterstützt werden. Langfristig ist die vollständig automatisierte Entscheidung eine anzustrebende Perspektive, die zum Beispiel in CASE-Tools für die Anwendungsentwicklung oder in Anwendungen selbst integriert werden könnte (sog. self assembling applications [OvTh2002]).

4.1 Schritt 1: Eliminierung der Variantenknoten

Zunächst werden die Variantenknoten wie bereits zu Beginn des Kapitels geschildert durch die Entscheidung für einen konkreten Anwendungstyp aus der Stückliste entfernt. Dies geschieht dadurch, dass von der tiefsten Ebene ausgehend für jeden Variantenknoten genau eine der möglichen Alternativen ausgewählt wird. Um diese Entscheidung zu unterstützen werden detaillierte Informationen benötigt. Daher wurde in [WeMü1981] vorgeschlagen, den Varian-

tenknoten sowie den möglichen Alternativen weitere Attribute zuzuordnen, die die Entscheidungsfindung unterstützen sollen. Ein Alternativknoten erhält zunächst eine Information darüber, welche Entscheidung zu treffen ist. Diese Information wird Einflussbereich (der Entscheidung) genannt. Zum Beispiel ist der Einflussbereich bei der Auswahl der richtigen Bilanzierungskomponente aus Kapitel 3 die Art der Bilanzierung. Die möglichen Alternativen bekommen passend zu diesem Einflussbereich ein Attribut, das beschreibt, in welcher Weise der Einflussbereich von der jeweiligen Alternative bedient wird. Diese Information wird Einflusselement genannt.

Ein Auswahlprozess kann mittels dieser Attribute beispielsweise in Form eines dialoggesteuerten Prozesses mit den Entwicklern stattfinden. Ein CASE-Tool kann so, beginnend mit der obersten Ebene der Variantenstückliste, systematisch den Entwickler befragen, welche Auswahl er zu einem gegebenen Einflussbereich treffen möchte. Im Beispiel könnte das System also fragen: „Welche Komponente soll zum Einflussbereich „Bilanzierung“ gewählt werden? Alternative a) IAS oder Alternative b) US-GAAP“. Die zu treffende Entscheidung wird dabei maßgeblich durch die Art der herzustellenden Variante bestimmt, die durch die Wünsche des Kunden explizit gegeben ist.

4.2 Schritt 2: Suche nach potenziell geeigneten Komponenten

Nachdem die Variantenknoten aus der Stückliste durch die entsprechenden Entscheidungen ersetzt worden sind, folgt nun sukzessive die Auflösung der Entscheidungsknoten. In diesem Schritt wird für jeden Entscheidungsknoten die dort hinterlegte Referenzspezifikation verwendet, um mit dieser auf den Komponentenmärkten nach geeigneten Komponenten zu suchen. Dabei kann sowohl eine Suche auf einem global zugänglichen Komponentenmarktplatz erwogen werden als auch eine Suche in einem unternehmensinternen Komponenten-Repository. Dabei ist es für die Vergleichbarkeit wichtig, dass Komponentenspezifikationen und Referenzspezifikationen dem gleichen Spezifikationsrahmen folgen.

Am Ende dieses Schrittes findet sich unter jedem Entscheidungsknoten eine Menge potenziell einsetzbarer Komponenten. Für das sich später anschließende Verfahren der Entscheidungsfindung ist es wichtig zu postulieren, dass mit dem Suchschritt jeweils alle möglichen Alternativen gefunden und somit berücksichtigt wurden. Daher ist dieser Schritt mit großer Sorgfalt durchzuführen.

4.3 Schritt 3: Sichtung der gefundenen Komponenten

Im dritten Schritt erfolgt für jeden Entscheidungsknoten eine erste Sichtung der gefundenen Alternativen. Hier werden zum Beispiel solche Alternativen entfernt, die die geforderten Satisfizierungsziele (also die K.O.-Kriterien) schon nicht erfüllen können oder solche, die von anderen Komponenten dominiert werden. Eine Komponente wird von einer anderen dominiert, falls diese in allen Attributen, die von der Referenzspezifikation vorgegeben werden, besser geeignet ist (also in jedem relevanten Attribut übertroffen) wird. Dabei kann die dominierende Komponente in den übrigen Attributen ihrer Spezifikation durchaus schlechter abschneiden – da diese laut Referenzspezifikation jedoch für die Entscheidung irrelevant sind, spielt dies im Sichtungs- und Entscheidungsprozess keine Rolle.

4.4 Schritt 4: Auswahl der am besten geeigneten Komponenten

Im vierten Schritt erfolgt dann die eigentliche Auswahl der am Besten geeigneten Komponen-

te aus der Menge der verbliebenen Alternativen. Dabei handelt es sich, wie im Beispiel gezeigt, um ein multiattributives Entscheidungsproblem [EiWe2003:115ff], d.h. es existieren mehrere zu berücksichtigende Ziele, die teilweise konkurrierend sein können. Darüber hinaus handelt es sich um eine Entscheidung unter Sicherheit, da die Alternativen nach der Suche und der sich anschließenden Vorauswahl unabhängig von weiteren Einflüssen sind.

Die Entscheidungsfindung kann auf zweierlei Arten ablaufen: Ist anzunehmen, dass die jeweiligen Entscheidungsknoten unabhängig voneinander sind, d.h. das die jeweils dort gewählten Komponenten keinen Einfluss auf die zu wählenden Komponenten in anderen Entscheidungsknoten haben, so kann jeder Knoten autark entschieden werden. Ist dies (was in der Praxis anzunehmen ist) nicht der Fall, so müssen sämtliche durch die Stückliste ermöglichten Komponentenkombinationen aufgeführt werden und einander zur Entscheidungsfindung gegenübergestellt werden. Dies entspricht im Prinzip auch der Vorgehensweise, die ein Entwickler ohne ein entscheidungsunterstützendes System (etwa durch die Verwendung eines morphologischen Kastens [Over2002a]) wählen könnte.

Bei multiattributiven Entscheidungsproblemen spielt insbesondere die Gewichtung der einzelnen Ziele, beispielsweise in Form von prozentualen Anteilen der einzelnen Ziele, eine entscheidende Rolle. Die Gewichte können dazu benutzt werden, aus den verschiedenen Zielen eine einzelne (gemeinsame) Zielfunktion zu bestimmen, indem eine gewichtete Summe aus den einzelnen Zielkriterien berechnet wird. Vergibt man die Zielgewichte nach Gefühl, wird dies Direct-Ratio-Verfahren genannt. Zum Beispiel könnte ein Entwickler angeben, dass ihm ein niedriger Preis ebenso wichtig ist wie eine schnelle Antwortzeit. In diesem Fall würde er jeweils zu 50% den tatsächlichen Preis bzw. die tatsächliche Antwortzeit in die Zielfunktion einfließen lassen. Da das Direct-Ratio Verfahren einige logische Mängel aufweist, wurden in der Literatur systematischere Verfahren wie das Trade-Off- [EiWe2003:125ff] oder das Swing-Verfahren entwickelt [EiWe2003:129f]. Eine genauere Analyse der verschiedenen Entscheidungstechniken zur Auswahl von Softwarekomponenten findet sich bei [NcDe2002].

Ein weiteres Problem entsteht durch die Bewertung der jeweiligen Alternativen anhand der einzelnen Ziele. Bei Extremalzielen stellt sich die Frage der Bewertung der Alternativen mittels normierter Zahlenwerte, also beispielsweise, ob ein Einkaufspreis von 100€ gegenüber einem Preis von 1000€ oder eine maximale Antwortzeit bei einem Methodenaufruf von 1ms gegenüber einer von 10ms besser ist. Hinzu kommt, dass häufig Angaben mit unterschiedlichen Einheiten sowie unterschiedlich großen Intervallen der möglichen Ausprägungen miteinander verglichen werden müssen.

Kontio [Kont1995] schlägt hierfür einen Bewertungsprozess vor, den er in zwei Fallstudien [KoCB1996], [Kont1996] auf seine Praxistauglichkeit untersucht hat. Zur Veranschaulichung des Prozesses wird hier die aus der Betriebswirtschaftslehre bekannte Nutzwertanalyse verwendet. Hierbei geht es darum, für jede Alternative hinsichtlich eines jeden Ziels einen Nutzwert zu bestimmen. Zum Beispiel würde eine Alternative, die 100€ kostet, gegenüber einer gleichwertigen Lösung zu einem Preis von 1000€ einen höheren Nutzwert erhalten. Zu jeder Alternative wird dann, wie oben erläutert, die gewichtete Summe der Nutzwerte bestimmt und davon die größte ausgewählt. Da die erwähnten Alternativen ansonsten gleichwertig sein sollten, wird ihre Antwortzeit auf denselben Wert von beispielsweise jeweils 7ms gesetzt. Der Nutzwert von 100€ sei beispielsweise 10, der Nutzwert von 1000€ sei 1 und der Nutzwert von einer Antwortzeit von 7ms sei 5. Bei den oben erwähnten 50%-Gewichten der Zielfunktionen ergibt sich also ein Nutzwert von 10 bei der ersten und von 6 bei der zweiten Alternative, d.h. die billigere Alternative würde erwartungsgemäß ausgewählt. Da die Nutzwertanalyse wie

wertanalyse wie das Direct-Ratio-Verfahren einige Nachteile besitzt, hat Kontio den Analytic Hierachy Process (AHP) [Saat1990] eingesetzt, der einige der Nachteile der Nutzwertanalyse wettzumachen versucht.

4.5 Schritt 5: Evaluation der getroffenen Komponentenauswahl

Nach der Ermittlung des Entscheidungsvorschlags im letzten Schritt wird dieser nun noch einmal auf seine tatsächliche Verwendbarkeit in der Praxis der Anwendungsentwicklung evaluiert. Dies ist entspricht sozusagen einer praktischen „Plausibilitätsprüfung“ bzw. Interpretation der Erkenntnisse, die durch ein systematisches Entscheidungsverfahren gewonnen wurden. Bei einer vollautomatischen Entscheidungsfindung mit einem ausgereiften Entscheidungsalgorithmus wird dieser Schritt entfallen.

4.6 Schritt 6: Freigabe der Komponentenauswahl

Nach der Bestätigung der Entscheidung für eine bestimmte Kombination von Komponenten wird diese zur Beschaffung freigegeben und zur Anwendungsentwicklung (Konfiguration) weitergegeben.

4.7 Schritt 7: Ex-Post-Analyse des Entscheidungsprozesses

Der Auswahlprozess endet in einem letzten Schritt mit einer Ex-Post-Analyse der getroffenen Entscheidung, wenn sich absehen lässt, wie gut die gefundene Auswahl war. Die hierbei gewonnenen Erkenntnisse werden verwendet, um die beschriebene Vorgehensweise zu kontrollieren und permanent weiter zu verbessern.

5 Stücklistenbasiertes Änderungsmanagement

Neben einer Unterstützung beim Auswahlmanagement können Stücklisten auch beim Änderungsmanagement sinnvoll eingesetzt werden. Wird beispielsweise eine häufig in der Entwicklung verwendete Komponente durch eine andere ersetzt (sei es eine neue Version, eine fremde Komponente etc.), so lassen sich die nunmehr zu wartenden Anwendungen und Baugruppen durch Nachverfolgen derjenigen Stücklisten ermitteln, die diese Komponente jeweils als Teil beinhalten.

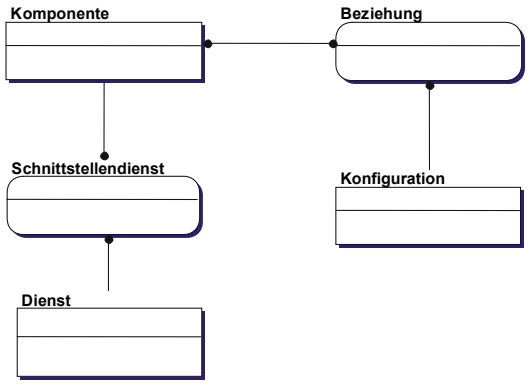

Abbildung 5: Metaschema des Komponentenmodells (basierend auf [ABC+2002]) und angeschlossene Stücklistenorganisation, dargestellt als ER-Diagramm

Ein solcher Verwendungsnachweis einer Komponente ist in der Regel in einem Dokumentationssystem (z.B. einem Repository [Ortn1999]), also einer speziellen Entwicklungsdatenbank, gespeichert. Abbildung 5 zeigt, wie eine Stücklistenorganisation beispielsweise auf das im GI-Spezifikationsrahmen [Turo2002] enthaltene Dokumentationsschema einer Komponente aufgesetzt werden kann. Hierdurch ist es – ausgehend von der betroffenen Komponente – leicht möglich, alle Stücklisten, die diese Komponente enthalten, zu ermitteln und anschließend nach zu wartenden Konfigurationen (Anwendungen oder Baugruppen) auszuwerten.

Über die von Wartungen betroffenen Konfigurationen hinaus vermag eine Stückliste auch anzugeben, wie oft eine Komponente als Teil während der jeweiligen Anwendungsentwicklung verwendet wurde. Hierdurch lässt sich also sicherstellen, dass jede Anwendung vollständig gewartet und an Änderungen angepasst wird. Gleichzeitig dient dieser mengenmäßige Nachweis auch der Schätzung des bei einer Änderung zu erwartenden Wartungsaufwands.

Etwas komplizierter stellt sich die Verwendung von Variantenstücklisten für das Änderungsmanagement dar. In diesem Fall werden zusätzlich zu der eigentlichen Stückliste auch die bei der Komponentenauswahl getroffenen Entscheidungen (als Eingabewerte für die Auswertung der Stückliste) benötigt.

6 Zusammenfassung und Ausblick

Es wurde gezeigt, wie der Einsatz von Stücklisten als Methode im Konfigurationsmanagement den Prozess der komponentenorientierten Anwendungsentwicklung weiter verbessern kann. Insbesondere der Nutzen bei der Komponentenauswahl und im Änderungsmanagement wurde dabei angesprochen.

Durch die Verbindung eines Standards zur Spezifikation von Fachkomponenten, grundlegender Verfahren aus der Entscheidungstheorie sowie Variantenstücklisten wurde ein systematischer (rationaler) Prozess zur Entscheidungsfindung bei der Auswahl von Komponenten vorgestellt, der die bislang üblichen Ad-hoc-Entscheidungen ablösen könnte. Insbesondere die Methoden der Entscheidungstheorie können weiter auf diesen Einsatzzweck angepasst werden. Zum Beispiel könnte die Einführung von Präferenzfunktionen, die auf einem Komponentenmarktplatz gewonnen werden könnten, dazu dienen, den Kunden beim Kauf einer Komponente gleich mitzuteilen, mit welchen anderen Komponenten diese bevorzugt verwendet wird, um somit mögliches Cross-Selling Potential auszuschöpfen.

Auch eine weitergehende Untersuchung der Ebenen des Spezifikationsrahmens sollte dazu führen, den Auswahlprozess weiter zu verbessern. So könnte eine erste Untersuchung mögliche Satisfizierungsziele und Extremalziele unterscheiden. Daneben wäre ein Algorithmus für den Kompatibilitätstest auf der Basis von Komponentenspezifikationen wünschenswert. Dieser ist derzeit in Planung. Eine Herausforderung besteht darin, die verschiedenen Ebenen des Spezifikationsrahmens auf Kompatibilität zu prüfen, beispielsweise also auf der Terminologieebene festzustellen, ob eine Komponente „Bilanzierung nach US-GAAP“ kompatibel zu einer Komponente „Bilanzierung nach IAS“ ist. Schließlich wäre es denkbar, auf dem Spezifikationsrahmen eine Zielhierarchie aufzubauen, die in vielen Entscheidungsmethoden (z.B. dem AHP) verwendet werden kann. Diese theoretischen Untersuchungen am Spezifikationsrahmen sollten dabei durch den Versuch ergänzt werden, das Konzept auch als Ganzes in einem CASE-Tool zu implementieren. Insbesondere der Einsatz von Variantenstücklisten im Anwendungsentwurf muss dabei durch entsprechende Erfahrungen in der Praxis noch weiter untersucht werden.

Literatur

[ABC+2002] *Ackermann, J.; Brinkop, F.; Conrad, S.; Fettke, P.; Frick, A.; Glistau, E.; Jaekel, H.; Kotlar, O.; Loos, P.; Mrech, H.; Ortner, E.; Overhage, S.; Raape, U.; Sahm, S.; Schmietendorf, A.; Teschke, T.; Turowski, K.:* Vereinheitlichte Spezifikation von Fachkomponenten. Gesellschaft für Informatik (GI), Arbeitskreis 5.10.3, Augsburg, 2002. http://www.fachkomponenten.de, Abruf am 20.11.2002.

[EiWe2003] *Eisenführ, F.; Weber, M.:* Rationales Entscheiden. 4. Auflage, Springer Verlag, Berlin 2003.

[Hahn2000] *Hahn, H.:* Ein Modell zur Ermittlung der Reife des Softwaremarktes. In: *Turowski, K. (Hrsg.):* Tagungsband 4. Workshop Komponentenorientierte betriebliche Anwendungssysteme. Universität Augsburg, Juni 2002, S. 75 – 85.

[HoNo2001] *Houston, K.; Norris, D.:* Software Components and the UML. In: *Heineman, G. T.; Councill, W. T. (Hrsg.):* Component-Based Software Engineering – Putting the Pieces Together. Addison-Wesley, Upper Saddle River, 2001, S. 243 – 262.

[KoCB1996] *Kontio, J.; Caldiera, G.; Basili, V.R.:* Defining Factors, Goals and Criteria for Reusable Component Evaluation, Presented at the CASCON '96 conference, Toronto, Canada, 1996.

[KoHe1996] *Kolisch, R.; Hempel, K.:* Auswahl von Standardsoftware, dargestellt am Beispiel von Programmen für das Projektmanagement. In: Wirtschaftsinformatik 38 (1996) 4, S. 399 – 410.

[Kont1995] *Kontio, J.:* OTSO: A Systematic Process for Reusable Software Component Selection, Institute for Advanced Computer Studies and Department of Computer Science, University of Maryland, Collage Park, USA. http://www.soberit.hut.fi/~jkontio/jko_pub.htm, Abruf am 22.11.2002.

[Kont1996] *Kontio, J.:* A Case Study in Applying a Systematic Method for COTS Selection, Proceedings: 18th International Conference on Software Engineering in Berlin, IEEE, 1996.

[LaCr2000] *Larsson, M.; Crnkovic, I.:* Component Configuration Management. In: *Szyperski, C. (Hrsg.):* Proceedings of WCOP 2000. http://www.mrtc.mdh.se/publication/0236.pdf, Abruf am 22.11.2002.

[Mcil1968] *McIlroy, M. D.:* Mass Produced Software Components. In: *Naur, P., Randell, B. (Hrsg.):* Software Engineering: Report on a Conference by the NATO Science Committee. NATO Scientific Affairs Division, Brussels, 1968, S. 138–150.

[NcDe2002] *Ncube, C.; Dean, J. C.:* The Limitations of Current Decision-Making Techniques in the Procurement of COTS Software Components. In: *Dean, J. C.; Gravel, A. (Hrsg.):* COTS-Based Software Systems, ICCBSS 2002, Lecture Notes in Computer Science 2255, Springer 2002, S. 176 – 187

[Ortn1998] *Ortner, E.:* Ein Multipfad-Vorgehensmodell für die Entwicklung von Informationssystemen – dargestellt am Beispiel von Workflow-Management-Anwendungen. In: Wirtschaftsinformatik 40 (1998) 4, S. 329 – 337.

[OrLK1999] *Ortner, E.; Lang, K.-P.; Kalkmann, J.:* Anwendungsentwicklung mit Komponenten. In: Information Management & Consulting 14 (1999) 2, S. 35 – 45.

[Ortn1999] *Ortner, E.:* Repository Systems. Teil 1: Mehrstufigkeit und Entwicklungsumgebung. In: Informatik Spektrum 22 (1999) 4, S. 235 – 251. Teil 2: Aufbau und Betrieb eines Entwicklungsrepositoriums. In: Informatik Spektrum 22 (1999) 5, S. 351 – 363.

[Ortn2001] *Ortner, E.:* Komponentenorientierte Anwendungsentwicklung. In: Management & Consulting 15 (2001) 4, S. 62 – 72.

[Over2002a] *Overhage, S.:* Die Spezifikation – kritischer Erfolgsfaktor der Komponentenorientierung. In: *Turowski, K. (Hrsg.):* Tagungsband 4. Workshop Komponentenorientierte betriebliche Anwendungssysteme. Universität Augsburg, Juni 2002, S. 1 – 17.

[Over2002b] *Overhage, S.:* Komponentenkataloge auf Basis eines einheitlichen Spezifikationsrahmens – ein Implementierungsbericht. In: *Turowski, K. (Hrsg.):* Tagungsband 3. Workshop Modellierung

und Spezifikation von Fachkomponenten. Universität Augsburg, September 2002, S. 1 – 16.

[OvTh2002] *Overhage, S.; Thomas, P.:* On Specifying Web Services Using UDDI Improvements. In: *Dittmar, T. et. al (Hrsg.):* Proceedings Netobjectdays Workshops 2002. IGT, Leipzig, 2002, S. 535 – 550.

[Saat1990] *Saaty, T. L.:* The Analytic Hierarchy Process. McGraw-Hill, New York, 1990.

[Schn1999] *Schneeweiß, C.:* Einführung in die Produktionswirtschaft. 7. Auflage, Springer Verlag, Berlin 1999.

[ShGa1996] *Shaw, M., Garlan, D.:* Software Architecture – Perspectives on an Emerging Discipline. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[Szyp1998] *Szyperski, C.:* Component Software: Beyond Object-Oriented Programming. Addison-Wesley, Harlow 1998.

[WeMü1981] *Wedekind, H.; Müller, T.:* Stücklistenorganisation bei einer großen Variantenzahl. In: Angewandte Informatik 23 (1981) 9, S. 377 – 383.

[Zimm1988] *Zimmermann, G.:* Produktionsplanung variantenreicher Erzeugnisse mit EDV. Springer Verlag, Berlin 1988.

Ein Komponenten-Framework für die situationsabhängige Adaption Web-Service-basierter Standardsoftware

Michael Amberg, Shota Okujava, Jens Wehrmann

Lehrstuhl für Wirtschaftsinformatik III, Friedrich-Alexander-Universität Erlangen-Nürnberg, Lange Gasse 20, 90403 Nürnberg, Deutschland, Tel.: +49(911)580796 - 40, Fax: - 42, amberg@wiso.uni-erlangen.de, shota.okujava@wiso.uni-erlangen.de, wehrmann@wiso.uni-erlangen.de, URL: http://www.wi3.uni-erlangen.de

Zusammenfassung: Die Intelligenz hinter einer Anwendung und das Verständnis für die Intensionen der Anwender wird für betrieblicher Standardsoftware immer wichtiger. Für die Neu- und Weiterentwicklung verteilter Standardsoftware werden Ansätze benötigt, die situationsabhängig eine selbstständige Anpassung der Software an die speziellen Bedürfnisse der Anwender und ihrer Tätigkeiten unterstützen.

In dieser Arbeit wird zunächst der Gesamtkontext adaptiver Software für betriebliche Standardsoftware diskutiert, bevor ein Komponenten-Framework für die situationsabhängige Adaption Web-Service-basierter Standardsoftware vorgestellt wird. Die wesentlichen Bestandteile des Komponenten-Frameworks sind die Grundkonzeption der situationsabhängigen Adaption mit Web-Services, die dazu erforderlichen Komponenten zur Protokollierung, die zentralen Komponenten des Adaptionsprozesses, sowie eine Gesamtarchitektur, die das Zusammenspiel der verwendeten Komponenten beschreibt.

Schlüsselworte: Web-Services, Situationsabhängigkeit, Situationsabhängige Protokollierung und Adaption, Adaptive Software, Komponenten-Framework, Standardsoftware, Pull- und Push-Services

1 Motivation

Die Entwicklung der Informationstechnologien ist rasant. Die technischen Möglichkeiten sind kaum noch ein limitierender Faktor in der Softwareentwicklung. Für zukünftige Softwareentwicklungen, insbesondere im Bereich betrieblicher Standardsoftware, wird die Intelligenz hinter einer Anwendung und das Verständnis für die Intensionen der Anwender ein immer bedeutenderer Wettbewerbsfaktor.

Eine Möglichkeit, adaptive bzw. intelligente Software zu realisieren, besteht in der Analyse des Nutzungsverhaltens und in der Speicherung bzw. selbständigen Verwendung von Nutzungsinformationen [OrGD1999]. Dies wird im folgenden als situationsabhängige Adaption bezeichnet. Nutzungsinformationen werden beispielsweise als Nutzungsprofile und Protokolldaten gespeichert. Diese können teilweise sehr detaillierte Informationen über das Verhalten der Anwender bei der Nutzung einer Software beinhalten. Mit Hilfe derartiger Informationen ist es möglich, Situationen zu erkennen und aus den erkannten Situationen

Informationen abzuleiten, die einen Mehrwert sowohl für den Anwender als auch für den Anbieter einer Standardsoftware generieren.

Die Verwendung von Nutzungsinformationen bei Internetanwendungen und mobilen Anwendungen kann bereits als gebräuchlich bezeichnet werden, jedoch fehlen unmittelbar anwendbare Standards sowie standardisierte Softwarekomponenten, die eine effiziente Entwicklung entsprechend adaptiver Software unterstützen. In dieser Arbeit soll am Beispiel Web-Service-basierter Standardsoftware ein Beitrag zur Beantwortung folgender Fragestellungen geleistet werden:

- Welche Arten von situationsabhängiger Adaption lassen sich bei Web-Service-basierter Standardsoftware im Wesentlichen unterscheiden? Welche Vorteile und Auswirkungen sind damit verknüpft?
- Wie sieht ein Komponenten-Framework aus, welches die situationsabhängige Protokollierung und Adaption von Web-Service-basierter Standardsoftware unterstützt? Welche Arten von situationsabhängiger Adaption lassen sich mit dem aufgezeigten Komponenten-Framework wie unterstützen?

Der hier vorgestellte Beitrag wurde im Umfeld eines durch die Bayerische Staatsregierung geförderten Projektes erarbeitet. Im Projektkontext plant ein KMU-Unternehmen als Softwarehersteller die Neuentwicklung einer Controllingsoftware als betriebliche Standardsoftware unter Nutzung von Web-Services. Die Software soll beispielsweise Energieversorgungsunternehmen angeboten werden, wo ca. zehn bis hundert Mitarbeiter des Unternehmens die Software in unterschiedlichen Rollen nutzen werden. Um sich von Wettbewerbern abzuheben, soll eine situationsabhängige Adaption in der Art integriert werden, dass unter Zuhilfenahme von Komponenten-Framework und Standardkomponenten die Adaption möglichst flexibel und in Bezug zur Kernsoftware weitgehend lose gekoppelt ist. An dieser Stelle sei angemerkt, dass die Ausführungen im Grundsatz auch auf die Entwicklung von Individualsoftware mit Multi-Tier-Architektur übertragen werden können.

2 Situationsabhängige Protokollierung und Adaption in Web-Service-basierter Standardsoftware

Im Folgenden wird zunächst der Gesamtkontext aufgezeigt und darauf eingegangen, was unter situationsabhängiger Adaption zu verstehen ist. In Kapitel 2.1 wird skizziert, welche Ziele bei der Einführung situationsabhängiger Adaption in betrieblicher Standardsoftware verfolgt werden. Kapitel 2.2 setzt sich mit der Entwicklung Web-Service-basierter Standardsoftware auseinander.

Der Gesamtanwendungskontext der situationsabhängigen Adaption betrieblicher Standardsoftware ist in Bild 1 vereinfacht dargestellt. Ein Softwarehersteller stellt einem Unternehmen eine vorgefertigte betriebliche Standardsoftware bereit und führt die Weiterentwicklung und Pflege der Software durch. Das Unternehmen (auch als Software-Lizenznehmer bezeichnet) ist u.a. für die unternehmensspezifische Anpassung, die Systemeinführung und den Systembetrieb verantwortlich. Die Software wird im Allgemeinen von mehreren Mitarbeitern und sonstigen Anwendern (z.B. Kunden) verwendet, wobei verschiedene Anwender in der Regel unterschiedliche Softwarefunktionalitäten nutzen. Als Interessensgruppen lassen sich auch unabhängig von der konkreten Software die Softwarehersteller bzw. Entwickler, sowie (End-) Anwender, Administratoren und das (Unternehmens-) Management unterscheiden.

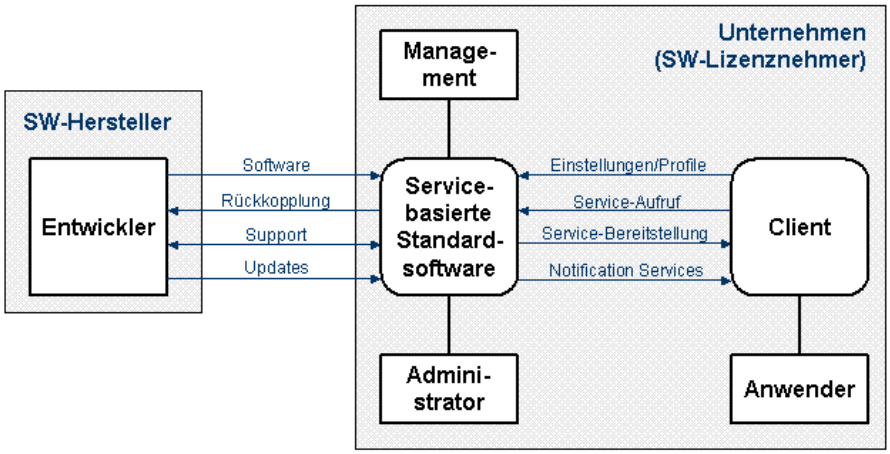

Bild 1 - Gesamtanwendungskontext der situationsabhängigen Adaption betrieblicher Standardsoftware

In diesem Beitrag wird von der Annahme ausgegangen, dass die Anpassung der Funktionalität einer betrieblichen Standardsoftware an die aktuelle Nutzungssituation eines Anwenders einen konkreten Mehrwert darstellt. Dabei wird unterstellt, dass die Problemstellung des Anwenders mit dieser Nutzungssituation korrespondiert. Eine Software, die Informationen über die Situation nutzt, kann sehr viel besser zur Problemlösung beitragen als dies ohne Situationsinformationen möglich wäre. Dabei werden Informationen verarbeitet, die auch für ergänzende Aufgaben und weitere Interessensgruppen hilfreich sind. Art und Umfang einer situationsabhängigen Adaption sind maßgeblich davon abhängig, wie der Situationsbegriff systematisiert und konkretisiert wird.

Ursprünglich entstammt die Diskussion über den Situationsbegriff aus dem Umfeld der mobilen Dienste. In der Literatur finden sich unterschiedliche Ansätze, die den Situationsbegriff bearbeiten und jeweils eigenständig interpretieren (z.B. [Gase2001}, [May2001], [Zobe2001], [AmFW2002] und [AmWe2002]). Hitz et al. [HiKG2002] betrachten in Ihrer Arbeit darüber hinaus explizit die Modellierung von ubiquitären Web-Anwendungen. Sie nehmen die Unterscheidung zwischen *natürlichem* (Ort, Zeit), *technischem* (Endgerät, Browser, Netzwerk, Status) und *sozialem Kontext* (Benutzerprofil und -verhalten) vor. Diese Differenzierung ist auch für betriebliche Standardsoftware geeignet.

Unter dem Begriff der situationsabhängigen Adaption wird die Anpassung einer Softwarefunktionalität an die aktuelle Nutzungssituation verstanden. Dabei sollen die Softwarekomponenten im Sinne einer intelligenten Software in der Lage sein, sich selbstständig an die speziellen Bedürfnisse der Anwender anzupassen und die Anwender in ihren Tätigkeiten aktiv zu unterstützen.

2.1 Ziele der situationsabhängigen Protokollierung und Adaption

Das übergeordnete Ziel für die Einführung der situationsabhängigen Adaption ist es, durch Ausrichtung bzw. Anpassung der Softwarefunktionalität an die spezifischen Bedürfnisse der Anwender einen erkennbaren Mehrwert zu schaffen, die Nutzungshäufigkeit der Software zu steuern sowie die Effektivität der Nutzung zu steigern. Hierzu muss die Software das Nutzungsverhalten protokollieren, wiederholt analysieren und Nutzungsinformationen speichern bzw. Nutzungsprofile anpassen und verwalten.

Über die **inhaltliche Anpassung der Softwarefunktionalität** auf die Bedürfnisse der Anwender hinaus bietet der Umgang mit derartigen Nutzungsinformationen in betrieblicher Standardsoftware die Gelegenheit, eine oder mehrere der folgenden Zielsetzungen zu realisieren:

Gestaltung adaptiver Nutzerschnittstellen
Mit Gestaltung adaptiver Nutzerschnittstellen wird hier einerseits die Bereitstellung der Softwarefunktionalität in einer für den Anwender intuitiven Art und Weise sowie andererseits das situationsabhängige Anbieten von Hilfestellungen und Zusatzinformationen bezeichnet. Ersteres umfasst beispielsweise Anpassungen der Menüs, eine Reduktion auf anwendbare Funktionen oder die Bereitstellung individualisierter Zusatzfunktionen. Letzteres umfasst beispielsweise sogenannte Notification-Services, um Anwender Hinweise auf anstehende Tätigkeiten bzw. geänderte Datenbestände zu geben und Verbesserungsvorschläge für ein effektives Arbeiten zu unterbreiten. Die benötigten Funktionalitäten sollen so effizienter zur Verfügung gestellt werden und ein intuitiveres Benutzen der Software ermöglichen.

Nutzungsabhängige Leistungsverteilung und Performanzmanagement
Die Protokollierung des Nutzungsverhaltens ermöglicht die Messung und Analyse der nutzungsabhängigen Systemlast. Entsprechende Ergebnisse erlauben es der Software, nutzungsspezifisch (Zwischen-) Ergebnisse gegebenenfalls im Vorfeld einer Anfrage zu berechnen und zwischen zu speichern. Dies kann Engpasssituationen vermeiden und helfen, das Systemverhalten zu verbessern.

Nutzungsabhängige interne Leistungsverrechnung
Die Protokollierung des Nutzungsverhaltens kann weiterhin Grundlage einer internen Leistungsverrechnung werden. Es kann festgestellt werden, welcher Mitarbeiter was, wann, wie oft und insbesondere in welcher Art und Weise einsetzt. Dies kann sowohl für die nutzungsabhängige Aufteilung auf innerbetriebliche Kostenstellen als auch für die Auswertung zu Controllingzwecken verwendet werden. Derartige Informationen helfen weiterhin, die Aktualität der Datenbestände zu sichern sowie den Bedarf für Schulungsmaßnahmen und Anreizsysteme zur Steigerung der Softwarenutzung zu identifizieren.

Nutzungsabhängige Softwareanalyse, –wartung und –pflege
Die Nutzungsinformationen einer Software stellen für den Hersteller der Standardsoftware eine wertvolle Informationsquelle dar. So erlauben entsprechende Analysen (Welche Funktionen der Software werden wie und wie häufig genutzt?) eine Bewertung der Nützlichkeit von Softwarefunktionen und liefern Hinweise für zu ergänzende oder zu überarbeitende Funktionen. Eine Schnittstelle für den regelmäßigen Austausch von Nutzungsinformationen zwischen Anbieter und Nutzer schafft ein stetiges Verbesserungspotenzial, damit die Software permanent weiterentwickelt und ohne lange Versionszyklen verbessert werden kann. Dies ist insbesondere bei der Erstentwicklung einer Standardsoftware und bei Pilotanwendungen hilfreich.

Erweitertes Application Service Providing (ASP)
Im Zusammenhang mit der Bereitstellung der Software als Application Service können Nutzungsinformationen hilfreich sein, um leistungsabhängige Lizenzierungsverfahren zu realisieren. Diese werden nicht wie herkömmliche Lizenzen einmalig oder periodisch sondern nach dem Grad der Softwarenutzung abgerechnet. Zudem kann die nutzungsabhängige Leistungsverteilung und Performanzmanagement berücksichtigt und die situationsabhängige Betreuung der Anwender verbessert werden.

Diese Aufzählung verdeutlicht, dass die situationsabhängige Protokollierung und Adaption bei betrieblicher Standardsoftware einen erkennbaren Mehrwert für die beteiligten Interessensgruppen bieten. In **Bild 2** wird der Zusammenhang zwischen Interessensgruppen und Zielsetzungen der situationsabhängigen Protokollierung und Adaption dargestellt.

Kategorien / Inter-essengruppen	Adaptive Nutzerschnitt-stellen	Leistungsvert. & Performance-management	Interne (Leistungs-) Verrechnung	Analyse, Wartung & Pflege	Erweitertes ASP
Software-Anbieter				X	X
Management			X		X
Administrator		X		X	
Endanwender	X	X			

Bild 2 – Übersicht der Interessensgruppen einer situationsabhängigen Protokollierung und Adaption

Kernproblem der situationsabhängigen Adaption ist der Umgang mit den Nutzungs- und Personendaten. Es sind die Aspekte des Datenschutzes zu beachten und eine Überwachung der Mitarbeiter zu verhindern. Gegebenenfalls kann durch Verschlüsselung und Verwendung von Pseudonymen sichergestellt werden, dass derartige Informationen nur in einer Art und Weise an Personen weitergegeben werden, dass diese nicht auf konkrete Anwender rückschließen können. Auf diese Problematik sowie mögliche Lösungsansätze soll in diesem Beitrag nicht weiter eingegangen werden.

Die Datenschutzproblematik wird hier dadurch abgeschwächt, dass im Folgenden die Kategorien von situationsabhängiger Adaption weiter betrachtet werden, die sich auf die Nutzergruppe *Endanwender* fokussieren und insbesondere automatisiert verarbeitet werden. Bei einer automatisierten Verarbeitung können Informationen nicht ohne weiteres von Dritten eingesehen werden. Für die automatisierte Verarbeitung in Web-Service-basierter Standardsoftware sind insbesondere die folgenden Subkategorien von Interesse:

- ***Adaptive Pull-Services:*** Ein Serviceaufruf eines Anwenders wird interpretiert und das Ergebnis entsprechend der Nutzungsinformationen modifiziert. Der Anwender erhält eine auf seinen Nutzungskontext angepasste Softwarefunktionalität.
- ***Adaptive Push-Services:*** Ein Service wird von sich aus aktiv und informiert den Anwender geeignet. Es können Hinweise auf anstehende Tätigkeiten, über geänderte Datenbestände oder Verbesserungsvorschläge für ein effizienteres Arbeiten gegeben werden. Diese Form von Services wird im Folgenden als Notification-Service bezeichnet.
- ***Adaptive Nutzeroberfläche (GUI):*** Die Ergebnisse eines Serviceaufrufes werden hinsichtlich der individuellen Bedürfnisse (individuelle Vorlieben, technische Möglichkeiten des Endgerätes) in der Darstellung angepasst. Daraus können auch Anpassungen der Menüführung resultieren (Reduktion auf anwendbare Funktionen oder die Bereitstellung individualisierter Zusatzfunktionen).

Im Folgenden soll näher darauf eingegangen werden, wie die situationsabhängige Adaption bei der Entwicklung Web-Service-basierter Standardsoftware berücksichtigt werden kann.

2.2 Entwicklungsaspekte Web-Service-basierter Standardsoftware

Insbesondere bei der Neu- und Weiterentwicklung betrieblicher Standardsoftware besitzt die komponentenbasierte Softwareentwicklung gegenüber der konventionellen Entwicklung zahlreiche Vorteile, wie beispielsweise geringere Kosten und die Möglichkeit, die Software adäquat und wirtschaftlich an die individuellen Bedürfnisse der Unternehmen anzupassen [OrHE1996]. Darüber hinaus stellt gerade die Wiederverwendung von Komponenten für die Hersteller von Standardsoftware eine Möglichkeit dar, die zunehmende Komplexität ihrer Anwendungssysteme zu beherrschen [Turo1999].

Web-Service-basierte Software wird als eine besondere Klasse komponentenbasierter Software aufgefasst. Web-Services unterteilen eine Software in grob granulare Softwarekomponenten, die eher lose miteinander gekoppelt sind. Softwarehersteller erwarten bei der Entwicklung verteilter Anwendungen Vorteile durch die anwendungsnahe Spezifikation von Softwarekomponenten, die sich an den Geschäftsprozessen orientieren. Die Funktionen eines Geschäftsprozesses werden eins zu eins auf Web-Services abgebildet und können über mehrere Rechner verteilt werden. Durch die Unabhängigkeit von Plattformen, Betriebssystemen und Programmiersprachen und die Nutzung von etablierten Web-Standards kann eine Software auch über die Unternehmensgrenzen hinaus verwendet werden.

<table>
<tr><th>Kategorien / Granularität</th><th>Adaptive Pull-Services</th><th>Adaptive Push-Services</th><th>Adaptive Nutzer-oberfläche</th><th>Leistungsvert. & Performance-management</th><th>Interne (Leistungs-) Verrechnung</th><th>Analyse, Wartung & Pflege</th><th>Erweitertes ASP</th></tr>
<tr><td>Web-Service Request</td><td>Anpassung bezügl. Person, Zeit und Ort</td><td rowspan="3">Versenden einer Information (Notification Service), falls ein Wert einen definierten Bereich verlässt</td><td>Berücksichtigung von Userprofilen</td><td>Vorberechnung aller möglicher Anfragen</td><td>Separate Abrechnung beim Abruf recheninten-siver Berichte</td><td rowspan="2">Fehlermeldung, Exeption Management, Feedback bei Fehlern, Optimierung des Services im Zeitverlauf</td><td rowspan="2">Nutzungs-abhängige Abrechnung</td></tr>
<tr><td>Web-Service</td><td rowspan="2">Wertehistory, Rezensionen, Empfehlungen, Notizen, Tips, Kommentare</td><td rowspan="2">Anpassung der Menüs, letzte Funktionen, Favoriten (clientseitig)</td><td rowspan="2">Individuelle kontextab-hängige Vor-berechnung (mittels Muster-erkennung) oder Caching von lokalen Informationen</td><td>Interne Vergabe von Rechten für die Benutzung spezifischer Dienste</td></tr>
<tr><td>Teil-Applikation</td><td rowspan="2">Abrechnung nach Benutzer-gruppen (die auf spezifische Komponenten zugreifen) und Verteilung der Kosten auf entsprechende Kostenstellen</td><td rowspan="2">Benutzungs-statistiken, Prioritätenma-nagement für die Weiter-entwicklung</td></tr>
<tr><td>Gesamt-Applikation</td><td>Integration von Inhalten, Daten und Funktionen</td><td>Anpassung der Default-GUI-Werte, Favoriten (serverseitig)</td><td>Lastverteilung, Verdichtungs-operationen</td><td>Wahl des optimalen Abrechnungs-modells</td></tr>
</table>

Bild 3 - Beispiele situationsabhängiger Adaption in Web-Service-basierter Standardsoftware

Aus Anwendungssicht stellt eine Web-Service-basierte (Standard-) Software die aufrufbare Softwarefunktionalität in Form von Web-Services bereit. Inhaltlich zusammenhängende Web-Services werden gegebenenfalls mehrstufig zu größeren Softwarekomponenten zusammengefasst. Die sich ergebende Granularität von Softwarekomponenten kann für die

Identifikation und Systematisierung der unterschiedlichen Arten einer situationsabhängigen Adaption hilfreich sein.

In **Bild 3** werden Beispiele für unterschiedliche Arten einer situationsabhängigen Adaption in Web-Service-basierter Standardsoftware aufgezeigt. Zur Strukturierung werden einerseits die in Kapitel 2.1 aufgezeigten Kategorien und andererseits Granularitätsstufen verwendet. Es werden hier vereinfacht vier Granularitätsstufen unterschieden: Web-Serviceaufruf (Web-Service Request), Web-Service, Softwarekomponente (Teil-Applikation) und Gesamtsoftware (Gesamt-Applikation). Die Abbildung zeigt am Beispiel Controllingsoftware auf, dass für eine situationsabhängige Adaption ein sehr breites Anwendungsspektrum existiert.

3 Ein Komponenten-Framework für die situationsabhängige Adaption

In diesem Kapitel wird ein Komponenten-Framework vorgestellt, welches auf eine umfassende Unterstützung der situationsabhängigen Adaption in Web-Service-basierter Standardsoftware ausgerichtet ist. Das Komponenten-Framework soll folgenden Anforderungen gerecht werden:

- **Lose Kopplung zwischen Adaption und Kernsoftware:** Adaption und Kernsoftware sollen möglichst unabhängig voneinander und nur lose miteinander gekoppelt sein. Inhalt und Umfang der Adaption sollen sich zur Laufzeit flexibel an veränderte Anforderungen anpassen können.
- **Flexibilität hinsichtlich Adaption:** Inhalt und Umfang der situationsabhängigen Adaption soll ausschließlich durch die Anforderungen der Kernsoftware und nicht durch die Implementierung des Komponenten-Frameworks festgelegt werden.
- **Gewährleistung hoher Performanz:** Einbußen in der Performanz durch die Adaption sollen weitestgehend minimiert bzw. vermieden werden. Eine direkte Kommunikation sowie eine gegebenenfalls parallelisierte Bearbeitung werden bevorzugt.
- **Realisierung mit Standardkomponenten:** Der Adaptionsprozess soll weitgehend automatisiert und standardisiert werden. Der Aufwand für die softwareindividuelle Adaption ist weitgehend zu reduzieren.
- **Orientierung an offenen Standards:** Aufbauend auf der Entwicklungsplattform Java 2 Enterprise Edition (J2EE) und Web-Services sollen möglichst offene Standards (z.B. XML, XSLT) und Open-Source-Komponenten (z.B. JAXP) eingesetzt werden.

Im Folgenden werden wesentliche Bestandteile des Komponenten-Framework vorgestellt. Zunächst wird die Grundkonzeption der situationsabhängigen Adaption mit Web-Services aufgezeigt (Kapitel 4). Anschließend werden in Kapitel 5 zentrale Komponenten des Adaptionsprozesses eingeführt. Die Erweiterungen werden in eine Gesamtarchitektur integriert und das Zusammenspiel der verwendeten Komponenten beschrieben (Kapitel 6).

4 Konzeption der situationsabhängigen Protokollierung und Adaption mit Web-Services

Die Grundkonzeption der situationsabhängigen Adaption mit Web-Services zielt auf eine lose Kopplung zwischen Adaptions- und sonstiger Kernsoftwarefunktionalität ab, um so eine nachträgliche Berücksichtigung bzw. Erweiterung der situationsabhängigen Protokollierung und Adaption zu unterstützen.

Die Grundkonzeption ist in **Bild 4** dargestellt. Es wird in diesem Ansatz davon ausgegangen, dass eine Client-Komponente bei einem Serviceaufruf keine Kenntnisse über Art und Umfang der Adaption hat bzw. erhält. Die aufgerufene Server-Komponente benötigt derartige Informationen lediglich, wenn eine inhaltliche Anpassung der Softwarefunktionalität erfolgen soll. Jede andere Form der Adaption erfolgt durch die Analyse, Interpretation und gegebenenfalls Modifikation der Kommunikationsnachrichten durch eine Standardkomponente für die situationsabhängige Adaption (SDA-Komponente). Die SDA-Komponente besteht aus einer XSLT-Engine für die Transformation von XML-Nachrichten mit Stylesheets und einem Stylesheet-Generator, der die Stylesheets abhängig vom Kontext und Situation generiert.

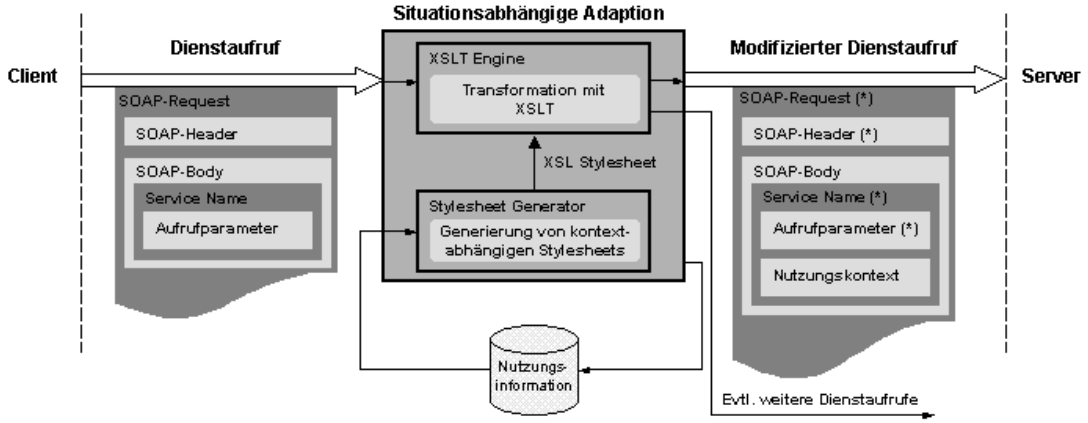

Bild 4 - Grundkonzept der situationsabhängigen Protokollierung und Adaption mit Web-Services

Der grundsätzliche Ablauf lässt sich wie folgt beschreiben: Ein Serviceaufruf (SOAP-Request) wird von der SDA-Komponente entgegengenommen, mit Hilfe einer Standard-XML-Transformation modifiziert und gegebenenfalls um Informationen zum relevanten Nutzungskontext ergänzt. Im modifizierten Serviceaufruf können sowohl der Header (z.B. Ergänzung von Metadaten für den Adaptionsprozess), der Servicename (z.B. Nutzung eines anderen Services) als auch die Aufrufparameter (z.B. Erweiterung des Aufrufs um den Nutzungskontext) verändert worden sein. Das für den Transformationsprozess notwendige Stylesheet wird unter Nutzung eines Stylesheet-Generators gegebenenfalls dynamisch aus gespeicherten Nutzungsinformationen generiert. Der Transformationsprozess wird zudem verwendet, um Informationen über das Nutzungsverhalten abzuleiten bzw. weitere Serviceaufrufe für weiterführende bzw. ergänzende Adaptionen zu generieren.

Die gleichen Adaptionsschritte können auch auf Ebene der Aufrufergebnisse (SOAP-Response) durchgeführt werden. Die Ergebnisse eines Aufrufs werden entsprechend des Kontextes und der Situation modifiziert bzw. erweitert. Der beschriebene Adaptionsprozess kann neben der Transformation genutzt werden, um das Nutzungsverhalten zu protokollieren oder sonstige Aktionen zu initiieren. So ist beispielsweise beim Eintreten eines vorgegebenen Nutzungsereignisses eine Benachrichtigung von Endanwendern über die Nutzeroberfläche, per Email oder SMS möglich.

5 Komponenten des situationsabhängigen Adaptionsprozesses

Für den situationsabhängigen Adaptionsprozess lassen sich folgende Gruppen von Komponenten unterscheiden:

- **Komponenten der Kernsoftware:** Umfasst die Komponenten der Kernsoftware unterteilt in Client-Komponenten und Server-Komponenten inklusive Web-Service-Interfaces.
- **Softwareindividuelle Adaptionskomponenten:** Umfasst die Komponenten, die eine situationsabhängige Adaption softwareindividuell realisieren. Sie können in serverseitige sowie clientseitige Adaptionskomponenten gegliedert werden.
- **Standard-Adaptionskomponenten:** Umfasst die Standardkomponenten zur Adaption, die unmittelbar auf den situationsabhängigen Adaptionsprozess ausgerichtet sind und unabhängig von einer konkreten Software eingesetzt werden können.

Im folgenden wird insbesondere auf die dritte Komponentengruppe fokussiert und das Zusammenspiel der Komponenten aus dem Blickwinkel der Standardkomponenten betrachtet. Die zentralen Designcharakteristika für das Zusammenspiel der benötigten Komponenten sind:

- **Mehrstufiger Adaptionsprozess:** Für die umfassende Unterstützung der situationsabhängigen Adaption eines Serviceaufrufes wird ein mehrstufiger Bearbeitungsprozess zugrundegelegt. Die Bearbeitungsschritte können beispielsweise nach der Zielsetzung der Adaption oder dem Bearbeitungsaufwand differenziert werden. Die Bearbeitung erfolgt zunächst auf dem Client (z.B. für die Ergänzung des Nutzungskontextes und für eine clientseitige Adaption), anschließend durch Standardkomponenten, die allgemeine Adaptionsfunktionalität bereitstellen (z.B. für eine serviceunabhängige Adaption) und abschließend auf dem Server (z.B. für eine serverseitige Adaption).
- **Performanz und Lastverteilung:** Um der Entstehung von Systemengpässen vorzubeugen, sollen die Anzahl der Bearbeitungskomponenten sowie die Ablauffolge variabel gehandhabt werden. Abhängig vom konkreten Serviceaufruf und allgemeinen Adaptionseinstellungen sollen lediglich die benötigten Bearbeitungskomponenten angesprochen werden (Variable Kommunikationsrouten). Wenn möglich, werden voneinander unabhängige Bearbeitungskomponenten parallel aufgerufen (Autonome Bearbeitung). Informationen (z.B. Protokollierungsinformationen), die für den Echtzeitbetrieb nicht zwingend erforderlich sind, werden asynchron mit Messages übertragen. Hierzu werden Komponenten für die Administration benötigt.

Das resultierende Komponentendesign ist in Bild 5 dargestellt. Es basiert auf den Standards Java 2 Enterprise Edition (J2EE) und Web-Services und erweitert die Architektur für Web-Service-basierte Software um Komponenten für die situationsabhängige Adaption.

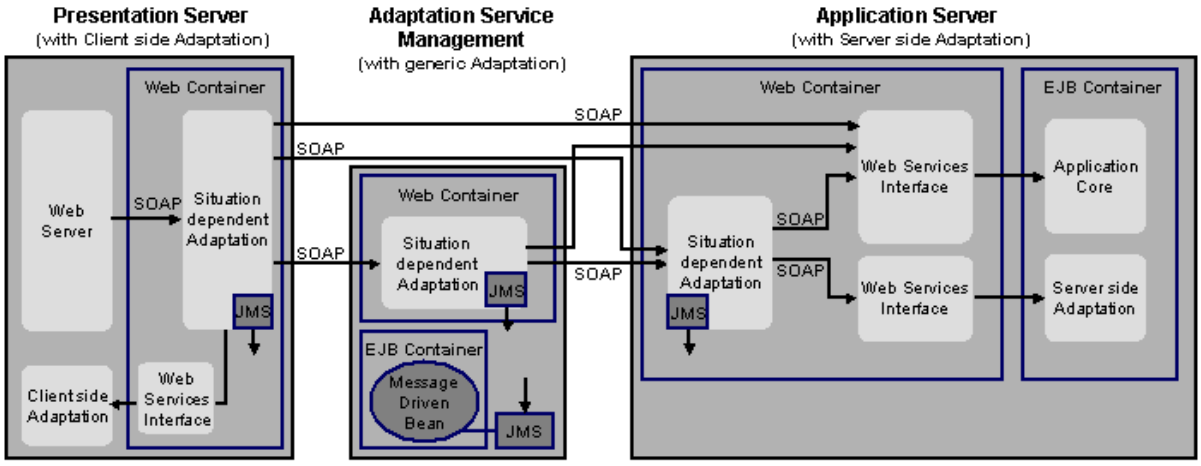

Bild 5 - Um Komponenten für die situationsabhängige Adaption erweiterte Multi-Tier-Architektur

Eine Web-Service-basierte Standardsoftware ist in der Regel als Multi-Tier-Architektur realisiert, die im Allgemeinen vier grundlegende Komponenten unterscheidet: (Thin-) Client, Presentation-Server, Application-Server und Datenbank-Server. Für die situationsabhängige Protokollierung und Adaption sind insbesondere die in der Abbildung dargestellten Komponenten von Interesse, die um einen weiteren Server ergänzt werden.

Der **Präsentations-(Web-)Server** gibt einem oder mehreren (Thin-) Clients die Möglichkeit, Web-Services eines Applikations-Servers zu nutzen. Dieser ist als Webserver konzipiert und um eine Clientseitige Adaptionskomponente (CSA) ergänzt. Der Web-Server nimmt die Anfrage eines Client entgegen, wandelt diese in einen Web-Service-Aufruf (SOAP-Request) um und leitet den Aufruf an eine situationsabhängige Adaptionskomponente weiter. Diese Adaptionskomponente erweitert den Web-Service-Aufruf um spezifische Informationen über den Nutzungskontext und legt die weitere Bearbeitungsfolge fest. Ein Web-Service eines Applikations-Servers kann sowohl direkt, als auch indirekt über eine speziellen Adaptions-Server (Adaptation Service Management) aufgerufen werden. Ein Anwendungsbeispiel für eine clientseitige Adaption ist die Hervorhebung der Werte eines Berichtes, die sich seit der letzten Anzeige des Berichtes verändert haben.

Der **Applikations-Server** stellt die funktionalen Dienste der Software als Web-Services zentral zur Verfügung. Er besteht aus einem Web Container und einem Enterprise Java Beans (EJB)-Container. Der Web Container beinhaltet eine Komponente zur situationsabhängigen Adaption sowie Web-Service-Schnittstellen zum EJB Container. Dieser beinhaltet die Kernfunktionalität der Software und wird um eine Serverseitige Adaptionskomponente (SSA) erweitert. Ein Anwendungsbeispiel hierfür ist die Vorverdichtung von zeit- und ressourcenintensiven Berechnungen, die mittels einer Mustererkennung zeitlich vorgelagert ausgeführt werden kann.

Für die situationsabhängige Adaption wird weiterhin ein spezieller **Adaptions-Server** (Adaptation Service Management) benötigt. Dieser besteht aus einen Web-Container für situationsabhängige Adaptionskomponenten und einem EJB-Container. Ersteres ist für rechenintensive oder serviceunabhängige Adaptionen hilfreich. Als Anwendungsbeispiel kann das Modifizieren eines Nutzerprofils genannt werden. Letzteres dient der Bearbeitung nicht zeitkritischer Adaptionsaufgaben, z.B. das Protokollieren der Web-Service-Aufrufe für die interne Leistungsverrechnung. Hierzu nimmt eine Java Messaging Service (JMS)-

Komponente entsprechende Anfragen aller Komponenten als Messages entgegen. Message-Driven-Beans im EJB-Container führen die entsprechenden Adaptionsoperationen durch.

6 Gesamtarchitektur des Komponenten-Frameworks

Die Gesamtarchitektur des Komponenten-Frameworks geht insbesondere auf die für das Management der Adaptionsprozesse benötigten Komponenten ein und zeigt das Zusammenspiel mit weiteren administrativen Komponenten verteilter Software auf. Es werden die folgenden Designkriterien zugrundegelegt:

- **Standardisiertes Kommunikations- und Informationsmanagement:** Die notwendigen Informationen zur Durchführung eines Adaptionsprozesses sind in der Regel auf den betroffenen Komponenten gespeichert. Lediglich bei der ersten Durchführung und bei Veränderung der Adaption ist eine Synchronisation der Komponenten notwendig. Über einen Synchronisationsbus werden die entsprechenden Komponenten mit den benötigten Informationen einheitlich versorgt.
- **Erweiterbarkeit auf mehrere Softwarekomponenten:** Die Architektur soll eine einheitliche Plattform darstellen, um die situationsabhängige Adaption bei mehreren Komponenten gegebenenfalls unterschiedlicher Softwaresysteme zu unterstützen. Der Entwicklungsaufwand für die Unterstützung neue Softwarekomponenten soll reduziert werden.

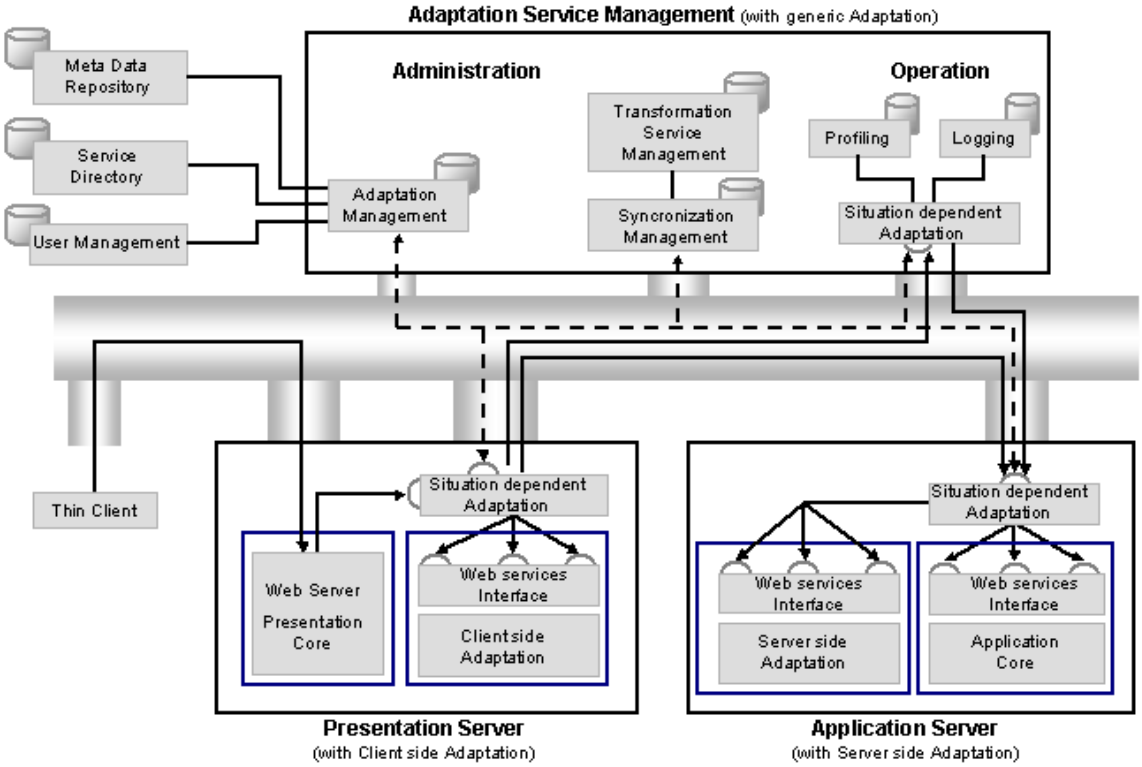

Bild *6* - Gesamtarchitektur des Komponenten-Frameworks

In der Gesamtarchitektur werden Komponenten zum Management des Gesamtsystems ergänzt (vgl. Bild 6). Neutrale Komponenten der Administration in verteilter Software sind Komponenten für die Benutzerverwaltung (User-Management), zum Verwalten von Services

(Service Directory) sowie zur Beschreibung der Services (Meta Data Repository). Diese werden auch für die situationsabhängige Adaption benötigt. Für das Adaptionsmanagement (Adaptation Service Management) werden mehrere Komponenten benötigt. Diese werden in administrative und operationale Komponenten unterteilt. Als **administrative Komponenten** werden hier die Komponenten für das übergeordnete Adaptionsmanagement (Adaptation Management), für das Synchronisationsmanagement der SDA-Komponenten (Synchronization Management) und zur Verwaltung der Transformationsvorschriften (Transformation Service Management) bezeichnet. Die wesentlichen Interaktionsbeziehungen sind ebenfalls in Bild 6 dargestellt. Die gestrichelten Kanten zeigen die Synchronisation im Gesamtsystem. Die Synchronisations-Management-Komponente aktualisiert die Daten (z.B. Profile) und synchronisiert andere Komponenten.

Als **operationale Komponenten** werden im Wesentlichen Komponenten zur Protokollierung von Transaktionsdaten, Benutzerverhalten, Antwortzeiten und sonstigen Nutzungsinformationen (Logging) sowie zur Mustererkennung in Nutzungsinformationen und für die automatischen Profilgenerierung (Profiling) benötigt. Die generierten Nutzungssituationen und Profile werden von den administrativen Komponenten verwendet. Weiterhin werden generische Adaptionskomponenten hinzugezählt.

Die Vorteile dieser Architektur liegen neben der losen Kopplung zur Kernsoftware insbesondere in der Performanz durch Reduktion des Verwaltungsaufwandes sowie in der Flexibilität hinsichtlich der Adaptionsprozesse. Durch die Komponentenorientierung wird das Gesamtsystem skalierbar und leichter erweiterbar. Die Aufteilung in Präsentations-Server und Applikations-Server bietet die nötige Flexibilität und die leichte Erweiterbarkeit für unterschiedliche Clients. Sogenannte Thin Clients (Web Browser, PDA-Browser,...) können über den Präsentations-Server flexibel eingebunden werden.

7 Zusammenfassung und Ausblick

Unter dem Begriff der situationsabhängigen Adaption wird die Anpassung einer Softwarefunktionalität an die aktuelle Nutzungssituation der Anwender verstanden. Eine situationsabhängige Software soll in der Lage sein, sich selbstständig an die speziellen Bedürfnisse der Anwender anzupassen und in ihren Tätigkeiten aktiv zu unterstützen. Aufgrund der besonderen Vorteile komponentenbasierter Software wurde im Rahmen dieser Arbeit die Entwicklung situationsabhängiger Adaption in Web-Service-basierter Standardsoftware näher untersucht. Web-Services als besondere Form der komponentenbasierten Software sind bei einem anwendungsnahen Design besonders gut für die Entwicklung adaptiver Software geeignet.

Zunächst wurde der Gesamtkontext der situationsabhängigen Adaption betrieblicher Standardsoftware beschrieben und es wurden die unterschiedlichen Teilziele einer Anwendung aufgezeigt. Der Kern der Arbeit besteht aus dem systematischen Aufbau eines Komponenten-Frameworks für die situationsabhängige Protokollierung und Adaption Web-Service-basierter Standardsoftware. Die Grundkonzeption zielt auf eine lose Kopplung zwischen Kernsoftware- und Adaptionsfunktionalität ab, um so eine nachträgliche Berücksichtigung bzw. Erweiterung der situationsabhängigen Adaption zu unterstützen. Für eine mehrstufige und performante Adaption sind spezielle Komponenten erforderlich. Diese werden in die Architektur des Gesamtsystems integriert. Sie stellt einen, gegebenenfalls auf mehrere Anwendungssysteme erweiterbaren standardisierten Rahmen für situationsabhängige Web-Service-basierte Standardsoftware bereit.

Die hier vorgestellten Konzepte für eine situationsabhängige Protokollierung und Adaption in Web-Service-basierter Standardsoftware werden bereits in einem Projekt angewendet und befinden sich in der Implementierungsphase. Nach Projektabschluss werden konkrete Erfahrungen über Aufwand und Nutzen des Komponenten-Frameworks vorliegen. Weiterhin ist noch zu klären, inwieweit der hier vorgestellte Ansatz auf andere Arten von Software übertragen werden kann.

8 Literatur

[AmFW2002] Amberg, M.; Figge, S.; Wehrmann, J. (2002): Compass – Ein Kooperationsmodell für situationsabhängige mobile Dienste: in Hampe, J. F.; Schwabe, G. (Hrsg.), Proceedings zur Teilkonferenz Mobile and Collaborative Business der Multikonferenz Wirtschaftsinformatik (MKWI 2002), Nürnberg, Deutschland.

[AmWe2002] Amberg, M.; Wehrmann, J. (2002): A Framework for the Classification of Situation Dependent Services: Eighth Americas Conference on Information Systems Proceedings (AMCIS 2002), pp. 1838-1843, Dallas, USA.

[BlBI2000] Blair, G.; Blair, L.; Issarny, V.; Tuma, P.; Zarras, A. (2000): The Role of Software Architecture in Containing Adaptation in Component-based Middleware Plattforms. Proceedings of Middleware 2000, IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing, Hudson River Valley (NY), USA.

[BaFK2000] Badrinath, B.; Fox, A.; Kleinrock, L.; Popek, G. (2000): A Conceptual Framework for Network and Client Adaptation. ACM MONET Journal, Vol. 5, No. 4, pp. 221-231.

[Gase2001] Gasenzer, R. (2001): Positionsbasierte Leistungsangebote für den mobilen Handel. In: Heilmann, H. (Hrsg.): *HMD - Praxis der Wirtschaftsinformatik*, Heft 220, S. 37-51, Heidelberg, Deutschland.

[GeJe2001] Gessler, S.; Jesse, K. (2001): Advanced Location Modeling to enable sophisticated LBS Provisioning in 3G networks. In: Beigl, M.; Gray, P.; Salber, D. (Hrsg.): *Proceedings of the Workshop on Location Modeling for Ubiquitous Computing*. Atlanta, USA.

[HiKR2002] Hitz, M.; Kappel, G.; Retschitzegger, W.; Schwinger, W. (2002): Ein UML-basiertes Framework zur Modellierung ubiquitärer Web-Anwendungen. In Wirtschaftsinformatik 44 (3/2002), S.225-235, Wiesbaden, Deutschland.

[KeBC2002] Ketfi, A.; Belkhatir, N.; Cunin, P.-Y. (2002): Automatic Adaptation of Component-based Software. PDPTA'02, Las Vegas, USA.

[May2001] May, P. (2001): *Mobile Commerce – Opportunities, Applications, and Technologies of Wireless Business*. Cambridge University Press, Cambridge, UK.

[OrGT1999] Oreizy, P.; Gorlick, M.M.; Taylor, R.N.; Heimbigner, D.; Johnson, G.; Medvidovic, N.; Quilici, A.; Rosenblum, D.S.; Wolf, A.L. (1999): An Architecture-Based Approach to Self-Adaptive Software in: IEEE Intelligent Systems, May/June 1999 (Vol. 14, No. 3), pp 54-62.

[OrHE1996] Orfali, R.; Harkey, D.; Edwards, J. (1996): The Essential Distributed Objects Survival Guide. John Wiley & Sons, New York, USA.

[Turo1999] Turowski, K. (1999): Ordnungsrahmen für komponentenbasierte betriebliche Anwendungssysteme. In: K. Turowski (Hrsg.): Tagungsband des 1. Workshops Komponentenorientierte betriebliche Anwendungssysteme (WKBA 1), S. 3-14, Magdeburg, Deutschland.

[WaSc2000] Wang, X.; Schulzrinne, H. (2000): An Integrated Resource Negotiation, Pricing, and QoS Adaptation Framework for Multimedia Applications. IEEE Journal on Selected Areas in Communications (JSAC), Vol. 18, No. 12, pp. 2514-2529.

[Zobe2001] Zobel, J. (2001): *Mobile Business und M-Commerce*. München, Deutschland.

Zur Spezifikation der Parameter von Fachkomponenten

Jörg Ackermann

Von-der-Tann-Str. 42, 69126 Heidelberg, Deutschland, E-Mail: joerg.ackermann.hd@t-online.de

Zusammenfassung. Die Spezifikation einer Softwarekomponente ist eine wesentliche Voraussetzung für ihre erfolgreiche Wiederverwendung. Von der GI-Arbeitsgruppe 5.10.3. wurde ein Vorschlag erstellt, wie die Spezifikation von Fachkomponenten standardisiert werden kann. Nicht untersucht wurde bisher, wie ein möglicher Parametrisierungsspielraum von Fachkomponenten spezifiziert werden soll. Dieser Beitrag beschäftigt sich mit dem Teilproblem, wie die Parameter einer Fachkomponente spezifiziert werden können. Außerdem wird ein Ausblick gegeben, wie die Auswirkungen der Parametereinstellungen beschreibbar sind.

Schlüsselworte: Fachkomponente; Softwarekomponente; Spezifikation; Standardisierung; Customizing; Parametrisierung

Mit der Komponententechnologie wird die Hoffnung verbunden, dass - wie in anderen Ingenieursdisziplinen üblich - komplexe Systeme aus vorgefertigten Bausteinen zusammengesetzt werden können. Davon verspricht man sich eine höhere Wiederverwendungsrate und eine damit verbundene Erhöhung von Effektivität und Qualität der Softwareentwicklung. Für betriebliche Anwendungen wird außerdem erwartet, dass man Anwendungssysteme bauen kann, die möglichst genau auf die Bedürfnisse von Verwendern zugeschnitten sind. Dies ist mit der heutigen Standardsoftware nur eingeschränkt möglich (siehe z.B. [Szyp1998, S.5]).

Dies setzt voraus, dass Softwarekomponenten zur Verfügung stehen, die möglichst genau auf die spezifischen Kundenwünsche ausgerichtet sind. Dazu sind prinzipiell zwei Ansätze denkbar: a) es gibt sehr viele Komponenten, welche verschiedenen Anforderungen jeweils punktgenau genügen, und ein Verwender wählt die passende Komponente aus; b) es gibt wenige, aber variable Komponenten und ein Verwender passt eine Komponente auf seine konkreten Bedürfnisse an.

Man kann zwar davon ausgehen, dass für Anforderungen, die sich deutlich voneinander unterscheiden, verschiedene Komponenten entstehen werden. Aus praktischen Gesichtspunkten ist es jedoch unwahrscheinlich, dass ein Hersteller für jede Kombination von Detailanforderungen eine eigene Komponente produzieren wird. Deshalb ist es notwendig, dass Softwarekomponenten in einem gewissen Rahmen anpassbar sind. Es müssen geeignete Mechanismen entwickelt werden, die ermöglichen, Komponenten effektiv auf Kundenbedürfnisse anzupassen.

Neben den technischen Hilfsmitteln zur Anpassung sind dem Verwender einer Komponente auch alle inhaltlichen Informationen zur Verfügung zu stellen, die dieser benötigt, um die möglichen und notwendigen Einstellungen vorzunehmen. Dies bedeutet, dass der Spielraum für Anpassungen und die Konsequenzen einer Anpassung für die Arbeitsweise der Komponente zu spezifizieren sind.

Dieser Beitrag beschäftigt sich mit der Anpassung von Fachkomponenten über Parametrisierung. und stellt Zwischenergebnisse einer laufenden Forschungsarbeit vor. (Für weitere Anpassungsstrategien siehe Abschnitt 1.2.) Wir untersuchen, wie Parameter einer Fachkomponente spezifiziert werden können. Im Kapitel 1 stellen wir kurz den derzeitigen Stand der Forschung dar. Im Kapitel 2 definieren wir, was wir unter Parametrisierung verstehen. Im Kapitel 3 unterbreiten wir Vorschläge, wie Parameter von Fachkomponenten spezifiziert werden können. Diese Vorschläge wurden in einer Fallstudie umgesetzt. Über die dabei gewonnenen Erkenntnisse berichten wir im Kapitel 4. Im Kapitel 5 wird schließlich ein Ausblick gegeben, wie der gesamte Parametrisierungsspielraum einer Fachkomponente spezifiziert werden könnte.

1 Stand der Forschung

Zum Thema „Spezifikation der Parametrisierbarkeit von Fachkomponenten“ sind uns keine Arbeiten bekannt (außer eigenen Vorarbeiten). Es gibt jedoch verschiedene Arbeiten zu Teilaspekten. Diese wurden in [Acke2002] ausführlich vorgestellt und diskutiert. In diesem Kapitel sollen die Ergebnisse kurz zusammengefasst werden.

1.1 Spezifikation von Fachkomponenten

Zur Spezifikation von Fachkomponenten ist das Memorandum „Vorschlag zur Vereinheitlichung der Spezifikation von Fachkomponenten“ [Turo2002] besonders hervorzuheben. Dieser Vorschlag wurde im Rahmen der Arbeitsgruppe 5.10.3. der Gesellschaft für Informatik von Vertretern aus Forschung und Praxis erstellt.

Dem Vorschlag liegt als „Leitbild, im Sinne eines idealen zukünftigen Zustands, die Idee einer kompositorischen, plug-and-play-artigen Wiederverwendung von Black-Box-Komponenten zu Grunde, deren Realisierung einem Verwender verborgen bleibt und die auf einem Softwaremarkt gehandelt werden.“ Im Memorandum [Turo2002] werden die Begriffe (Software-) Komponente und Fachkomponente wie folgt definiert:

- „Eine *Komponente* besteht aus verschiedenartigen (Software-) Artefakten. Sie ist wiederverwendbar, abgeschlossen und vermarktbar, stellt Dienste über wohldefinierte Schnittstellen zur Verfügung, verbirgt ihre Realisierung und kann in Kombination mit anderen Komponenten eingesetzt werden, die zur Zeit der Entwicklung nicht unbedingt vorhersehbar ist.“
- „Eine *Fachkomponente* ist eine Komponente, die eine bestimmte Menge von Diensten einer betrieblichen Anwendungsdomäne anbietet.“

Im weiteren Verlauf dieses Beitrages folgen wir diesen Definitionen und dem vorgestellten Leitbild.

Turowski et al. formulieren methodische Standards, wie Fachkomponenten eindeutig und vollständig spezifiziert werden können. Es wird „postuliert, dass die Spezifikation von Fachkomponenten auf verschiedenen Abstraktionsebenen erfolgen muss“: Schnittstellenebene, Verhaltensebene, Abstimmungsebene, Qualitätsebene, Terminologieebene, Aufgabenebene und Vermarktungsebene. Zu allen Abstraktionsebenen werden die zu spezifizierenden Objekte, eine primäre Spezifikationstechnik sowie gegebenenfalls ergänzende sekundäre Spezifikationstechniken festgelegt.

Variabilität und Anpassbarkeit von Fachkomponenten wurden aufgrund fehlender Vorarbeiten bisher nicht berücksichtigt.

1.2 Anpassbarkeit von Softwarekomponenten

Es ist davon auszugehen, dass beim Erstellen von betrieblichen Anwendungen aus Fachkomponenten der Bedarf besteht, einzelne Komponenten oder die gesamte Anwendung kundenspezifisch anzupassen. Verschiedene Autoren haben sich mit der Anpassbarkeit von Softwarekomponenten beschäftigt. Siehe dafür z.B. [BRS+2000], [FlGu1996] oder [StCr1998]. Die allgemeine Problematik Variabilität und Wiederverwendung wird z.B. von Jacobsen et al. im Standardwerk „Software Reuse“ ausführlich diskutiert [JaGJ1997].

Zusammenfassend lässt sich feststellen, dass verschiedene Mechanismen und Techniken zur Anpassung und Variabilität von komponentenbasierten Anwendungen bekannt sind. Diese können grob in zwei Gruppen unterteilt werden:

- Anpassung der Komponentenarchitektur sowie der Auswahl und des Zusammenspiels der einzelnen Komponenten (z.B. Änderung der Komposition von Komponenten, Verwendung von Wrapperkomponenten, Komposition mit Adaptorkomponenten),
- Anpassung einzelner Komponenten selbst (z.B. Vererbung, Änderung der Implementierung, Erweiterungen durch Extension classes, Setzen von Parameterwerten).

Bei der zweiten Gruppe bewegen sich die meisten Techniken auf der Programmierebene und es wird zumeist nicht davon ausgegangen, dass vom Komponentenhersteller eine Variabilität vorgesehen wurde.

Wenig untersucht wird der Fall, dass eine Komponente selbst eine gewisse Variabilität mit sich bringt. Mögliche Techniken sind das Setzen von Parameterwerten, das Programmieren eigener Teilaspekte in vorgedachten Erweiterungspunkten und das Einbinden alternativer, vorgedachter Varianten. Eine detaillierte Beschreibung dieser Techniken erfolgt jedoch nicht.

1.3 Parametrisierung betrieblicher Standardsoftware

Zur Parametrisierung und Anpassbarkeit betrieblicher Anwendungssysteme liegen umfangreiche Erfahrungen für monolithische Standardsoftware vor. Solche Anwendungssysteme sind hochkomplex parametrisierbar, um eine möglichst genaue Anpassung an die konkreten Kundenwünsche zu ermöglichen. Die Parametrisierung wird bei betrieblicher Standardsoftware oft auch als Customizing bezeichnet.

Umfangreiche Untersuchungen über die Kopplung von Geschäftsprozessmodellen und Anwendungssystemen wurden im WEGA-Projekt vorgenommen. Bei WEGA (Wiederverwendbare und erweiterbare Geschäftsprozess- und Anwendungssystem-architekturen) handelt es sich um ein Verbundprojekt zwischen der Universität Bamberg, der SAP AG und der KPMG Unternehmensberatung. „Ziel des Projektes WEGA [war] die Entwicklung und Untersuchung von Architekturen für Geschäftsprozessmodelle und Anwendungssysteme in industriellen Größenordnungen ...“. Besonders berücksichtigt werden sollte dabei, dass die Beherrschung von Komplexität, Variantenreichtum und Interoperabilität unterstützt wird. Für Details und umfangreiche Literaturverweise siehe den WEGA-Abschlussbericht [FSH+1998].

Im Ergebnis des WEGA-Projektes entstand das heutige Vorgehen beim Customizing von SAP R/3. (Für Details dazu siehe das SAP R/3 Referenz(prozess)modell [SAP1997]). Der Fokus liegt dabei hauptsächlich auf einer Komplexitätsreduktion im Großen. Es werden die folgenden Ziele verfolgt: Beherrschung von Komplexität und Variantenreichtum, Aufzeigen von Abhängigkeiten zwischen Geschäftsprozessen und Customizing, modellgestütztes Customizing, sowie die Reduktion der Komplexität auf die kundenspezifischen Anforderungen.

Nicht im Fokus lag allerdings die detaillierte Beschreibung einzelner Parametereinstellungen. So werden z.B. im System SAP R/3 die einzelnen Customizing-Aktivitäten überblicksartig dokumentiert, eine formale Beschreibung erfolgt jedoch nicht. Bedingungen und Abhängigkeiten auf der Ebene einzelner Parameter sind oft nur anhand von Systemmeldungen nachvollziehbar.

1.4 Referenzmodellierung

Zur Referenzmodellierung sei z.B. auf [Schü1998] verwiesen. Referenzmodelle können verkürzt „[...] als Darstellung unternehmensklassenspezifischer Strukturen und Abläufe mit Sollcharakter definiert werden [...]“ [Schü1998, S.1] Referenzmodelle können als Grundlage dienen, um unternehmensspezifische Informationsmodelle zu erstellen. Dazu sind in einem Referenzmodell „[...] explizit Alternativen abzubilden, die nur im Referenzmodell gültig sind und entsprechend den Anforderungen des Modellanwenders in einem konkreten Modell eine Änderung erfahren.“ [Schü1998, S.82]

Zur Abbildung von Alternativen werden bei Schütte neben den bekannten Operatoren im Prozessmodell (dort als Run-Time-Operatoren bezeichnet) auch sogenannte Build-Time-Operatoren definiert, die nur im Referenzmodell Anwendung finden. Analog werden die Beziehungstypen im Datenmodell (Run-Time-Beziehungstypen genannt) um Build-Time-Beziehungstypen ergänzt. Bei der Konfiguration eines unternehmensspezifischen Modells aus einem Referenzmodell besteht für den Modellierer (innerhalb vorgegebener Regeln) die Wahlfreiheit, wie die Build-Time-Operatoren bzw. -Beziehungstypen in Run-Time-Operatoren bzw. -Beziehungstypen überführt werden können.

1.5 Eigene Vorarbeiten

In [Acke2002] wurde die in Kapitel 3 aufgeführte Liste von Thesen aufgestellt. Die Thesen betreffen die Fragestellung, welche Aspekte berücksichtigt werden müssen, um die Parameter von Fachkomponenten zu spezifizieren. Da es keine Literatur zur Parametrisierung von Fachkomponenten gab, wurde in [Acke2002] ein Teilbereich des Customizings von SAP R/3 untersucht. Danach wurde versucht, die dabei gewonnenen Erkenntnisse auf Fachkomponenten zu übertragen. Dieser Beitrag ist eine Fortsetzung zu [Acke2002].

2 Anpassung durch fachliche Parametrisierung

Was wir unter Parametrisierung verstehen, wurde in [Acke2002] genauer beschrieben. Dort wurden einige Begriffe definiert, erläutert und zum Teil mit Beispielen belegt. Für ein besseres Verständnis dieser Arbeit werden hier die wichtigsten Aspekte wiederholt.

Unter *Parameter* verstehen wir ein Datenfeld mit vorgegebenem oder freien Wertebereich. Der Anwender kann dieses mit einem *Parameterwert* füllen und damit die Funktionsweise der

Software beeinflussen. Während des Ablaufs von Transaktionen und Prozessen verhalten sich Parameter wie Konstanten und unterscheiden sich damit von Programmiervariablen. Allerdings sind Parameter nicht auf alle Zeiten konstant, sondern können zwischen verschiedenen Transaktionen (in gewissem Rahmen) geändert werden.

Parametrisierung heißt eine Anpassungsstrategie, die sich durch folgende Merkmale auszeichnet:

- Die Anpassungsmöglichkeiten werden vom Hersteller der Software vorgedacht.
- Der Hersteller definiert Parameter, deren Bedeutung sowie deren Auswirkungen auf die Funktionsweise der Software.
- Der Verwender belegt die Parameter mit Werten.
- Die Parameterbelegungen sind persistent und werden zur Laufzeit ausgewertet.

Unter *Parametrisierung* verstehen wir nicht nur die Anpassungsstrategie, sondern auch den Prozess, die vorgegebenen Parameter mit Werten zu füllen. Weitgehend synonym zu Parametrisierung ist der oft verwendete Begriff *Customizing*. Der Begriff *Parametrisierungsspielraum* beschreibt die Gesamtheit der Variabilität (in Datenstrukturen und Verhalten) einer Fachkomponente, welche durch Parametrisierung erreicht werden kann.

Unter *fachlicher Parametrisierung* verstehen wir solche Einstellungen, die betriebswirtschaftlich und aufgabenbezogen sind. Mögliche Beispiele dafür sind:

- Definition von organisatorischen Einheiten und Stammdaten (z.B. Werke, Materialen),
- Auswahl unter vorgegebenen Prozessvarianten,
- Definition von Steuerdaten (z.B. erlaubte Anlieferungszeiten an einem Lager),
- Definition von Daten zur Dialog- und Benutzersteuerung (z.B. Vorschlagswerte).

Unter *technischer Parametrisierung* verstehen wir Einstellungen, die sich auf einer rein technischen Ebene befinden (z.B. verwendete Datenbank, verwendetes Betriebssystem). In unseren weiteren Untersuchungen interessiert uns nur die fachliche Parametrisierung. Der Einfachheit halber werden wir oft von Parametrisierung sprechen, meinen damit aber immer die fachliche Parametrisierung.

Unserer Meinung nach wird die Parametrisierung eine wichtige Rolle bei der Anpassung von Fachkomponenten spielen. Dafür spricht, dass dieser Anpassungsmechanismus vom Verwender keine Kenntnis der Implementierung erfordert und damit gut zur Black-Box-Wiederverwendung passt (vgl. Abschnitt 1.1). Außerdem ist Parametrisierung eine weit verbreitete Technik zur Anpassung betrieblicher Standardsoftware, was eine gute Eignung für die Anpassung betrieblicher Anwendungen nahe legt. Eine genauere Untersuchung und Beschreibung der Parametrisierung von Fachkomponenten erfolgte bisher nicht, erscheint jedoch wünschenswert.

Im weiteren Verlauf der Arbeit werden wir zwischen operativen Diensten und Diensten zur Parametrisierung unterscheiden. Unter *operativen Diensten* verstehen wir die Dienste, mit deren Hilfe die Fachkomponente die ihr zugeordneten betrieblichen Aufgaben erfüllt. Die *Dienste zur Parametrisierung* dienen dazu, Parameter mit geeigneten Werten zu belegen. In der Praxis wird die Abgrenzung zwischen operativen Diensten und Diensten zur Parametrisierung nicht unbedingt eindeutig sein. Die Unterscheidung hilft uns jedoch, die Aufgabenstellung dieser Arbeit zu verdeutlichen.

Bild 1 setzt unser Forschungsvorhaben in Bezug zur Spezifikation von Fachkomponenten. Im Memorandum [Turo2002] wurde festgelegt, dass die operativen Dienste einer Komponente spezifiziert werden müssen, und wie dies geschehen sollte (vgl. Punkt 1). Die Dienste zur Parametrisierung gehören ebenso zur Außensicht einer Komponente und müssen ebenfalls spezifiziert werden (Punkt 2). Abhängig von gewählten Parameterwerten können sich die operativen Dienste unterschiedlich verhalten. Diese Abhängigkeiten der operativen Dienste von den Parametereinstellungen sind daher ebenso zu spezifizieren (Punkt 3).

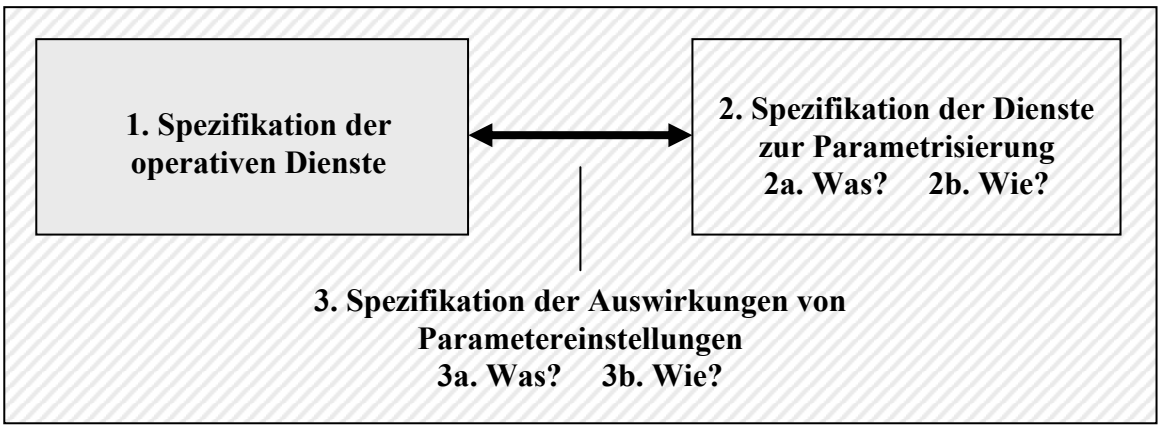

Bild 1: Aufgabenstellungen für die Spezifikation von Fachkomponenten

Da es zur Parametrisierung von Fachkomponenten keine Vorarbeiten gibt, muss für die Punkte 2 und 3 sowohl untersucht werden, welche Aspekte zu spezifizieren sind (2a und 3a), als auch, wie diese Spezifikation erfolgen kann (2b und 3b).

In [Acke2002] wurde Punkt 2a behandelt. Darauf aufbauend unterbreiten wir in dieser Arbeit Vorschläge, wie Punkt 2b gelöst werden kann. Im Kapitel 5 geben wir einen Ausblick auf mögliche Lösungen für die Punkte 3a und 3b.

3 Spezifikation von Parametern einer Fachkomponente

3.1 Einführende Überlegungen

1. Es liegen noch keine Erfahrungen vor, wie Fachkomponenten parametrisiert werden können. In [Acke2002] haben wir deshalb einen Teilbereich des Customizings von SAP R/3 untersucht. Danach wurde bewertet, inwieweit sich die gewonnenen Erkenntnisse auf Fachkomponenten übertragen lassen. Als Ergebnis entstand eine Liste von Aspekten, die berücksichtigt werden müssen, um die Parameter von Fachkomponenten zu spezifizieren. Diese wurden in Thesenform formuliert. Die Thesen sind Ausgangspunkt unserer Überlegungen (siehe Abschnitt 3.2).

2. Als Technik für die Spezifikation von Parametern wählen wir die OMG Unified Modeling Language (UML) (siehe [OMG2001]) aus folgenden Gründen:

- Die Thesen beschreiben eine Mischung aus Daten- und Prozesssicht, wofür die objektorientierte Sichtweise der UML gut geeignet ist.
- Die UML hat die Mächtigkeit, alle relevanten Aspekte abbilden zu können.
- Die UML ist eine formale Sprache, d.h. sie hat eine eindeutige Syntax und Semantik.

- Bei der UML handelt es sich um einen weit verbreiteten und bekannten Standard.
- Im Memorandum [Turo2002] werden auf der Verhaltens- und der Abstimmungsebene ebenfalls Elemente der UML verwendet. Durch diese Wahl wird ein Methodenbruch zwischen der Spezifikation der operativen Dienste und der Dienste zur Parametrisierung vermieden.

Teil der UML ist die Object Constraint Language (OCL), mit welcher Bedingungen zwischen Modellelementen formuliert werden können. Die OCL wird ebenfalls verwendet.

3. Die Ausführungen in Abschnitt 3.2 werden durch Beispiele näher erläutert. Alle Beispiele beziehen sich auf das UML-Klassendiagramm in Bild 2. Das Diagramm beschreibt Teile der Fachkomponente *Flugticketverkauf*, die im Anhang B spezifiziert wird. Zur Vereinfachung enthält Bild 2 nur einige Dienste zur Parametrisierung. Klassen mit operativen Diensten werden in Bild 2 nicht dargestellt. Für das vollständige Klassendiagramm der Fachkomponente siehe Anhang B, Abschnitt 3.

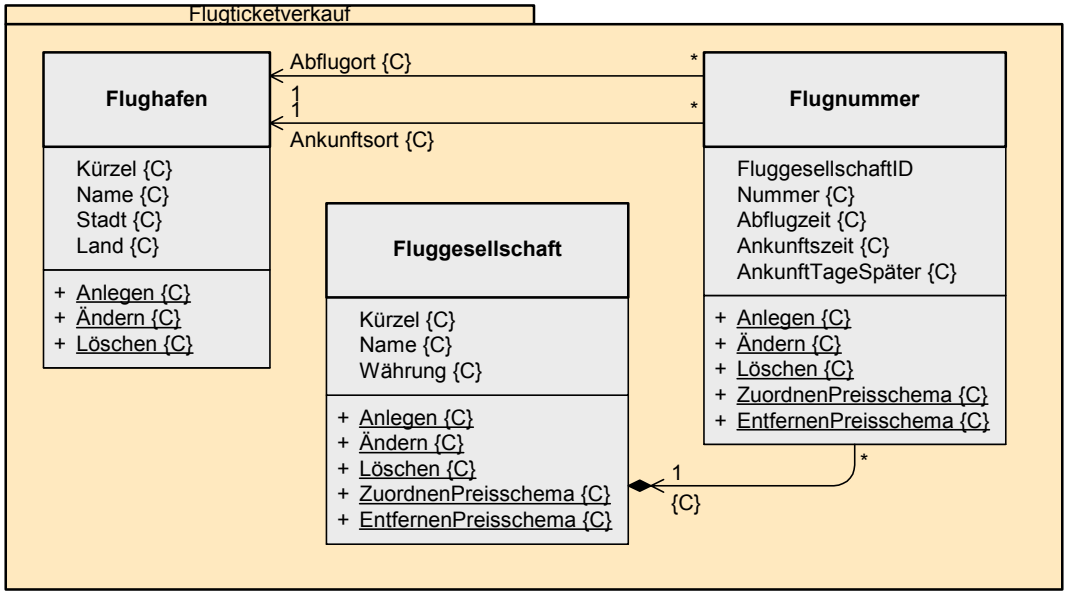

Bild 2: Ausschnitt aus dem UML-Klassendiagramm zur Fachkomponente *Flugticketverkauf*

Zur Erläuterung des Beispiels: Flugnummern bezeichnen regelmäßige Flüge einer Fluggesellschaft. Eigenschaften von Flugnummern sind die Fluggesellschaft, Nummer, Abflugort, Ankunftsort, Abflugzeit, Ankunftszeit. An welchen Tagen die Flüge stattfinden, wird in der Klasse *Flug* festgelegt, die in Bild 2 nicht enthalten ist.

> Beispiel: Die Fluggesellschaft LH (Lufthansa) bietet regelmäßige Flüge mit der Flugnummer LH 400 an. Abflugort ist der Flughafen FRA (Frankfurt) und Ankunftsort ist der Flughafen JFK (New York). Die Flüge starten immer 10.10 Uhr und landen 12.55 Uhr (jeweils Ortszeit) des gleichen Tages.

Die Fachkomponente *Flugticketverkauf* bietet Dienste an, die den Verkauf von Flugtickets unterstützt. Die Daten zu Flughäfen und Fluggesellschaften sowie der Flugplan (Flugnummern und Flüge) werden für die operativen Dienste der Fachkomponente vorausgesetzt und werden durch Parametrisierung gepflegt.

3.2 Vorschläge zur Spezifikation von Parametern

Im Folgenden finden Sie die Thesen aus [Acke2002] zur Fragestellung, welche Aspekte von Parametern bei der Spezifikation berücksichtigt werden müssen. Diese sind jeweils kursiv dargestellt. Zu den Thesen unterbreiten wir Vorschläge, wie diese Aspekte mit Hilfe der UML bei der Spezifikation abgebildet werden können. Wir sprechen im weiteren Verlauf immer von den *Abbildungsvorschriften* zu den Thesen.

Zur Erinnerung wird zunächst die Definition wiederholt: Parametrisierung von Fachkomponenten bedeutet, dass der Entwickler bestimmte Parameter vordefiniert, die der Anwender mit Werten belegt. D.h. es gibt Parameter und es gibt Aktivitäten, mit denen die Parameterwerte gesetzt bzw. verändert werden können.

1. *Es lassen sich mehrere Parameter zu logischen Einheiten zusammenfassen, die üblicherweise zusammen gepflegt werden. Diese Einheiten nennen wir im Folgenden Customizing-Gruppen (CG). Gruppen der Größe 1 sind möglich.*

Die Customizing-Gruppen werden in UML durch Klassen in einem Klassendiagramm repräsentiert. Die zugehörigen Parameter sind Attribute oder Assoziationen der Klasse. Für Details zur Darstellung der Parameter siehe Punkt 3.

> Beispiel: Die CG *Flugnummer* wird durch die gleichnamige Klasse in Bild 2 repräsentiert. Zur CG gehören die Parameter *Fluggesellschaft*, *Nummer*, *Abflugort*, *Ankunftsort*, *Abflugzeit*, *Ankunftszeit* und *AnkunftTageSpäter*.

2. *Zu den Customizing-Gruppen kann es zur gleichen Zeit mehrere Belegungen von Parameterwerten geben.*

Die Belegungen werden durch die Instanzen der Klasse repräsentiert. Strukturell entsprechen die Instanzen der Klasse, aber die Parameter können in den einzelnen Instanzen unterschiedliche Werte annehmen. Einschränkungen, wie viele Instanzen zu einer Klasse existieren dürfen, können durch einen OCL-Ausdruck beschrieben werden.

> Beispiel: Bei den Flugnummern LH 400 und LH 401 handelt es sich um verschiedene Belegungen der CG *Flugnummer*. Parameterwerte von LH 400 sind: Abflug 10.10 Uhr in FRA, Ankunft 12.55 Uhr in JKF, *AnkunftTageSpäter* = 0. Die Parameterwerte von LH 401 sind: Abflug 16.00 Uhr in JFK, Ankunft 05.35 Uhr in FRA, *AnkunftTageSpäter* = 1.

3. *Die Parameter lassen sich aufgrund ihres Wertebereichs in zwei Gruppen unterteilen:*

   a) *Parameter mit nicht-customizingabhängigem Wertebereich*

Ein solcher Parameter wird durch ein Attribut repräsentiert. Das Attribut wird mit dem Eigenschaftswert (Tagged Value) {C} versehen und damit als parametrisierungsrelevant gekennzeichnet. Der Wertebereich wird durch den Datentyp des Attributs näher bestimmt.

Besteht der Wertebereich aus vorgegebenen Festwerten, gibt es zwei Möglichkeiten zur Darstellung: entweder wird ein Aufzählungs-Datentyp (`enumeration`) verwendet, oder die erlaubten Werte werden im Rahmen einer OCL-Bedingung angegeben.

> Beispiel: *Nummer*, *Abflugzeit, Ankunftszeit* und *AnkunftTageSpäter* sind die Parameter der CG *Flugnummer*, deren Wertebereiche nicht-customizingabhängig sind. Diese Parameter finden sich in Bild 2 als Attribute der Klasse Flugnummer und sind mit {C} gekennzeichnet. Die Parameter *Abflugzeit* und *Ankunftszeit* sind z.B. vom Typ „Uhrzeit“, welcher ihren Wertebereich vorgibt.

b) Parameter mit customizingabhängigem Wertebereich, d.h. der Wertebereich des Parameters besteht aus den Belegungen einer anderen Customizing-Gruppe

Ein solcher Parameter wird durch eine Assoziation zwischen den beiden Klassen repräsentiert. Die Assoziation wird am Assoziationsende mit dem Eigenschaftswert {C} versehen und damit als parametrisierungsrelevant gekennzeichnet. Der so modellierte Parameter gehört zur Klasse am Assoziationsanfang, d.h. zu der Klasse, an deren Ende sich das {C} nicht befindet.

Bei Bedarf kann das Assoziationsende mit einem Rollennamen versehen werden. In einem solchen Fall ist der Name des Parameters gleich dem Rollennamen, ansonsten gleich der assoziierten Klasse. Die Art der Beziehung kann über die Art der Assoziation ausgedrückt werden. Dafür stehen Komposition, Aggregation und normale Assoziation zur Verfügung. Üblicherweise ist eine Navigation zur assoziierten Klasse möglich.

Beispiel 1: Flugnummern gehören immer zu einer Fluggesellschaft. D.h. der Wertebereich des Parameters *Fluggesellschaft* (der CG *Flugnummer*) ist die Menge der Fluggesellschaften, die in der CG *Fluggesellschaft* definiert wurden. Deshalb wird der Parameter *Fluggesellschaft* (der CG *Flugnummer*) durch eine Assoziation von der Klasse *Flugnummer* zur Klasse *Fluggesellschaft* beschrieben. Diese Assoziation wird am Assoziationsende mit {C} gekennzeichnet.

Beispiel 2: Abflugort und Ankunftsort (einer Flugnummer) können nur Flughäfen sein, die in der CG *Flughafen* definiert sind. Daher werden die Parameter *Abflugort* und *Ankunftsort* (der CG *Flugnummer*) durch jeweils eine Assoziation von der Klasse *Flugnummer* zur Klasse *Flughafen* beschrieben. Beide Assoziation sind mit {C} gekennzeichnet. Da die Namen der Parameter ungleich *Flughafen* (der Name der assoziierten Klasse) sind, wurden die Rollennamen *Abflugort* und *Ankunftsort* vergeben.

4. *Es kann Parameter geben, welche einzelne Belegungen einer CG eindeutig identifizieren. Diese werden auch als Schlüsselattribute der CG bezeichnet.*

Dieser Sachverhalt kann durch eine Bedingung in OCL ausgedrückt werden.

Beispiel: Der Parameter *Kürzel* identifiziert die Belegungen der CG *Fluggesellschaft* eindeutig. Mögliche Werte des Parameters *Kürzel* sind „LH“ oder „AA“. Dieser Sachverhalt kann durch folgende OCL-Bedingung beschrieben werden:

```
Flugticketverkauf::Fluggesellschaft
  inv: Fluggesellschaft.forAll(fg1, fg2 |
                   fg1 <> fg2 implies fg1.Kürzel <> fg2.Kürzel)
```

Zunächst wird mit `Flugticketverkauf::Fluggesellschaft` der Kontext angegeben, für welchen die Bedingung gilt. Das Kürzel „inv“ steht für Invariante und beschreibt, dass diese Bedingung immer gelten soll. Die Bedingung selbst ist so zu lesen: Nimmt man zwei beliebige, aber verschiedene Instanzen von *Fluggesellschaft*, dann müssen diese sich im Attribut *Kürzel* unterscheiden.

5. *Es kann beliebige Aktivitäten geben, mit denen man die Parameter mit geeigneten Werten belegen bzw. die Wertebelegung wieder rückgängig machen kann.*

Diese Aktivitäten werden durch Methoden an den entsprechenden Klassen repräsentiert. Dabei sollte jede Methode einen elementaren Arbeitsschritt darstellen. Durch den Eigenschaftswert {C} bei den Methoden wird ausgedrückt, dass sie zur Manipulation der

Parameterwerte vorgesehen sind.

Zu jeder Methode muss ihre Schnittstelle spezifiziert werden. Die Spezifikation der Dienste zur Parametrisierung soll später in die Gesamtspezifikation der Fachkomponente integriert werden (siehe Kapitel 5). Wir verwenden daher, wie im Memorandum [Turo2002] allgemein für Dienste vorgeschlagen, die OMG IDL als Spezifikationstechnik, um die Schnittstellen der Methoden zu beschreiben.

Gibt es Einschränkungen bei der Verwendung einer Methode oder Abhängigkeiten zwischen Parametern, die bei der Pflege zu beachten sind, dann werden diese durch OCL-Bedingungen abgebildet. (Siehe dazu auch Punkte 6-8.)

Beispiel: Zur CG *Fluggesellschaft* gibt es die Methoden *Anlegen*, *Ändern*, *Löschen*, *ZuordnenPreisschema* und *EntfernenPreisschema*.

Die Schnittstelle der Methode *Fluggesellschaft.Anlegen* kann in OMG IDL folgendermaßen ausgedrückt werden:

```
interface Fluggesellschaft {
   struct FluggesellschaftKeyTyp {
                    string<3>         Kürzel; };
   struct FluggesellschaftDatenTyp {
                    string<20>        Name;
                    string<20>        Währung; };
   void Anlegen(
         in  FluggesellschaftKeyTyp    FluggesellschaftKey,
         in  FluggesellschaftDatenTyp  FluggesellschaftDaten); }
```

6. *Die Pflege von Parametern kann optional oder obligatorisch sein. Optionale Parameter können mit Defaultwerten vorbelegt sein.*

Die Notwendigkeit, einen Parameter pflegen zu müssen, kann weder mit der OMG IDL noch im UML-Modell adäquat ausgedrückt werden. Daher wird vorgeschlagen, für jede Methode eine Vorbedingungen in OCL zu formulieren, welche die obligatorischen Parameter kennzeichnet. Defaultwerte werden ebenso durch eine OCL-Bedingung angegeben. (Mit der OMG IDL kann man zwar ganzen Parametern Defaultwerte zuweisen, nicht jedoch einzelnen Feldern strukturierter Parameter.)

Beispiel: Bei der Definition einer Fluggesellschaft ist die Pflege aller Parameter obligatorisch. In einer OCL-Bedingung wird daher beschrieben, dass alle Felder (*Kürzel*, *Name* und *Währung*) beim Aufruf der Methode *Fluggesellschaft.Anlegen* gefüllt sein müssen:

```
Flugticketverkauf::Fluggesellschaft::Anlegen(Key, Daten)
  pre: Key.Kürzel <> ´´ and Daten.Name <> ´´ and Daten.Währung <> ´´
```

7. *Es können Bedingungen auftreten, welche die Reihenfolge verschiedener Aktivitäten vorschreiben.*

Reihenfolgebedingungen werden mit den temporalen Operatoren ausgedrückt, welche in [CoTu2000] als Ergänzung zur OCL vorgeschlagen wurden.

Beispiel: Wird eine Fluggesellschaft definiert, muss ihr auch ein Preisschema zugeordnet werden. (Bei Preisschema handelt es sich um eine weitere CG der Komponente *Flugticketverkauf*. Diese ist im Bild 2 nicht abgebildet, findet sich aber im vollständigen

Klassendiagramm im Anhang B, Kapitel 3).

```
Flugticketverkauf::Fluggesellschaft::Anlegen(Key, Daten)
 post: sometime after(Fluggesellschaft::ZuordnenPreisschema(Key,Prs))
```

Der Kontext dieser Bedingung ist die Methode *Anlegen* der Klasse *Fluggesellschaft*. Mit dem Kürzel „post“ wird gekennzeichnet, dass es sich um eine Nachbedingung handelt. Die Bedingung ist folgendermaßen zu lesen: Wurde die Methode *Anlegen* erfolgreich ausgeführt (Nachbedingung), dann muss in Zukunft auch die Methode *ZuordnenPreisschema* ausgeführt werden.

8. *Es können Abhängigkeiten zwischen verschiedenen Parametern auftreten. Wir treffen die Annahme, dass sich diese als Bedingungen mithilfe der UML OCL ausdrücken lassen.*

Abhängigkeiten zwischen einzelnen Parametern werden durch OCL-Ausdrücke beschrieben, welche die beteiligten Klassen und Attribute enthalten. Sind Bedingungen immer gültig, werden sie als Invarianten abgebildet. Treten Bedingungen nur bei der Ausführung von Methoden auf, dann handelt es sich um Vor- und Nachbedingungen zu den Methoden.

Beispiel: Abflugort und Ankunftsort einer Flugnummer müssen verschieden sein. Dieser Sachverhalt wird durch eine Invariante ausgedrückt, welche die Parameter *Abflugort* und *Ankunftsort* der CG *Flugnummer* verknüpft:

```
Flugticketverkauf::Flugnummer
  inv: self.Abflugort <> self.Ankunftsort
```

9. *Auftretende Bedingungen werden meist innerhalb einer Fachkomponente bestehen, können aber auch komponentenübergreifend sein.*

Bestehen Bedingungen an Parameter, welche außerhalb der Fachkomponente definiert sind, müssen diese Parameter ebenfalls in das UML-Modell aufgenommen werden. Analog zum Vorgehen bei der Spezifikation der operativen Dienste (vergleiche [Acke2001] und [Turo2002]) sollten diese Parameter als *Extern* gekennzeichnet werden.

Abschließen wollen wir dieses Kapitel mit zwei Modellierungsaspekten:

- Bei dem verwendeten UML-Klassendiagramm (siehe z.B. Bild 2) handelt es sich um ein semantisches Modell auf konzeptioneller Ebene. Dieses Modell trifft keinerlei Aussage über Implementierungsaspekte. Die Verwendung eines solchen Modells ist angemessen, da die Parametrisierung hauptsächlich die Datensicht betrifft. Darüber hinaus wurde schon in [Acke2001] gezeigt, dass die Verwendung eines Modells zu einer einfacheren und wirtschaftlicheren Spezifikation führt.
- Es ist denkbar, dass eine Customizing-Gruppe nur einen Teil (oder bestimmte Aspekte) eines umfassenderen Objekts darstellt. Sollte das Objekt an sich (oder andere Aspekte) in der Spezifikation einer Fachkomponente von Interesse sein, dann sollte das gesamte Objekt durch eine Klasse repräsentiert werden. Dieses beinhaltet dann als Teilmenge die ihm zugeordneten Parameter, welche durch `{C}` gekennzeichnet sind.

4 Fallstudie zur Spezifikation von Parametern

In Kapitel 3 wurden Vorschläge unterbreitet, wie Parameter und Parameterwerte von Fachkomponenten spezifiziert werden können. Anhand eines konkreten Beispiels sollen diese Vorschläge auf ihre Umsetzbarkeit hin überprüft werden.

Dazu untersuchten wir die Komponente *Flugticketverkauf*. Es handelt sich dabei um einen Teil einer Schulungsanwendung, mit der SAP verschiedene Technologien demonstriert. Die Komponente *Flugticketverkauf* stellt betriebliche Dienste zur Verfügung, die ein Reisebüro zum Verkauf von Flugtickets benötigt.

Die operativen Dienste der Komponente wurden schon in [Acke2001] spezifiziert. Diese Spezifikation diente während der Diskussionen zum Memorandum [Turo2002] als Beispiel. Der Parametrisierungsaspekt wurde damals aufgrund fehlender Vorarbeiten allerdings nicht berücksichtigt.

Im Rahmen dieser Arbeit wurde die Spezifikation der Komponente *Flugticketverkauf* um die Spezifikation der Parameter ergänzt. Dazu waren zwei Arbeitsschritte nötig: erstens die Parameter der Komponente zu ermitteln und zu analysieren, und zweitens die so ermittelten Parameter zu spezifizieren.

4.1 Ermittlung der Parameter

Im ersten Schritt wurden Customizing-Aktivitäten und einstellbare Parameter der Komponente ermittelt und klassifiziert. Dies erfolgte auf der Grundlage der Klassifikationsschemata aus [Acke2002]. (Für detaillierte Ergebnisse siehe Anhang A.) An dieser Stelle wird kurz zusammengefasst, welche Erkenntnisse sich bei diesem Arbeitsschritt ergeben haben:

- Die Klassifikationsschemata waren für das Beispiel gut geeignet.
- Zu einigen Details haben sich zusätzliche Erkenntnisse ergeben:
  - Es sind abgeleitete Attribute aufgetreten. Diese können selbst nicht gepflegt werden, sondern berechnen sich aus anderen Parameterwerten. Um die Verständlichkeit zu erhöhen, ist es sinnvoll, diese auch zu erfassen und zu spezifizieren (siehe Anhang A, Nr. 3b).
  - Es ist nicht immer sinnvoll, gesetzte Parameter ändern oder löschen zu können (siehe Anhang A, Nr. 6a).
  - Bei Parametern mit customizingabhängigem Wertebereich können Abhängigkeiten zu anderen Parametern entstehen, wenn die referenzierte Customizing-Gruppe mehr als ein Schlüsselattribut hat (siehe Anhang A, Nr. 3b).
- Wie schon bei der Fallstudie in [Acke2002] sind auch hier Bedingungen an einzelne Parameter und Abhängigkeiten zwischen Parametern aufgetreten. Diese werden im Anhang A jeweils unter Besonderheiten aufgeführt. Alle diese Bedingungen und Abhängigkeiten lassen sich mit Hilfe der OCL beschreiben. Dies bestätigt unsere Annahme in der These 8 (in Kapitel 3).

Die Analyse der Parameter der Fachkomponente *Flugticketverkauf* bestätigt die Vorgehensweise und die Klassifizierung der Parameter in [Acke2002]. Aufgrund der zusätzlichen Erkenntnisse wurden die Thesen in Kapitel 3 leicht angepasst. Dabei wurden die etwas zu restriktiven Thesen in [Acke2002] allgemeingültiger formuliert.

4.2 Spezifikation der Parameter

Im zweiten Arbeitsschritt wurden die Dienste zur Parametrisierung der Fachkomponente

Flugticketverkauf vollständig spezifiziert. Dazu haben wir die Schnittstellen der Dienste, das Verhalten der Dienste und deren Abstimmung untereinander beschrieben. Als Grundlage dienten die Abbildungsvorschriften aus Kapitel 3 und die Vorschriften aus dem Memorandum [Turo2002].

Insgesamt wurden 6 Customizing-Gruppen identifiziert, die zusammen 25 Dienste anbieten. Das Verhalten dieser Dienste wird durch 19 Invarianten, 67 Vorbedingungen und 25 Nachbedingungen beschrieben. Hinzu kommen zwei Bedingungen, welche die Reihenfolge von Dienstaufrufen einschränken. Die vollständige Spezifikation der Dienste zur Parametrisierung findet sich im Anhang B in den Abschnitten 2.1.4 (Schnittstellen), 3.7 bis 3.12 (Verhalten) und 4.6 bis 4.7 (Abstimmung).

Durch die Fallstudie wurden für die Spezifikation von Parametern folgende Erkenntnisse gewonnen:

- Die Abbildungsvorschriften aus Kapitel 3 waren insgesamt gut geeignet, um die Parameter(werte) zu beschreiben.
- Verwendet man die Abbildungsvorschriften aus Kapitel 3, so ergibt sich eine Spezifikation der Parameter, die die gleiche Struktur hat wie die Spezifikation der operativen Dienste der Fachkomponente. Die Dienste zur Parametrisierung können damit analog zu den operativen Diensten beschrieben werden (siehe auch Kapitel 5).
- Die Dienste zur Parametrisierung sind weniger komplex als die operativen Dienste. Entsprechend sind auch die meisten Bedingungen zum Verhalten sehr einfach.
- Es haben sich einige Muster oft wiederkehrender Bedingungen ergeben:
  - Schlüsselattribute identifizieren die Belegungen eindeutig. (6 Invarianten)
  - Obligatorische Parameter müssen gefüllt werden. (25 Vorbedingungen)
  - Anzulegende Belegungen dürfen noch nicht existieren. (10 Vorbedingungen)
  - Zu ändernde oder zu löschende Belegungen müssen schon existieren. (15 Vorbedingungen)
  - Methode wurde erfolgreich ausgeführt, d.h. Belegung wurde angelegt, geändert bzw. gelöscht. (25 Nachbedingungen)
- Durch die Verwendung der OMG IDL und der UML gab es einige kleinere Einschränkungen bezüglich der Notation: Bei der OMG IDL kann man nicht einzelne Parameter einer Methode als optional deklarieren, und man kann keine semantisch reicheren Datentypen definieren (wie Datum oder Uhrzeit). Mit der OCL lassen sich bestimmte semantische Beziehungen zwischen Attributen nicht ausdrücken. (Beispiel: Ein Betrag im Attribut „Währungsbetrag“ bezieht sich auf die im Attribut „Währung“ angegebene Währung.) Außerdem erlaubt OCL nicht, sich in Bedingungen auf die Exportparameter einer Methode zu beziehen. Diese Einschränkungen sind auch schon bei der Spezifikation der operativen Dienste aufgetreten und wurden in [Acke2001] ausführlicher diskutiert. Die Einschränkungen konnten meist durch Umwege umgangen werden.
- In [Acke2001] wurde bei der Spezifikation der operativen Dienste festgestellt, dass der Zeitaufwand für die Spezifikation ziemlich hoch ist. Bereinigt um Einmaleffekte (Einarbeitung in OCL, etc.) wurde geschätzt, dass der Aufwand für die Spezifikation

genauso hoch ist wie für Implementierung und herkömmliche Dokumentation zusammen. Dies hat sich auch bei der Spezifikation der Parameter bestätigt. Insbesondere die Formulierung der OCL-Bedingungen ist aufwändig und fehleranfällig.

- Die Spezifikation der Parameter(werte) ist ungefähr 30 Seiten lang und macht damit etwa ein Drittel der Gesamtspezifikation aus. Dies impliziert für einen Verwender einen entsprechenden Leseaufwand.

Zusammenfassend lässt sich feststellen, dass die Spezifikation der Parameter(werte) anhand der in Kapitel 3 vorgestellten Abbildungsvorschriften sehr gut möglich war. Zu beachten ist der hohe Arbeitsaufwand für die Erstellung. Daher sollte untersucht werden, ob man alternative Darstellungsmöglichkeiten für die oben angegebenen Muster wiederkehrender Bedingungen findet. Diese Bedingungen sind nicht nur oft aufgetreten (etwa 70% aller Bedingungen), sondern auch weitgehend trivial und selbstverständlich. Eventuell könnte auch eine Toolunterstützung sowohl beim Erstellen als auch beim Lesen der Spezifikation Erleichterung bringen.

5 Ausblick: Spezifikation des Parametrisierungsspielraums

Um eine vollständige Spezifikation einer Fachkomponente zu erhalten, müssen noch die folgenden Fragen geklärt werden:

- Wie kann die Spezifikation der Dienste zur Parametrisierung mit der Spezifikation der operativen Dienste in ein Dokument zusammengefasst werden?
- Wie kann die Spezifikation der operativen Dienste so erweitert werden, dass sie auch die Auswirkungen von Parametereinstellungen berücksichtigt?

Diese beiden Fragen wurden anhand unseres Beispiels (Fallstudie in Anhang B) bearbeitet. Die dabei gewonnenen Erkenntnisse werden in diesem Abschnitt diskutiert.

Zunächst wurde die Spezifikation der Parameter (siehe Kapitel 4) in die Spezifikation der operativen Dienste [Acke2001] integriert. Ein Ergebnis des Kapitels 4 war, dass im betrachteten Beispiel die Spezifikation der Dienste zur Parametrisierung die gleiche Struktur hat wie die Spezifikation der operativen Dienste. Die verschiedenen Aspekte der Spezifikation der Dienste zur Parametrisierung (Schnittstelle, Verhalten und Abstimmung) konnten damit auf die entsprechenden Ebenen einer Gesamtspezifikation verteilt werden.

Es ergaben sich folgende Erweiterungen an der bisherigen Spezifikation [Acke2001]:

- Die Customizing-Gruppen wurden in das UML-Klassendiagramm der Fachkomponente integriert (Anhang B, Abschnitt 3). Die einzustellenden Parameter sind anhand des Eigenschaftswertes `{C}` zu erkennen.
- Auf der Schnittstellenebene wurden im Modul *Flugticketverkauf* zusätzliche Interfaces definiert, welche die Schnittstellen der Dienste zur Parametrisierung beschreiben (Anhang B, 2.1.4).
- Verhaltens- und Abstimmungsebene wurden um die Bedingungen ergänzt, welche bei der Parametrisierung zu beachten sind (Anhang B, 3.7 bis 3.12 bzw. 4.6 und 4.7).
- Außerdem wurden die Dienste zur Parametrisierung kurz auf der Aufgabenebene beschrieben (Anhang B, Abschnitt 1.5.2) und einige zusätzliche Begriffe wurden auf der Terminologieebene hinzugefügt (Anhang B, Abschnitt 7).

Diese Änderungen waren problemlos möglich und fügen sich nahtlos in die bisherige Spezifikation ein. Die eben beschriebenen Änderungen sind allesamt Erweiterungen, welche die bisherige Spezifikation nicht veränderten.

Die Arbeitsweise der operativen Dienste hängt von Parametereinstellungen ab. Daher muss in der Spezifikation beschrieben werden, welche Auswirkungen einzelne Parameter(werte) auf die Arbeitsweise der operativen Dienste haben. Dies wurde in der Fallstudie umgesetzt und hatte folgende Auswirkungen auf die Spezifikation:

- Einige der parametrisierungsrelevanten Klassen waren schon zuvor im UML-Diagramm enthalten, jedoch waren ihre Attribute nicht änderbar. Ihre Attribute wurden jetzt als änderbar und parametrisierungsrelevant gekennzeichnet.

  Beispiel: In der früheren Spezifikation wurde davon ausgegangen, dass bestimmte Stammdaten wie *Flughafen*, *Fluggesellschaft* und *Flug* in der Fachkomponente vorhanden sind. Es wurde nicht näher ausgeführt, wie diese gepflegt werden. Es ergaben sich allerdings Bedingungen an operative Dienste, die von diesen Stammdaten abhingen. Daher wurden diese Stammdaten als Klassen ins UML-Diagramm aufgenommen und in den Bedingungen verwendet. Jetzt wurden die Attribute mit {C} gekennzeichnet und es wurden Methoden angegeben, wie die Attribute gepflegt werden können.

- Andere parametrisierungsrelevante Klassen im UML-Diagramm sind neu hinzugekommen.

  Beispiel: In der Spezifikation gibt es eine Klasse für die Preise eines Fluges. Bisher wurde davon ausgegangen, dass die Preise (analog zu den Flügen selbst) der Komponente bekannt sind. Jetzt ist es möglich, über die Parameter der CG *Preisschema* zu beeinflussen, wie die Preise ermittelt werden. Daher wurde eine neue Klasse *Preisschema* aufgenommen und eine Bedingung beschreibt, wie sich aus den Preisschemata die Preise von Flügen ergeben (siehe Anhang B, Abschnitt 3.10).

- Neben Erweiterungen gab es Verschiebungen im UML-Diagramm.

  Beispiel: Abflugzeit und Ankunftszeit sind für alle Flüge einer Flugnummer gleich und wurden deshalb zu Attributen der neu aufgenommenen Klasse *Flugnummer*.

- An den Bedingungen für Verhalten und Abstimmung der operativen Dienste gab es einige, wenige Änderungen. Es ist eine Bedingung neu hinzugekommen, drei sind entfallen und 5 mussten syntaktisch leicht verändert werden. Verglichen mit den insgesamt 60 Bedingungen waren also nur wenige Anpassungen notwendig.

- Auf den anderen Spezifikationsebenen waren im untersuchten Beispiel keine Änderungen zu verzeichnen.

Zusammenfassend kann man feststellen, dass beim betrachteten Beispiel die Auswirkungen von Parametereinstellungen gut in der Spezifikation der operativen Dienste abgebildet werden konnten. Besonders hervorzuheben ist dabei, dass keine Änderungen oder Erweiterungen an der Spezifikationstechnik notwendig waren.

Die Untersuchungen in diesem Kapitel wurden anhand der konkreten Fachkomponente *Flugticketverkauf* durchgeführt. Diese Komponente ist jedoch nicht unbedingt repräsentativ für beliebige Fachkomponenten. So bestand die Beispielkomponente weitgehend aus Parametern für Stammdaten und Steuerdaten, enthielt aber keine Parameter zur Auswahl von

Prozessvarianten. Aus unserer Sicht ist es daher notwendig, dass die Auswirkungen von Parametern systematisch untersucht werden. Die Ergebnisse dieses Kapitels bieten dafür einen ersten Anhaltspunkt.

6 Zusammenfassung und Ausblick

Fachkomponenten sollten so erstellt werden, dass ein Verwender sie in gewissem Rahmen an seine Bedürfnisse anpassen kann. Parametrisierung ist eine Technik, die dem Verwender dies ermöglicht. Sieht der Hersteller vor, dass eine Fachkomponente parametrisierbar ist, so muss der Parametrisierungsspielraum auch spezifiziert werden.

In [Acke2002] wurde vorgeschlagen, welche Aspekte bei der Spezifikation von Parametern zu berücksichtigen sind. Darauf aufbauend haben wir im Kapitel 3 Abbildungsvorschriften formuliert, wie diese Aspekte mit Hilfe der OMG UML spezifiziert werden können. Anhand einer Fallstudie wurde festgestellt, dass die Spezifikation der Parameter mit den Abbildungsvorschriften sehr gut möglich ist (siehe Kapitel 4). Die einzige Einschränkung liegt im hohen Arbeitsaufwand bei der Erstellung. Im Kapitel 5 wurden erste Erkenntnisse vorgestellt, wie die Auswirkungen der Parametereinstellungen spezifiziert werden können.

Zukünftige Forschungen sollten sich mit der Frage befassen, wie sich Parameter auf die operativen Dienste einer Fachkomponente auswirken. Außerdem wären Untersuchungen wünschenswert, wie der Aufwand beim Erstellen der Spezifikation verringert werden kann. Außerdem muss die Spezifikation des Parametrisierungsspielraums noch in das Memorandum „Vorschlag zur Vereinheitlichung der Spezifikation von Fachkomponenten“ [Turo2002] integriert werden.

Literatur

[Acke2001] *Ackermann, J.*: Fallstudie zur Spezifikation von Fachkomponenten. In: *K. Turowski (Hrsg.)*: Modellierung und Spezifikation von Fachkomponenten. 2. Workshop, Bamberg 2001, S. 1 – 66.

[Acke2002] *Ackermann, J.*: Spezifikation des Parametrisierungsspielraums von Fachkomponenten – Erste Überlegungen. In: *K. Turowski (Hrsg.)*: Modellierung und Spezifikation von Fachkomponenten. 3. Workshop, Nürnberg 2002, S. 17 – 68.

[BRS+2000] *Bergner, K.; Rausch, A.; Sihling, M.; Vilbig, A.:* Adaptation Strategies in Componentware. In: Proceedings 2000 Australian Software Engineering Conference. IEEE Computer Society 2000, S. 87 – 95.

[CoTu2000] *Conrad, S.; Turowski, K.*: Vereinheitlichung der Spezifikation von Fachkomponenten auf der Basis eines Notationsstandards. In: Tagungsband Modellierung 2000. St. Goar 2000, S. 179 – 194.

[FeLT2001] *Fettke, P.; Loos, P.; von der Tann, M.*: Eine Fallstudie zur Spezifikation von Fachkomponenten eines Informationssystems für Virtuelle Finanzdienstleister – Beschreibung und Schlussfolgerungen. In: *K. Turowski (Hrsg.)*: Modellierung und Spezifikation von Fachkomponenten. 2. Workshop, Bamberg 2001, S. 75 – 94.

[FlGu1996] *Floch, J.; Gulla, B.:* Enabling Reuse with a Configuration Language. In: Proceedings of Fourth International Conference on Software Reuse. IEEE Computer Society, Los Alamitos 1996, S. 176 – 185.

[FSH+1998] *Ferstl, O.K.; Sinz, E.J.; Hammel, C.; Schlitt, M.; Wolf, S.; Popp, K.; Kehlenbeck, R.; Pfister, A.; Kniep, H.; Nielsen, N.; Seitz, A.:* WEGA – Wiederverwendbare und erweiterbare Geschäftsprozeß- und Anwendungssystemarchitekturen. Abschlussbericht des Verbundprojektes. Walldorf 1998.

[JaGJ1997] *Jacobson, I.; Griss, M.; Jonsson, P.:* Software Reuse. ACM Press/Addison Wesley Longman, New York 1997.

[OMG2001] *OMG (Hrsg.):* Unified Modeling Language Specification: Version 1.4, September 2001. http://www.omg.org/technology/documents/formal/uml.htm, Abruf am 2001-12-18.

[Saak1993] *Saake, G.*: Objektorientierte Spezifikation von Informationssystemen. Teubner Verlag, Stuttgart 1993.

[SAP1997] *SAP (Hrsg.*): R/3-Referenz(prozeß)modell 4.0 im R/3 Business Engineer – Zielsetzung, Inhalte, Vorgehensweise. Walldorf 1997.

[Schü1998] *Schütte, R.:* Grundsätze ordnungsmäßiger Referenzmodellierung. Gabler Verlag, Wiesbaden 1998.

[StCr1998] *Stiemerling, O.; Cremers, A. B.:* Tailorable Component Architectures for CSCW-Systems. In: Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Programming. IEEE Press, Madrid 1998, S. 302-308.

[Szyp1998] Component Software: Beyond Object-Oriented Programming. 2. Aufl., Addison-Wesley, Harlow 1998.

[Turo2002] *Turowski, K. (Hrsg.).:* Vorschlag zur Vereinheitlichung der Spezifikation von Fachkomponenten. Memorandum des Arbeitskreises 5.10.3 der Gesellschaft für Informatik. Augsburg 2002.

Anhang A: Analyse des Customizings der Fachkomponente „Flugticketverkauf“

Im Rahmen einer Fallstudie wurde eine Komponente *Flugticketverkauf* spezifiziert. Besonderer Augenmerk lag dabei auf der Spezifikation der Parameter und möglicher Parameterwerte der Komponente. Dazu wurden in einem ersten Schritt die Parameter ermittelt und klassifiziert. Die Ergebnisse dazu finden sich hier im Anhang A. In einem zweiten Schritt wurde dann die Spezifikation anhand der Abbildungsregeln in Kapitel 3 angefertigt. Die Spezifikation der Komponente finden sich im Anhang B.

SAP R/3 kann mit Hilfe von sogenannten Customizing-Aktivitäten (CA) parametrisiert werden. Customizing-Aktivitäten fassen alle die elementaren Einstellungen zusammen, die zum selben betriebswirtschaftlichen Kontext gehören und zusammen durchzuführen sind. In [Acke2002] wurde untersucht, welche Aspekte von CAs für die Spezifikation des Customizings von Interesse sind. Dazu wurden Klassifikationsschemata entwickelt, welche die Eigenschaften von CAs und der einzustellenden Parameter beschreiben. Die Klassifikationsschemata finden sich in den Tabellen A-1 bis A-3 am Ende des Anhangs. Für weitere Erläuterungen zu den Merkmalen und Merkmalsausprägungen verweisen wir auf [Acke2002].

Im ersten Schritt der Fallstudie wurde das Customizing der Komponente *Flugticketverkauf* analysiert. Die Funktionalität zum *Flugticketverkauf* ist Teil einer Schulungsanwendung, mit der SAP verschiedene Technologien anhand eines einfachen betriebswirtschaftlichen Beispiels demonstriert und vermittelt. Die Funktionalität existiert seit Release 6.10, wurde aber eigentlich nicht als Komponente implementiert. Sie ist jedoch weitgehend gekapselt und kann deshalb als Fachkomponente betrachtet werden. Es handelt sich um ein real existierendes Beispiel. Die Komponente ist unabhängig von der Fallstudie entstanden und wurde nicht extra für die Fallstudie erstellt.

Da es sich um eine Beispielanwendung handelt, bei der das Customizing nicht im Vordergrund steht, gibt es bisher keine expliziten Customizing-Aktivitäten (CAs) zur Pflege der Parameter. Die Parametrisierung der Komponente erfolgt derzeit über direktes Setzen von Werten auf Datenbankebene. Um die Parameter der Komponente einfacher anhand der Schemata A-1 bis A-3 klassifizieren zu können, werden die möglichen Einstellungen im Anhang A trotzdem als Customizing-Aktivitäten dargestellt.

Bei der Analyse des Customizings der Komponente wurden insgesamt 6 Customizing-Aktivitäten identifiziert. Dabei handelt es sich um drei einfache CAs und 3 komplexe CAs, wobei diese wiederum aus insgesamt 7 einfachen CAs bestehen. Im Folgenden werden die CAs mit ihren Eigenschaften im Detail dargestellt.

Jede der Customizing-Aktivitäten (CA) wird anhand des folgenden Schemas beschrieben:

Nr *Name der Customizing-Aktivität*
- Beschreibung
- Zusammenhang
- Klassifizierung

- Parameter
- Beziehung
- Besonderheiten

Die CAs wurden durchnummeriert. Dies erhöht die Übersichtlichkeit und erlaubt einfache Referenzen auf andere CAs. Unter *Beschreibung* wird kurz erklärt, welche fachliche Bedeutung die CA hat. Unter *Zusammenhang* wird auf eventuelle Verbindungen zu anderen CAs verwiesen. Unter *Klassifizierung* werden die Eigenschaften der CA anhand des „Klassifikationsschemas für einfache CAs“ beschrieben (vergleiche Tabelle A-1). Unter *Parameter* werden alle Parameter der CA aufgeführt. Die Schlüsselparameter werden durch „Key“ gekennzeichnet. Außerdem werden die Parameter anhand des „Klassifikationsschemas für Parameter“ eingeordnet (vergleiche Tabelle A-2). Der Kürze halber wird auf die parameterspezifische Angabe der Eigenschaften verzichtet; die Klassifizierung erfolgt summiert für alle Parameter der CA. Hat die CA Parameter mit customizingabhängigem Wertebereich, dann werden diese unter *Beziehung* näher untersucht. Es wird angegeben, welche CA den Wertebereich für diesen Parameter vorgibt. Außerdem werden die so entstandenen Beziehungen anhand des „Klassifikationsschemas für die Beziehungen zwischen CAs, die durch einen customizingabhängigen Wertebereich entstehen“ eingeordnet (vergleiche Tabelle A-3). Unter *Besonderheiten* werden weitere Bedingungen und Abhängigkeiten aufgeführt, die in einer Spezifikation zu berücksichtigen sind. Bei komplexen CAs werden die einzelnen Teilschritte aufgeführt und jeder der Teilschritte wird anhand des oben genannten Schemas beschrieben.

1 *Flughäfen pflegen*
- Beschreibung: Es werden mögliche Flughäfen mit Kürzel und Beschreibung gepflegt.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Stammdaten
    - Anzahl von Entitäten: Beliebig
    - Abhängigkeiten: Keine
- Parameter
    - Kürzel (Key), Name, Stadt, Land
    - Wertebereich: 1x Festwerte (Land), 3x Beliebig
    - Notwendigkeit: 4x obligatorisch
- Besonderheiten: keine

2 *Fluggesellschaften pflegen*
- Beschreibung: Es werden Fluggesellschaften mit Kürzel, Beschreibung und Eigenschaften gepflegt.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Stammdaten
    - Anzahl von Entitäten: Beliebig
    - Abhängigkeiten: Keine
- Parameter
    - Kürzel (Key), Name, Währung
    - Wertebereich: 1x Festwerte (Währung), 2x Beliebig
    - Notwendigkeit: 3x obligatorisch
- Besonderheiten: keine

3 *Flüge definieren*

- Beschreibung: Hier werden die Eigenschaften von Flügen definiert. Dazu zählen z.B. Abflug- und Ankunftsort sowie die Daten, an denen die Flüge stattfinden.
- Es handelt sich um eine komplexe CA mit 2 Teilschritten.

3a *Flugnummern definieren*

- Beschreibung: Hier werden Flugnummern (z.B. LH 400) definiert und die Eigenschaften von Flügen angegeben, die bei allen Flügen dieser Flugnummer gleich sind.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Stammdaten
    - Anzahl von Entitäten: Beliebig
    - Abhängigkeiten: Innerhalb des Bereichs
- Parameter
    - Fluggesellschaft (Key), Flugnummer (Key), Abflugort, Abflugzeit, Ankunftsort, Ankunftszeit, Ankunft Tage später
    - Wertebereich: 3x Customizingabhängig (Fluggesellschaft, Abflugort, Ankunftsort), 4x Beliebig
    - Notwendigkeit: 1x optional mit Default (Ankunft Tage später), 6x obligatorisch
- Beziehung 1
    - Wertebereich der Fluggesellschaft wird durch die CA 2 vorgegeben
    - Semantischer Charakter: Gliederung / Strukturierung
    - Art: Hierarchie
    - Abhängigkeit: Innerhalb des Bereichs
- Beziehung 2-3
    - Wertebereich von Abflugort und Ankunftsort werden durch die CA 1 vorgegeben
    - Semantischer Charakter: beide Vorschrift
    - Art: beide Obligatorische Referenz
    - Abhängigkeit: beide Innerhalb des Bereichs
- Besonderheiten:
    - Der Abflugort ist ungleich dem Ankunftsort.

3b *Abflugdaten definieren*

- Beschreibung: Hier wird definiert, an welchen Tagen Flüge einer Flugnummer stattfinden.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Stammdaten
    - Anzahl von Entitäten: Beliebig
    - Abhängigkeiten: Innerhalb des Bereichs
- Parameter
    - Fluggesellschaft (Key), Flugnummer (Key), Abflugdatum (Key), Standardpreis, Steuer
    - Wertebereich: 2x Customizingabhängig (Fluggesellschaft, Flugnummer), 3x Beliebig
    - Notwendigkeit: 5x obligatorisch
- Beziehung 1-2
    - Wertebereiche von Fluggesellschaft und Flugnummer werden zusammen durch die CA 3a vorgegeben
    - Semantischer Charakter: Gliederung / Strukturierung

    - Art: Hierarchie
    - Abhängigkeit: Innerhalb des Bereichs
- Besonderheiten:
    - Es sind nur solche Flugnummern erlaubt, die für die entsprechende Fluggesellschaft definiert wurden (zusammengesetzter Schlüssel).
    - Zum Flug gehören darüber hinaus noch die Eigenschaften Ankunftsdatum und Währung. Diese werden allerdings nicht gepflegt, sondern ergeben sich folgendermaßen:
        - Die Währung von Standardpreis und Steuer ist die Währung der Fluggesellschaft.
        - Der Parameter *Ankunft Tage später* der zugehörigen Flugnummer beschreibt, wie viele Tage das Ankunftsdatum nach dem Abflugdatum liegt.

4 *Preisschemata definieren*
- Beschreibung: Die Flugpreise für die einzelnen Kategorien (Economy, Business, First Class) sowie die Ermäßigungen errechnen sich durch Auf- und Abschläge von einem Standardpreis. Hier können Preisschemata definiert werden, in welchen abstrakt die Faktoren für Auf- und Abschläge festgelegt werden.
- Zusammenhang: In der CA 5 werden die Preisschemata Fluggesellschaften, Flugnummern oder einzelnen Flügen zugewiesen.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Strategische Steuerdaten
    - Anzahl von Entitäten: Beliebig
    - Abhängigkeiten: Keine
- Parameter
    - Nummer (Key), Faktor Business, Faktor First, Faktor Kind, Faktor Kleinkind
    - Wertebereich: 5x Beliebig
    - Notwendigkeit: 4x optional mit Default, 1x obligatorisch (Nummer)
- Besonderheiten: Die Parameter *Faktor Business* und *Faktor First* sind größer oder gleich eins und die Parameter *Faktor Kind* und *Faktor Kleinkind* sind kleiner oder gleich eins.

5 *Preisschema zuordnen*
- Beschreibung: Hier kann ein Preisschema zu Fluggesellschaften, Flugnummern und einzelnen Flügen zugeordnet werden. Bei der Berechnung des Preises wird in der folgenden Reihenfolge nach einem gültigen Preisschema gesucht: Flug, Flugnummer, Fluggesellschaft.
- Es handelt sich um eine komplexe CA mit 3 Teilschritten.

5a *Preisschema für Fluggesellschaften festlegen*
- Beschreibung: Hier wird jeder Fluggesellschaft ein Preisschema zugeordnet, welches als Standardschema für diese Fluggesellschaft dient.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Administrative Steuerdaten
    - Anzahl von Entitäten: Abhängig von anderer/n CA
    - Abhängigkeiten: Innerhalb des Bereichs
- Parameter

    - Fluggesellschaft (Key), Preisschema
    - Wertebereich: 2x Customizingabhängig
    - Notwendigkeit: 2x obligatorisch
- Beziehung 1
    - Wertebereich der Fluggesellschaft wird durch die CA 2 vorgegeben
    - Semantischer Charakter: Gliederung / Strukturierung
    - Art: Aggregation
    - Abhängigkeit: Innerhalb des Bereichs
- Beziehung 2
    - Wertebereich des Preisschemas wird durch die CA 4 vorgegeben
    - Semantischer Charakter: Vorschrift
    - Art: Obligatorische Referenz
    - Abhängigkeit: Innerhalb des Bereichs
- Bemerkung: Jeder Fluggesellschaft muss ein Preisschema zugeordnet werden.

5b *Preisschema für Flugnummern festlegen*
- Beschreibung: Hier kann einer Flugnummer (einer Fluggesellschaft) ein Preisschema zugeordnet werden.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Administrative Steuerdaten
    - Anzahl von Entitäten: Abhängig von anderer/n CA
    - Abhängigkeiten: Innerhalb des Bereichs
- Parameter
    - Fluggesellschaft (Key), Flugnummer (Key), Preisschema
    - Wertebereich: 3x Customizingabhängig
    - Notwendigkeit: 3x obligatorisch
- Beziehung 1-2
    - Wertebereiche von Fluggesellschaft und Flugnummer werden zusammen durch die CA 3a vorgegeben
    - Semantischer Charakter: Gliederung / Strukturierung
    - Art: Aggregation
    - Abhängigkeit: Innerhalb des Bereichs
- Beziehung 3
    - Wertebereich des Preisschemas wird durch die CA 4 vorgegeben
    - Semantischer Charakter: Vorschrift
    - Art: Obligatorische Referenz
    - Abhängigkeit: Innerhalb des Bereichs
- Besonderheiten:
    - Es muss nicht jeder Flugnummer ein Preisschema zugeordnet werden.
    - Es sind nur solche Flugnummern erlaubt, die für die entsprechende Fluggesellschaft definiert wurden (zusammengesetzter Schlüssel).

5c *Preisschema für Flüge definieren*
- Beschreibung: Hier kann einem einzelnen Flug ein Preisschema zugeordnet werden.
- Klassifizierung
    - Betriebswirtschaftlicher Zweck: Administrative Steuerdaten
    - Anzahl von Entitäten: Abhängig von anderer/n CA
    - Abhängigkeiten: Innerhalb des Bereichs

- Parameter
  - Fluggesellschaft (Key), Flugnummer (Key), Abflugdatum (Key), Preisschema
  - Wertebereich: 4x Customizingabhängig
  - Notwendigkeit: 4x obligatorisch
- Beziehung 1-3
  - Wertebereiche von Fluggesellschaft, Flugnummer und Abflugdatum werden zusammen durch die CA 3b vorgegeben
  - Semantischer Charakter: Gliederung / Strukturierung
  - Art: Aggregation
  - Abhängigkeit: Innerhalb des Bereichs
- Beziehung 4
  - Wertebereich des Preisschemas wird durch die CA 4 vorgegeben
  - Semantischer Charakter: Vorschrift
  - Art: Obligatorische Referenz
  - Abhängigkeit: Innerhalb des Bereichs
- Besonderheiten:
  - Es muss nicht jeder Flugnummer ein Schema zugeordnet werden.
  - Es sind nur solche Flugnummern und Abflugdaten erlaubt, die für die entsprechende Fluggesellschaft definiert wurden (zusammengesetzter Schlüssel).

6 *Flugverbindungen pflegen*
- Beschreibung: Hier werden die Eigenschaften von Flugverbindungen definiert. Eine Flugverbindung ist ein Angebot eines Reisebüros. Sie bezieht sich auf einen Startort, einen Zielort und eine bestimmte Abflugzeit (Datum, Uhrzeit). Sie besteht aus einer oder mehreren Teilstrecken.
- Es handelt sich um eine komplexe CA mit 2 Teilschritten.

6a *Flugverbindungen definieren*
- Beschreibung: Hier werden für ein Reisebüro die Flugverbindungen definiert.
- Klassifizierung
  - Betriebswirtschaftlicher Zweck: Stammdaten
  - Anzahl von Entitäten: Beliebig
  - Abhängigkeiten: Außerhalb des Bereichs
- Parameter
  - Reisebüronummer (Key), Verbindungsnummer (Key)
  - Wertebereich: 1x Customizingabhängig (Reisebüronummer), 1x Beliebig
  - Notwendigkeit: 2x obligatorisch
- Beziehung 1
  - Wertebereich des Reisebüros wird durch eine CA außerhalb der Komponente vorgegeben
  - Semantischer Charakter: Gliederung / Strukturierung
  - Art: Aggregation
  - Abhängigkeit: Außerhalb des Bereichs
- Besonderheiten: Diese CA enthält nur Schlüsselfelder. Änderungen an erfassten Entitäten sind daher nicht sinnvoll, sie können nur wieder gelöscht und neu erfasst werden.

6b *Teilstrecken festlegen*
- Beschreibung: Hier wird definiert, aus welchen Teilstrecken eine Flugverbindung besteht.

- Klassifizierung
  - Betriebswirtschaftlicher Zweck: Stammdaten
  - Anzahl von Entitäten: Beliebig
  - Abhängigkeiten: Innerhalb des Bereichs
- Parameter
  - Reisebüronummer (Key), Verbindungsnummer (Key), Teilstreckennummer, Fluggesellschaft, Flugnummer, Abflug Tage später
  - Wertebereich: 4x Customizingabhängig, 2x Beliebig (Teilstreckennummer, Abflug Tage später)
  - Notwendigkeit: 6x obligatorisch
- Beziehung 1-2
  - Wertebereiche von Reisebüro und Verbindungsnummer werden zusammen durch die CA 6a vorgegeben
  - Semantischer Charakter: Gliederung / Strukturierung
  - Art: Hierarchie
  - Abhängigkeit: Innerhalb des Bereichs
- Beziehung 3-4
  - Wertebereiche von Fluggesellschaft und Flugnummer werden zusammen durch die CA 3a vorgegeben
  - Semantischer Charakter: Aufbau von Zuordnungen
  - Art: Obligatorische Referenz
  - Abhängigkeit: Innerhalb des Bereichs
- Besonderheiten:
  - Es sind nur solche Verbindungsnummern erlaubt, die für das entsprechende Reisebüro definiert wurden (zusammengesetzter Schlüssel).
  - Es sind nur solche Flugnummern erlaubt, die für die entsprechende Fluggesellschaft definiert wurden (zusammengesetzter Schlüssel).
  - Eine Flugverbindung muss mindestens eine Teilstrecke haben.
  - Soll eine Flugverbindung n Teilstrecken haben, dann müssen die Teilstrecken-nummern 1 bis n vergeben werden.
  - Bei den Teilstrecken zu einer Flugverbindung muss es sich um verschiedene Flugnummern handeln, d.h. eine Flugnummer kann nicht in mehreren Teilstrecken derselben Flugverbindung verwendet werden.
  - Das Attribut *Abflug Tage später* der Teilstrecke m muss größer oder gleich dem Attribut *Abflug Tage später* der Teilstrecke m-1 sein.
  - Ist das Attribut *Abflug Tage später* der Teilstrecke m gleich dem Attribut *Abflug Tage später* der Teilstrecke m-1, dann muss die Ankunftszeit der Teilstrecke m-1 vor der Abflugszeit der Teilstrecke m sein.
  - Der Abflugort der Teilstrecke m muss gleich dem Ankunftsort der Teilstrecke m-1 sein.

Quantitativ wurden in der Fallstudie 10 einfache Customizing-Aktivitäten untersucht (3 einfache CAs und 3 komplexe CAs, die sich aus insgesamt 7 einfachen CAs zusammensetzen). Diese haben 41 Parameter, von denen 19 Parameter einen customizingabhängigen Wertebereich haben. Abschließend wird noch die quantitative Verteilung der Eigenschaften anhand der entsprechenden Klassifikationsschemata vorgestellt.

- Die Eigenschaften der 10 CAs verteilen sich wie folgt:

MERKMAL	MERKMALSAUSPRÄGUNG			
Bwl. Zweck	Org. Einheiten / Stammdaten (6)	Strategische Steuerdaten (1)	Administrative Steuerdaten (3)	Benutzersteuerung / Darstellung (0)
Max. Anzahl Ausprägungen	Eine (0)	Feste Anzahl (0)	Abhängig von anderen CA (3)	Beliebig (7)
Abhängigkeiten	Keine (3)	Innerhalb des Bereichs (6)	Außerhalb des Bereichs (1)	Innerhalb und außerhalb des Bereichs (0)

Tabelle A-1: Klassifikationsschema für einfache Customizing-Aktivitäten

- Die Eigenschaften der 41 Parameter verteilen sich wie folgt:

MERKMAL	MERKMALSAUSPRÄGUNG			
Wertebereich	Boolean (0)	Festwerte (2)	Customizing-abhängig (19)	Beliebig (20)
Notwendigkeit	Optional (0)	Optional mit Default (5)	Obligatorisch (36)	

Tabelle A-2: Klassifikationsschema für Parameter

- Die Eigenschaften der 19 Beziehungen, die aufgrund der Parameter mit customizingsabhängigem Wertebereich entstanden sind, verteilen sich wie folgt:

MERKMAL	MERKMALSAUSPRÄGUNG				
Semantischer Charakter	Gliederung / Strukturierung (12)	Aufbau von Zuordnungen (2)	Vorschrift (5)	Klassifizierung (0)	Prozessintegration (0)
Art	Hierarchie (5)	Aggregation (7)	Obligatorische Referenz (7)	Optionale Referenz (0)	
Abhängigkeiten	Innerhalb des Bereichs (18)	Außerhalb des Bereichs (1)			

Tabelle A-3: Klassifikationsschema für die Beziehungen zwischen CAs, die durch einen customizingabhängigen Wertebereich entstehen

Anhang B: Spezifikation der Fachkomponente „Flugticketverkauf“

Der Anhang B enthält die vollständige Spezifikation der Fachkomponente „Flugticketverkauf“. Die Spezifikation der operativen Dienste erfolgte anhand der Vorgaben aus dem Memorandum [Turo2002]. Die Spezifikation des Parametrisierungsspielraums erfolgte anhand der Ergebnisse dieses Beitrags.

Im Anhang B werden zwei Schriftarten verwendet:

- Normal: Spezifikation der Komponente an sich. Die Normalschrift kennzeichnet den Teil, den ein Komponentenintegrator sehen würde
- *Kursiv: Erfahrungen und offene Punkte bei der Spezifikation. Dieser Teil ist für den Arbeitskreis interessant und zeigt, wie gut die bisherigen Konzepte umgesetzt werden können und an welchen Stellen weiterer Ideenbedarf besteht.*

Inhalt

1 Aufgabenebene

1.1 Funktionalität der Fachkomponente

Die Fachkomponente „Flugticketverkauf“ stellt Dienste zur Verfügung, die ein Reisebüro zum Verkauf von Flugtickets benötigt. Dazu zählen Verwaltung und Auswahl von Flugverbindungen und Verwaltung und Verkauf von Flugreisen. Für eine genaue Liste der unterstützten betrieblichen Aufgaben siehe Abschnitt 1.2.

Die Komponente enthält keine Funktionen zur Kundenverwaltung, da davon ausgegangen wird, dass ein Reisebüro schon ein Kundenverwaltungsprogramm im Einsatz hat. Eine solche Kundenverwaltung außerhalb der hier angebotenen Komponente wird allerdings vorausgesetzt.

Des weiteren ist eine Verbindung zu einem entsprechenden Flugbuchungssystem der Fluggesellschaften notwendig, damit für die verkauften Flugreisen auch die Plätze bei den Fluggesellschaften reserviert werden können.

Die Komponente bietet ihre Dienste nur als Programmierschnittstellen an und beinhaltet keine Benutzeroberfläche. Ein Verwender muss sich diese selbst erstellen.

Die Komponente bietet verschiedene Möglichkeiten zur initialen Datenpflege und Parametrisierung. Diese Aspekte wurden ebenfalls spezifiziert.

1.2 Unterstützte betriebliche Aufgaben

Die Fachkomponente „Flugticketverkauf“ unterstützt die folgenden betrieblichen Aufgaben eines Reisebüros:

- Angebotserstellung für Flugreisen
- Verkauf von Flugreisen
- Verwaltung von Flugreisen
- Verwaltung von Flugverbindungen

Die folgende Tabelle zeigt, welche Softwaredienste für die Erfüllung der betrieblichen Aufgaben zur Verfügung stehen. Dabei handelt es sich einerseits um Dienste der Fachkomponente selbst, andererseits um Dienste, die von anderen Komponenten zu erbringen sind. Die angebotenen Dienste der Fachkomponente werden in den Abschnitten 1.4 und 1.5 näher erläutert.

Betriebliche Aufgabe	Dienst der Komponente	Externer Dienst
Angebotserstellung für Flugreisen	Flugverbindung::LiefereListe Flugverbindung::LiefereDetails	Flugverfügbarkeit::Check
Verkauf von Flugreisen	Flugreise::Anlegen	Flugkunde::PrüfeExistenz Flugkunde::Anlegen Flugbuchung::Anlegen
Verwaltung von Flugreisen	Flugreise::LiefereListe	Reisebüro::PrüfeExistenz Flugkunde::PrüfeExisten

		z
Verwaltung von Flugverbindungen	Flugverbindung::LiefereListe	Reisebüro::PrüfeExistenz

1.3 Verwendungsmöglichkeiten

Die Fachkomponente „Flugticketverkauf“ ist für folgende Benutzergruppen in den aufgeführten Konstellationen von Interesse:

- Verwendung durch ein Reisebüro
- Verwendung durch eine Fluggesellschaft für den Bereich Flugticketverkauf
  Da die Hauptzielgruppe dieser Komponente Reisebüros sind, wird in diesem Dokument immer der Begriff „Reisebüro“ verwendet. Die Komponente ist aber auch für Fluggesellschaften interessant. In diesem Falle sollten die verwendeten Begriffe folgendermaßen verstanden werden: Reisebüro = Verkaufsbereich der Fluggesellschaft; Fluggesellschaft = Flugbetriebsbereich der Fluggesellschaft
- Verwendung durch die Reiseabteilung eines Unternehmens

Es ist möglich, die Fachkomponente für mehrere Reisebüros einzusetzen. Dadurch ergeben sich auch folgende Verwendungsmöglichkeiten:

- gemeinsame Verwendung durch mehrere Reisebüros
- Verwendung durch einen Application Service Provider, der diese Funktionalität über das Internet mehreren Reisebüros anbietet

1.4 Beschreibung der wichtigsten Entitätstypen und ihrer Aufgaben

Die wichtigsten Entitätstypen der Fachkomponente „Flugticketverkauf“ sind die Flugverbindung und die Flugreise. Die angebotenen Dienste beziehen sich auch auf diese beiden Entitätstypen. In diesem Abschnitt werden Definition, Aufgaben und Besonderheiten der beiden Entitätstypen vorgestellt.

1.4.1 Entitätstyp Flugverbindung

Mit *Flugverbindung* wird eine Strecke und eine Abflugzeit beschrieben, für welche ein Reisebüro Beförderungsleistungen verkauft. Eine Flugverbindung bezieht sich auf einen Startort und einen Zielort und auf eine bestimmte Abflugzeit (Datum, Uhrzeit). Eine Flugverbindung besteht aus einer oder mehreren Teilstrecken.

Eine Flugverbindung wird identifiziert durch: Reisebüronummer, Verbindungsnummer, Datum

Eigenschaften einer Flugverbindung sind: Abflugort, Ankunftsort, Abflugzeit, Ankunftszeit, Ticketpreis und Währung, Liste der Teilstrecken (Fluggesellschaft, Flugnummer, Datum), insgesamt vorhandene und freie Plätze pro Teilstrecke. (Bei allen angegebenen Zeiten handelt es sich jeweils um die lokalen Zeiten der Flughäfen.)

Jede Teilstrecke bezieht sich auf genau einen Flug einer Fluggesellschaft. In einer Flugverbindung können auch Flüge verschiedener Fluggesellschaften kombiniert werden.

Beispiel:

Reisebüro 110, Verbindung 27, 10.08.2001; von Berlin-Tegel (TXL) nach New York (JFK); Abflug 7.10 MEZ, Ankunft 10.30 ET; Preis 1800 DEM (Economy)

- 1. Teilstrecke: von TXL nach FRA mit LH 2407; Abflug 7.10 Uhr MEZ, Ankunft 8.15 Uhr MEZ; Economy: 120 Plätze gesamt, noch 44 freie Plätze
- 2. Teilstrecke: von FRA nach JFK mit LH 400; Abflug 9.30 MEZ, Ankunft 10.30 ET; Economy: 350 Plätze gesamt, noch 134 freie Plätze

Zum Entitätstyp Flugverbindung stehen die Dienste *LiefereListe* und *LiefereDetails* zur Verfügung. Die Definition, welche Flugverbindungen vom Reisebüro verkauft werden können, erfolgt zur Konfigurationszeit. (Es ist vorstellbar, in einer späteren Version eine komfortablere Pflegevariante zur Verfügung zu stellen.)

1.4.2 Entitätstyp Flugreise

Die Buchung einer *Flugreise* ist ein Vertrag zwischen einem Flugkunden und einem Reisebüro. Eine Flugreise besteht aus einem Hinflug und (optional) einem Rückflug. Eine Flugreise kann mehrere Passagiere umfassen. Durch die Buchung einer Flugreise erwirbt der Flugkunde das Recht, dass alle benannten Passagiere auf dem Hinflug und (optional) dem Rückflug befördert werden. Der Flugkunde bezahlt dafür einen Preis.

Eine Flugreise wird identifiziert durch: Reisebüronummer, Reisenummer

Eigenschaften einer Flugreise sind zum z.B. Kundennummer, Verbindungen für Hin- und Rückflug, Flugklasse, Anzahl der Passagiere, Gesamtpreis und Währung, sowie Name und Geburtsdatum der Passagiere.

Hin- und Rückflug einer Flugreise beziehen sich auf jeweils eine Flugverbindung des Reisebüros. Es können sowohl Hin- als auch Rückflug aus mehreren Teilstrecken zusammengesetzt sein.

Beim Buchen einer Flugreise wird vom Reisebüro für jede Teilstrecke für jeden Passagier eine Flugbuchung bei der entsprechenden Fluggesellschaft ausgeführt. Dadurch werden die notwendigen Platzreservierungen vorgenommen.

Beispiel:
Reisebüro 110, Reisenummer 4711; Kunde 1868; Hinflug: Verbindung 27 (TXL – JFK) am 10.08.2001, kein Rückflug; Economy Class; 2 Erwachsene; Preis: 3600 DEM;
Passagier 1: Herr Max Mustermann, 01.01.1960; Passagier 2: Frau Maxime Mustermann, 31.12.1959

Zum Entitätstyp Flugreise stehen die Dienste *LiefereListe* und *Anlegen* zur Verfügung. (Prinzipiell wäre auch ein Dienst *Stornieren* wünschenswert. Dieser ist jedoch erst für eine spätere Version der Fachkomponente geplant.)

1.5 Überblick über angebotene und erwartete Dienste

1.5.1 Angebotene Dienste

Die Komponente *Flugticketverkauf* bietet eine Reihe von Diensten an, mit welchen potentielle Clients die Funktionalität der Komponente nutzen können.

1. `Flugticketverkauf::Flugverbindung::LiefereListe`
   - Dieser Dienst liefert eine Liste aller von einem Reisebüro angebotenen Flugverbindungen. Dabei kann optional die Selektion auf einen bestimmten Abflugort, Ankunftsort, Datumsbereich und/oder Fluggesellschaft eingeschränkt werden.
2. `Flugticketverkauf::Flugverbindung::LiefereDetails`
   - Dieser Dienst liefert zu einer konkreten Flugverbindung weitere Details, insbesondere die Verfügbarkeit auf allen Teilstrecken. Dazu muss die gewünschte Flugverbindung angegeben werden.
3. `Flugticketverkauf::Flugreise::Anlegen`
   - Dieser Dienst ermöglicht das Anlegen einer neuen Flugreise. Dazu müssen vom Client alle notwendigen Informationen bereitgestellt werden. Das Anlegen einer Flugreise beinhaltet insbesondere, dass bei den entsprechenden Fluggesellschaften für jede Teilstrecke und jeden Passagier eine `Flugbuchung` durchgeführt wird.
4. `Flugticketverkauf::Flugreise::LiefereListe`
   - Dieser Dienst liefert eine Liste aller von einem Reisebüro verkauften Flugreisen. Dabei kann optional die Selektion auf einen bestimmten Flugkunden, Datumsbereich für Hinflug und/oder Datumsbereich für Reisebuchung eingeschränkt werden.

Eine ausführliche Dokumentation der angebotenen Dienste ist Teil der Fachkomponente. Diese beschreibt, welche Aufgaben die Dienste im Detail erfüllen, wie sie zu verwenden sind und welche Abhängigkeiten bestehen. Die Dokumentation soll an dieser Stelle nicht wiederholt werden. Außerdem finden sich diese Informationen auch auf der Verhaltens- und der Abstimmungsebene wieder.

1.5.2 Dienste zur Parametrisierung

Die Fachkomponente stellt insgesamt sechs Customizing-Gruppen mit verschiedenen Diensten zur Manipulation der Parameter zur Verfügung. Damit können Stammdaten wie Flughäfen und Fluggesellschaften sowie Steuerdaten wie die vom Reisebüro angebotenen Flugverbindungsnummern erfasst werden. Diese werden im Folgenden zusammen mit ihren Diensten aufgeführt.

1. `Flugticketverkauf::Flughafen` mit den Methoden `Anlegen`, `Ändern` und `Löschen`.
2. `Flugticketverkauf::Fluggesellschaft` mit den Methoden `Anlegen`, `Ändern`, `Löschen`, `ZuordnenPreisschema` und `EntfernenPreisschema`.
3. `Flugticketverkauf::Flugnummer` mit den Methoden `Anlegen`, `Ändern`, `Löschen`, `ZuordnenPreisschema` und `EntfernenPreisschema`.
4. `Flugticketverkauf::Flug` mit den Methoden `Anlegen`, `Ändern`, `Löschen`, `ZuordnenPreisschema` und `EntfernenPreisschema`.
5. `Flugticketverkauf::Flugverbindungsnummer` mit den Methoden `Anlegen`, `Löschen`, `AnlegenTeilstrecke` und `LöschenTeilstrecke`.
6. `Flugticketverkauf::Preisschema` mit den Methoden `Anlegen`, `Ändern` und `Löschen`.

Eine ausführliche Dokumentation der Dienste zur Parametrisierung ist Teil der Fachkomponente. Diese beschreibt, welche Einstellungsmöglichkeiten im Detail bestehen

und wie sie vorgenommen werden können. Die Dokumentation soll an dieser Stelle nicht wiederholt werden. Außerdem finden sich diese Informationen auch auf der Verhaltens- und der Abstimmungsebene wieder.

1.5.3 Erwartete Dienste

Die Komponente *Flugticketverkauf* benötigt zur Abarbeitung ihrer Aufgaben eine Reihe von Diensten. Dabei handelt es sich um Dienste zu den Entitätstypen *Flugverfügbarkeit, Flugbuchung*, *Flugkunde* und *Reisebüro*.

Man beachte, dass dabei keinerlei Aussage getroffen wird, von wem diese Dienste implementiert werden. Es kann sich dabei um ein oder mehrere andere Komponenten handeln. Es könnte aber auch ein Legacy-Programm sein, das nicht dem Komponentenparadigma entspricht. Um dies auszudrücken, wurde der Kontext der erwarteten Dienste mit „Extern“ bezeichnet.

1. `Extern::Reisebüro::PrüfeExistenz`
   - Von diesem Dienst wird erwartet, dass die Existenz eines Reisebüros überprüft wird. Dieser Dienst ist von einer außerhalb der Komponente liegenden Reisebüroverwaltung zu implementieren.
2. `Extern::Reisebüro::LiefereWährung`
   - Von diesem Dienst wird erwartet, dass die Währung zurückgeliefert wird, in der das Reisebüro abrechnet. Dieser Dienst ist von einer außerhalb der Komponente liegenden Reisebüroverwaltung zu implementieren.
3. `Extern::Flugkunde::PrüfeExistenz`
   - Von diesem Dienst wird erwartet, dass die Existenz eines Flugkunden überprüft wird. Dieser Dienst ist von einer außerhalb der Komponente liegenden Flugkundenverwaltung zu implementieren.
4. `Extern::Flugkunde::LiefereRabatt`
   - Von diesem Dienst wird erwartet, dass der Rabattsatz des Kunden zurückgeliefert wird. Dieser Dienst ist von einer außerhalb der Komponente liegenden Flugkundenverwaltung zu implementieren.
5. `Extern::Flugverfügbarkeit::Check`
   - Von diesem Dienst wird zu einem konkreten Flug die Verfügbarkeit erwartet. Dazu wird der gewünschte Flug angegeben. Dieser Dienst ist in einem Flugbuchungssystem aufzurufen.
6. `Extern::Flugbuchung::Anlegen`
   - Von diesem Dienst wird erwartet, dass eine neue Flugbuchung von der Fluggesellschaft angelegt wird und insbesondere ein Platz reserviert wird. Dazu werden alle notwendigen Informationen bereitgestellt. Dieser Dienst ist in einem Flugbuchungssystem aufzurufen.

2 Schnittstellenebene

Dieses Kapitel beschreibt die Schnittstellen der Fachkomponente „Flugticketverkauf“. Die genaue Syntax aller angebotenen und erwarteten Dienste findet sich in den Abschnitten 2.1. und 2.2. im OMG IDL Format.

Im Abschnitt 2.3. findet sich eine Liste aller Statusmeldungen, die von der Komponente zurückgegeben werden können.

Zuvor werden noch einige Konventionen und Einschränkungen erläutert, die sich durch die Verwendung der OMG IDL ergaben.

Die OMG IDL auf der Schnittstellenebene und die OCL auf der Verhaltensebene verwenden unterschiedliche Konstrukte zur Modularisierung. Die folgende Tabelle zeigt, welche Konstrukte in der weiteren Spezifikation einander entsprechen:

Fachkonzept	**Beispiel**	**OMG IDL**	**OCL / UML**
Fachkomponente	Flugticketverkauf	module	package
Entitätstyp	Flugreise	interface	class
Dienst	Anlegen	operation	method

Entsprechend dieser Konvention werden auf Schnittstellenebene zwei OMG IDL-Module definiert: das Modul *Flugticketverkauf* steht für die Fachkomponente selbst und das Modul *Extern* beschreibt die von außerhalb der Fachkomponente benötigten Dienste.

Alle Dienste der Komponente sind funktional implementiert. Bei der Verwendung handelt es sich also um funktionale Aufrufe mit Datenübergabe. Es können keine Objekte in der Komponente instanziiert werden und es werden keine Objektreferenzen an der Schnittstelle übergeben (weder im Import noch im Export). Dies trifft keine Aussage, ob die Komponente objektorientiert implementiert ist oder nicht.

An einigen Stellen werden bei der Spezifikation OO-Konstrukte verwendet (z.B. bei der OMG IDL und bei der UML OCL). An diesen Stellen werden alle Dienste als Klassenmethoden abgebildet.

Alle beschriebenen Dienste werden in Anlehnung an objektorientierte Modellierung in der Form `Entitätstyp::Methode` (Beispiel `Flugreise::Anlegen`) dargestellt. Dies dient der besseren Übersichtlichkeit und Strukturierung. Aber es sind wie oben beschrieben funktionale Dienste und man kann den Namen eines Dienstes einfach gesamt als „`Entitätstyp::Methode`“ betrachten.

Auf der Schnittstellenebene kommt es durch die Verwendung der OMG IDL (Version 2.4.2.) zu folgenden Einschränkungen:

- Es ist nicht möglich, einzelne Parameter einer Methode als optional zu deklarieren.
- Es ist nur schwer oder gar nicht möglich, semantisch reichere Datentypen zu definieren. Beispiele dafür sind:
  - OMG IDL kennt weder Datum noch Uhrzeit. Als Konsequenz habe ich z.B. Datum als *fixed<8,0>* beschrieben. Damit müssen jedoch an verschiedensten Stellen zusätzliche Bedingungen angegeben werden, dass nicht alle achtstelligen Zahlen ein gültiges Datum darstellen. Dieser Mehraufwand entfällt, wenn das Datum als eigener Datentyp vorliegt.

- In der ABAP-Implementierung werden an verschiedenen Stellen sogenannte Numerical Character-Typen verwendet. Dabei handelt es sich um Character-Typen einer vorgegebenen Länge, die allerdings nur Ziffern als Zeichen enthalten dürfen. (Ein Beispiel ist die achtstellige Buchungsnummer oder die vierstellige Flugnummer.) Auch dies lässt sich in der OMG IDL so nicht abbilden. Man könnte solche Typen alternativ als *fixed<n,0>* darstellen. Dies ist aber auch unkorrekt, da es sich um keine Zahlen handelt. Insbesondere sollen keine Rechenoperationen, sondern Stringoperationen anwendbar sein. (Ich habe diese Datentypen hier vorläufig als *string<n>* beschrieben, was auch nicht korrekt ist.)

- Die Fehlerbehandlung der Fachkomponente entspricht nicht den Konventionen der OMG IDL. Die Dienste der Komponente liefern keine *exceptions*. Stattdessen hat jeder Dienst einen Export-Parameter *Statusmeldungen*. Dieser enthält alle Meldungen, die sich auf den Status der Dienstabarbeitung beziehen. Dabei kann es sich neben Fehlermeldungen auch um Erfolgs- oder Informationsmeldungen handeln. (Siehe dafür auch Abschnitt 2.3.)

2.1 Schnittstellen der angebotenen Dienste

Dieser Abschnitt enthält die Schnittstellen der Fachkomponente *Flugticketverkauf*. Dies umfasst die betrieblichen Dienste und die Dienste zur Parametrisierung. Zur besseren Lesbarkeit wurden die Definitionen gegliedert und teilweise kommentiert. Die exakte Version der OMG IDL-Syntax ergibt sich einfach daraus, dass alle Zwischentexte weggelassen werden.

```
module Flugticketverkauf {
```

2.1.1 Definition übergreifender Datentypen

Im ersten Teil werden einige übergreifende Datentypen definiert, die für das Interface Flugverbindung, das Interface Flugreise und die Interfaces zur Parametrisierung von Interesse sind.

```
typedef fixed<8,0> DatumTyp;
typedef string<3> FluggesellschaftTyp;
typedef string<3> FlughafenTyp;
typedef string<4> FlugnummerTyp;
typedef string<4> FlugverbindungsnummerTyp;
typedef string<3> LandTyp;
typedef integer   ListenlängeTyp;
typedef string<8> ReisebüronummerTyp;
typedef string<30> StadtTyp;
typedef string<1> TeilstreckennummerTyp;
typedef fixed<4,0> UhrzeitTyp;
typedef fixed<19,4> WährungsbetragTyp;
typedef string<3> WährungsTyp;

struct DatumsbereichTyp {
      enum SignTyp {I, E}     Sign;
      enum OptionTyp {EQ, BT} Option;
      DatumTyp                Low;
      DatumTyp                High; };
typedef sequence<DatumsbereichTyp> DatumsbereichlistenTyp;

struct StatusTyp {
      string<1>   Typ;
      string<3>   Nummer;
```

```
        string<255> Nachricht; };
    typedef sequence<StatusTyp> StatuslistenTyp; };
```

2.1.2 Definition des Interfaces für die Flugverbindung

In diesem Abschnitt wird das Interface für den Entitätstyp Flugverbindung und dazu benötigte Datentypen definiert. Das Interface verwendet außerdem einige der in 2.1.1. definierten Datentypen.

```
interface Flugverbindung {

    typedef fixed<3,0> SitzanzahlTyp;

    struct FlugverbindungsdatenTyp {
          ReisebüronummerTyp      Reisebüronummer;
          FlugverbindungsnummerTyp Verbindungsnummer;
          DatumTyp                Abflugdatum;
          UhrzeitTyp              Abflugzeit;
          FlughafenTyp            Startflughafen;
          StadtTyp                Abflugstadt;
          DatumTyp                Ankunftsdatum;
          UhrzeitTyp              Ankunftszeit;
          FlughafenTyp            Zielflughafen;
          StadtTyp                Ankunftsstadt;
          integer                 Flugdauer;
          integer                 AnzahlTeilstrecken; };
    typedef sequence<FlugverbindungsdatenTyp> FlugverbindungslistenTyp;

    struct OrtTyp {
          FlughafenTyp Flughafenkürzel;
          StadtTyp     Stadt;
          LandTyp      Land; };

    struct TeilstreckenTyp {
          TeilstreckennummerTyp Nummer;
          FluggesellschaftTyp Fluggesellschaft;
          string<20>  Fluggesellschaftsname;
          FlugnummerTyp Flugnummer;
          FlughafenTyp Startflughafen;
          StadtTyp    Abflugstadt;
          LandTyp     Abflugland;
          FlughafenTyp Zielflughafen;
          StadtTyp    Ankunftsstadt;
          LandTyp     Ankunftsland;
          DatumTyp    Abflugdatum;
          UhrzeitTyp  Abflugzeit;
          DatumTyp    Ankunftsdatum;
          UhrzeitTyp  Ankunftszeit;
          string<20>  Flugzeugtyp; };
    typedef sequence<TeilstreckenTyp> TeilstreckenlistenTyp;

    struct VerbindungspreisTyp {
          WährungsbetragTyp EcoErw;
          WährungsbetragTyp EcoKind;
          WährungsbetragTyp EcoKleinkind;
          WährungsbetragTyp BusErw;
          WährungsbetragTyp BusKind;
          WährungsbetragTyp BusKleinkind;
          WährungsbetragTyp FirstErw;
          WährungsbetragTyp FirstKind;
          WährungsbetragTyp FirstKleinkind;
          WährungsbetragTyp Steuern;
```

```
      WährungsTyp       Währung; };

struct VerfügbarkeitsTyp {
      TeilstreckennummerTyp   Teilstreckennummer;
      SitzanzahlTyp           EcoFrei;
      SitzanzahlTyp           EcoMax;
      SitzanzahlTyp           BusFrei;
      SitzanzahlTyp           BusMax;
      SitzanzahlTyp           FirstFrei;
      SitzanzahlTyp           FirstMax;  };
typedef sequence<VerfügbarkeitsTyp> VerfügbarkeitslistenTyp;


void LiefereListe(
      in  ReisebüronummerTyp        Reisebüronummer,
      in  FluggesellschaftTyp       Fluggesellschaft,
      in  OrtTyp                    Abflugort,
      in  OrtTyp                    Ankunftsort,
      in  DatumsbereichlistenTyp    Datumsbereich,
      in  ListenlängeTyp            Listenlänge,
      out FlugverbindungslistenTyp  Flugverbindungsdaten,
      out StatuslistenTyp           Statusmeldungen);

void LiefereDetails(
      in  ReisebüronummerTyp        Reisebüronummer,
      in  FlugverbindungsnummerTyp  Verbindungsnummer,
      in  DatumTyp                  Abflugdatum,
      out FlugverbindungsdatenTyp   Flugverbindungsdaten,
      out TeilstreckenlistenTyp     Teilstreckenliste,
      out VerbindungspreisTyp       Preis,
      out VerfügbarkeitslistenTyp   Verfügbarkeit,
      out StatuslistenTyp           Statusmeldungen); };
```

2.1.3 Definition des Interfaces für die Flugreise

In diesem Abschnitt wird das Interface für die Entitätstyp Flugreise und dazu benötigte Datentypen definiert. Das Interface verwendet außerdem einige der in 2.1.1. definierten Datentypen.

```
interface Flugreise {

enum              FlugklasseTyp {Y, C, F};
typedef string<8> FlugkundeTyp;
typedef string<8> ReisenummerTyp;

struct FlugreisedatenTyp {
      ReisebüronummerTyp            Reisebüronummer;
      ReisenummerTyp                Reisenummer;
      FlugkundeTyp                  Flugkundennummer;
      FlugverbindungsnummerTyp      Hinflugverbindung;
      DatumTyp                      Hinflugdatum;
      FlugverbindungsnummerTyp      Rückflugverbindung;
      DatumTyp                      Rückflugdatum;
      FlugklasseTyp                 Flugklasse;
      DatumTyp                      Buchungsdatum;
      enum BuchungsstatusTyp {B, C} Buchungsstatus;
      integer                       AnzahlErwachsene;
      integer                       AnzahlKinder;
      integer                       AnzahlKleinkinder; };
typedef sequence<FlugreisedatenTyp> FlugreiselistenTyp;

struct PassagierTyp {
```

```
        string<25> Name;
        string<15> Anrede;
        DatumTyp   Geburtsdatum; };
    typedef sequence<PassagierTyp> PassagierlistenTyp;

    struct ReisedatenTyp {
        ReisebüronummerTyp          Reisebüronummer;
        FlugkundeTyp                Flugkundennummer;
        FlugverbindungsnummerTyp    Hinflugverbindung;
        DatumTyp                    Hinflugdatum;
        FlugverbindungsnummerTyp    Rückflugverbindung;
        DatumTyp                    Rückflugdatum;
        FlugklasseTyp               Flugklasse; };

    struct ReisepreisTyp {
        WährungsbetragTyp Summe;
        WährungsbetragTyp Steuern;
        WährungsTyp       Währung; };


    void LiefereListe(
        in  ReisebüronummerTyp      Reisebüronummer,
        in  FlugkundeTyp            Flugkundennummer,
        in  DatumsbereichlistenTyp  Flugdatumsbereich,
        in  DatumsbereichlistenTyp  Buchungsdatumsbereich,
        in  ListenlängeTyp          Listenlänge,
        out FlugreiselistenTyp      Flugreisedaten,
        out StatuslistenTyp         Statusmeldungen);

    void Anlegen(
        in  ReisedatenTyp           Reisedaten,
        in  PassagierlistenTyp      Passagierliste,
        out ReisebüronummerTyp      Reisebüronummer,
        out ReisenummerTyp          Reisenummer,
        out ReisepreisTyp           Reisepreis,
        out StatuslistenTyp         Statusmeldungen); };
```

2.1.4 Definition der Interfaces für die Parametrisierung

In diesem Abschnitt werden die Dienste spezifiziert, die das Erfassen, Ändern und Löschen von Parametern erlauben, die zur Fachkomponente *Flugticketverkauf* gehören. Dazu wird für jeden parametrisierungsrelevanten Entitätstypen ein Interface mit geeigneten Operationen definiert. Diese Interfaces verwenden ebenfalls einige der in 2.1.1. definierten Datentypen.

Bemerkung: Der Abschnitt 2.1.4 bezieht sich auf die Parametrisierung und ist vollständig neu hinzugekommen.

```
interface Flughafen {

    struct FlughafenKeyTyp {
        FlughafenTyp                Kürzel; };

    struct FlughafenDatenTyp {
        string<20>                  Name;
        StadtTyp                    Stadt;
        LandTyp                     Land; };


    void Anlegen(
        in  FlughafenKeyTyp         FlughafenKey,
        in  FlughafenDatenTyp       FlughafenDaten);
```

```
    void Ändern (
          in  FlughafenKeyTyp           FlughafenKey,
          in  FlughafenDatenTyp         FlughafenDaten);

    void Löschen(
          in  FlughafenKeyTyp           FlughafenKey); };


interface Fluggesellschaft {

    struct FluggesellschaftKeyTyp {
          FluggesellschaftTyp           Kürzel; };

    struct FluggesellschaftDatenTyp {
          string<20>                    Name;
          WährungsTyp                   Währung; };


    void Anlegen(
          in  FluggesellschaftKeyTyp    FluggesellschaftKey,
          in  FluggesellschaftDatenTyp  FluggesellschaftDaten);

    void Ändern (
          in  FluggesellschaftKeyTyp    FluggesellschaftKey,
          in  FluggesellschaftDatenTyp  FluggesellschaftDaten);

    void Löschen(
          in  FluggesellschaftKeyTyp    FluggesellschaftKey);

    void ZuordnenPreisschema (
          in  FluggesellschaftKeyTyp    FluggesellschaftKey,
          in  PreisschemaKeyTyp         PreisschemaKey);

    void EntfernenPreisschema (
          in  FluggesellschaftKeyTyp    FluggesellschaftKey);    };


interface Flugnummer {

    struct FlugnummerKeyTyp {
          FluggesellschaftTyp           Fluggesellschaft;
          FlugnummerTyp                 Nummer; };

    struct FlugnummerDatenTyp {
          FlughafenTyp                  Startflughafen
          UhrzeitTyp                    Abflugzeit;
          FlughafenTyp                  Zielflughafen
          UhrzeitTyp                    Ankunftszeit;
          integer                       Flugdauer;
          integer                       AnkunftTageSpäter; };


    void Anlegen(
          in  FlugnummerKeyTyp          FlugnummerKey,
          in  FlugnummerDatenTyp        FlugnummerDaten);

    void Ändern (
          in  FlugnummerKeyTyp          FlugnummerKey,
          in  FlugnummerDatenTyp        FlugnummerDaten);

    void Löschen(
```

```
            in  FlugnummerKeyTyp          FlugnummerKey);

      void ZuordnenPreisschema (
            in  FlugnummerKeyTyp          FlugnummerKey,
            in  PreisschemaKeyTyp         PreisschemaKey);

      void EntfernenPreisschema (
            in  FlugnummerKeyTyp          FlugnummerKey);    };


  interface Flug {

      struct FlugKeyTyp {
            FluggesellschaftTyp           Fluggesellschaft;
            FlugnummerTyp                 Flugnummer;
            DatumTyp                      Abflugdatum; };

      struct FlugDatenTyp {
            WährungsbetragTyp             Standardpreis;
            WährungsbetragTyp             Steuer; };


      void Anlegen(
            in  FlugKeyTyp                FlugKey,
            in  FlugDatenTyp              FlugDaten);

      void Ändern (
            in  FlugKeyTyp                FlugKey,
            in  FlugDatenTyp              FlugDaten);

      void Löschen(
            in  FlugKeyTyp                FlugKey);

      void ZuordnenPreisschema (
            in  FlugKeyTyp                FlugKey,
            in  PreisschemaKeyTyp         PreisschemaKey);

      void EntfernenPreisschema (
            in  FlugKeyTyp                FlugKey);    };


interface Flugverbindungsnummer {

      struct FlugverbindungsnummerKeyTyp {
            ReisebüronummerTyp            Reisebüronummer;
            FlugverbindungsnummerTyp      Verbindungsnummer; };

      struct TeilstreckennummerDatenTyp {
            FluggesellschaftTyp           Fluggesellschaft;
            FlugnummerTyp                 Flugnummer;
            integer                       AbflugTageSpäter; };


      void Anlegen(
            in  FlugverbindungsnummerKeyTyp    FlugverbindungsnummerKey);

      void Löschen(
            in  FlugverbindungsnummerKeyTyp    FlugverbindungsnummerKey);

      void AnlegenTeilstrecke(
            in  FlugverbindungsnummerKeyTyp    FlugverbindungsnummerKey,
            in  TeilstreckennummerTyp          HopNr,
```

```
            in  TeilstreckennummerDatenTyp     TeilstreckennummerDaten);

      void LöschenTeilstrecke(
            in  FlugverbindungsnummerKeyTyp    FlugverbindungsnummerKey,
            in  TeilstreckennummerTyp          HopNr); };


  interface Preisschema {

      struct PreisschemaKeyTyp {
            integer                 Schemanummer; };

      struct PreisschemaDatenTyp {
            fixed<1,2>              FaktorBus;
            fixed<1,2>              FaktorFirst;
            fixed<1,2>              FaktorKind;
            fixed<1,2>              FaktorKleinkind; };


      void Anlegen(
            in  PreisschemaKeyTyp         PreisschemaKey,
            in  PreisschemaDatenTyp       PreisschemaDaten);

      void Ändern(
            in  PreisschemaKeyTyp         PreisschemaKey,
            in  PreisschemaDatenTyp       PreisschemaDaten);

      void Löschen(
            in  PreisschemaKeyTyp         PreisschemaKey);  };

};
```

2.2 Schnittstellen der erwarteten Dienste

In diesem Abschnitt wird das Modul Extern definiert, welches alle von der Komponente benötigten Dienste und deren Schnittstellen enthält. Da dieses nicht so umfangreich ist, wird auf eine weitere Untergliederung verzichtet.

Außerdem werden auch hier einige der in 2.1.1. definierten Datentypen benötigt. Diese könnten jedoch direkt nur dann verwendet werden, wenn das Modul Flugticketverkauf hier inkludiert wird. Dies soll aber bewusst nicht geschehen, weil dadurch die Trennung der Module wieder aufgehoben wird. Stattdessen werden die benötigten Datentypen hier namensgleich noch einmal definiert. Das ist möglich, da jedes OMG IDL Modul einen eigenen Namensraum bildet. (Theoretisch könnte man hier also unter gleichen Namen syntaktisch andere Datentypen definieren. Dies geschieht aber aus Übersichtlichkeitsgründen nicht. Gleiche Namen bedeuten auch gleiche Datentypen.)

```
module Extern {

      typedef string<8> BuchungsnummerTyp;
      typedef fixed<8,0> DatumTyp;
      typedef string<3> FluggesellschaftTyp;
      enum              FlugklasseTyp {Y, C, F};
      typedef string<8> FlugkundeTyp;
      typedef string<4> FlugnummerTyp;
      typedef string<8> ReisebüronummerTyp;
      typedef fixed<3,0> SitzanzahlTyp;
      typedef fixed<19,4> WährungsbetragTyp;
      typedef string<3> WährungsTyp;
```

```
    enum              XFlagTyp {X, ,'};

    struct StatusTyp {
          string<1>   Typ;
          string<3>   Nummer;
          string<255> Nachricht; };
    typedef sequence<StatusTyp> StatuslistenTyp; };


interface Reisebüro {

    XFlagTyp    PrüfeExistenz(
          in  ReisebüronummerTyp  Reisebüronummer);

    WährungsTyp LiefereWährung(
          in  ReisebüronummerTyp  Reisebüronummer); };


interface Flugkunde {

    typedef fixed<3,0> RabattTyp;

    XFlagTyp   PrüfeExistenz(
          in  FlugkundeTyp  Flugkundennummer);

    RabattTyp  LiefereRabatt(
          in  FlugkundeTyp  Flugkundennummer); };


interface Flugverfügbarkeit {

    struct FlugverfügbarkeitsTyp {
          SitzanzahlTyp          EcoFrei;
          SitzanzahlTyp          EcoMax;
          SitzanzahlTyp          BusFrei;
          SitzanzahlTyp          BusMax;
          SitzanzahlTyp          FirstFrei;
          SitzanzahlTyp          FirstMax;  };

    void Check(
          in  FluggesellschaftTyp      Fluggesellschaft,
          in  FlugnummerTyp            Flugnummer,
          in  DatumTyp                 Flugdatum,
          out FlugverfügbarkeitsTyp    Flugverfügbarkeit,
          out StatuslistenTyp          Statusmeldungen); };


interface Flugbuchung {

    struct BuchungsdatenTyp {
          FluggesellschaftTyp     Fluggesellschaft;
          FlugnummerTyp           Flugnummer;
          DatumTyp                Flugdatum;
          FlugkundeTyp            Flugkunde;
          ReisebüronummerTyp      Reisebüronummer;
          FlugklasseTyp           Flugklasse;
          string<25>              PassagierName;
          string<15>              PassagierAnrede;
          DatumTyp                PassagierGeburtsdatum; };

    struct FlugbuchungspreisTyp {
          WährungsbetragTyp Preis;
```

```
            WährungsbetragTyp Steuern;
            WährungsTyp       Währung; };

      void Anlegen(
            in  BuchungsdatenTyp          Buchungsdaten,
            out FluggesellschaftTyp       Fluggesellschaft,
            out BuchungsnummerTyp         Buchungsnummer,
            out FlugbuchungspreisTyp      Flugpreis,
            out StatuslistenTyp           Statusmeldungen); };
};
```

2.3 Definition der Statusmeldungen

Jeder der bereitgestellten Dienste enthält einen Output-Parameter Statusmeldungen. Dieser enthält detaillierte Informationen, ob der Dienst erfolgreich abgearbeitet werden konnte oder welche Probleme aufgetreten sind. Dabei wird jeder Statussatz durch einen Typ, eine Nummer und einen Meldungstext beschrieben.

Bei Typ sind nur die Werte {‘S’, ‘E’, ‘W’} mit folgender Bedeutung möglich:

S (Success): Erfolgsmeldung; Dienst wurde erfolgreich abgearbeitet
W (Warning): Warnung; die Meldung enthält Hinweise auf eventuelle Einschränkungen, die bei der Dienstbearbeitung aufgetreten sind
E (Error): Fehlermeldung; die Meldung enthält Hinweise, warum die Dienstbearbeitung nicht erfolgreich war

Die auftretenden Statusnummern sind im folgenden mit Kurztext angegebenen. (Die zugehörigen Langtexte werden an dieser Stelle weggelassen.) Ausdrücke der Art &1 sind Parameter, die dynamisch mit den entsprechenden Werten gefüllt werden. Die Statusmeldungen werden bei den Bedingungen auf der Verhaltensebene referenziert.

000 Ausführung der Methode war erfolgreich
001 Es sind Fehler aufgetreten
006 Technischer Fehler bei der Verarbeitung
010 Datum &1 ist ungültig
017 ISO-Länderkürzel &1 unbekannt

050 Fluggesellschaft &1 unbekannt
051 Flughafen mit Kürzel &1 unbekannt
052 Ort &1 &2 unbekannt
053 Ort &1 wurde mehrfach (in verschiedenen Ländern) gefunden
057 Gewünschtes Flugdatum &1 liegt in der Vergangenheit

106 Geburtsdatum &1 liegt in der Zukunft
107 Flugklasse kann nur Y, C oder F sein

150 Flugkunde mit Nummer &1 unbekannt
151 Reisebüro mit Nummer &1 unbekannt

250 Flugverbindung &1 nicht vorhanden
251 Es entsprachen keine Flugverbindungen den Selektionsbedingungen
252 Gewünschtes Reisedatum &1 liegt in der Vergangenheit

253 Fehler bei Verfügbarkeitsermittlung für Teilstrecke &1
254 Fehler bei Preisermittlung für die Flugverbindung

300 Flugreise &1 nicht vorhanden
301 Es entsprachen keine Flugreisen den Selektionsbedingungen
302 Rückflugdatum &2 liegt vor dem Hinflugdatum &1
303 Zu einem Passagier wurde kein Name übergeben
304 Reservierung für Flug &1 &2 war nicht möglich
305 Es konnte keine Reisenummer vergeben werden
306 Passagierliste wurde nicht übergeben

3 Verhaltensebene

Zunächst stellen wir in einem UML-Modell alle wesentlichen Entitätstypen und deren Beziehung zueinander dar. Dieses Modell bildet die Grundlage für die in diesem Abschnitt formulierten OCL-Bedingungen. Die im Modell definierten Bezeichner für Entitätstypen, Methoden, Assoziationen etc. werden in den OCL-Bedingungen verwendet.

Man beachte, dass das Modell nur ein Spezifikationsartefakt ist. Es ist ein Hilfsmittel, um die Spezifikation verständlich und ausdrucksstark zu machen. Das Modell abstrahiert von der konkreten Implementierung. Die angegebenen Klassen müssen nicht tatsächlich existieren. Bei einer Nicht-OO-Implementierung existieren ja gar keine Klassen. (Deshalb verwende ich den Ausdruck „Entitätstyp“ statt „Klasse“). Um den Spezifikationsgedanken zu unterstreichen, wurde die Sichtbarkeit aller Elemente nicht angegeben. Eine Ausnahme bilden die tatsächlich vorkommenden Dienste. In das Modell wurden nur Daten/Eigenschaften aufgenommen, über die in der Spezifikation sowieso Aussagen getroffen werden. Damit verstößt dieses Vorgehen nicht gegen den Black-Box-Gedanken!

Bemerkungen zum Modell:

- *Das UML-Modell enthält die zwei Pakete Flugticketverkauf und Extern. Diese wiederum enthalten verschiedene Entitätstypen (als Klassen) mit Methoden und Attributen sowie deren Beziehungen untereinander.*
- *Parametrisierungsrelevante Attribute und Methoden wurden mit `{C}` gekennzeichnet.*
- *Assoziationen wurden immer mit Navigierbarkeit versehen. Fehlende Navigierbarkeit bedeutet, dass eine Navigation im Modell nicht möglich ist. In den folgenden OCL-Bedingungen werden Assoziationen immer nur in Navigationsrichtung verwendet.*
- *Ich habe bei den Attributen im Diagramm zur Vereinfachung und besseren Lesbarkeit die Datentypen weggelassen. Die Attribute tragen den gleichen Typ wie die vergleichbaren Felder an den Schnittstellen der Dienste (siehe Schnittstellenebene). Alle folgenden Aussagen in Kapitel 3 sind typkonform, auch wenn die Typen formal im Modell fehlen.*
- *Ebenso werden die Parameter der Methoden hier nicht näher angegeben. Diese finden sich bei den gleichnamigen Operationen auf der Schnittstellenebene.*

Bemerkung 2: Bei der Modellierung der Customizing-Aktivitäten (CAs) als Entitätstypen sind folgende Aspekte von Interesse:

- *Die Abgrenzung von CAs erfolgt eher prozessgetrieben. Die Abbildung in UML-Klassen mit Methoden bedingt eine eher strukturelle Sicht. Daher ist eine 1:1 Modellierung nicht immer sinnvoll.*
  Beispiel: Die Zuordnung von Preisschemata wird unter Prozesssicht als eine Aufgabe gesehen. Entsprechend gibt es dazu die komplexe CA 5 (siehe Anhang A). In den 3 Teilschritten können Preisschemata zu Fluggesellschaften, Flugnummern und Flügen zugeordnet werden. Aus struktureller Sicht handelt es sich eher um Eigenschaften von Fluggesellschaften etc., so dass dort entsprechende Methoden definiert wurden.
- *Werden durch eine CA Zuordnungen zwischen zwei Entitätstypen hergestellt, dann gibt es zwei Modellierungsalternativen:*
  - *Die Zuordnung wird durch einen eigenen Entitätstyp repräsentiert und trägt entsprechende Methoden.*

- *Die Zuordnung wird durch eine Assoziation zwischen den beiden Entitätstypen repräsentiert. Eine der beiden Entitätstypen trägt zusätzliche Methoden zur Verwaltung der Zuordnungen..*

Der zweite Fall ist vor allem dann interessant, wenn die Zuordnung keine weiteren Attribute trägt oder selbst keine so wichtige Rolle spielt.
Beispiel: Die Zuordnung eines Preisschemas zu einer Fluggesellschaft wird durch die Assoziation dargestellt, und der Entitätstyp Fluggesellschaft trägt die Methoden ZuordnenPreisschema und EntfernenPreisschema.

- *Bei der Festlegung von Methoden, welche die möglichen Arbeitsschritte bei CAs beschreiben, haben sich die folgenden Konventionen als praktikabel erwiesen:*
  - *Eine parametrisierungsrelevante Klasse erhält die drei Standardmethoden Anlegen, Ändern und Löschen. Diese beziehen sich immer auf einzelne Instanzen.*
  - *Hat die Klasse keine änderbaren Attribute (z.B. wenn sie nur Schlüsselattribute hat), dann kann auf die Methode Ändern verzichtet werden.*
  - *Gibt es zu Entitäten stark abhängige Entitäten, kann es sinnvoll sein, die Methoden zur Manipulation der abhängigen Entität bei der unabhängigen Entität aufzunehmen.*
- *Triviale Reihenfolgebeziehungen, welche sich aufgrund von Existenzbedingungen ergeben, werden auf der Verhaltensebene abgebildet.*
  Beispiele: Eine Entität kann nur geändert werden, wenn sie zuvor angelegt wurde. Trägt eine Entität Parameter mit customizingabhängigem Wertebereich, dann kann diese nur angelegt werden, wenn die benötigten Werte des anderen Entitätstyps zuvor angelegt wurden.
- *Auf der Abstimmungsebene finden sich damit nur die nicht-trivialen Reihenfolge-bedingungen zwischen den Diensten.*
- *Für die Modellierung komplexer CA, die aus mehreren einfachen CA bestehen, wurde folgende Konvention festgelegt:*
  - *Erfolgte die Zusammenlegung der einfachen CAs nur aus Usabilitygründen, ohne dass eine zwingende Reihenfolgebeziehung vorliegt, dann wurden nur die einfachen CAs angegeben.*
  - *Besteht zwischen den einfachen CAs eine zwingende Reihenfolgebeziehung, dann wurden diese auf der Abstimmungsebene modelliert.*
    Beispiel: Siehe Abschnitt 4.8.

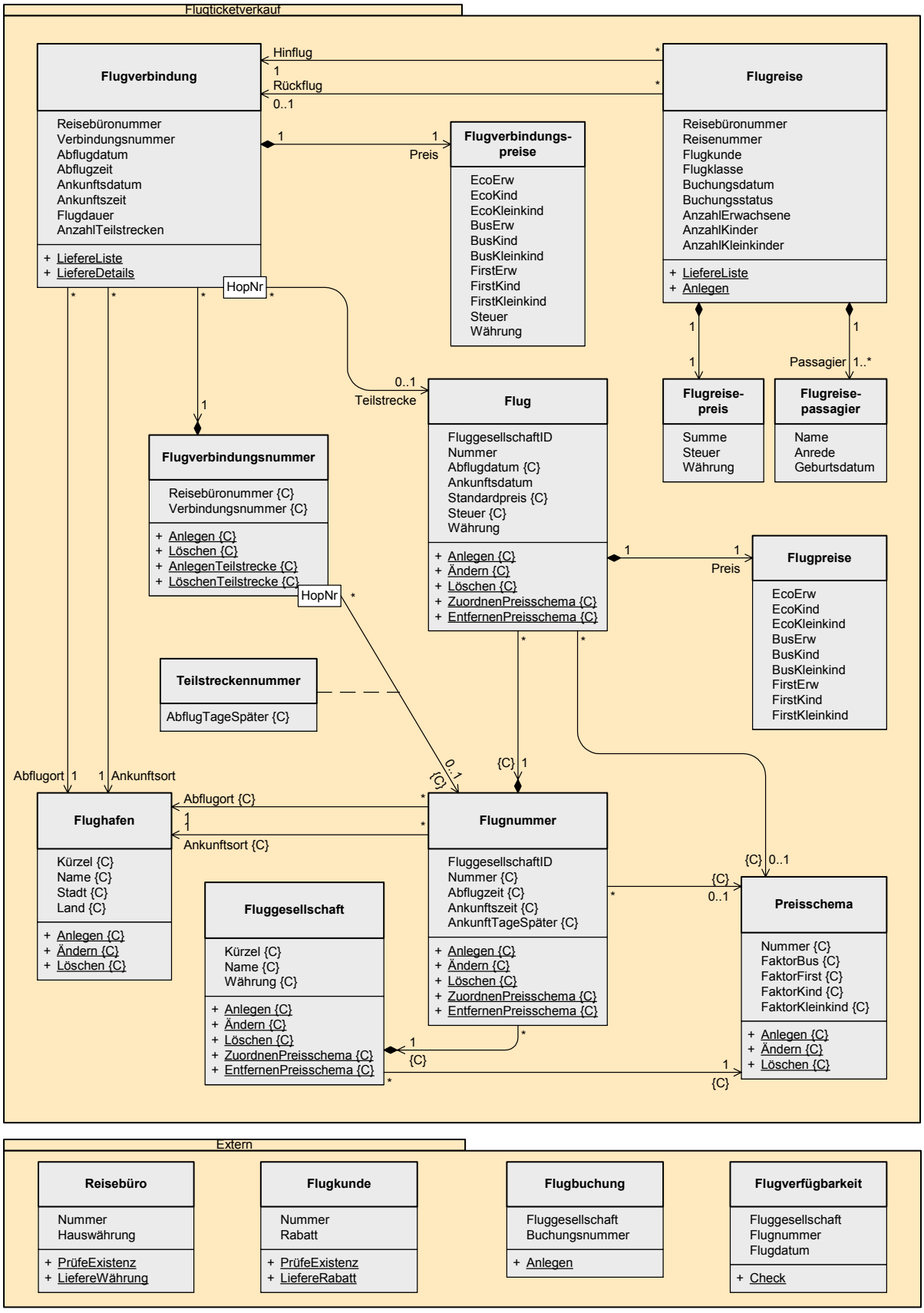

Bild B-1: UML-Modell der Fachkomponente *Flugticketverkauf*

3.1 Flugverbindung allgemein

In diesem Abschnitt werden eine Reihe von Invarianten aufgeführt, die von Flugverbindungen jederzeit erfüllt werden.

3.1.1 Identifikation einer Flugverbindung

Eine *Flugverbindung* wird durch die Attribute *Reisebüronummer*, *Verbindungsnummer* und *Abflugdatum* eindeutig identifiziert.

Flugticketverkauf

```
inv: self.Flugverbindung->forAll(fvb1, fvb2 | fvb1 <> fvb2 implies
          fvb1.Reisebüronummer    <> fvb2.Reisebüronummer or
          fvb1.Verbindungsnummer  <> fvb2.Verbindungsnummer or
          fvb1.Abflugdatum        <> fvb2.Abflugdatum )
```

3.1.2 Reisebüro existiert

Das die *Flugverbindung* anbietende *Reisebüro* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung.

Flugticketverkauf::Flugverbindung

```
inv: Extern::Reisebüro::PrüfeExistenz(self.Reisebüronummer) = ´X´
```

3.1.3 Anzahl der Teilstrecken

Die Anzahl der Teilstrecken einer Flugverbindung (Kardinalität der mit der *Flugverbindung* verbundenen *Flüge*) wird durch das Attribut *AnzahlTeilstrecken* beschrieben.

Flugticketverkauf::Flugverbindung

```
inv: self.AnzahlTeilstrecken = self.Teilstrecke->size
```

3.1.4 Zusammenhang mit der Flugverbindungsnummer

Eine Flugverbindungsnummer ist gewissermaßen eine Schablone, aus der sich die einzelnen Flugverbindungen ableiten. Dabei stimmen die Attribute Reisebüronummer und Verbindungsnummer bei einer Flugverbindung und der mit ihr verknüpften Flugverbindungsnummer überein.

Flugticketverkauf::Flugverbindung

```
inv: self.Reisebüronummer   = self.Flugverbindungsnummer.Reisebüronummer
     self.Verbindungsnummer = self.Flugverbindungsnummer.Verbindungsnummer
```

Die Eigenschaften einer Flugverbindung werden durch die Eigenschaften der Flugverbindungsnummer festgelegt. Zu einer Flugverbindungsnummer gehören Teilstrecken, bei denen es sich um Flugnummern handelt.

Analog gehören zu einer Flugverbindung (= Flugverbindungsnummer + Abflugdatum) Teilstrecken, bei denen es sich um Flüge (= Flugnummer + Abflugdatum) handelt.

Dabei müssen die Teilstrecken der Flugverbindung genau die gleichen Flugnummern haben, wie dies von der Flugverbindungsnummer vorgegeben wird.

Flugticketverkauf::Flugverbindung

```
inv: self.Flugverbindungsnummer.Flugnummer = self.Teilstrecke.Flugnummer
```

Bemerkung: Aufgrund dieser Korrespondenz übertragen sich einige der Eigenschaften der Flugverbindungsnummer automatisch auf die Flugverbindung:

- *Zu einer Flugverbindung gibt es mindestens eine Teilstrecke.*
- *Jede der Teilstrecken einer Flugverbindung wird durch ihre Nummer eindeutig identifiziert. Diese Nummer liegt zwischen 1 und AnzahlTeilstrecken.*
- *Der Abflug eines Teilstreckenfluges kann nur nach der Ankunft des vorherigen Teilstreckenfluges erfolgen.*
- *Der Abflugort eines Teilstreckenfluges ist gleich dem Ankunftsort des vorherigen Teilstreckenfluges.*

Bemerkung: Diese Bedingung ist neu hinzugekommen. Sie beschreibt, wie sich eine Flugverbindung aus den Parameter-Einstellungen zur Flugverbindungsnummer ergibt.

3.1.5 Abflug der Teilstreckenflüge

Bemerkung: Diese Bedingung entfällt, da sie jetzt in Abschnitt 3.1.4 enthalten ist.

3.1.6 Abflug der Flugverbindung

Abflugort und –zeit der Gesamtflugverbindung entsprechen Abflugort und –zeit der ersten Teilstrecke.

```
Flugticketverkauf::Flugverbindung
  inv: self.Abflugdatum  = self.Teilstrecke[1].Abflugdatum
       self.Abflugort    = self.Teilstrecke[1].Flugnummer.Abflugort
       self.Abflugzeit   = self.Teilstrecke[1].Flugnummer.Abflugzeit
```

Bemerkung: Der Ausdruck `self.Teilstrecke[1]` steht für die erste Teilstrecke der Flugverbindung.

3.1.7 Ankunft der Flugverbindung

Analog entsprechen Ankunftsort und –zeit der Gesamtflugverbindung Ankunftsort und –zeit der letzten Teilstrecke.

```
Flugticketverkauf::Flugverbindung
  inv: let n = self.AnzahlTeilstrecken in
         self.Ankunftsdatum    = self.Teilstrecke[n].Ankunftsdatum
    and self.Ankunftsort       = self.Teilstrecke[n].Flugnummer.Ankunftsort
    and self.Ankunftszeit      = self.Teilstrecke[n].Flugnummer.Ankunftszeit
```

Bemerkung: Die Bedingungen in den Abschnitten 3.1.6 und 3.1.7 konnten aufgrund des erweiterten UML-Modells vereinfacht werden.

3.1.8 Zeitzonen

Alle Zeiten sind in der jeweiligen Zeitzone des Flughafens angegeben.

3.1.9 Flugdauer

Die Flugdauer einer Flugverbindung ist gleich der Differenz zwischen Ankunftszeit und Abflugszeit unter Berücksichtigung der Zeitverschiebung.

Bemerkung: Mit Hilfe der OCL kann man sehr gut Bedingungen zwischen verschiedenen Elementen eines Modells beschreiben. Die Grenzen der OCL werden aber in 3.1.8. und 3.1.9. sichtbar. Beide Aussagen sind in Prosaform intuitiv und meiner Meinung nach unmissverständlich. Um diese in OCL auszudrücken, müsste das Konzept von Zeitzonen und Zeitverschiebungen explizit in das Modell aufgenommen werden.

3.1.10 Währungsangaben bei Flugverbindungen

Alle Preise einer Flugverbindung werden in der Hauswährung des Reisebüros angegeben.

```
Flugticketverkauf::Flugverbindung
  inv: Extern::Reisebüro::LiefereWährung(self.Reisebüronummer)
                                                = self.Preis.Währung
```

3.1.11 Berechnung der Preise einer Flugverbindung

Bei den Preisen für eine Flugverbindung wird zwischen den verschiedenen Flugklassen (Economy, Business und First Class) sowie nach Tarifen (Erwachsene, Kinder, Kleinkinder) unterschieden. Für nähere Erklärungen zu den Tarifen siehe auch 3.4.9.

Die Preise einer *Flugverbindung* ergeben sich daraus, dass die entsprechenden Preise der Einzelflüge (= Teilstrecken) addiert werden.

```
Flugticketverkauf::Flugverbindung
  inv: self.Preis.EcoErw         = self.Teilstrecke.Preis.EcoErw->sum
   and self.Preis.EcoKind        = self.Teilstrecke.Preis.EcoKind->sum
   and self.Preis.EcoKleinkind   = self.Teilstrecke.Preis.EcoKleinkind->sum
   and self.Preis.BusErw         = self.Teilstrecke.Preis.BusErw->sum
   and self.Preis.BusKind        = self.Teilstrecke.Preis.BusKind->sum
   and self.Preis.BusKleinkind   = self.Teilstrecke.Preis.BusKleinkind->sum
   and self.Preis.FirstErw       = self.Teilstrecke.Preis.FirstErw->sum
   and self.Preis.FirstKind      = self.Teilstrecke.Preis.FirstKind->sum
   and self.Preis.FirstKleinkind =
                                 self.Teilstrecke.Preis.FirstKleinkind->sum
   and self.Preis.Steuer         = self.Teilstrecke.Steuer->sum
```

Bemerkung 1: Es wird davon ausgegangen, dass alle Währungsbeträge sich auf die angegebene Währung beziehen. So ist z.B. für die Flugverbindung *self.Preis.Währung* die Währung zu *self.Preis.EcoErw* und für den Flug *self.Teilstrecke.Flug.Preis.Währung* die Währung zu *self.Teilstrecke.Flug.Preis.EcoErw*.

Bemerkung 2: In dieser Bedingung wird der Einfachheit halber vorausgesetzt, dass alle Währungsbeträge in der selben Währung vorliegen. Es ist aber streng genommen notwendig, die Währungsbeträge mit Hilfe einer Servicefunktion umzurechnen. Darauf wird in dieser ersten Version der Spezifikation verzichtet (ist aber so implementiert).

Bemerkung 3: Diese Bedingung hat sich aufgrund des veränderten UML-Modells syntaktisch leicht verändert, ist aber semantisch gleich geblieben.

3.2 Flugverbindung::LiefereListe

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugverbindung::LiefereListe*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugverbindung { ...

      void LiefereListe(
            in  ReisebüronummerTyp        Reisebüronummer,
            in  FluggesellschaftTyp       Fluggesellschaft,
            in  OrtTyp                    Abflugort,
            in  OrtTyp                    Ankunftsort,
            in  DatumsbereichlistenTyp    Datumsbereich,
            in  ListenlängeTyp            Listenlänge,
            out FlugverbindungslistenTyp  Flugverbindungsdaten,
            out StatuslistenTyp           Statusmeldungen); ... }; ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugverbindung::LiefereListe(
                  In    Rbnr:   ReisebüronummerTyp,
                        Fg:     FluggesellschaftTyp,
                        Ab:     OrtTyp,
                        An:     OrtTyp,
                        Dab:    DatumsbereichlistenTyp,
                        Anz:    ListenlängeTyp,
                  Out   Fvbd:   FlugverbindungslistenTyp,
                        Status: StatuslistenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer nur abgekürzt und ohne Datentypen angegeben.

3.2.1 Reisebüro existiert

Das angegebene *Reisebüro* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung. Ist die Vorbedingung nicht erfüllt, wird ein entsprechender Fehler zurückgegeben.

```
Flugticketverkauf::Flugverbindung::LiefereListe(Rbnr, ..., Status)
 pre ReisebüroExistiert:
     Extern::Reisebüro::PrüfeExistenz(Rbnr) = ´X´

 post: ReisebüroExistiert = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´151´)
```

Bemerkung 1: Die Komponente wurde fehlertolerant erstellt. Bei Verletzung einer Vorbedingung wird ein entsprechender Fehler zurückgegeben. Die Spezifikation enthält im Abschnitt 2.3 die Information, auf welche Situationen mit welchem Fehler reagiert wird.

Zu jeder Vorbedingung erscheint eine Nachbedingung, die den entsprechenden Fehler aufführt, der bei nicht erfüllter Vorbedingung zurückgegeben wird. Die Liste aller

Fehlermeldungen und deren Bedeutung findet sich auf der Schnittstellenebene (Abschnitt 2.3).

Bemerkung 2: Zur Vereinfachung wurde der Vorbedingung ein Name gegeben, auf den dann in der Nachbedingung Bezug genommen wird. OCL erlaubt dazu, Bedingungen einen Namen zu geben. (Ich bin mir aber nicht sicher, ob man auf diesen Namen auch im Rahmen anderer Bedingungen Bezug nehmen darf).

3.2.2 Fluggesellschaft existiert

Die angegebene *Fluggesellschaft* existiert. Ist die Vorbedingung nicht erfüllt, wird ein entsprechender Fehler zurückgegeben.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Fg, ..., Status)
 pre FluggesellschaftExistiert:  (Fg <> ´´) implies
           Fluggesellschaft->exists(fg | fg.Kürzel = Fg)

 post: FluggesellschaftExistiert = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´050´)
```

Man beachte, dass der Parameter *Fluggesellschaft* optional ist. Deshalb ist die Überprüfung der Fluggesellschaft nur sinnvoll, wenn der Parameter gefüllt ist.

3.2.3 Konsistenz des Parameters Abflugort

Ist der Parameter *Abflugort* nicht leer, sucht der Dienst geeignete Einträge in folgender Reihenfolge: *Flughafen*, *Stadt* und *Lan*d, nur *Stadt*, nur *Land*. Wird zu einer der Varianten ein Eintrag gefunden, wird überprüft, ob die Eingabe korrekt ist. Falls nicht, wird ein entsprechender Fehler zurückgegeben. Es wird danach nicht mehr überprüft, ob auch andere (in der Reihenfolge spätere) Varianten gültig sind bzw. ob diese konsistent sind. (Beispiel: Es wird als Flughafen FRA und als Stadt Berlin übergeben. Für den Flughafen wurde das korrekte Kürzel FRA übergeben. Es werden alle Flugverbindungen ab Frankfurt gesucht. Der Eintrag Berlin wird ignoriert.)

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ..., Status)
 pre AbflugortExistiert1: (Ab.Flughafenkürzel <> ´´) implies
     Flughafen->exists(airp | airp.Kürzel = Ab.Flughafenkürzel)

 post: AbflugortExistiert1 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´051´)

Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ... , Status)
 pre AbflugortExistiert2:
     (Ab.Flughafenkürzel = ´´) and (Ab.Stadt <> ´´) and (Ab.Land <> ´´)
      implies Flughafen->exists(airp |
           airp.Stadt = Ab.Stadt and airp.Land = Ab.Land)

 post: AbflugortExistiert2 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´052´)

Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ... , Status)
 pre AbflugortExistiert3:
     (Ab.Flughafenkürzel = ´´) and (Ab.Stadt <> ´´) and (Ab.Land = ´´)
      implies Flughafen->exists(airp | airp.Stadt = Ab.Stadt)
```

```
 post: AbflugortExistiert3 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´052´)

Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ... , Status)
 pre AbflugortExistiert4:
     (Ab.Flughafenkürzel = ´´) and (Ab.Stadt = ´´) and (Ab.Land <> ´´)
      implies Flughafen->exists(airp | airp.Land = Ab.Land)

 post: AbflugortExistiert4 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´017´)
```

3.2.4 Konsistenz des Parameters Ankunftsort

Der Parameter Ankunftsort wird analog zum Abflugort behandelt. Siehe auch die Vorbedingungen in 3.2.3.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ..., Status)
 pre AnkunftsortExistiert1: (An.Flughafenkürzel <> ´´) implies
     Flughafen->exists(airp | airp.Kürzel = An.Flughafenkürzel)

 post: AnkunftsortExistiert1 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´051´)

Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ... , Status)
 pre AnkunftsortExistiert2:
     (An.Flughafenkürzel = ´´) and (An.Stadt <> ´´) and (An.Land <> ´´)
      implies Flughafen->exists(airp |
           airp.Stadt = An.Stadt and airp.Land = An.Land)

 post: AnkunftsortExistiert2 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´052´)

Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ... , Status)
 pre AnkunftsortExistiert3:
     (An.Flughafenkürzel = ´´) and (An.Stadt <> ´´) and (An.Land = ´´)
      implies Flughafen->exists(airp | airp.Stadt = An.Stadt)

 post: AnkunftsortExistiert3 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´052´)

Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ... , Status)
 pre AnkunftsortExistiert4:
     (An.Flughafenkürzel = ´´) and (An.Stadt = ´´) and (An.Land <> ´´)
      implies Flughafen->exists(airp | airp.Land = An.Land)

 post: AnkunftsortExistiert4 = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´017´)
```

3.2.5 Zusammenhang Verbindungsliste und Flugverbindungen

Die im Tabellenparameter *Flugverbindungsdaten* zurückgegebenen Einträge repräsentieren Flugverbindungen. Das bedeutet, zu jedem Eintrag in *Flugverbindungsdate*n gibt es eine (in UML als Objekt modellierte) korrespondierende Entität von *Flugverbindung*. Insbesondere stimmen die entsprechenden Daten überein.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Fvbd, ...)
 post: Fvbd->forAll(fvbd | Flugverbindung->exists (fvb |
            fvbd.Reisebüronummer    =  fvb.Reisebüronummer
       and  fvbd.Verbindungsnummer  =  fvb.Verbindungsnummer
       and  fvbd.Abflugdatum        =  fvb.Abflugdatum
       and  fvbd.Abflugzeit         =  fvb.Abflugzeit
       and  fvbd.Startflughafen     =  fvb.Abflugort.Kürzel
       and  fvbd.Abflugstadt        =  fvb.Abflugort.Stadt
       and  fvbd.Ankunftsdatum      =  fvb.Ankunftsdatum
       and  fvbd.Ankunftszeit       =  fvb.Ankunftszeit
       and  fvbd.Zielflughafen      =  fvb.Ankunftsort.Kürzel
       and  fvbd.Ankunftsstadt      =  fvb.Ankunftsort.Stadt
       and  fvbd.Flugdauer          =  fvb.Flugdauer
       and  fvbd.AnzahlTeilstrecken =  fvb.AnzahlTeilstrecken) )
```

Bemerkung: Da es sich bei den Einträgen aus der Flugverbindungsliste um Flugverbindungen handelt, gelten somit alle Invarianten aus Kapitel 3.1.

3.2.6 Selektion bezüglich Reisebüro

Alle Flugverbindungen aus dem Parameter *Flugverbindungsdaten* werden von dem Reisebüro angeboten, welches durch den Parameter *Reisebüronummer* spezifiziert wurde.

```
Flugticketverkauf::Flugverbindung::LiefereListe(Rbnr, ..., Fvbd, ...)
 post: Fvbd->forAll(fvbd |  fvbd.Reisebüronummer = Rbnr)
```

3.2.7 Selektion bezüglich Fluggesellschaft

Wurde im Import eine *Fluggesellschaft* spezifiziert, dann werden nur *Flugverbindungen* selektiert, bei denen alle Teilstrecken von der *Fluggesellschaft* durchgeführt werden:

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Fg, ..., Fvbd, ...)
 post: (Fg <> ´´) implies
           Fvbd->forAll(fvbd | Flugverbindung->exists(fvb1 |
                  ( fvbd.Reisebüronummer   = fvb1.Reisebüronummer)
              and ( fvbd.Verbindungsnummer = fvb1.Verbindungsnummer )
              and ( fvbd.Abflugdatum       = fvb1.Abflugdatum )
               and ( fvb1.Teilstrecke->forAll(fl |
                             fl.FluggesellschaftID = Fg ) ) )
```

Bemerkung: Da von den *Flugverbindungsdaten* keine direkte Navigation zu den Teilstrecken möglich ist, muss hier der Umweg über die Entität *Flugverbindung* gegangen werden.

Bemerkung 2: Diese Bedingung hat sich aufgrund des veränderten UML-Modells syntaktisch leicht verändert, ist aber semantisch gleich geblieben.

3.2.8 Selektion bezüglich Abflugort

Ist der Parameter *Abflugort* nicht leer, dann sucht der Dienst geeignete Einträge in folgender Reihenfolge: Flughafen, Stadt und Land, nur Stadt, nur Land. Wird zu einer der Varianten ein gültiger Eintrag gefunden, dann dient dieser zur Selektion des Abflugortes. Eventuell andere, in obiger Reihenfolge später betrachtete Einträge werden ignoriert. (Beispiel: Es wird als Flughafen FRA und als Stadt Berlin übergeben. Es werden alle Flugverbindungen ab Frankfurt gesucht. Der Eintrag Berlin wird ignoriert.)

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ..., Fvbd, ...)
```

```
 post: (Ab.Flughafenkürzel <> ´´) implies
           Fvbd->forAll(fvbd | fvbd.Startflughafen = Ab.Flughafenkürzel )

Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ..., Fvbd, ...)
 post: (Ab.Flughafenkürzel = ´´) and (Ab.Stadt <> ´´) and (Ab.Land <> ´´)
           implies Fvbd->forAll(fvbd | Flugverbindung->exists(fvb1 |
                 ( fvbd.Reisebüronummer   = fvb1.Reisebüronummer)
             and ( fvbd.Verbindungsnummer = fvb1.Verbindungsnummer )
             and ( fvbd.Abflugdatum       = fvb1.Abflugdatum )
             and ( fvb1.Abflugort.Stadt   = Ab.Stadt )
             and ( fvb1.Abflugort.Land    = Ab.Land  ) ) )

Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ..., Fvbd, ...)
 post: (Ab.Flughafenkürzel = ´´) and (Ab.Stadt <> ´´) and (Ab.Land = ´´)
           implies Fvbd->forAll(fvbd | fvbd.Abflugstadt = Ab.Stadt)
```

Bemerkung: Da in dieser Bedingung nur die Abflugstadt vorkommt und nicht das Land (welches nicht Teil von *Flugverbindungsdaten* ist!), ist diese Bedingung einfacher als die vorherige und die folgende auszudrücken.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Ab, ..., Fvbd, ...)
 post: (Ab.Flughafenkürzel = ´´) and (Ab.Stadt = ´´) and (Ab.Land <> ´´)
           implies Fvbd->forAll(fvbd | Flugverbindung->exists(fvb1 |
                 ( fvbd.Reisebüronummer   = fvb1.Reisebüronummer)
             and ( fvbd.Verbindungsnummer = fvb1.Verbindungsnummer )
             and ( fvbd.Abflugdatum       = fvb1.Abflugdatum )
             and ( fvb1.Abflugort.Land    = Ab.Land  ) ) )
```

3.2.9 Selektion bezüglich Ankunftsort

Die Selektion bezüglich des Parameters *Ankunftsort* erfolgt analog zum *Abflugort*. Siehe auch die Nachbedingungen in 3.2.8.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ..., Fvbd, ...)
 post: (An.Flughafenkürzel <> ´´) implies
           Fvbd->forAll(fvbd | fvbd.Zielflughafen = An.Flughafenkürzel )

Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ..., Fvbd, ...)
 post: (An.Flughafenkürzel = ´´) and (An.Stadt <> ´´) and (An.Land <> ´´)
           implies Fvbd->forAll(fvbd | Flugverbindung->exists(fvb1 |
                 ( fvbd.Reisebüronummer   = fvb1.Reisebüronummer)
             and ( fvbd.Verbindungsnummer = fvb1.Verbindungsnummer )
             and ( fvbd.Abflugdatum       = fvb1.Abflugdatum )
             and ( fvb1.Ankunftsort.Stadt = An.Stadt )
             and ( fvb1.Ankunftsort.Land  = An.Land  ) ) )

Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ..., Fvbd, ...)
 post: (An.Flughafenkürzel = ´´) and (An.Stadt <> ´´) and (An.Land = ´´)
           implies Fvbd->forAll(fvbd | fvbd.Ankunftsstadt = An.Stadt)

Flugticketverkauf::Flugverbindung::LiefereListe(..., An, ..., Fvbd, ...)
 post: (An.Flughafenkürzel = ´´) and (An.Stadt = ´´) and (An.Land <> ´´)
           implies Fvbd->forAll(fvbd | Flugverbindung->exists(fvb1 |
                 ( fvbd.Reisebüronummer   = fvb1.Reisebüronummer)
             and ( fvbd.Verbindungsnummer = fvb1.Verbindungsnummer )
             and ( fvbd.Abflugdatum       = fvb1.Abflugdatum )
             and ( fvb1.Ankunftsort.Land  = An.Land  ) ) )
```

3.2.10 Selektion bezüglich Datumsbereich

Ist der Parameter *Datumsbereich* nicht leer, so werden nur *Flugverbindungen* zu den gewünschten Flugdaten selektiert. Dabei wird immer bezüglich des Abflugdatums selektiert. Ein Datum ist für die Selektion gültig, wenn es eingeschlossen (Sign = I) und gleichzeitig nicht ausgeschlossen (Sign = E) wurde. Dabei können entweder einzelne Daten (Option = EQ, Datum in Low) oder ganze Intervalle (Option = BT, untere Intervallgrenze in Low, obere Intervallgrenze in High; Intervallgrenzen gehören mit zum Intervall) ein- oder ausgeschlossen werden.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Dab, ..., Fvbd, ...)
 post: Dab->size <> 0 implies
           Fvbd->forAll(fvbd |
                 Dab->exists(da | (da.Sign = ´I´) and (da.Option = ´EQ´)
                              and (da.Low = fvbd.Abflugdatum) )
              or Dab->exists(da | (da.Sign = ´I´) and (da.Option = ´BT´)
                              and (da.Low  <= fvbd.Abflugdatum)
                              and (da.High >= fvbd.Abflugdatum) )
         and not Dab->exists(da | (da.Sign = ´E´) and (da.Option = ´EQ´)
                              and (da.Low = fvbd.Abflugdatum) )
         and not Dab->exists(da | (da.Sign = ´E´) and (da.Option = ´BT´)
                              and (da.Low  <= fvbd.Abflugdatum)
                              and (da.High >= fvbd.Abflugdatum) )
```

3.2.11 Einschränkung der Listenlänge

Ist der Parameter *Listenlänge* mit n > 0 gefüllt, dann werden maximal n Flugverbindungen im Parameter *Flugverbindungsdaten* zurückgegeben. Diese Einschränkung vermeidet Performancenachteile bei zu ungenauen Selektionsbedingungen. Es kann keine Aussage darüber getroffen werden, auf welche n *Flugverbindungen* die Selektion eingeschränkt wird:

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Anz, Fvbd, ...)
 post: (Anz > 0) implies Fvbd->size <= Anz
```

3.2.12 Vollständigkeit der Flugverbindungsliste

Gilt für den Parameter *Listenlänge* n = 0 oder enthält die Flugverbindungsliste < n Einträge, dann wurde die Trefferliste nicht durch den Parameter *Listenlänge* beschränkt. In diesem Fall kann davon ausgegangen werden, dass im Parameter *Flugverbindungsdaten* alle Flugverbindungen zurückgegeben werden, die den gestellten Bedingungen genügen.

Enthält der Parameter *Flugverbindungsdaten* genau n Einträge, dann kann nicht entschieden werden, ob die Trefferliste durch den Parameter *Listenlänge* = n beschränkt wurde oder ob auch ohne Einschränkung die Trefferliste aus genau n Einträgen besteht.

Bemerkung: Diese Aussage kann man nur in OCL formulieren, indem alle notwendigen Bedingungen leicht modifiziert noch einmal aufgeführt werden. Dies ist sehr unschön. Deshalb verzichten wir an dieser Stelle auf die Formulierung der Bedingung in OCL.

3.2.13 Warnung bei leerer Liste von Flugverbindungsdaten

Der Parameter *Statusmeldungen* enthält eine entsprechende Warnung, wenn der Dienst erfolgreich abgearbeitet wurde, jedoch keine *Flugverbindung* gefunden werden konnte, die den Selektionsbedingungen genügt.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Fvbd, ..., Status)
  post: (not Status->exists(st | st.Typ = ´E´) and (Fvbd->size = 0)) implies
            Status->exists(st | st.Typ = ´W´ and st.Nummer = ´251´)
```

3.2.14 Statusmeldung über Dienstabarbeitung

Im Parameter *Statusmeldungen* wird neben den konkreten Meldungen auch ein Gesamtstatus der Dienstabarbeitung übergeben. Dies erfolgt entweder in Form einer Erfolgsmeldung oder einer zusammenfassenden Fehlermeldung.

```
Flugticketverkauf::Flugverbindung::LiefereListe(..., Status)
  post: not Status->exists(st | st.Typ = ´E´) implies
            Status->exists(st | st.Typ = ´S´ and st.Nummer = ´000´)

Flugticketverkauf::Flugverbindung::LiefereListe(..., Fvbd, Status):
  post: Status->exists(st | st.Typ = ´E´) implies
            (Fvbd->size = 0) and
            Status->exists(st | st.Typ = ´E´ and st.Nummer = ´001´)
```

3.3 Flugverbindung::LiefereDetails

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugverbindung::LiefereDetails*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugverbindung { ...

      void LiefereDetails(
            in  ReisebüronummerTyp  Reisebüronummer,
            in  FlugverbindungsnummerTyp  Verbindungsnummer,
            in  DatumTyp                  Abflugdatum,
            out FlugverbindungsdatenTyp   Flugverbindungsdaten,
            out TeilstreckenlistenTyp     Teilstreckenliste,
            out VerbindungspreisTyp       Preis,
            out VerfügbarkeitslistenTyp   Verfügbarkeit,
            out StatuslistenTyp           Statusmeldungen); }; ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugverbindung::LiefereDetails(
                        In    Rbnr:   ReisebüronummerTyp,
                              Vbnr:   FlugverbindungsnummerTyp,
                              Fldat:  DatumTyp,
                        Out   Fvbd:   FlugverbindungsdatenTyp,
                              Tstrl:  TeilstreckenlistenTyp,
                              Preis:  VerbindungspreisTyp,
                              Vfb:    VerfügbarkeitslistenTyp,
                              Status: StatuslistenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer nur abgekürzt und ohne Datentypen angegeben.

3.3.1 Flugverbindung existiert

Die angegebene *Flugverbindung* (definiert durch *Reisebüronummer* und *Verbindungsnummer*) existiert. Ist die Vorbedingung nicht erfüllt, wird ein entsprechender Fehler zurückgegeben.

Flugticketverkauf::Flugverbindung::LiefereDetails(Rbnr, Vbnr, Fldat, ...)

```
 pre FlugverbindungExistiert:
      Flugverbindung->exists(fvb | fvb.Reisebüronummer   = Rbnr
                                 and fvb.Verbindungsnummer = Vbnr
                                 and fvb.Abflugdatum       = Fldat)

 post: FlugverbindungExistiert = false implies
            Status->exists(st | st.Typ = ´E´ and st.Nummer = ´250´)
```

3.3.2 Zusammenhang Verbindungsdaten und Flugverbindung

Die vom Dienst zurückgegebenen Daten beschreiben eine konkrete Flugverbindung. Das heißt, es gibt eine (in OCL als Objekt modellierte) Entität von *Flugverbindung*, deren Attribute mit den Exportdaten des Dienstes übereinstimmen. Dies auszudrücken, ist etwas länglich:

Flugticketverkauf::Flugverbindung::LiefereDetails(..., Fvbd, ...)

```
 post: Flugverbindung->exists (fvb |
            Fvbd.Reisebüronummer    =  fvb.Reisebüronummer
       and  Fvbd.Verbindungsnummer  =  fvb.Verbindungsnummer
       and  Fvbd.Abflugdatum        =  fvb.Abflugdatum
       and  Fvbd.Abflugzeit         =  fvb.Abflugzeit
       and  Fvbd.Startflughafen     =  fvb.Abflugort.Kürzel
       and  Fvbd.Abflugstadt        =  fvb.Abflugort.Stadt
       and  Fvbd.Ankunftsdatum      =  fvb.Ankunftsdatum
       and  Fvbd.Ankunftszeit       =  fvb.Ankunftszeit
       and  Fvbd.Zielflughafen      =  fvb.Ankunftsort.Kürzel
       and  Fvbd.Ankunftsstadt      =  fvb.Ankunftsort.Stadt
       and  Fvbd.Flugdauer          =  fvb.Flugdauer
       and  Fvbd.AnzahlTeilstrecken =  fvb.AnzahlTeilstrecken)
```

Ist n = AnzahlTeilstrecken, dann enthält der Parameter *Tstrl* genau n Einträge, welche von 1 bis n durchnummeriert sind.

Flugticketverkauf::Flugverbindung::LiefereDetails(..., Fvbd, Tstrl, ...)

```
 post: let n = Fvbd.AnzahlTeilstrecken in
            Tstrl->size = n
       and integer->forAll(m | 1 <= m <= n implies
                 Tstrl->exists (tstr | tstr.Nummer = m) )
```

Außerdem entsprechen die Daten der Einträge in *Tstrl* den entsprechenden Teilstrecken der Flugverbindung.

Flugticketverkauf::Flugverbindung::LiefereDetails(..., Fvbd, Tstrl, ...)

```
 post: let fvb = Flugverbindung->any(fvb1 |
            fvb.Reisebüronummer    =  Fvbd.Reisebüronummer
       and  fvb.Verbindungsnummer  =  Fvbd.Verbindungsnummer
       and  fvb.Abflugdatum        =  Fvbd.Abflugdatum )   in

     Tstrl->forAll(tstr | let fl = fvb.Teilstrecke[tstr.Nummer] in

                 tstr.Fluggesellschaft   = fl.FluggesellschaftID
```

```
        and   tstr.Flugnummer          = fl.Nummer
        and   tstr.Startflughafen      = fl.Flugnummer.Abflugort.Kürzel
        and   tstr.Abflugstadt         = fl.Flugnummer.Abflugort.Stadt
        and   tstr.Abflugland          = fl.Flugnummer.Abflugort.Land
        and   tstr.Zielflughafen       = fl.Flugnummer.Ankunftsort.Kürzel
        and   tstr.Ankunftsstadt       = fl.Flugnummer.Ankunftsort.Stadt
        and   tstr.Ankunftsland        = fl.Flugnummer.Ankunftsort.Land
        and   tstr.Abflugdatum         = fl.Abflugdatum
        and   tstr.Abflugzeit          = fl.Flugnummer.Abflugzeit
        and   tstr.Ankunftsdatum       = fl.Ankunftsdatum
        and   tstr.Ankunftszeit        = fl.Flugnummer.Ankunftszeit )
```

Bemerkung: Diese Bedingung hat sich aufgrund des veränderten UML-Modells syntaktisch leicht verändert, ist aber semantisch gleich geblieben.

Analog stimmen auch die Preise überein:

`Flugticketverkauf::Flugverbindung::LiefereDetails(..., Fvbd, ..., Preis, ...)`

```
 post: Flugverbindung->exists (fvb |
           Fvbd.Reisebüronummer    =  fvb.Reisebüronummer
     and   Fvbd.Verbindungsnummer  =  fvb.Verbindungsnummer
     and   Fvbd.Abflugdatum        =  fvb.Abflugdatum
     and   Preis.EcoErw            =  fvb.Preis.EcoErw
     and   Preis.EcoKind           =  fvb.Preis.EcoKind
     and   Preis.EcoKleinkind      =  fvb.Preis.EcoKleinkind
     and   Preis.BusErw            =  fvb.Preis.BusErw
     and   Preis.BusKind           =  fvb.Preis.BusKind
     and   Preis.BusKleinkind      =  fvb.Preis.BusKleinkind
     and   Preis.FirstErw          =  fvb.Preis.FirstErw
     and   Preis.FirstKind         =  fvb.Preis.FirstKind
     and   Preis.FirstKleinkind    =  fvb.Preis.FirstKleinkind
     and   Preis.Steuer            =  fvb.Preis.Steuer
     and   Preis.Währung           =  fvb.Preis.Währung )
```

Bemerkung: Da es sich hier um eine *Flugverbindung* handelt, gelten alle Invarianten aus Kapitel 3.1.

3.3.3 Verfügbarkeit auf den Teilstrecken

Der Parameter *Verfügbarkeit* enthält Informationen zur Platzverfügbarkeit auf den Teilstrecken der Flugverbindung. Dabei handelt es sich gerade um die Verfügbarkeit der Einzelflüge (der jeweiligen Teilstrecke). Diese muss mithilfe eines externen Dienstes ermittelt werden:

`Flugticketverkauf::Flugverbindung::LiefereDetails(Rbnr, Vbnr, ..., Vfb,...)`

```
 post: Vfb->forAll(vfb | Tstrl->exists(tstr | (tstr.Nummer = vfb.Nummer)
  and let ( Fg1  = tstr.Fluggesellschaft and
            Fnr1 = tstr.Flugnummer and
            Fda1 = tstr.Abflugdatum ) in

      let (flvb = Extern::Flugverfügbarkeit::Check
                  (Fg1, Fnr1, Fda1, Fvb1, St).Flugverfügbarkeit) in

            vfb.EcoFrei   = flvb.EcoFrei   and
            vfb.EcoMax    = flvb.EcoMax    and
            vfb.BusFrei   = flvb.BusFrei   and
            vfb.BusMax    = flvb.BusMax    and
            vfb.FirstFrei = flvb.FirstFrei and
```

```
            vfb.FirstMax   = flvb.FirstMax    ) )
```

Bemerkung 1: Die OCL-Bedingung ist folgendermaßen zu lesen: Für jeden Eintrag *vfb* des Parameters *Vfb* wird der zugehörige Eintrag *tstr* im Parameter *Tstrl* bestimmt. Für den mit *tstr* verbundenen Flug wird über den externen Dienst die Verfügbarkeit *flvb* ermittelt. Die Daten der Verfügbarkeit des Fluges müssen identisch zu den Daten im entsprechenden Eintrag *vfb* sein.

Bemerkung 2: Leider enthält die OCL keinen Hinweis darauf, wie man bei einer Methode Bezug auf die Exportparameter nehmen kann. Deshalb haben wir die übliche Teilkomponenten-Notation mit **.** verwendet.

D.h. `Check(..,Fvb1,...).Flugverfügbarkeit` ist vom Typ des Parameters *Flugverfügbarkeit* und enthält gerade die Daten, die beim Aufruf der Methode *Check* im Parameter Flugverfügbarkeit zurückgegeben werden.

3.3.4 Statusmeldung über Dienstabarbeitung

Im Parameter *Statusmeldungen* wird neben den konkreten Meldungen auch ein Gesamtstatus der Dienstabarbeitung übergeben. Dies erfolgt entweder in Form einer Erfolgsmeldung oder einer zusammenfassenden Fehlermeldung.

```
Flugticketverkauf::Flugverbindung::LiefereDetails(..., Status)
 post: not Status->exists(st | st.Typ = ´E´) implies
           Status->exists(st | st.Typ = ´S´ and st.Nummer = ´000´)

Flugticketverkauf::Flugverbindung::LiefereDetails(..., Status)
 post: Status->exists(st | st.Typ = ´E´) implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´001´)
```

3.4 Flugreise allgemein

In diesem Abschnitt werden eine Reihe von Invarianten aufgeführt, die von *Flugreisen* jederzeit erfüllt werden.

3.4.1 Identifikation einer Flugreise

Eine Flugreise wird durch die Attribute *Reisebüronummer* und *Reisenummer* eindeutig identifiziert.

```
Flugticketverkauf
 inv: self.Flugreise->forAll(flr1, flr2 | flr1 <> flr2 implies
           flr1.Reisebüronummer    <> flr2.Reisebüronummer or
           flr1.Reisenummer        <> flr2.Verbindungsnummer )
```

3.4.2 Reisebüro existiert

Das die Flugreise verkaufende *Reisebüro* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung.

```
Flugticketverkauf::Flugreise
 inv: Extern::Reisebüro::PrüfeExistenz(self.Reisebüronummer) = ´X´
```

3.4.3 Flugkunde existiert

Der die Flugreise kaufende Flugkunde existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung.

```
Flugticketverkauf::Flugreise
  inv: Extern::Flugkunde::PrüfeExistenz(self.Flugkunde) = ´X´
```

3.4.4 Bedingungen an Hinflug und Rückflug

Hinflug und Rückflug beziehen sich jeweils auf eine vom (die Flugreise verkaufenden) Reisebüro angebotene Flugverbindung.

```
Flugticketverkauf::Flugreise
  inv: self.Reisebüronummer = self.Hinflug.Reisebüronummer
  inv:  self.Rückflug->notEmpty implies
            self.Reisebüronummer = self.Rückflug.Reisebüronummer
```

3.4.5 Bedingungen an das Datum von Hinflug und Rückflug

Hinflug und Rückflug dürfen nicht vor dem *Buchungsdatum* liegen. Der Rückflug darf nicht vor dem Hinflug liegen.

```
Flugticketverkauf::Flugreise
  inv: self.Buchungsdatum <= self.Hinflug.Abflugdatum
  inv: self.Rückflug->notEmpty implies
            self.Hinflug.Abflugdatum <= self.Rückflug.Abflugdatum
```

Bemerkung: Das Datum ist hier ein String der Länge 8 der Form JJJJMMTT. Die „<=“-Beziehung für Strings entspricht aber gerade der zeitlichen Abfolge verschiedener Daten, wie wir dies erwarten würden.

3.4.6 Gültige Flugklassen

Bei der *Flugklasse* kann es sich nur um eine der drei Klassen handeln: Y (Economy Class), C (Business Class) oder F (First Class).

```
Flugticketverkauf::Flugreise
  inv: (self.Flugklasse = `Y`) or (self.Flugklasse = `C`) or
        (self.Flugklasse = `F`)
```

3.4.7 Gültiger Buchungsstatus

Beim *Buchungsstatus* kann es sich nur um einen der zwei Werte handeln: B (gebucht), C (storniert).

```
Flugticketverkauf::Flugreise
  inv: (self.Buchungsstatus = `B`) or (self.Buchungsstatus = `C`)
```

3.4.8 Anzahl der Passagiere

Die Gesamtanzahl der Passagiere (Kardinalität der mit der Flugreise verbundenen Passagierliste) ist die Summe der Attribute *AnzahlErwachsener*, *AnzahlKinder* und *AnzahlKleinkinder*.

```
Flugticketverkauf::Flugreise
  inv: self.Passagier->size = self.AnzahlErwachsene + self.AnzahlKinder
```

```
                                    + self.AnzahlKleinkinder
```

Das Attribut *AnzahlKinder* beschreibt die Anzahl der Passagiere, die am Tage des Hinflugs mindestens 2 und höchstens 11 Jahre alt sind. Dabei beschreibt *PassAlter* das Alter des Passagiers in ganzen Jahren.

```
Flugticketverkauf::Flugreise
  inv: self.AnzahlKinder = self.Passagiere->select(pass |
            let (GebJahr  = pass.Geburtsdatum.div(10000)) in
            let (GebTag   = pass.Geburtsdatum - GebJahr * 10000) in
            let (FlugJahr = self.Hinflug.Abflugdatum.div(10000)) in
            let (FlugTag  = self.Hinflug.Abflugdatum - FlugJahr * 10000) in
            let   if    GebTag <= FlugTag
                  then  PassAlter = FlugJahr - GebJahr
                  else  PassAlter = FlugJahr - GebJahr - 1 endif  in

            (1 < PassAlter) and (PassAlter < 12) )->size
```

Das Attribut *AnzahlKleinkinder* beschreibt die Anzahl der Passagiere, die am Tage des Hinflugs jünger als 2 Jahre sind. Dabei beschreibt *PassAlter* das Alter des Passagiers am Abflugtag in ganzen Jahren.

```
Flugticketverkauf::Flugreise
  inv: self.AnzahlKleinkinder = self.Passagier->select(pass |
            let (GebJahr  = pass.Geburtsdatum.div(10000)) in
            let (GebTag   = pass.Geburtsdatum - GebJahr * 10000) in
            let (FlugJahr = self.Hinflug.Abflugdatum.div(10000)) in
            let (FlugTag  = self.Hinflug.Abflugdatum - FlugJahr * 10000) in
            let   if    GebTag <= FlugTag
                  then  PassAlter = FlugJahr - GebJahr
                  else  PassAlter = FlugJahr - GebJahr - 1 endif  in

            PassAlter < 2 )->size
```

Bemerkung: Die Berechnung des Alters erfolgt so: Ist das Geburtsdatum 19901003 (= 3.10.1990), so ist *GebJahr* = 1990 und *GebTag* = 1003. Analog werden *FlugJahr* und *FlugTag* bestimmt. Nun ist *PassAlter* die Differenz von *GebJahr* und *FlugJahr*, wenn der *FlugTag* nicht vor dem *GebTag* liegt. Ansonsten ist *PassAlter* um eins geringer.

3.4.9 Abrechnungswährung der Flugreise

Die Flugreise wird in der Hauswährung des Reisebüros abgerechnet.

```
Flugticketverkauf::Flugreise
  inv: Extern::Reisebüro.LiefereWährung(self.Reisebüronummer)
                          = self.Flugreisepreis.Währung
```

3.4.10 Gesamtpreis der Flugreise

Der Gesamtpreis der Flugreise wird folgendermaßen ermittelt: Der Preis für den Hinflug (*preis1*) ist gerade die Summe der Preise für die einzelnen Passagiere. Diese ergeben sich aus den Preisen für die Flugverbindung, wobei der gültige Tarif und die Flugklasse berücksichtigt werden. Analog ergibt sich der Preis für den Rückflug (*preis2*), falls der Rückflug existiert. Anderenfalls ist der Preis für den Rückflug null. Der Gesamtpreis ist nun die Summe von Hinflug- und Rückflugspreis unter Abzug eines eventuellen Kundenrabatts (*rabatt*).

Die Steuer der Flugreise ist einfach die Summe der Flugverbindungssteuern für alle Passagiere.

Flugticketverkauf::Flugreise

```
 inv: let ((Flugklasse = `Y` implies
      preis1 = self.Hinflug.Preis.EcoErw * self.AnzahlErwachsene
                + self.Hinflug.Preis.EcoKind * self.AnzahlKinder
                + self.Hinflug.Preis.EcoKleinkind *
                                        self.AnzahlKleinkinder ) and
          (Flugklasse = `C` implies
      preis1 = self.Hinflug.Preis.BusErw * self.AnzahlErwachsene
                + self.Hinflug.Preis.BusKind * self.AnzahlKinder
                + self.Hinflug.Preis.BusKleinkind *
                                        self.AnzahlKleinkinder ) and
          (Flugklasse = `F` implies
           preis1 = self.Hinflug.Preis.FirstErw * self.AnzahlErwachsene
                     + self.Hinflug.Preis.FirstKind * self.AnzahlKinder
                     + self.Hinflug.Preis.FirstKleinkind *
                                        self.AnzahlKleinkinder ) ) in

     let (if self.Rückflug->notEmpty then
           (Flugklasse = `Y` implies
            preis2 = self.Rückflug.Preis.EcoErw * self.AnzahlErwachsene
                     + self.Rückflug.Preis.EcoKind * self.AnzahlKinder
                     + self.Rückflug.Preis.EcoKleinkind *
                                        self.AnzahlKleinkinder ) and
           (Flugklasse = `C` implies
            preis2 = self.Rückflug.Preis.BusErw * self.AnzahlErwachsene
                     + self.Rückflug.Preis.BusKind * self.AnzahlKinder
                     + self.Rückflug.Preis.BusKleinkind *
                                        self.AnzahlKleinkinder ) and
           (Flugklasse = `F` implies
            preis2 = self.Rückflug.Preis.FirstErw * self.AnzahlErwachsene
                     + self.Rückflug.Preis.FirstKind * self.AnzahlKinder
                     + self.Rückflug.Preis.FirstKleinkind *
                                        self.AnzahlKleinkinder )
         else (preis2 = 0) endif) in

   let (rabatt = Extern::Flugkunde.LiefereRabatt(self.Flugkunde)) in

     self.Flugreisepreis.Summe = (preis1 + preis2) * (1 - rabatt)

 inv: let (st1 = self.Hinflug.Preis.Steuer  * self.Passagier->size) in
      let (if self.Rückflug->notEmpty then
                (st2 = self.Rückflug.Preis.Steuer * self.Passagier->size)
           else (st2 = 0) endif) in

           self.Flugreisepreis.Steuer = st1 + st2
```

Bemerkung: Da der Rückflug optional ist, sind alle Ausdrücke *self.Rückflug* nur definiert, wenn der Rückflug auch existiert. Deshalb ist beim Rückflug immer die Zusatzabfrage *self.Rückflug->notEmpty* notwendig.

Bemerkung2: Da es sich bei der Währung der Flugverbindungen und bei der Währung der Flugreise jeweils um die Hauswährung des Reisebüros handelt, kann der Preis der Flugreise ohne weitere Währungsumrechnung ermittelt werden.

3.5 Flugreise::LiefereListe

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugreise::LiefereListe*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugreise { ...

      void LiefereListe(
            in  ReisebüronummerTyp       Reisebüronummer,
            in  FlugkundeTyp             Flugkundennummer,
            in  DatumsbereichlistenTyp   Flugdatumsbereich,
            in  DatumsbereichlistenTyp   Buchungsdatumsbereich,
            in  ListenlängeTyp           Listenlänge,
            out FlugreiselistenTyp       Flugreisedaten,
            out StatuslistenTyp          Statusmeldungen); ... };
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugreise::LiefereListe(
                        Rbnr:   ReisebüronummerTyp,
                        Fknr:   FlugkundeTyp,
                        Fdab:   DatumsbereichlistenTyp,
                        Bdab:   DatumsbereichlistenTyp,
                        Anz:    ListenlängeTyp,
                        Flrd:   FlugreiselistenTyp,
                        Status: StatuslistenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer nur abgekürzt und ohne Datentypen angegeben.

3.5.1 Reisebüro existiert

Das angegebene *Reisebüro* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung. Ist die Vorbedingung nicht erfüllt, wird ein entsprechender Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::LiefereListe(Rbnr, ..., Status)
 pre ReisebüroExistiert:
           Extern::Reisebüro::PrüfeExistenz(Rbnr) = ´X´

 post: ReisebüroExistiert = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´151´)
```

3.5.2 Flugkunde existiert

Der *Flugkunde* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung. Ist die Vorbedingung nicht erfüllt, wird ein entsprechender Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::LiefereListe(..., Fknr, ..., Status)
 pre FlugkundeExistiert:  (Fknr <> ´´) implies
           Extern::Flugkunde::PrüfeExistenz(Fknr) = ´X´
```

```
post: FlugkundeExistiert = false implies
          Status->exists(st | st.Typ = ´E´ and st.Nummer = ´150´)
```

Man beachte, dass der Parameter *Flugkundennummer* optional ist. Deshalb ist die Überprüfung des Flugkunden nur sinnvoll, wenn der Parameter gefüllt ist.

3.5.3 Zusammenhang Flugreisedaten und Flugreisen

Die im Tabellenparameter *Flugreisedaten* zurückgegebenen Einträge repräsentieren Flugreisen. Das bedeutet, zu jedem Eintrag in *Flugreisedaten* gibt es eine (in OCL als Objekt modellierte) korrespondierende Entität von *Flugreise*. Insbesondere stimmen die entsprechenden Daten überein.

```
Flugticketverkauf::Flugreise::LiefereListe(..., Flrd, ...)
post: Flrd->forAll(flrd | Flugreise->exists (flr |
          flrd.Reisebüronummer    =  flr.Reisebüronummer
     and  flrd.Reisenummer        =  flr.Reisenummer
     and  flrd.Flugkundennummer   =  flr.Flugkunde
     and  flrd.Hinflugverbindung  =  flr.Hinflug.Verbindungsnummer
     and  flrd.Hinflugdatum       =  flr.Hinflug.Abflugdatum
     and  (if flr.Rückflug->notEmpty
          then (flrd.Rückflugverbindung = flr.Rückflug.Verbindungsnummer
                and flrd.Rückflugdatum  = flr.Rückflug.Abflugdatum )
          else (flrd.Rückflugverbindung = ´´
                and flrd.Rückflugdatum  = ´´ )
          endif)
     and  flrd.Flugklasse         =  flr.Flugklasse
     and  flrd.Buchungsdatum      =  flr.Buchungsdatum
     and  flrd.Buchungsstatus     =  flr.Buchungsstatus
     and  flrd.AnzahlErwachsene   =  flr.AnzahlErwachsene
     and  flrd.AnzahlKinder       =  flr.AnzahlKinder
     and  flrd.AnzahlKleinkinder  =  flr.AnzahlKleinkinder    ) )
```

Bemerkung: Bei einer *Flugreise* ist der Rückflug optional. Ist der Rückflug vorhanden, dann entsprechen die Daten zum Rückflug in *Flugreisedaten* denen der korrespondierenden *Flugreise*. Ist kein Rückflug vorhanden, dann sind die entsprechenden Felder in *Flugreisedaten* leer.

Bemerkung 2: Da es sich bei den Einträgen aus der Flugreiseliste um Flugreisen handelt, gelten alle Invarianten aus Kapitel 3.4.

3.5.4 Selektion bezüglich Reisebüro

Alle Flugreisen aus dem Parameter *Flugreisedaten* wurden von dem Reisebüro verkauft, welches durch den Parameter *Reisebüronummer* spezifiziert wurde.

```
Flugticketverkauf::Flugreise::LiefereListe(Rbnr, ..., Flrd, ...)
post: Flrd->forAll(flrd |  flrd.Reisebüronummer = Rbnr)
```

3.5.5 Selektion bezüglich Flugkunde

Wurde im Import ein Flugkunde spezifiziert, werden nur Flugreisen selektiert, die von diesem Flugkunden gebucht wurden.

```
Flugticketverkauf::Flugreise::LiefereListe(..., Fknr, ..., Flrd, ...)
post: (Fknr <> ´´) implies
```

```
Flrd->forAll(flrd | flrd.Flugkundennummer = Fknr )
```

3.5.6 Selektion bezüglich Buchungsdatumsbereich

Ist der Parameter *Buchungsdatumsbereich* nicht leer, so werden nur die Flugreisen zu den gewünschten Buchungsdaten selektiert. Ein Datum ist für die Selektion gültig, wenn es eingeschlossen (Sign = I) und gleichzeitig nicht ausgeschlossen (Sign = E) wurde. Dabei können entweder einzelne Daten (Option = EQ, Datum in Low) oder ganze Intervalle (Option = BT, untere Intervallgrenze in Low, obere Intervallgrenze in High; Intervallgrenzen gehören mit zum Intervall) ein- oder ausgeschlossen werden.

Flugticketverkauf::Flugreise::LiefereListe(..., Bdab, ..., Flrd, ...)

```
post: Bdab->size <> 0 implies
          Flrd->forAll(flrd |
                Bdab->exists(da | (da.Sign = ´I´) and (da.Option = ´EQ´)
                                  and (da.Low = flrd.Buchungsdatum) )
         or     Bdab->exists(da | (da.Sign = ´I´) and (da.Option = ´BT´)
                                  and (da.Low  <= flrd.Buchungsdatum)
                                  and (da.High >= flrd.Buchungsdatum) )
        and not Bdab->exists(da | (da.Sign = ´E´) and (da.Option = ´EQ´)
                                  and (da.Low = flrd.Buchungsdatum) )
       and not  Bdab->exists(da | (da.Sign = ´E´) and (da.Option = ´BT´)
                                  and (da.Low  <= flrd.Buchungsdatum)
                                  and (da.High >= flrd.Buchungsdatum)
```

3.5.7 Selektion bezüglich Flugdatumsbereich

Ist der Parameter *Flugdatumsbereich* nicht leer, so werden nur die Flugreisen zu den gewünschten Flugdaten selektiert. Dabei wird immer bezüglich des Hinflugdatums selektiert. Ein Datum ist für die Selektion gültig, wenn es eingeschlossen (Sign = I) und gleichzeitig nicht ausgeschlossen (Sign = E) wurde. Dabei können entweder einzelne Daten (Option = EQ, Datum in Low) oder ganze Intervalle (Option = BT, untere Intervallgrenze in Low, obere Intervallgrenze in High; Intervallgrenzen gehören mit zum Intervall) ein- oder ausgeschlossen werden.

Flugticketverkauf::Flugreise::LiefereListe(..., Fdab, ..., Flrd, ...)

```
post: Fdab->size <> 0 implies
          Flrd->forAll(flrd |
                Fdab->exists(da | (da.Sign = ´I´) and (da.Option = ´EQ´)
                                  and (da.Low = flrd.Hinflugdatum) )
         or     Fdab->exists(da | (da.Sign = ´I´) and (da.Option = ´BT´)
                                  and (da.Low  <= flrd.Hinflugdatum)
                                  and (da.High >= flrd.Hinflugdatum) )
        and not Fdab->exists(da | (da.Sign = ´E´) and (da.Option = ´EQ´)
                                  and (da.Low = flrd.Hinflugdatum) )
        and not Fdab->exists(da | (da.Sign = ´E´) and (da.Option = ´BT´)
                                  and (da.Low  <= flrd.Hinflugdatum)
                                  and (da.High >= flrd.Hinflugdatum) )
```

3.5.8 Einschränkung der Listenlänge

Ist der Parameter *Listenlänge* mit n > 0 gefüllt, dann werden maximal n Flugreisen im Parameter *Flugreisedaten* zurückgegeben. Diese Einschränkung vermeidet Performancenachteile bei zu ungenauen Selektionsbedingungen. Es kann keine Aussage darüber getroffen werden, auf welche n *Flugreisen* die Selektion eingeschränkt wird.

Flugticketverkauf::Flugreise::LiefereListe(..., Anz, Flrd, ...)

```
post: (Anz > 0) implies Flrd->size <= Anz
```

3.5.9 Vollständigkeit der Liste der Flugreisedaten

Gilt für den Parameter *Listenlänge* n = 0 oder enthält der Parameter *Flugreisedaten* < n Einträge, dann wurde die Trefferliste nicht durch den Parameter *Listenlänge* beschränkt. In diesem Fall kann davon ausgegangen werden, dass im Parameter *Flugreisedaten* alle Flugreisen zurückgegeben werden, die den gestellten Bedingungen genügen.

Enthält *Flugreisedaten* genau n Einträge, dann kann nicht entschieden werden, ob die Trefferliste durch den Parameter *Listenlänge* = n beschränkt wurde oder ob auch ohne Einschränkung die Trefferliste aus genau n Einträgen besteht.

Bemerkung: Diese Aussage kann man nur in OCL formulieren, indem alle notwendigen Bedingungen leicht modifiziert noch einmal aufgeführt werden. Dies ist sehr unschön. Deshalb verzichten wir an dieser Stelle auf die Formulierung der Bedingung in OCL.

3.5.10 Warnung bei leerer Liste von Flugreisedaten

Der Parameter *Statusmeldungen* enthält eine entsprechende Warnung, wenn der Dienst erfolgreich abgearbeitet wurde, jedoch keine Flugreise gefunden werden konnte, die den Selektionsbedingungen genügt.

```
Flugticketverkauf::Flugreise::LiefereListe(..., Flrd, ..., Status)
 post: (not Status->exists(st | st.Typ = ´E´) and (Flrd->size = 0)) implies
           Status->exists(st | st.Typ = ´W´ and st.Nummer = ´301´)
```

3.5.11 Statusmeldung über Dienstabarbeitung

Im Parameter *Statusmeldungen* wird neben den konkreten Meldungen auch ein Gesamtstatus der Dienstabarbeitung übergeben. Dies erfolgt entweder in Form einer Erfolgsmeldung oder einer zusammenfassenden Fehlermeldung.

```
Flugticketverkauf::Flugreise::LiefereListe(..., Status)
 post: not Status->exists(st | st.Typ = ´E´) implies
           Status->exists(st | st.Typ = ´S´ and st.Nummer = ´000´)

Flugticketverkauf::Flugreise::LiefereListe(..., Flrd, Status)
 post: Status->exists(st | st.Typ = ´E´) implies
           (Flrd->size = 0) and
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´001´)
```

3.6 Flugreise::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugreise::Anlegen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugreise { ...

      void Anlegen(
            in  ReisedatenTyp            Reisedaten,
            in  PassagierlistenTyp       Passagierliste,
            out ReisebüronummerTyp       Reisebüronummer,
            out ReisenummerTyp           Reisenummer,
            out ReisepreisTyp            Reisepreis,
```

```
            out StatuslistenTyp          Statusmeldungen); };
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugreise::Anlegen(
                        Rd:     ReisedatenTyp,
                        Passl:  PassagierlistenTyp,
                        Rbnr:   ReisebüronummerTyp,
                        Rnr:    ReisenummerTyp,
                        Rprs:   ReisepreisTyp,
                        Status: StatuslistenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer nur abgekürzt und ohne Datentypen angegeben.

3.6.1 Reisebüro existiert

Das angegebene *Reisebüro* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung. Ist die Vorbedingung nicht erfüllt, wird ein entsprechender Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(Rd, ..., Status)
 pre ReisebüroExistiert:
           Extern::Reisebüro::PrüfeExistenz(Rd.Reisebüronummer) = ´X´

 post: ReisebüroExistiert = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´151´)
```

3.6.2 Flugkunde existiert

Der *Flugkunde* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung. Ist die Vorbedingung nicht erfüllt, wird ein entsprechender Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(Rd, ..., Status)
 pre FlugkundeExistiert:
           Extern::Flugkunde::PrüfeExistenz(Rd.Flugkundennummer) = ´X´

 post: FlugkundeExistiert = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´150´)
```

3.6.3 Bedingungen an die Flugverbindung für Hinflug

Die gewünschte *Flugverbindung* für den Hinflug muss existieren und genügend freie Plätze in der gewählten Flugklasse haben. Außerdem darf der Hinflug nicht vor dem Buchungsdatum liegen. Sind die Vorbedingungen nicht erfüllt, werden entsprechende Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(Rd, Passl, ..., Status)
 pre HinflugverbindungExistiert: Flugverbindung->exists(fvb |
          (fvb.Reisebüronummer   = Rd.Reisebüronummer)
      and (fvb.Verbindungsnummer = Rd.Hinflugverbindung)
      and (fvb.Abflugdatum       = Rd.Hinflugdatum)

      and fvb.Teilstrecke->forAll(fl |
```

```
            let (par1 = fl.FluggesellschaftID and par2 = fl.Nummer
                  and par3 = fl.Abflugdatum) in
            let (flvb = Extern::Flugverfügbarkeit::Check
                        (par1, par2, par3, ...).Flugverfügbarkeit) in

            (Rd.Flugklasse = `Y` implies flvb.EcoFrei >= Passl->size )
          and (Rd.Flugklasse = `C` implies flvb.BusFrei >= Passl->size )
          and (Rd.Flugklasse = `F` implies flvb.FirstFrei >= Passl->size )))

  post: HinflugverbindungExistiert = false implies
            Status->exists(st | st.Typ = ´E´ and st.Nummer = ´250´)

  pre GültigesHinflugdatum:   Today() <= Rd.Hinflugdatum

  post: GültigesHinflugdatum = false implies
            Status->exists(st | st.Typ = ´E´ and st.Nummer = ´252´)
```

Bemerkung: Today() ist ein Dienst des Komponentenframeworks, welcher das aktuelle Datum zurückliefert. Dieser Dienst müsste noch näher spezifiziert werden. Dieser steht auf dem Komponentenframework des SAP Web Application Servers in Form einer Attributabfrage zur Verfügung.

Bemerkung: Diese Bedingung hat sich aufgrund des veränderten UML-Modells syntaktisch leicht verändert, ist aber semantisch gleich geblieben.

3.6.4 Bedingungen an die Flugverbindung für Rückflug

Wird ein Rückflug gewünscht, dann muss die entsprechende Flugverbindung existieren und genügend freie Plätze in der gewählten Flugklasse haben. Außerdem darf das Rückflugdatum nicht vor dem Hinflugdatum liegen. Sind die Vorbedingungen nicht erfüllt, werden entsprechende Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(Rd, Passl, ..., Status)
  pre RückflugverbindungExistiert: (Rd.Rückflugverbindung <> ``) implies
      Flugverbindung->exists(fvb |
          (fvb.Reisebüronummer   = Rd.Reisebüronummer)
      and (fvb.Verbindungsnummer = Rd.Rückflugverbindung)
      and (fvb.Abflugdatum       = Rd.Rückflugdatum)

      and fvb.Teilstrecke->forAll(fl |
            let (par1 = fl.FluggesellschaftID and par2 = fl.Nummer
                  and par3 = fl.Abflugdatum) in
            let (flvb = Extern::Flugverfügbarkeit::Check
                        (par1, par2, par3, ...).Flugverfügbarkeit) in

            (Rd.Flugklasse = `Y` implies flvb.EcoFrei >= Passl->size )
          and (Rd.Flugklasse = `C` implies flvb.BusFrei >= Passl->size )
          and (Rd.Flugklasse = `F` implies flvb.FirstFrei >= Passl->size )))

  post: RückflugverbindungExistiert = false implies
            Status->exists(st | st.Typ = ´E´ and st.Nummer = ´250´)

  pre GültigesRückflugdatum:  (Rd.Rückflugverbindung <> ``) implies
                        Rd.Hinflugdatum <= Rd.Rückflugdatum

  post: GültigesRückflugdatum = false implies
```

```
            Status->exists(st | st.Typ = ´E´ and st.Nummer = ´302´)
```

Bemerkung: Diese Bedingung hat sich aufgrund des veränderten UML-Modells syntaktisch leicht verändert, ist aber semantisch gleich geblieben.

3.6.5 Gültige Flugklassen

Bei der gewünschten Flugklasse kann es sich nur um eine der drei Klassen handeln: Y (Economy Class), C (Business Class) oder F (First Class). Ist die Vorbedingung nicht erfüllt, dann wird ein entsprechender Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(Rd, ..., Status)
 pre GültigeFlugklasse:(Rd.Flugklasse = `Y`) or
           (Rd.Flugklasse = `C`) or (Rd.Flugklasse = `F`)

 post: GültigeFlugklasse = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´107´)
```

3.6.6 Bedingungen an die Passagierliste

Die *Passagierliste* muss mindestens einen Eintrag enthalten. Zu jedem Eintrag ist der Name des Passagiers obligatorisch, während Anrede und Geburtsdatum optional sind. Wird das Geburtsdatum angegeben, so muss es sich um ein gültiges Datum in der Vergangenheit handeln. Sind die Vorbedingungen nicht erfüllt, werden entsprechende Fehler zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(..., Passl, ...)
 pre PassagierlisteVorhanden:Passl->size > 0

 post: PassagierlisteVorhanden = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´306´)

 pre PassagiernameVorhanden: Passl->forAll(pass | pass.Name <> ``)

 post: PassagiernameVorhanden = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´303´)

 pre GültigesGeburtsdatum:   Passl->forAll(pass |
           pass.Geburtsdatum <> `` implies
           IsDate(pass.Geburtsdatum) and (pass.Geburtsdatum <= Today() ) )

 post: GültigesGeburtsdatum = false implies
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´010´) or
           Status->exists(st | st.Typ = ´E´ and st.Nummer = ´106´)
```

Bemerkung: IsDate(dat) ist ein Dienst des Komponentenframeworks, welcher einen boolschen Wert zurückliefert. Der Wert ist wahr, wenn es sich bei dat um ein gültiges Datum handelt. Dieser Dienst müsste noch näher spezifiziert werden.

Sowohl in ABAP (Implementierungssprache der Komponente) als auch in WSDL ist es im Gegensatz zur OMG IDL möglich, einen Datentypen für ein Datum zu definieren. Dann würde schon das Typsystem sicherstellen, dass ein übergebenes Datum korrekt ist. Auf diesen Punkt könnte dann hier verzichtet werden.

3.6.7 Neu angelegte Flugreise

Wenn kein Verarbeitungsfehler aufgetreten ist, dann wurde eine neue *Flugreise* angelegt. Die identifizierenden Attribute *Reisebüronummer* und *Reisenummer* werden in den gleichlautenden Parametern zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(..., Rbnr, Rnr, ..., Status)
 post: Status->exists(st | st.Typ = ´S´ and st.Nummer = ´000´) implies
          Flugreise->any(flr |  flr.Reisebüronummer = Rbnr
                             and flr.Reisenummer     = Rnr).oclIsNew
```

Bemerkung: Durch Verwendung des Operators oclIsNew wird ausgedrückt, dass die Entität zum Zeitpunkt post existiert und zum Zeitpunkt pre noch nicht existierte.

3.6.8 Zusammenhang Flugreise und Eingabedaten

Die in den Parametern *Reisedaten* und *Passagierliste* übergebenen Werte finden sich an entsprechender Stelle in der soeben angelegten *Flugreise* wieder.

```
Flugticketverkauf::Flugreise::Anlegen(Rd, Passl , ...)
 post: Flugreise->exists (flr |
          flr.Reisebüronummer           =  Rbnr
     and  flr.Reisenummer               =  Rnr
     and  flr.Reisebüronummer           =  Rd.Reisebüronummer
     and  flr.Flugkunde                 =  Rd.Flugkundennummer
     and  flr.Hinflug.Verbindungsnummer =  Rd.Hinflugverbindung
     and  flr.Hinflug.Abflugdatum       =  Rd.Hinflugdatum
     and  (if (Rd.Rückflugverbindung <> ´´)
           then (flr.Rückflug.Verbindungsnummer = Rd.Rückflugverbindung
                 and flr.Rückflug.Abflugdatum   = Rd.Rückflugdatum)
           else (flr.Rückflug->size = 0) endif)
     and  flr.Flugklasse          =  Rd.Flugklasse
     and  flr.Buchungsdatum       =  Today()
     and  flr.Buchungsstatus      =  ´B´
     and  flr.Passagier->size     = Passl->size
     and  flr.Passagier->forAll(pass | Passl->exists(pass2 |
                          pass.Name         = pass2.Name   and
                          pass.Anrede       = pass2.Anrede and
                          pass.Geburtsdatum = pass2.Geburtsdatum) ) )
     and  Passl->forAll(pass2 | flr.Passagier->exists(pass |
                          pass.Name         = pass2.Name   and
                          pass.Anrede       = pass2.Anrede and
                          pass.Geburtsdatum = pass2.Geburtsdatum) ) )
```

Bemerkung: Da es sich wiederum um eine *Flugreise* handelt, gelten alle Invarianten aus Kapitel 3.4. Dort ist insbesondere beschrieben, welche Werte die Attribute *AnzahlErwachsener*, *AnzahlKinder* und *AnzahlKleinkinder* annehmen. Dieser Punkt soll hier nicht wiederholt werden, auch wenn man dies als eine Nachbedingung betrachten könnte.

3.6.9 Rückgabe des Reisepreises

Beim Anlegen der *Flugreise* wird vom Dienst der Gesamtpreis für die Flugreise ermittelt. Die Berechnungsvorschrift wurde im Abschnitt 3.4.10. beschrieben. Der Gesamtpreis und die Steuern werden im Parameter *Reisepreis* an den Aufrufer zurückgegeben.

```
Flugticketverkauf::Flugreise::Anlegen(... ,Rbnr, Rnr, Rprs, ...)
 post: Flugreise->exists (flr |
```

```
            flr.Reisebüronummer           =  Rbnr
      and   flr.Reisenummer               =  Rnr
      and   flr.Flugreisepreis.Summe      =  Rprs.Summe
      and   flr.Flugreisepreis.Steuer     =  Rprs.Steuer
      and   flr.Flugreisepreis.Währung    =  Rprs.Währung )
```

3.6.10 Statusmeldung über Dienstabarbeitung

Im Parameter *Statusmeldungen* wird neben den konkreten Meldungen auch ein Gesamtstatus der Dienstabarbeitung übergeben. Dies erfolgt entweder in Form einer Erfolgsmeldung oder einer zusammenfassenden Fehlermeldung.

```
Flugticketverkauf::Flugreise::Anlegen(..., Status)
  post: not Status->exists(st | st.Typ = ´E´) implies
            Status->exists(st | st.Typ = ´S´ and st.Nummer = ´000´)

Flugticketverkauf::Flugreise::Anlegen(..., Rbnr, Rnr, ..., Status)
  post: Status->exists(st | st.Typ = ´E´) implies
                  Rbnr = 0 and Rnr = 0 and
            Status->exists(st | st.Typ = ´E´ and st.Nummer = ´001´)
```

Bemerkung: Die folgenden Abschnitte 3.7 bis 3.12 beziehen sich auf die Parametrisierung und sind vollständig neu hinzugekommen.

3.7 Parametrisierung von Flughafen

Dieser Abschnitt enthält alle Bedingungen, die für die Parametrisierung von *Flughafen* gelten.

3.7.1 Flughafen allgemein

In diesem Abschnitt wird eine Invariante aufgeführt, die von *Flughafen* jederzeit erfüllt wird.

- Ein Flughafen wird eindeutig durch das Attribut *Kürzel* identifiziert.

```
Flugticketverkauf
  inv: self.Flughafen->forAll(fh1, fh2 | fh1 <> fh2 implies
                                  fh1.Kürzel  <> fh2.Kürzel )
```

3.7.2 Flughafen::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flughafen::Anlegen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flughafen { ...

      void Anlegen(
            in  FlughafenKeyTyp          FlughafenKey,
            in  FlughafenDatenTyp        FlughafenDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flughafen::Anlegen(
```

```
                                Key:     FlughafenKeyTyp,
                                Daten:   FlughafenDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flughafen::Anlegen(Key, Daten)
  pre: Key.Kürzel <> ´´ and Daten.Name <> ´´ and Daten.Stadt <> ´´
             and Daten.Land <> ´´
```

- Der anzulegende Flughafen darf noch nicht existieren, d.h. es gibt noch keinen Flughafen mit dem gleichen Kürzel.

```
Flugticketverkauf::Flughafen::Anlegen(Key, Daten)
  pre: not Flughafen.exists->(fh | fh.Kürzel = Key.Kürzel)
```

- Das Ergebnis des Dienstes ist, dass ein neuer Flughafen definiert wurde. Die Attribute des Flughafens stimmen mit den entsprechenden Feldern der Input-Parameter überein.

```
Flugticketverkauf::Flughafen::Anlegen(Key, Daten)
  post: Flughafen.any->(fh | fh.Kürzel = Key.Kürzel).oclIsNew

  post: Flughafen.exists->(fh |
                    fh.Kürzel  = Key.Kürzel
                and fh.Name    = Daten.Name
                and fh.Stadt   = Daten.Stadt
                and fh.Land    = Daten.Land )
```

Bemerkung 1: Durch Verwendung des Operators oclIsNew wird ausgedrückt, dass die Entität zum Zeitpunkt *post* existiert und zum Zeitpunkt *pre* noch nicht existierte.

Bemerkung 2: Es fällt auf, dass die hier angegebenen Bedingungen alle selbstverständlich erscheinen. Für eine vollständige Spezifikation müssen sie allerdings aufgenommen werden. Bei den parametrisierungsrelevanten Entitätstypen sind 70% der Bedingungen von dieser Qualität. Es sollte daher untersucht werden, ob solche wiederkehrenden Bedingungen nicht anders berücksichtigt werden können. Dies würde die Verhaltensebene deutlich verkürzen.

3.7.3 Flughafen::Ändern

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flughafen::Ändern*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flughafen { ...

      void Ändern(
            in  FlughafenKeyTyp          FlughafenKey,
            in  FlughafenDatenTyp        FlughafenDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flughafen::Ändern(
                                Key:     FlughafenKeyTyp,
                                Daten:   FlughafenDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flughafen::Ändern(Key, Daten)
  pre: Key.Kürzel <> ´´ and Daten.Name <> ´´ and Daten.Stadt <> ´´
            and Daten.Land <> ´´
```

- Der zu ändernde Flughafen muss schon existieren.

```
Flugticketverkauf::Flughafen::Ändern(Key, Daten)
  pre: Flughafen.exists->(fh | fh.Kürzel = Key.Kürzel)
```

- Das Ergebnis des Dienstes ist, dass die Attribute des Flughafens die entsprechenden Werte der Input-Parameter angenommen haben.

```
Flugticketverkauf::Flughafen::Ändern(Key, Daten)
  post: Flughafen.exists->(fh |
                  fh.Kürzel   = Key.Kürzel
              and fh.Name     = Daten.Name
              and fh.Stadt    = Daten.Stadt
              and fh.Land     = Daten.Land )
```

3.7.4 Flughafen::Löschen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flughafen::Löschen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flughafen { ...

      void Löschen(
            in  FlughafenKeyTyp           FlughafenKey); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flughafen::Löschen(Key:FlughafenKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Flughafen::Löschen(Key)
  pre: Key.Kürzel <> ´´
```

- Der zu löschende Flughafen muss existieren:

```
Flugticketverkauf::Flughafen::Löschen(Key)
  pre: Flughafen.exists->(fh | fh.Kürzel = Key.Kürzel)
```

- Der Flughafen kann nur gelöscht werden, wenn er in keiner Flugnummer mehr referenziert wird:

```
Flugticketverkauf::Flughafen::Löschen(Key)
  pre: Flugnummer.select->(fnr |
             fnr.Abflugort.Kürzel    = Key.Kürzel
          or fnr.Ankunftsort.Kürzel  = Key.Kürzel)->isEmpty()
```

- Im Ergebnis des Dienstes wurde der Flughafen gelöscht.

```
Flugticketverkauf::Flughafen::Löschen(Key)
  post: not Flughafen.exists->(fh | fh.Kürzel = Key.Kürzel)
```

3.8 Parametrisierung von Fluggesellschaft

Dieser Abschnitt enthält alle Bedingungen, die für die Parametrisierung von *Fluggesellschaft* gelten.

3.8.1 Fluggesellschaft allgemein

In diesem Abschnitt wird eine Invariante aufgeführt, die von *Fluggesellschaft* jederzeit erfüllt wird.

- Eine Fluggesellschaft wird eindeutig durch das Attribut *Kürzel* identifiziert.

```
Flugticketverkauf
  inv: self.Fluggesellschaft->forAll(fg1, fg2 | fg1 <> fg2 implies
                  fg1.Kürzel  <> fg2.Kürzel )
```

3.8.2 Fluggesellschaft::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Fluggesellschaft::Anlegen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Fluggesellschaft { ...

      void Anlegen(
            in  FluggesellschaftKeyTyp    FluggesellschaftKey,
            in  FluggesellschaftDatenTyp  FluggesellschaftDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Fluggesellschaft::Anlegen(
                        Key:     FluggesellschaftKeyTyp,
                        Daten:   FluggesellschaftDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Fluggesellschaft::Anlegen(Key, Daten)
  pre: Key.Kürzel <> ´´ and Daten.Name <> ´´ and Daten.Währung <> ´´
```

- Die anzulegende Fluggesellschaft darf noch nicht existieren, d.h. es gibt noch keine Fluggesellschaft mit dem gleichen Kürzel.

```
Flugticketverkauf::Fluggesellschaft::Anlegen(Key, Daten)
  pre: not Fluggesellschaft.exists->(fg | fg.Kürzel = Key.Kürzel)
```

- Das Ergebnis des Dienstes ist, dass eine neue Fluggesellschaft definiert wurde. Die Attribute der Fluggesellschafts stimmen mit den entsprechenden Feldern der Input-Parameter überein.

```
Flugticketverkauf::Fluggesellschaft::Anlegen(Key, Daten)
 post: Fluggesellschaft.any->(fg | fg.Kürzel = Key.Kürzel).oclIsNew

 post: Fluggesellschaft.exists->(fg |
                fg.Kürzel   = Key.Kürzel
             and fg.Name    = Daten.Name
             and fg.Währung  = Daten.Währung )
```

3.8.3 Fluggesellschaft::Ändern

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Fluggesellschaft::Ändern*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Fluggesellschaft { ...

      void Ändern(
            in  FluggesellschaftKeyTyp    FluggesellschaftKey,
            in  FluggesellschaftDatenTyp  FluggesellschaftDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Fluggesellschaft::Ändern(
                        Key:    FluggesellschaftKeyTyp,
                        Daten:  FluggesellschaftDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Fluggesellschaft::Ändern(Key, Daten)
 pre: Key.Kürzel <> ´´ and Daten.Name <> ´´ and Daten.Währung <> ´´
```

- Die zu ändernde Fluggesellschaft muss schon existieren:

```
Flugticketverkauf::Fluggesellschaft::Ändern(Key, Daten)
 pre: Fluggesellschaft.exists->(fg | fg.Kürzel = Key.Kürzel)
```

- Das Ergebnis des Dienstes ist, dass die Attribute der Fluggesellschaft die entsprechenden Werte der Input-Parameter angenommen haben:

```
Flugticketverkauf::Fluggesellschaft::Ändern(Key, Daten)
 post: Fluggesellschaft.exists->(fg |
                fg.Kürzel   = Key.Kürzel
             and fg.Name    = Daten.Name
             and fg.Währung  = Daten.Währung )
```

3.8.4 Fluggesellschaft::Löschen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Fluggesellschaft::Löschen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Fluggesellschaft { ...

      void Löschen(
            in  FluggesellschaftKeyTyp    FluggesellschaftKey); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Fluggesellschaft::Löschen(Key:FluggesellschaftKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Fluggesellschaft::Löschen(Key)
  pre: Key.Kürzel <> ´´
```

- Die zu löschende Fluggesellschaft muss existieren:

```
Flugticketverkauf::Fluggesellschaft::Löschen(Key)
  pre: Fluggesellschaft.exists->(fg | fg.Kürzel = Key.Kürzel)
```

- Die Fluggesellschaft kann nur gelöscht werden, wenn sie in keiner Flugnummer mehr referenziert wird:

```
Flugticketverkauf::Fluggesellschaft::Löschen(Key)
  pre: Flugnummer.select->(fnr |
                   fnr.FluggesellschaftID= Key.Kürzel)->isEmpty()
```

- Im Ergebnis des Dienstes wurde die Fluggesellschaft gelöscht:

```
Flugticketverkauf::Fluggesellschaft::Löschen(Key)
  post: not Fluggesellschaft.exists->(fg | fg.Kürzel = Key.Kürzel)
```

3.8.5 Fluggesellschaft::ZuordnenPreisschema

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Fluggesellschaft::ZuordnenPreisschema*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Fluggesellschaft { ...

      void ZuordnenPreisschema(
            in  FluggesellschaftKeyTyp          FluggesellschaftKey,
            in  PreisschemaKeyTyp               PreisschemaKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Fluggesellschaft::ZuordnenPreisschema(
                          Fg:         FluggesellschaftKeyTyp,
                          Prs:        PreisschemaKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Fluggesellschaft::ZuordnenPreisschema(Fg, Prs)
  pre: Fg.Kürzel <> ´´  and Prs.Schemanummer <> ´´
```

- Die angegebene Fluggesellschaft und das angegebene Preisschema müssen existieren.

```
Flugticketverkauf::Fluggesellschaft::ZuordnenPreisschema(Fg, Prs)
  pre: Fluggesellschaft.exists->(fg | fg.Kürzel = Fg.Kürzel)
  pre: Preisschema.exists->(prs | prs.Nummer = Prs.Schemanummer)
```

- Der Fluggesellschaft darf noch kein Preisschema zugeordnet sein.

```
Flugticketverkauf::Fluggesellschaft::ZuordnenPreisschema(Fg, Prs)
  pre: let fg = Fluggesellschaft.any->(fg1 | fg1.Kürzel = Fg.Kürzel) in
            fg.Preisschema->size = 0
```

Bemerkung: Mit `fg` wird zunächst die betrachtete Fluggesellschaft bezeichnet, damit die Bedingung übersichtlicher wird.

- Das Ergebnis des Dienstes ist, dass der Fluggesellschaft das angegebene Preisschema zugeordnet wurde.

```
Flugticketverkauf::Fluggesellschaft::ZuordnenPreisschema(Fg, Prs)
  post: let fg = Fluggesellschaft.any->(fg1 | fg1.Kürzel = Fg.Kürzel) in
            fg.Preisschema->size = 1
        and fg.Preisschema.Nummer = Prs.Schemanummer
```

3.8.6 Fluggesellschaft::EntfernenPreisschema

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Fluggesellschaft::EntfernenPreisschema*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes.

```
module Flugticketverkauf { ...
  interface Fluggesellschaft { ...

      void EntfernenPreisschema(
            in  FluggesellschaftKeyTyp          FluggesellschaftKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Fluggesellschaft::EntfernenPreisschema(
                          Fg:          FluggesellschaftKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Fluggesellschaft::EntfernenPreisschema(Fg)
  pre: Fg.Kürzel <> ´´
```

- Die angegebene Fluggesellschaft muss existieren.

```
Flugticketverkauf::Fluggesellschaft::EntfernenPreisschema(Fg)
  pre: Fluggesellschaft.exists->(fg | fg.Kürzel = Fg.Kürzel)
```

- Der Fluggesellschaft muss schon ein Preisschema zugeordnet sein.

```
Flugticketverkauf::Fluggesellschaft::EntfernenPreisschema(Fg)
  pre: let fg = Fluggesellschaft.any->(fg1 | fg1.Kürzel = Fg.Kürzel) in
            fg.Preisschema->size = 1
```

- Im Ergebnis des Dienstes wurde die Zuordnung des Preisschemas zur Fluggesellschaft aufgehoben.

```
Flugticketverkauf::Fluggesellschaft::EntfernenPreisschema(Fg)
  post: let fg = Fluggesellschaft.any->(fg1 | fg1.Kürzel = Fg.Kürzel) in
            fg.Preisschema->size = 0
```

3.9 Parametrisierung von Flugnummer

Dieser Abschnitt enthält alle Bedingungen, die für die Parametrisierung von F*lugnummer* gelten.

3.9.1 Flugnummer allgemein

In diesem Abschnitt werden Invarianten aufgeführt, die von *Flugnummer* jederzeit erfüllt werden.

- Eine Flugnummer wird eindeutig durch die Attribute *FluggesellschaftID* und *Nummer* identifiziert.

```
Flugticketverkauf
  inv: self.Flugnummer->forAll(fnr1, fnr2 | fnr1 <> fnr2 implies
            fnr1.FluggesellschaftID <> fnr2.FluggesellschaftID or
            fnr1.Nummer             <> fnr2.Nummer )
```

- Das Attribut *FluggesellschaftID* kann nicht gepflegt werden, sondern ist automatisch gleich dem Attribut *Fluggesellschaft.Kürzel*. Es wurde zusätzlich aufgenommen, um die Navigation im Modell zu erleichtern.

```
Flugticketverkauf::Flugnummer
  inv: self.FluggesellschaftID = self.Fluggesellschaft.Kürzel
```

- Der Abflugort muss ungleich dem Ankunftsort sein.

```
Flugticketverkauf::Flugnummer
  inv: self.Abflugort <> self.Ankunftsort
```

3.9.2 Flugnummer::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugnummer::Anlegen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugnummer { ...

      void Anlegen(
            in  FlugnummerKeyTyp          FlugnummerKey,
            in  FlugnummerDatenTyp        FlugnummerDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugnummer::Anlegen(
                        Key:    FlugnummerKeyTyp,
                        Daten:  FlugnummerDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Einige Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flugnummer::Anlegen(Key, Daten)
 pre: Key.Fluggesellschaft <> ´´   and Key.Nummer <> ´´
  and  Daten.Startflughafen <> ´´  and Daten.Abflugzeit <> ´´
  and  Daten.Zielflughafen <> ´´   and Daten.Ankunftszeit <> ´´
  and  Daten.Flugdauer <> ´´
```

- Die anzulegende Flugnummer darf noch nicht existieren, d.h. es gibt noch keine Flugnummer mit den gleichen Keyattributen.

```
Flugticketverkauf::Flugnummer::Anlegen(Key, Daten)
 pre: not Flugnummer.exists->(fnr |
                fnr.FluggesellschaftID  = Key.Fluggesellschaft
           and  fnr.Nummer              = Key.Nummer)
```

- Das Ergebnis des Dienstes ist, dass eine neue Flugnummer definiert wurde. Die Attribute der Flugnummer stimmen mit den entsprechenden Feldern der Input-Parameter überein. Außerdem wurden Verknüpfungen zu den entsprechenden Entitäten von *Flughafen* und *Fluggesellschaft* hergestellt.

```
Flugticketverkauf::Flugnummer::Anlegen(Key, Daten)
 post: Flugnummer.any->(fnr |
               (fnr.FluggesellschaftID  = Key.Fluggesellschaft
           and  fnr.Nummer              = Key.Nummer      ).oclIsNew

 post: Flugnummer.exists->(fnr |
                fnr.FluggesellschaftID  = Key.Fluggesellschaft
           and  fnr.Nummer              = Key.Nummer
           and  fnr.Abflugort.Kürzel    = Daten.Startflughafen
           and  fnr.Abflugzeit          = Daten.Abflugzeit
           and  fnr.Ankunftsort.Kürzel  = Daten.Zielflughafen
           and  fnr.Ankunftszeit        = Daten.Ankunftszeit
           and  fnr.Flugdauer           = Daten.Flugdauer
           and  ( if Daten.AnkunftTageSpäter <> ´´
                    then fnr.AnkunftTageSpäter = Daten.AnkunftTageSpäter
                      else fnr.AnkunftTageSpäter = 0 ) )
```

Bemerkung 1: Außerdem wird eine Assoziation zu einer Entität von *Fluggesellschaft* aufgebaut. Dies wurde allerdings schon als Invariante in Abschnitt 3.9.1. formuliert und wird deshalb hier nicht wiederholt.

3.9.3 Flugnummer::Ändern

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugnummer::Ändern*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugnummer { ...

      void Ändern(
            in  FlugnummerKeyTyp          FlugnummerKey,
            in  FlugnummerDatenTyp        FlugnummerDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugnummer::Ändern(
                        Key:    FlugnummerKeyTyp,
                        Daten:  FlugnummerDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Einige Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

Flugticketverkauf::Flugnummer::Ändern(Key, Daten)

```
 pre: Key.Fluggesellschaft <> ´´   and Key.Nummer <> ´´
  and  Daten.Startflughafen <> ´´  and Daten.Abflugzeit <> ´´
  and  Daten.Zielflughafen <> ´´   and Daten.Ankunftszeit <> ´´
  and  Daten.Flugdauer <> ´´
```

- Die zu ändernde Flugnummer muss schon existieren.

Flugticketverkauf::Flugnummer::Ändern(Key, Daten)

```
 pre: Flugnummer.exists->(fnr |
           fnr.FluggesellschaftID  = Key.Fluggesellschaft
       and fnr.Nummer              = Key.Nummer)
```

- Das Ergebnis des Dienstes ist, dass die Attribute der Flugnummer die entsprechenden Werte der Input-Parameter angenommen haben.

Flugticketverkauf::Flugnummer::Ändern(Key, Daten)

```
 post: Flugnummer.exists->(fnr |
           fnr.FluggesellschaftID  = Key.Fluggesellschaft
      and  fnr.Nummer              = Key.Nummer
      and  fnr.Abflugort.Kürzel    = Daten.Startflughafen
      and  fnr.Abflugzeit          = Daten.Abflugzeit
      and  fnr.Ankunftsort.Kürzel  = Daten.Zielflughafen
      and  fnr.Ankunftszeit        = Daten.Ankunftszeit
      and  fnr.Flugdauer           = Daten.Flugdauer
      and  ( if Daten.AnkunftTageSpäter <> ´´
               then fnr.AnkunftTageSpäter = Daten.AnkunftTageSpäter
               else fnr.AnkunftTageSpäter = 0 ) )
```

3.9.4 Flugnummer::Löschen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugnummer::Löschen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugnummer { ...

      void Löschen(
           in  FlugnummerKeyTyp          FlugnummerKey); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

Flugticketverkauf::Flugnummer::Löschen(Key:FlugnummerKeyTyp)

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

Flugticketverkauf::Flugnummer::Löschen(Key)

```
 pre: Key.Fluggesellschaft <> ´´   and Key.Nummer <> ´´
```

- Die zu löschende Flugnummer muss existieren:

Flugticketverkauf::Flugnummer::Löschen(Key)

```
 pre: Flugnummer.exists->(fnr |
           fnr.FluggesellschaftID  = Key.Fluggesellschaft
       and fnr.Nummer              = Key.Nummer)
```

- Die Flugnummer kann nur gelöscht werden, wenn sie in keinem Flug und von keiner Flugverbindungsnummer mehr referenziert wird.

```
Flugticketverkauf::Flugnummer::Löschen(Key)
 pre: Flug.select->(fl |
           fl.FluggesellschaftID   = Key.Fluggesellschaft
       and fl.Nummer               = Key.Nummer      )->isEmpty()

 pre: Flugverbindungsnummer.select->(fvbnr |
     fvbnr.Flugnummer.FluggesellschaftID = Key.Fluggesellschaft
  and fvbnr.Flugnummer.Nummer            = Key.Nummer      )->isEmpty()
```

- Im Ergebnis des Dienstes wurde die Flugnummer gelöscht:

```
Flugticketverkauf::Flugnummer::Löschen(Key)
 post: not Flugnummer.exists->(fnr |
           fnr.FluggesellschaftID  = Key.Fluggesellschaft
       and fnr.Nummer              = Key.Nummer)
```

3.9.5 Flugnummer::ZuordnenPreisschema

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugnummer::ZuordnenPreisschema*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugnummer { ...

      void ZuordnenPreisschema(
           in  FlugnummerKeyTyp          FlugnummerKey,
           in  PreisschemaKeyTyp         PreisschemaKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugnummer::ZuordnenPreisschema(
                        Fnr:        FlugnummerKeyTyp,
                        Prs:        PreisschemaKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Flugnummer::ZuordnenPreisschema(Fnr, Prs)
 pre: Fnr.Fluggesellschaft <> ´´ and Fnr.Nummer <> ´´ and
           Prs.Schemanummer <> ´´
```

- Die angegebene Flugnummer und das angegebene Preisschema müssen existieren.

```
Flugticketverkauf::Flugnummer::ZuordnenPreisschema(Fnr, Prs)
 pre: Flugnummer.exists->(fnr |
           fnr.FluggesellschaftID  = Fnr.Fluggesellschaft
       and fnr.Flugnummer          = Fnr.Nummer )

 pre: Preisschema.exists->(prs | prs.Nummer = Prs.Schemanummer)
```

- Der Flugnummer darf noch kein Preisschema zugeordnet sein.

```
Flugticketverkauf::Flugnummer::ZuordnenPreisschema(Fnr, Prs)
 pre: let fnr = Flugnummer.any->(fnr1 |
           fnr1.FluggesellschaftID = Fnr.Fluggesellschaft
       and fnr1.Flugnummer         = Fnr.Nummer) in
           fnr.Preisschema->size = 0
```

Bemerkung: Mit `fnr` wird zunächst die betrachtete Flugnummer bezeichnet, damit die Bedingung übersichtlicher wird.

- Das Ergebnis des Dienstes ist, dass der Flugnummer das angegebene Preisschema zugeordnet wurde.

```
Flugticketverkauf::Flugnummer::ZuordnenPreisschema(Fnr, Prs)
 post: let fnr = Flugnummer.any->(fnr1 |
           fnr1.FluggesellschaftID = Fnr.Fluggesellschaft
       and fnr1.Flugnummer         = Fnr.Nummer ) in
                 fnr.Preisschema->size = 1
             and fnr.Preisschema.Nummer = Prs.Schemanummer
```

3.9.6 Flugnummer::EntfernenPreisschema

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugnummer::EntfernenPreisschema*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugnummer { ...

      void EntfernenPreisschema(
           in  FlugnummerKeyTyp          FlugnummerKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugnummer::EntfernenPreisschema(Fnr: FlugnummerKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Flugnummer::EntfernenPreisschema(Fnr)
 pre: Fnr.Fluggesellschaft <> ´´ and Fnr.Nummer <> ´´
```

- Die angegebene Flugnummer muss existieren.

```
Flugticketverkauf::Flugnummer::EntfernenPreisschema(Fnr)
 pre: Flugnummer.exists->(fnr |
           fnr.FluggesellschaftID  = Fnr.Fluggesellschaft
       and fnr.Flugnummer          = Fnr.Nummer )
```

- Der Flugnummer muss schon ein Preisschema zugeordnet sein.

```
Flugticketverkauf::Flugnummer::EntfernenPreisschema(Fnr)
 pre: let fnr = Flugnummer.any->(fg1 |
           fnr1.FluggesellschaftID = Fnr.Fluggesellschaft
       and fnr1.Flugnummer         = Fnr.Nummer ) in
                 fnr.Preisschema->size = 1
```

- Im Ergebnis des Dienstes wurde die Zuordnung des Preisschemas zur Flugnummer aufgehoben.

```
Flugticketverkauf::Flugnummer::EntfernenPreisschema(Fg)
  post: let fg = Flugnummer.any->(fg1 |
              fnr1.FluggesellschaftID = Fnr.Fluggesellschaft
        and fnr1.Flugnummer         = Fnr.Nummer ) in
                  fnr.Preisschema->size = 0
```

3.10 Parametrisierung von Flug

Dieser Abschnitt enthält alle Bedingungen, die für die Parametrisierung von *Flug* gelten.

3.10.1 Flug allgemein

In diesem Abschnitt werden einige Invarianten aufgeführt, die von *Flug* jederzeit erfüllt werden.

- Ein Flug wird eindeutig durch die Attribute *FluggesellschaftID*, *Nummer* und *Abflugdatum* identifiziert.

```
Flugticketverkauf
  inv: self.Flug->forAll(fl1, fl2 | fl1 <> fl2 implies
             fl1.FluggesellschaftID  <> fl2.FluggesellschaftID or
             fl1.Nummer              <> fl2.Nummer or
             fl1.Abflugdatum         <> fl2.Abflugdatum )
```

- Die Attribute *FluggesellschaftID* und *Nummer* können nicht gepflegt werden, sondern ergeben sich automatisch. Sie wurden zusätzlich aufgenommen, um die Navigation im Modell zu erleichtern.

```
Flugticketverkauf::Flug
  inv: self.FluggesellschaftID= self.Flugnummer.FluggesellschaftID
   and self.Nummer            = self.Flugnummer.Nummer
```

- Ist das Attribut *AnkunftTageSpäter* der entsprechenden Flugnummer = 0, dann ist das Ankunftsdatum des Fluges gleich dem Abflugdatum. Ist das Attribut *AnkunftTageSpäter* = n, dann liegt das Ankunftsdatum n Tage nach dem Abflugdatum.

```
Flugticketverkauf::Flug
  inv: self.Ankunftsdatum =
             self.Abflugdatum + self.Flugnummer.AnkunftTageSpäter
```

Bemerkung 1: Streng genommen ist die angegebene Addition nicht korrekt. Ist zum Beispiel das Abflugdatum der 31.12.2002 und die Ankuft 1 Tag später, dann ist das Ankunftsdatum natürlich der 01.01.2003 und nicht der 32.12.2002. Man müsste an dieser Stelle also die Datumsberechnung in OCL ausführlich herleiten. Darauf wird aber verzichtet, da eine solche Herleitung die Verständlichkeit eher verschlechtert als verbessert. Außerdem wird in der Implementierung der ABAP-Datentyp DATS (Datum) für Abflug- und Ankunftsdatum verwendet, der obige Addition korrekt ausführt.

Bemerkung 2: Es gibt Attribute, die über die Parametrisierung nicht pflegbar sind. Erleichtern die Attribute das Verständnis, können sie folgendermaßen berücksichtigt werden:

- *Das Attribut wird im UML-Diagramm der entsprechenden Klasse zugeordnet.*

- *Das Attribut wird in den Schnittstellen der Methoden nicht aufgenommen.*
- *Auf der Verhaltensebene wird erläutert, wie sich der Wert des Attributs ergibt.*

- Die Preise für einen Flug werden nicht direkt gepflegt, sondern berechnen sich aus dem Standardpreis und verschiedenen Faktoren für Auf- und Abschläge. Zu jedem Flug gibt es ein gültiges Preisschema. Bei der Ermittlung des gültigen Preisschemas wird in folgender Reihenfolge gesucht, ob ein Preisschema zugewiesen wurde: Flug, Flugnummer, Fluggesellschaft. Aus dem gültigen Preisschema ergeben sich die Faktoren für Business und First Class sowie für Kinder und Kleinkinder.

```
Flugticketverkauf::Flug
  inv: if self.Preisschema->notEmpty
            let prs = self.Preisschema in
       else if self.Flugnummer.Preisschema->notEmpty
              let prs = self.Flugnummer.Preisschema in
            else
             let prs = self.Flugnummer.Fluggesellschaft.Preisschema in
            endif
       endif

       let stpr = self.Standardpreis in

     self.Preis.EcoErw         = stpr                                  and
     self.Preis.EcoKind        = stpr * prs.FaktorKind                 and
     self.Preis.EcoKleinkind   = stpr * prs.FaktorKleinkind            and
     self.Preis.BusErw         = stpr * prs.FaktorBus                  and
     self.Preis.BusKind        = stpr * prs.FaktorBus * prs.FaktorKind and
     self.Preis.BusKleinkind   =
                         stpr * prs.FaktorBus * prs.FaktorKleinkind    and
     self.Preis.FirstErw       = stpr * prs.FaktorFirst                and
     self.Preis.FirstKind      = stpr * prs.FaktorFirst * prs.FaktorKindand
     self.Preis.FirstKleinkind =
                         stpr * prs.FaktorFirst * prs.FaktorKleinkind
```

- Die Währung, in welcher die Preise zum Flug angegeben sind, ist immer die Währung der Fluggesellschaft. Das Attribut Währung ist im Rahmen der Parametrisierung nicht direkt pflegbar, sondern ergibt sich aus dieser Regel.

```
Flugticketverkauf::Flug
  inv: self.Währung = self.Flugnummer.Fluggesellschaft.Währung
```

3.10.2 Flug::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flug::Anlegen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flug { ...

       void Anlegen(
             in  FlugKeyTyp          FlugKey,
             in  FlugDatenTyp        FlugDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flug::Anlegen(Key: FlugKeyTyp, Daten: FlugDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flug::Anlegen(Key, Daten)
 pre: Key.Fluggesellschaft <> ´´   and Key.Flugnummer <> ´´
  and  Key.Abflugdatum <> ´´       and Daten.Standardpreis <> ´´
  and  Daten.Steuer <> ´´
```

- Der anzulegende Flug darf noch nicht existieren, d.h. es gibt noch keinen Flug mit den gleichen Keyattributen.

```
Flugticketverkauf::Flug::Anlegen(Key, Daten)
 pre: not Flug.exists->(fl |
                fl.FluggesellschaftID   = Key.Fluggesellschaft
            and fl.Nummer               = Key.Flugnummer
            and fl.Abflugdatum          = Key.Abflugdatum )
```

- Die Flugnummer, auf den sich der Flug beziehen soll, existiert.

```
Flugticketverkauf::Flug::Anlegen(Key, Daten)
 pre: Flugnummer.exists->(fnr |
                fnr.FluggesellschaftID  = Key.Fluggesellschaft
             and fnr.Nummer             = Key.Flugnummer )
```

- Das Ergebnis des Dienstes ist, dass ein neuer Flug definiert wurde. Die Attribute des Fluges stimmen mit den entsprechenden Feldern der Input-Parameter überein.

```
Flugticketverkauf::Flug::Anlegen(Key, Daten)
 post: Flug.any->(fl |
                fl.FluggesellschaftID   = Key.Fluggesellschaft
             and fl.Nummer              = Key.Flugnummer
             and fl.Abflugdatum         = Key.Abflugdatum      ).oclIsNew

 post: Flug.exists->(fl |
                fl.FluggesellschaftID   = Key.Fluggesellschaft
            and fl.Nummer               = Key.Flugnummer
            and fl.Abflugdatum          = Key.Abflugdatum
            and fl.Standardpreis        = Daten.Standardpreis
            and fl.Steuer               = Daten.Steuer )
```

Bemerkung: Für den Flug gelten damit alle Invarianten aus Abschnitt 3.10.1.

3.10.3 Flug::Ändern

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flug::Ändern*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flug { ...

      void Ändern(
            in  FlugKeyTyp          FlugKey,
            in  FlugDatenTyp        FlugDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flug::Ändern(Key: FlugKeyTyp, Daten: FlugDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flug::Ändern(Key, Daten)
 pre: Key.Fluggesellschaft <> ´´   and Key.Flugnummer <> ´´
  and  Key.Abflugdatum <> ´´       and Daten.Standardpreis <> ´´
  and  Daten.Steuer <> ´´
```

- Der zu ändernde Flug muss schon existieren.

```
Flugticketverkauf::Flug::Ändern(Key, Daten)
 pre: Flug.exists->(fl |
           fl.FluggesellschaftID = Key.Fluggesellschaft
       and fl.Nummer             = Key.Flugnummer
       and fl.Abflugdatum        = Key.Abflugdatum )
```

- Das Ergebnis des Dienstes ist, dass die Attribute des Fluges die entsprechenden Werte der Input-Parameter angenommen haben.

```
Flugticketverkauf::Flug::Ändern(Key, Daten)
 post: Flug.exists->(fnr |
           fl.FluggesellschaftID  = Key.Fluggesellschaft
      and  fl.Nummer              = Key.Flugnummer
      and  fl.Abflugdatum         = Key.Abflugdatum
      and  fl.Standardpreis       = Daten.Standardpreis
      and  fl.Steuer              = Daten.Steuer )
```

3.10.4 Flug::Löschen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flug::Löschen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flug { ...

      void Löschen(
           in  FlugKeyTyp          FlugKey); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flug::Löschen(Key: FlugKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flug::Löschen(Key)
 pre: Key.Fluggesellschaft <> ´´   and Key.Flugnummer <> ´´
  and  Key.Abflugdatum <> ´´
```

- Der zu löschende Flug muss existieren.

```
Flugticketverkauf::Flug::Löschen(Key)
 pre: Flug.exists->(fl |
           fl.FluggesellschaftID  = Key.Fluggesellschaft
      and  fl.Nummer              = Key.Flugnummer
```

```
      and  fl.Abflugdatum          = Key.Abflugdatum )
```

- Der Flug kann nur gelöscht werden, wenn er nicht Teil einer (nicht stornierten) Flugreise ist.

```
Flugticketverkauf::Flug::Löschen(Key)
  pre: Flugreise.select->(flr |
       flr.Buchungsstatus                          <> ´C´
   and flr.Hinflug.Teilstrecke.FluggesellschaftID = Key.Fluggesellschaft
   and flr.Hinflug.Teilstrecke.Nummer             = Key.Flugnummer)
                                                          ->isEmpty()
```

- Im Ergebnis des Dienstes wurde der Flug gelöscht.

```
Flugticketverkauf::Flug::Löschen(Key)
 post: not Flug.exists->(fl |
           fl.FluggesellschaftID   = Key.Fluggesellschaft
      and  fl.Nummer               = Key.Flugnummer
      and  fl.Abflugdatum          = Key.Abflugdatum )
```

3.10.5 Flug::ZuordnenPreisschema

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flug::ZuordnenPreisschema*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flug { ...

      void ZuordnenPreisschema(
           in  FlugKeyTyp               FlugKey,
           in  PreisschemaKeyTyp        PreisschemaKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flug::ZuordnenPreisschema(
                        Fl:         FlugKeyTyp,
                        Prs:        PreisschemaKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flug::ZuordnenPreisschema(Fl, Prs)
 pre: Fl.Fluggesellschaft <> ´´    and Fl.Flugnummer <> ´´ and
      Fl.Abflugdatum <> ´´         and Prs.Schemanummer <> ´´
```

- Der angegebene Flug und das angegebene Preisschema müssen existieren.

```
Flugticketverkauf::Flug::ZuordnenPreisschema(Fl, Prs)
 pre: Flug.exists->(fl |
                fl.FluggesellschaftID   = Fl.Fluggesellschaft
           and  fl.Nummer               = Fl.Flugnummer
           and  fl.Abflugdatum          = Fl.Abflugdatum)
 pre: Preisschema.exists->(prs | prs.Nummer = Prs.Schemanummer)
```

- Dem Flug darf noch kein Preisschema zugeordnet sein.

```
Flugticketverkauf::Flug::ZuordnenPreisschema(Fl, Prs)
  pre: let fl = Flug.any->(fl1 |
                fl1.FluggesellschaftID  = Fl.Fluggesellschaft
```

```
              and fl1.Nummer              = Fl.Flugnummer
              and fl1.Abflugdatum         = Fl.Abflugdatum) in

            fl.Preisschema->size = 0
```

Bemerkung: Mit `fg` wird zunächst die betrachtete Flug bezeichnet, damit die Bedingung übersichtlicher wird.

- Das Ergebnis des Dienstes ist, dass dem Flug das angegebene Preisschema zugeordnet wurde.

```
Flugticketverkauf::Flug::ZuordnenPreisschema(Fl, Prs)
 post: let fl = Flug.any->(fl1 |
                 fl1.FluggesellschaftID  = Fl.Fluggesellschaft
               and fl1.Nummer              = Fl.Flugnummer
               and fl1.Abflugdatum         = Fl.Abflugdatum) in

             fl.Preisschema->size = 1
               and fl.Preisschema.Nummer = Prs.Schemanummer
```

3.10.6 Flug::EntfernenPreisschema

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flug::EntfernenPreisschema*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flug { ...

void EntfernenPreisschema(
in  FlugKeyTyp          FlugKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flug::EntfernenPreisschema(Fl: FlugKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flug::EntfernenPreisschema(Fl)
 pre: Fl.Fluggesellschaft <> ´´ and Fl.Flugnummer <> ´´ and
      Fl.Abflugdatum <> ´´
```

- Der angegebene Flug muss existieren.

```
Flugticketverkauf::Flug::EntfernenPreisschema(Fg)
 pre: Flug.exists->(fl | fl.FluggesellschaftID = Fl.Fluggesellschaft
                    and  fl.Nummer             = Fl.Flugnummer
                    and  fl.Abflugdatum        = Fl.Abflugdatum)
```

- Dem Flug muss schon ein Preisschema zugeordnet sein.

```
Flugticketverkauf::Flug::EntfernenPreisschema(Fl)
 pre: let fl = Flug.any->(fl1 |
                 fl1.FluggesellschaftID  = Fl.Fluggesellschaft
               and fl1.Nummer              = Fl.Flugnummer
               and fl1.Abflugdatum         = Fl.Abflugdatum) in

             fl.Preisschema->size = 1
```

- Im Ergebnis des Dienstes wurde die Zuordnung des Preisschemas zur Flug aufgehoben.

```
Flugticketverkauf::Flug::EntfernenPreisschema(Fl)
 post: let fl = Flug.any->(fl1 |
                 fl1.FluggesellschaftID  = Fl.Fluggesellschaft
             and fl1.Nummer              = Fl.Flugnummer
             and fl1.Abflugdatum         = Fl.Abflugdatum) in

           fl.Preisschema->size = 0
```

3.11 Parametrisierung von Flugverbindungsnummer

Dieser Abschnitt enthält alle Bedingungen, die für die Parametrisierung von *Flugverbindungsnummer* gelten.

3.11.1 Flugverbindungsnummer allgemein

In diesem Abschnitt werden einige Invarianten aufgeführt, die von Flugverbindungsnummern jederzeit erfüllt werden.

- Eine Flugverbindungsnummer wird eindeutig durch die Attribute *Reisebüronummer* und *Verbindungsnummer* identifiziert.

```
Flugticketverkauf
 inv: self.Flugverbindungsnummer->forAll(fvbnr1, fvbnr2 |
      fvbnr1 <> fvbnr2 implies
       fvbnr1.Reisebüronummer      <> fvbnr2.Reisebüronummer or
       fvbnr1.Verbindungsnummer    <> fvbnr2.Verbindungsnummer )
```

- Das die Flugverbindung anbietende Reisebüro existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung.

```
Flugticketverkauf::Flugverbindungsnummer
 inv: Extern::Reisebüro::PrüfeExistenz(self.Reisebüronummer) = ´X´
```

- Bei den Teilstrecken zu einer Flugverbindungsnummer muss es sich um verschiedene Flugnummern handeln, d.h. eine Flugnummer kann nicht in mehreren Teilstrecken derselben Flugverbindung verwendet werden.

```
Flugticketverkauf::Flugverbindungsnummer
 inv: self.Teilstreckennummer->size = self.Flugnummer->size
```

Bemerkung: Es ist nicht auf den ersten Blick ersichtlich, dass die OCL-Bedingung den angegebenen Sachverhalt beschreibt: Angenommen, eine Flugverbindungsnummer hat n Teilstrecken. Dann ist die linke Seite der Gleichung = n, da jede Teilstreckennummer nur einmal vorkommen kann. Die rechte Seite kann aber nur = n sein, wenn alle Flugnummern verschieden sind. (`self.Flugnummer` ist in OCL eine Menge (Set), in welcher jedes Element nur einmal vorkommt.)

- Zu einer Flugverbindungsnummer muss es mindestens eine Teilstrecke geben.

```
Flugticketverkauf::Flugverbindungsnummer
 inv: self.Flugnummer->size > 0
```

- Hat eine Flugverbindungsnummer n Teilstrecken, dann tragen die Teilstrecken die Nummern 1 bis n.

```
Flugticketverkauf::Flugverbindungsnummer
  inv: let n = self.Flugnummer->size in
            integer->forAll(m | 1 <= m <= n implies
                              self.Flugnummer->size = 1)
```

- Ein Teilstreckenflug kann nur starten, wenn der vorhergehende Teilstreckenflug beendet wurde. Daraus ergibt sich eine entsprechende Bedingung an das Attribut *AbflugTageSpäter* der Teilstreckennummer sowie an die Attribute *Ankunftszeit* und *Abflugzeit*.

```
Flugticketverkauf::Flugverbindungsnummer
  inv: let n = self.Flugnummer->size in
       integer->forAll(m | 1 < m <= n implies
               (   self.Teilstreckennummer[m].AbflugTageSpäter
                 > self.Teilstreckennummer[m-1].AbflugTageSpäter +
                   self.Flugnummer[m-1].AnkunftTageSpäter )
             or (
               (   self.Teilstreckennummer[m].AbflugTageSpäter
                 = self.Teilstreckennummer[m-1].AbflugTageSpäter +
                   self.Flugnummer[m-1].AnkunftTageSpäter ) and
               (   self.Flugnummer[m].Abflugzeit
                 > self.Flugnummer[m-1].Ankunftszeit ) ) )
```

- Der Abflugort einer Teilstrecke muss gleich dem Ankunftsort der vorhergehenden Teilstrecke sein.

```
Flugticketverkauf::Flugverbindungsnummer
  inv: let n = self.Flugnummer->size in
       integer->forAll(m | 1 < m <= n implies
             self.Flugnummer[m].Abflugort.Kürzel
                 = self.Flugnummer[m-1].Ankunftsort.Kürzel )
```

Bemerkung: Bei der Assoziation von Flugverbindungsnummer zu Flugnummer handelt es sich um eine qualifizierte Assoziation. In den OCL-Ausdrücken steht `self.Flugnummer[m]` dabei für die Flugnummer, die mit der Teilstrecke mit HopNr = m verknüpft ist.

Bemerkung: Die Flugverbindungsnummer selbst besteht nur aus den Schlüsselattributen, die nicht geändert werden können. Daher wird hier die Methode Ändern nicht benötigt.

3.11.2 Flugverbindungsnummer::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugverbindungsnummer::Anlegen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugverbindungsnummer { ...

      void Anlegen(
            in  FlugverbindungsnummerKeyTyp     FlugverbindungsnummerKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugverbindungsnummer::Anlegen(
                        Key:     FlugverbindungsnummerKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Flugverbindungsnummer::Anlegen(Key)
  pre: Key.Reisebüronummer <> ´´ and Key.Verbindungsnummer <> ´´
```

- Das angegebene *Reisebüro* existiert in einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung.

```
Flugticketverkauf::Flugverbindungsnummer::Anlegen(Key)
  pre: Extern::Reisebüro::PrüfeExistenz(Key.Reisebüronummer) = ´X´
```

- Die anzulegende Flugverbindungsnummer darf noch nicht existieren, d.h. es gibt noch keine Flugverbindungsnummer mit den gleichen Keyattributen:

```
Flugticketverkauf::Flugverbindungsnummer::Anlegen(Key)
  pre: not Flugverbindungsnummer.exists->(fvbnr |
                  fvbnr.Reisebüronummer   = Key.Reisebüronummer
              and fvbnr.Verbindungsnummer = Key.Verbindungsnummer)
```

- Das Ergebnis des Dienstes ist, dass eine neue Flugverbindungsnummer definiert wurde:

```
Flugticketverkauf::Flugverbindungsnummer::Anlegen(Key)
  post: Flugverbindungsnummer.any->(fvbnr |
                  fvbnr.Reisebüronummer   = Key.Reisebüronummer
              and fvbnr.Verbindungsnummer = Key.Verbindungsnummer ).oclIsNew
```

3.11.3 Flugverbindungsnummer::Löschen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugverbindungsnummer::Löschen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugverbindungsnummer { ...

      void Löschen(
            in  FlugverbindungsnummerKeyTyp     FlugverbindungsnummerKey);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugverbindungsnummer::Löschen(
                          Key:  FlugverbindungsnummerKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Flugverbindungsnummer::Löschen(Key)
  pre: Key.Reisebüronummer <> ´´ and Key.Verbindungsnummer <> ´´
```

- Die zu löschende Flugverbindungsnummer muss existieren.

```
Flugticketverkauf::Flugverbindungsnummer::Löschen(Key)
  pre: Flugverbindungsnummer.exists->(fvbnr |
                  fvbnr.Reisebüronummer   = Key.Reisebüronummer
```

```
            and  fvbnr.Verbindungsnummer = Key.Verbindungsnummer)
```

- Die Flugverbindungsnummer kann nur gelöscht werden, wenn sie in keiner Flugverbindung mehr referenziert wird.

```
Flugticketverkauf::Flugverbindungsnummer::Löschen(Key)
 pre: Flugverbindung.select->(flvb |
           flvb.Reisebüronummer    = Key.Reisebüronummer
      and  flvb.Verbindungsnummer  = Key.Verbindungsnummer)->isEmpty()
```

- Im Ergebnis des Dienstes wurde die Flugverbindungsnummer gelöscht.

```
Flugticketverkauf::Flugverbindungsnummer::Löschen(Key)
 post: not Flugverbindungsnummer.exists->(fvbnr |
           fvbnr.Reisebüronummer   = Key.Reisebüronummer
       and fvbnr.Verbindungsnummer = Key.Verbindungsnummer)
```

3.11.4 Flugverbindungsnummer::AnlegenTeilstrecke

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugverbindungsnummer::AnlegenTeilstrecke*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugverbindungsnummer { ...

      void AnlegenTeilstrecke(
            in  FlugverbindungsnummerKeyTyp     FlugverbindungsnummerKey,
            in  TeilstreckennummerTyp           HopNr,
            in  TeilstreckennummerDatenTyp      TeilstreckennummerDaten);
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugverbindungsnummer::AnlegenTeilstrecke(
                        Key:        FlugverbindungsnummerKeyTyp,
                        HopNr:      TeilstreckennummerTyp,
                        Daten:      TeilstreckennummerDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Flugverbindungsnummer::AnlegenTeilstrecke(Key, HopNr,
Daten)
 pre: Key.Reisebüronummer <> ´´    and Key.Verbindungsnummer <> ´´
   and Daten.Fluggesellschaft <> ´´      and Key.Flugnummer <> ´´
   and HopNr <> ´´                   and Daten.AbflugTageSpäter <> ´´
```

- Die angegebene Flugverbindungsnummer muss existieren:

```
Flugticketverkauf::Flugverbindungsnummer::AnlegenTeilstrecke(Key, HopNr,
Daten)
 pre: Flugverbindungsnummer.exists->(fvbnr |
                fvbnr.Reisebüronummer   = Key.Reisebüronummer
            and fvbnr.Verbindungsnummer = Key.Verbindungsnummer)
```

- Die anzulegende Teilstrecke darf noch nicht existieren, d.h. zur Flugverbindungsnummer gibt es noch keine Teilstrecke mit dieser Nummer.

```
Flugticketverkauf::Flugverbindungsnummer::AnlegenTeilstrecke(Key, HopNr,
Daten)
 pre: let fvbnr = Flugverbindungsnummer.any->(vbnr |
                  vbnr.Reisebüronummer    = Key.Reisebüronummer
            and   vbnr.Verbindungsnummer  = Key.Verbindungsnummer) in

                  fvbnr.Flugnummer[HopNr]->size = 0
```

Bemerkung: Mit `fvbnr` wird zunächst die betrachtete Flugverbindungsnummer bezeichnet, damit die dann folgende Bedingung übersichtlicher wird.

- Die angegebene Flugnummer existiert.

```
Flugticketverkauf::Flugverbindungsnummer::AnlegenTeilstrecke(Key, HopNr,
Daten)
 pre: Flugnummer.exists->(fnr |
           fnr.FluggesellschaftID  = Daten.Fluggesellschaft
      and  fnr.Nummer              = Daten.Flugnummer )
```

- Das Ergebnis des Dienstes ist, dass eine Assoziation zwischen der Flugverbindungsnummer und der Flugnummer für die Teilstrecke hergestellt wurde.

```
Flugticketverkauf::Flugverbindungsnummer::AnlegenTeilstrecke(Key, HopNr,
Daten)
 post: let fvbnr = Flugverbindungsnummer.any->(vbnr |
           vbnr.Reisebüronummer    = Key.Reisebüronummer
      and  vbnr.Verbindungsnummer  = Key.Verbindungsnummer) in

      fvbnr.Flugnummer[HopNr]->size = 1
   and fvbnr.Flugnummer[HopNr].FluggesellschaftID = Daten.Fluggesellschaft
   and fvbnr.Flugnummer[HopNr].Nummer             = Daten.Flugnummer
   and fvbnr.Teilstreckennummer[HopNr].AbflugTageSpäter
                                                  = Daten.AbflugTageSpäter
```

3.11.5 Flugverbindungsnummer::LöschenTeilstrecke

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugverbindungsnummer::LöschenTeilstrecke*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Flugverbindungsnummer { ...

      void LöschenTeilstrecke(
           in  FlugverbindungsnummerKeyTyp    FlugverbindungsnummerKey,
           in  TeilstreckennummerTyp          HopNr); };
... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Flugverbindungsnummer::LöschenTeilstrecke(
                        Key:        FlugverbindungsnummerKeyTyp,
                        HopNr:      TeilstreckennummerTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Flugverbindungsnummer::LöschenTeilstrecke(Key, HopNr)
```

```
 pre: Key.Reisebüronummer <> ´´ and Key.Verbindungsnummer <> ´´
           and HopNr <> ´´
```

- Die zu löschende Teilstrecke muss existieren:

```
Flugticketverkauf::Flugverbindungsnummer::LöschenTeilstrecke(Key, HopNr)
 pre: let fvbnr = Flugverbindungsnummer.any->(vbnr |
                  vbnr.Reisebüronummer    = Key.Reisebüronummer
            and   vbnr.Verbindungsnummer  = Key.Verbindungsnummer) in

                  fvbnr.Flugnummer[HopNr]->size = 1
```

- Im Ergebnis des Dienstes wurde die Teilstreckennummer gelöscht:

```
Flugticketverkauf::Flugverbindungsnummer::LöschenTeilstrecke(Key, HopNr)
 post: let fvbnr = Flugverbindungsnummer.any->(vbnr |
                  vbnr.Reisebüronummer    = Key.Reisebüronummer
            and   vbnr.Verbindungsnummer  = Key.Verbindungsnummer) in

                  fvbnr.Flugnummer[HopNr]->size = 0
```

3.12 Parametrisierung von Preisschema

Dieser Abschnitt enthält alle Bedingungen, die für die Parametrisierung von *Preisschema* gelten.

3.12.1 Preissschema allgemein

In diesem Abschnitt werden einige Invarianten aufgeführt, die von den Preisschemata jederzeit erfüllt werden.

- Ein Preisschema wird eindeutig durch das Attribut *Nummer* identifiziert.

```
Flugticketverkauf
 inv: self.Preisschema->forAll(prs1, prs2 | prs1 <> prs2 implies
                  prs1.Nummer <> prs2.Nummer )
```

- Im Preisschema selbst wird festgelegt, mit welchen Faktoren der Standardpreis eines Fluges multipliziert werden muss, um die Preise für Business Class und First Class sowie die für Kinder und Kleinkinder zu erhalten. Dabei müssen die Faktoren für Business und First Class größer oder gleich eins sein, und die Faktoren für Kinder und Kleinkinder kleiner oder gleich eins.

```
Flugticketverkauf::Preisschema
 inv: self.FaktorBus >= 1 and self.FaktorFirst >= 1 and
      0 < self.FaktorKleinkind <= 1 and 0 < self.FaktorKind <= 1
```

3.12.2 Preisschema::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Preisschema::Anlegen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Preisschema { ...

      void Anlegen(
            in  PreisschemaKeyTyp         PreisschemaKey); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Preisschema::Anlegen(
                        Key:     PreisschemaKeyTyp,
                        Daten:   PreisschemaDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein:

```
Flugticketverkauf::Preisschema::Anlegen(Key, Daten)
 pre: Key.Schemanummer <> ´´ and Daten.FaktorBus <> ´´  and
      Daten.FaktorFirst <> ´´ and Daten.FaktorKind <> ´´  and
      Daten.FaktorKleinkind <> ´´
```

- Das anzulegende Preisschema darf noch nicht existieren, d.h. es gibt noch kein Preisschema mit der gleichen Nummer.

```
Flugticketverkauf::Preisschema::Anlegen(Key, Daten)
 pre: not Preisschema.exists->(prs | prs.Nummer      = Key.Schemanummer)
```

- Das Ergebnis des Dienstes ist, dass ein neues Preisschema definiert wurde. Die Attribute des Preisschemas stimmen mit den entsprechenden Feldern der Input-Parameter überein.

```
Flugticketverkauf::Preisschema::Anlegen(Key, Daten)
 post: Preisschema.any->(prs | prs.Nummer      = Key.Schemanummer).oclIsNew

 post: Preisschema.exists->(prs |
                 prs.Nummer             = Key.Schemanummer
       and  if Daten.FaktorBus <> ´´
            then prs.FaktorBus          = Daten.FaktorBus
            else prs.FaktorBus          = ´1´ endif
       and  if Daten.FaktorFirst <> ´´
            then prs.FaktorFirst        = Daten.FaktorFirst
            else prs.FaktorFirst        = ´1´ endif
       and  if Daten.FaktorKind <> ´´
            then prs.FaktorKind         = Daten.FaktorKind
            else prs.FaktorKind         = ´1´ endif
       and  if Daten.FaktorKleinkind <> ´´
            then prs.FaktorKleinkind    = Daten.FaktorKleinkind
            else prs.FaktorKleinkind    = ´1´ endif )
```

3.12.3 Preisschema::Ändern

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Preisschema::Ändern*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Preisschema { ...

      void Ändern(
            in  PreisschemaKeyTyp         PreisschemaKey,
            in  PreisschemaDatenTyp       PreisschemaDaten); ...
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Preisschema::Ändern(
                        Key:   PreisschemaKeyTyp,
                        Daten: PreisschemaDatenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Preisschema::Ändern(Key, Daten)
 pre: Key.Schemanummer <> ´´ and Daten.FaktorBus <> ´´  and
      Daten.FaktorFirst <> ´´ and Daten.FaktorKind <> ´´  and
      Daten.FaktorKleinkind <> ´´
```

- Das zu ändernde Preisschema muss schon existieren:

```
Flugticketverkauf::Preisschema::Ändern(Key, Daten)
 pre: Preisschema.exists->(prs | prs.Nummer = key.Schemanummer)
```

- Das Ergebnis des Dienstes ist, dass die Attribute des Preisschemas die entsprechenden Werte der Input-Parameter angenommen haben:

```
Flugticketverkauf::Preisschema::Ändern(Key, Daten)
 post: Preisschema.exists->(prs |
                  prs.Nummer             = Key.Schemanummer
        and  if Daten.FaktorBus <> ´´
             then prs.FaktorBus          = Daten.FaktorBus
             else prs.FaktorBus          = ´1´ endif
        and  if Daten.FaktorFirst <> ´´
             then prs.FaktorFirst        = Daten.FaktorFirst
             else prs.FaktorFirst        = ´1´ endif
        and  if Daten.FaktorKind <> ´´
             then prs.FaktorKind         = Daten.FaktorKind
             else prs.FaktorKind         = ´1´ endif
        and  if Daten.FaktorKleinkind <> ´´
             then prs.FaktorKleinkind    = Daten.FaktorKleinkind
             else prs.FaktorKleinkind    = ´1´ endif )
```

3.12.4 Preisschema::Löschen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Preisschema::Löschen*. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Flugticketverkauf { ...
  interface Preisschema { ...

      void Löschen(
            in  PreisschemaKeyTyp         PreisschemaKey); ... };
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Flugticketverkauf::Preisschema::Löschen(
                        Key:   PreisschemaKeyTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer ohne Datentypen angegeben.

- Alle Felder der Input-Parameter sind obligatorisch und müssen gefüllt sein.

```
Flugticketverkauf::Preisschema::Löschen(Key)
```

```
pre: Key.Schemanummer <> ´´
```

- Das zu löschende Preisschema muss existieren.

```
Flugticketverkauf::Preisschema::Löschen(Key)
pre: Preisschema.exists->(prs | prs.Nummer = Key.Schemanummer)
```

- Die Preisschema kann nur gelöscht werden, wenn es in keiner Fluggesellschaft, in keiner Flugnummer und in keinem Flug mehr referenziert wird.

```
Flugticketverkauf::Preisschema::Löschen(Key)
pre: Fluggesellschaft.select->(fg |
      fg.Preisschema.Nummer = Key.Schemanummer) ->isEmpty()

 pre: Flugnummer.select->(fnr |
      fnr.Preisschema.Nummer= Key.Schemanummer) ->isEmpty()

pre: Flug.select->(fl |
      fl.Preisschema.Nummer = Key.Schemanummer) ->isEmpty()
```

- Im Ergebnis des Dienstes wurde das Preisschema gelöscht.

```
Flugticketverkauf::Preisschema::Löschen(Key)
post: not Preisschema.exists->(prs | prs.Nummer = Key.Schemanummer)
```

3.13 Extern::Reisebüro::PrüfeExistenz

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Reisebüro::PrüfeExistenz*. Dieser Dienst wird von der Komponente nicht implementiert, sondern von ihr erwartet. Dieser Dienst muss von einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung implementiert werden. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Extern { ...
  interface Reisebüro {

     XFlagTyp    PrüfeExistenz(
           in  ReisebüronummerTyp  Reisebüronummer); ... };
};
```

3.13.1 Ergebnis der Überprüfung

Existiert das angegebene *Reisebüro* in einer außerhalb der Komponente liegenden Geschäftspartnerverwaltung, dann ist der Ergebnisparameter = ´X´, ansonsten = ´´.

```
Extern::Reisebüro::PrüfeExistenz(Rbnr:ReisebüronummerTyp):XFlagTyp
post: if Reisebüro->exists(rb | rb.Nummer = Rbnr)
       then result = ´X´
       else result = ´´
      endif
```

3.14 Extern::Reisebüro::LiefereWährung

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Reisebüro::LiefereWährung*. Dieser Dienst wird von der Komponente nicht implementiert, sondern von ihr erwartet. Dieser Dienst muss von einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung implementiert werden. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Extern { ...
  interface Reisebüro { ...
      WährungsTyp LiefereWährung(
            in  ReisebüronummerTyp  Reisebüronummer); ... };
};
```

3.14.1 Reisebüro existiert

Das angegebene *Reisebüro* existiert.

```
Extern::Reisebüro::LiefereWährung(Rbnr:ReisebüronummerTyp):WährungsTyp
  pre: Reisebüro->exists(rb | rb.Nummer = Rbnr)
```

3.14.2 Rückgabe der Hauswährung

Der Ergebnisparameter *Hauswährung* enthält die Hauswährung des angegebenen *Reisebüros*.

```
Extern::Reisebüro::LiefereWährung(Rbnr:ReisebüronummerTyp):WährungsTyp
  post: Reisebüro->exists(rb | rb.Nummer      = Rbnr and
                               rb.Hauswährung = result)
```

3.15 Extern::Flugkunde::PrüfeExistenz

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugkunde::PrüfeExistenz*. Dieser Dienst wird von der Komponente nicht implementiert, sondern von ihr erwartet. Dieser Dienst muss von einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung implementiert werden. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Extern { ...
  interface Flugkunde { ...
      XFlagTyp   PrüfeExistenz(
            in  FlugkundeTyp  Flugkundennummer); ... };
};
```

3.15.1 Ergebnis der Überprüfung

Existiert der angegebene *Flugkunde* in einer außerhalb der Komponente liegenden Geschäftspartnerverwaltung, dann ist der Ergebnisparameter = ´X´, ansonsten = ´´.

```
Extern::Flugkunde::PrüfeExistenz(Fknr:FlugkundeTyp):XFlagTyp
  post: if Flugkunde->exists(fk | fk.Nummer = Fknr)
        then result = ´X´
        else result = ´´
       endif
```

3.16 Extern::Flugkunde::LiefereRabatt

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugkunde::LiefereRabatt*. Dieser Dienst wird von der Komponente nicht implementiert, sondern von ihr erwartet. Dieser Dienst muss von einer (außerhalb der Komponente liegenden) Geschäftspartnerverwaltung implementiert werden. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Extern { ...
  interface Flugkunde { ...
      RabattTyp  LiefereRabatt(
            in  FlugkundeTyp  Flugkundennummer); };
```

```
};
```

3.16.1 Flugkunde existiert

Der angegebene *Flugkunde* existiert.

```
Extern::Flugkunde::LiefereRabatt(Fknr:FlugkundeTyp):RabattTyp
  pre: Flugkunde->exists(fk | fk.Nummer = Fknr)
```

3.16.2 Rückgabe der Kundenrabatts

Der Ergebnisparameter *Rabatt* enthält den Rabatt, der dem *Flugkunden* gewährt wird.

```
Extern::Flugkunde::LiefereRabatt(Fknr:FlugkundeTyp):RabattTyp
  post: Flugkunde->exists(fk | fk.Nummer = Fknr and
                               fk.Rabatt = result)
```

3.17 Extern::Flugverfügbarkeit::Check

Zum Dienst *Flugverfügbarkeit::Check* werden keine speziellen Vor- und Nachbedingungen spezifiziert.

Es wird natürlich erwartet, dass der Dienst die richtigen Informationen zur Verfügbarkeit zurückliefert. Dies kann jedoch nur in OCL ausgedrückt werden, wenn der externe Entitätstyp Flugverfügbarkeit ausmodelliert wird. Dies ist aber nicht sinnvoll.

3.18 Extern::Flugbuchung::Anlegen

Dieser Abschnitt enthält die Vor- und Nachbedingungen zum Dienst *Flugbuchung::Anlegen*. Dieser Dienst wird von der Komponente nicht implementiert, sondern von ihr erwartet. Dieser Dienst muss von einem (außerhalb der Komponente liegenden) Flugbuchungssystem implementiert werden. Zur Erinnerung noch einmal die Schnittstelle des Dienstes:

```
module Extern { ...
  interface Flugbuchung { ...
      void Anlegen(
            in  BuchungsdatenTyp          Buchungsdaten,
            out FluggesellschaftTyp       Fluggesellschaft,
            out BuchungsnummerTyp         Buchungsnummer,
            out FlugbuchungspreisTyp      Flugpreis,
            out StatuslistenTyp           Statusmeldungen); };
};
```

Der Kontext aller Bedingungen dieses Abschnitts ist der o.g. Dienst. In ausführlicher Form:

```
Extern::Flugbuchung::Anlegen(
                        Bd:     BuchungsdatenTyp,
                        Fg:     FluggesellschaftTyp,
                        Bnr:    BuchungsnummerTyp,
                        Fpr:    FlugbuchungspreisTyp,
                        Status: StatuslistenTyp)
```

Zur besseren Lesbarkeit wird dieser Kontext im weiteren Verlauf immer nur abgekürzt und ohne Datentypen angegeben.

3.18.1 Neu angelegte Flugbuchung

Wenn kein Verarbeitungsfehler aufgetreten ist, wurde eine neue *Flugbuchung* angelegt. Die identifizierenden Attribute *Fluggesellschaft* und *Buchungsnummer* werden in den gleichlautenden Parametern zurückgegeben.

```
Extern::Flugbuchung::Anlegen(Bd, Fg, Bnr, Status)
  post: Status->exists(st | st.Typ = ´S´ and st.Nummer = ´000´) implies
            Flugbuchung->forAll(flb | (flb.Fluggesellschaft  = Fg
                                       and flb.Buchungsnummer = Bnr)
                                implies flb.oclIsNew)
```

Bemerkung: Es wird natürlich erwartet, dass bei der neu angelegten Flugbuchung die im Parameter Buchungsdaten übergebenen Daten verwendet wurden. Dies kann jedoch nur in OCL ausgedrückt werden, wenn der externe Entitätstyp Flugbuchung ausmodelliert wird. Dies ist aber nicht sinnvoll.

4 Abstimmungsebene

Dieses Kapitel beschreibt die Abstimmungsebene. Es enthält alle Bedingungen, die sich für die Reihenfolge und Konsistenz verschiedener Dienste ergeben. Dazu zählt die Angabe, welche externen Dienste von den einzelnen Diensten der Komponente benötigt werden. Siehe dafür Abschnitte 4.1 bis 4.4. Im Abschnitt 4.5 findet sich eine Bedingung, unter welchen Umständen die Methode *Flugreise::Anlegen* gleichzeitig mehrfach ausgeführt werden kann. In den Abschnitten 4.6 und 4.7 werden zwei Reihenfolgebeziehungen zwischen parametrisierungsrekevanten Diensten ausgedrückt.

Die Syntax in diesem Abschnitt besteht aus OCL-Ausdrücken und speziellen temporalen Operatoren, um die die OCL erweitert werden kann. Diese wurden in [CoTu2000] vorgeschlagen.

In Ergänzung zu [CoTu2000] wird hier eine erweiterte Syntax verwendet, die sich z.B. an [Saak1993] anlehnt. Zur kurzen Erklärung:

- Der Ausdruck *after(Methode(par))* ist ein Ausdruck vom Typ Boolean. Er ist zu genau dem Zeitpunkt wahr, in welchem der Methodenaufruf (mit den spezifizierten Parametern *par*) erfolgreich beendet wurde. Ansonsten ist er falsch.
- Analog dazu ist *before(Methode(par))* zu genau dem Zeitpunkt wahr, in welchem der Methodenaufruf (mit den spezifizierten Parametern *par*) gestartet wird. Ansonsten ist er falsch.

Bemerkung zur Notation: Der Übersichtlichkeit halber werden auch hier die Kontexte der Bedingungen nur verkürzt angegeben. Die ausführliche Form wurde im Kapitel 3 eingeführt.

4.1 Von Flugverbindung::LiefereListe verwendete Dienste

Während der Abarbeitung des Dienstes *Flugverbindung::LiefereListe* ruft dieser Dienst den externen Dienst *Extern::Reisebüro::PrüfeExistenz*. Soll also der Dienst *Flugverbindung::LiefereListe* verwendet werden, muss der externe Dienst von einer anderen Komponente zur Verfügung gestellt werden.

```
Flugticketverkauf::Flugverbindung::LiefereListe(Rbnr, ...)
  post: before(Extern::Reisebüro::PrüfeExistenz(Rbnr)) sometime_since_last
            before(Flugverbindung::LiefereListe(Rbnr, ...))
    and after(Extern::Reisebüro::PrüfeExistenz(Rbnr)) sometime_since_last
            before(Extern::Reisebüro::PrüfeExistenz(Rbnr))
```

4.2 Von Flugverbindung::LiefereDetails verwendete Dienste

Während der Abarbeitung des Dienstes *Flugverbindung::LiefereDetails* ruft dieser Dienst den externen Dienst *Extern::Flugverfügbarkeit::Check*. Soll also der Dienst *Flugverbindung::LiefereDetails* verwendet werden, muss der externe Dienst von einer anderen Komponente zur Verfügung gestellt werden.

```
Flugticketverkauf::Flugverbindung::LiefereDetails(Rbnr, Vbnr, Fldat, ...)
  post: Tstrl->forAll(tstr | let (Fg1 = tstr.Fluggesellschaft and
       Fnr1 = tstr.Flugverbindungsnummer and Fda1 = tstr.Abflugdatum ) in

       before(Extern::Flugverfügbarkeit::Check(Fg1, Fnr1, Fda1, ...))
             sometime_since_last
```

```
      before(Flugverbindung::LiefereDetails(Rbnr, Vbnr, Fldat, ...))

  and after(Extern::Flugverfügbarkeit::Check(Fg1, Fnr1, Fda1, ...))
           sometime_since_last
      before(Extern::Flugverfügbarkeit::Check(Fg1, Fnr1, Fda1, ...)))
```

Bemerkung: Der externe Dienst wird für jeden Teilstreckenflug genau einmal aufgerufen, d.h. im allgemeinen erfolgen mehrere Aufrufe von *Extern::Flugverfügbarkeit::Check*.

4.3 Von Flugreise::LiefereListe verwendete Dienste

Während der Abarbeitung des Dienstes *Flugreise::LiefereListe* ruft dieser Dienst die externen Dienste *Extern::Reisebüro::PrüfeExistenz* und *Extern::Flugkunde::PrüfeExistenz*. Soll also der Dienst *Flugreise::LiefereListe* verwendet werden, müssen die externen Dienste von anderen Komponenten zur Verfügung gestellt werden.

Flugticketverkauf::Flugreise::LiefereListe(Rbnr, Fknr, ...)

```
 post: before(Extern::Reisebüro::PrüfeExistenz(Rbnr)) sometime_since_last
           before(Flugreise::LiefereListe(Rbnr, Fknr, ...))
   and after(Extern::Reisebüro::PrüfeExistenz(Rbnr)) sometime_since_last
           before(Extern::Reisebüro::PrüfeExistenz(Rbnr))

 post: before(Extern::Flugkunde::PrüfeExistenz(Fknr)) sometime_since_last
           before(Flugreise::LiefereListe(Rbnr, Fknr, ...))
   and after(Extern::Flugkunde::PrüfeExistenz(Fknr)) sometime_since_last
           before(Extern:: Flugkunde::PrüfeExistenz(Fknr))
```

4.4 Von Flugreise::Anlegen verwendete Dienste

Während der Abarbeitung des Dienstes *Flugreise::Anlegen* ruft dieser Dienst die externen Dienste *Extern::Reisebüro::PrüfeExistenz* und *Extern::Flugkunde::PrüfeExistenz*. Soll also der Dienst *Flugreise::Anlegen* verwendet werden, müssen die externen Dienste von anderen Komponenten zur Verfügung gestellt werden.

Flugticketverkauf::Flugreise::Anlegen(Rd, ...)

```
 post: let (rb1 = Rd.Reisebüronummer) in
         before(Extern::Reisebüro::PrüfeExistenz(rb1)) sometime_since_last
           before(Flugreise::Anlegen(Rd, ...))
    and after(Extern::Reisebüro::PrüfeExistenz(rb1)) sometime_since_last
           before(Extern::Reisebüro::PrüfeExistenz(rb1))

 post: let (fk1 = Rd.Flugkundennummer) in
         before(Extern::Flugkunde::PrüfeExistenz(fk1)) sometime_since_last
           before(Flugreise::Anlegen(Rd, ...))
    and after(Extern::Flugkunde::PrüfeExistenz(fk1)) sometime_since_last
           before(Extern:: Flugkunde::PrüfeExistenz(fk1))
```

Außerdem initiiert dieser Dienst für jeden Passagier und jede Teilstrecke eine Flugbuchung bei der entsprechenden Fluggesellschaft. Das geschieht dadurch, dass jeweils der externe Dienst *Extern::Flugbuchung::Anlegen* gerufen wird.

Flugticketverkauf::Flugreise::Anlegen(... ,Rbnr, Rnr, ...)

```
 post: Flugreise->exists (flr |
           flr.Reisebüronummer          =  Rbnr
     and   flr.Reisenummer              =  Rnr
     and   flr.Hinflug.Teilstrecke->forAll(tstr |
```

```
            flr.Passagier->forAll(pass |
            let Bda: BuchungsdatenTyp =
            (Bda.Fluggesellschaft  = tstr.Flug.Fluggesellschaft,
             Bda.Flugnummer        = tstr.Flug.Flugnummer,
             Bda.Flugdatum         = tstr.Flug.Abflugdatum,
             Bda.Flugkunde         = flr.Flugkunde,
             Bda.Flugklasse        = flr.Flugklasse,
             Bda.PassName          = pass.Name,
             Bda.PassAnrede        = pass.Anrede,
             Bda.PassGeburtsdatum  = pass.Geburtsdatum) in

     before(Extern::Flugbuchung::Anlegen(Bda, ...)) sometime_since_last
          before(Flugreise::Anlegen(Rd, Rbnr, Rnr, ...))
  and after(Extern::Flugbuchung::Anlegen(Bda, ...)) sometime_since_last
          before(Extern::Flugbuchung::Anlegen(Bda, ...)) ) )
```

Die analoge Bedingung gilt für den Rückflug.

Flugticketverkauf::Flugreise::Anlegen(... ,Rbnr, Rnr, ...)

```
 post: (Rd.Rückflugverbindung <> ´´) implies Flugreise->exists (flr |
          flr.Reisebüronummer           =  Rbnr
     and  flr.Reisenummer               =  Rnr
     and  flr.Rückflug.Teilstrecke->forAll(tstr |
           flr.Passagier->forAll(pass |
            let Bda: BuchungsdatenTyp =
           (Bda.Fluggesellschaft  = tstr.Flug.Fluggesellschaft,
            Bda.Flugnummer        = tstr.Flug.Flugnummer,
            Bda.Flugdatum         = tstr.Flug.Abflugdatum,
            Bda.Flugkunde         = flr.Flugkunde,
            Bda.Flugklasse        = flr.Flugklasse,
            Bda.PassName          = pass.Name,
            Bda.PassAnrede        = pass.Anrede,
            Bda.PassGeburtsdatum  = pass.Geburtsdatum) in

      before(Extern::Flugbuchung::Anlegen(Bda, ...)) sometime_since_last
          before(Flugreise::Anlegen(Rd, Rbnr, Rnr, ...))
  and after(Extern::Flugbuchung::Anlegen(Bda, ...)) sometime_since_last
          before(Extern::Flugbuchung::Anlegen(Bda, ...)) ) )
```

4.5 Bedingungen an die parallele Ausführung von Flugreise::Anlegen

Die Fachkomponente Flugticketverkauf ist so erstellt, dass Parallelverarbeitung unterstützt wird. Allerdings kann die Methode *Flugreise::Anlegen* nur unter bestimmten Bedingungen aufgerufen werden, solange noch ein anderer Aufruf derselben Methode abgearbeitet wird.

Die Bedingung ist, dass sich die jeweiligen Teilstreckenflüge voneinander unterscheiden. Ansonsten kann es bei der Buchung von Plätzen bei der Fluggesellschaft zu Sperrproblemen kommen.

Flugticketverkauf::Flugreise::Anlegen(Rd2, ...)

```
 pre: not after(Flugreise::Anlegen(Rd1, ...))
          sometime_since_last before(Flugreise::Anlegen(Rd1, ...))
     implies
     let (hinflug1: set(Flug) = Flugverbindung->select(fvb |
                fvb.Reisebüronummer     = Rd1.Reisebüronummer
          and   fvb.Verbindungsnummer   = Rd1.Hinflugverbindung
          and   fvb.Abflugdatum         = Rd1.Hinflugdatum).Flug) in

     let (rückflug1: set(Flug) = Flugverbindung->select(fvb |
                fvb.Reisebüronummer   = Rd1.Reisebüronummer
```

```
            and   fvb.Verbindungsnummer = Rd1.Rückflugverbindung
            and   fvb.Abflugdatum       = Rd1.Rückflugdatum).Flug) in

 let (hinflug2: set(Flug) = Flugverbindung->select(fvb |
                fvb.Reisebüronummer   = Rd2.Reisebüronummer
            and   fvb.Verbindungsnummer = Rd2.Hinflugverbindung
            and   fvb.Abflugdatum       = Rd2.Hinflugdatum).Flug) in

 let (rückflug2: set(Flug) = Flugverbindung->select(fvb |
                fvb.Reisebüronummer   = Rd2.Reisebüronummer
            and   fvb.Verbindungsnummer = Rd2.Rückflugverbindung
            and   fvb.Abflugdatum       = Rd2.Rückflugdatum).Flug) in

     let (flüge1: set(Flug) = hinflug1->union(rückflug1)) in

     let (flüge2: set(Flug) = hinflug2->union(rückflug2)) in

      fl1->forAll(fl1 | fl2->forAll(fl2 |
          fl1.Fluggesellschaft.Kürzel   <> fl2.Fluggesellschaft.Kürzel
        or fl1.Flugnummer               <> fl2.Flugnummer
        or fl1.Abflugdatum              <> fl2.Abflugdatum ) )
```

Bemerkung: Die Bedingung ist folgendermaßen zu lesen. Angenommen, der Dienst *Flugreise::Anlegen* wird mit dem Parameter *Rd2* gestartet, bevor der Aufruf mit dem Parameter *Rd1* beendet wurde. Dann gilt folgende Bedingung: *hinflug1* bezeichnet die Menge aller Flüge, die mit dem Hinflug aus *Rd1* verbunden sind. Analog bezeichnet *rückflug1* die Menge aller Flüge, die mit dem Rückflug verbunden sind (sofern dieser existiert) und *flüge1* ist die Vereinigung beider Mengen. Genauso wird *flüge2* aus *Rd2* ermittelt. Nun wird gefordert, dass alle Flüge aus *flüge1* verschieden von allen Flügen in *flüge2* sind.

4.6 Ausführung der Parametrisierungs-Methode Fluggesellschaft::ZuordnenPreisschema

Wurde eine Fluggesellschaft definiert, dann ist es obligatorisch, dieser ein Preisschema zuzuordnen.

```
Flugticketverkauf::Fluggesellschaft::Anlegen(Key, Daten)
 post: sometime after(Fluggesellschaft::ZuordnenPreisschema(Key, Prs))
```

Bemerkung: Aufgrund des gleichen Parameters `Key` handelt es sich um die selbe Fluggesellschaft.

4.7 Ausführung der Parametrisierungs-Methode Flugverbindungsnummer::AnlegenTeilstrecke

Wurde eine Flugverbindungsnummer definiert, dann ist es obligatorisch, für diese mindestens eine Teilstrecke zu definieren.

```
Flugticketverkauf::Flugverbindungsnummer::Anlegen(Key)
 post: sometime after(Flugverbindungsnummer::AnlegenTeilstrecke(Key, ...))
```

Bemerkung: Aufgrund des gleichen Parameters `Key` handelt es sich um die selbe Flugverbindungsnummer.

5 Qualitätsebene

Bemerkung: Im Memorandum [Turo2002] wurde bisher nicht konkret angegeben, welche Größen auf der Qualitätsebene zu spezifizieren sind. Daher haben wir uns an den Informationen orientiert, die in [FeLT2001] erfasst wurden.

Ausgangspunkt für die Messung der Qualitätseigenschaften ist folgende Systemumgebung:

- Prozessorarchitektur: 4x Intel Pentium III 550 MHz
- Hauptspeicher: 4 GB RAM (+ 8 GB virtuell)
- Windows NT 4.0 SP 5/6
- Microsoft SQL 8.00
- SAP Web Application Server 6.20

Unter dieser Systemumgebung wird exemplarisch der Dienst *Flugverbindung::LiefereDetails* folgendermaßen spezifiziert.

Qualitätseigenschaft	Spezifikation
Durchsatzzeit	Der Durchsatz beträgt 41s bei einer Arbeitslast von 1000 Dienstanforderungen.
Antwortzeit	Die Antwortzeit beträgt 44 ms.
Antwortzeitverhalten	Das Antwortzeitverhalten beträgt 3 ms.
Verfügbarkeit	keine Angabe
Wiederanlaufzeit	keine Angabe

Bemerkungen:

- Die Fachkomponente *Flugticketverkauf* läuft auf dem Komponentenframework des SAP Web Application Servers. Daher sind die Angaben zu Durchsatzzeit, Antwortzeit etc. immer abhängig von der Gesamtbelastung des Systems und kann für die Komponente nicht vollständig isoliert betrachtet werden. Zur Ermittlung der Qualitätseigenschaften wurde ein relativ wenig verwendetes Testsystem benutzt. Die Messungen wurden mehrfach ausgeführt und die Ergebnisse gemittelt.
- In der vorliegenden Testinstallation wurden die extern benötigten Dienste von SAP-Komponenten zu Flugbuchungen und Flugkundenverwaltung erbracht. Diese Komponenten wurden auf der selben Maschine installiert und ausgeführt. (Es ist möglich, die Flugbuchungen remote in einem anderen System auszuführen. In diesem Fall wären die Messwerte entsprechend höher.)
- Verfügbarkeit und Wiederanlaufzeit werden primär vom Komponenten-Framework und kaum von der Fachkomponente bestimmt.

Bemerkung 2: Das Beispiel zeigt, dass die Spezifikation in dieser Form wenig sinnvoll ist. Daraus ergibt sich die Notwendigkeit, die auf der Qualitätsebene zu erfassenden Messgrößen nochmals genauer zu untersuchen und Vorschriften anzugeben, wie diese operationalisiert werden können.

6 Vermarktungsebene

Name Flugticketverkauf
Version V 6.10
Lieferumfang Die Fachkomponente enthält alle Objekte des SAP-Pakets SAPBC_IBF2 (76 Objekte) und des SAP-Pakets SAPBC_DATAMODEL (291 Objekte). Es handelt sich dabei u.a. um Tabellen, Datentypen, Programme, Funktionsbausteine und Dokumentation. - Außerdem gehört eine Dokumentation zum Lieferumfang.
Komponententechnologie SAP Web Application Server mit Release 6.10 oder höher
Systemanforderungen Der SAP Web AS ist eine notwendige Voraussetzung für Installation und Betrieb der Fachkomponente. Der SAP Web AS ist in einer Vielzahl verschiedener Systemumgebungen lauffähig. Für die genauen Anforderungen an die Systemkonfiguration sei auf die entsprechende Dokumentation des SAP Web AS verwiesen. Die Fachkomponente selbst stellt keine weitergehenden Anforderungen an die Systemkonfiguration.
Hersteller SAP AG, Neurottstr. 16, D-69190 Walldorf / Baden
Ansprechpartner Jörg Ackermann, joerg.ackermann@sap.com
Vertragliche Konditionen Die Funktionalität zum „Flugticketverkauf“ wurde im Rahmen dieser Fallstudie als Fachkomponente spezifiziert. Es handelt es sich allerdings produktmäßig nicht um eine einzeln vermarktete Fachkomponente. Die hier beschriebene Funktionalität ist (ab SAP Web AS Release 6.10) Teil einer SAP-Schulungsanwendung, mit der SAP bestimmte Technologien anhand eines einfachen betriebswirtschaftlichen Beispiels demonstriert und vermittelt. Die Funktionalität gehört zur Standardauslieferung des SAP Web AS ab Release 6.10.

7 Terminologieebene

Auf der Terminologieebene werden die wichtigsten Begriffe der Fachkomponente *Flugticketverkauf* lexikonartig erklärt. Unterstrichene Wörter verweisen auf andere aufgeführte Begriffe.

Flug, planmäßiger Flug einer Fluggesellschaft zwischen einem Startort und einem Zielort an einem bestimmten Datum.

-nummer, vierstellige Zahl, die zusammen mit dem Fluggesellschaftskürzel und dem Datum einen Flug eindeutig identifiziert.

-preis, tarifabhängiger Geldbetrag, der für eine Beförderung an die Fluggesellschaft zu bezahlen ist.

Homonyme Verwendung: Umgangssprachlich wird oft von einem Flug gesprochen, obwohl es sich um eine Kombination von Einzelflügen handelt (Hin- und Rückflug, mehrere Teilstrecken). Letzteres wird hier präziser als Flugreise bezeichnet. Ein Flug ist hier immer ein einzelner Flug (siehe oben).

Flugbuchung, Vertrag zwischen einer Fluggesellschaft und einem Flugkunden. Die Fluggesellschaft befördert einen vom Flugkunden benannten Passagier auf einem bestimmten Flug. Der Flugkunde bezahlt dafür einen festgelegten Preis. Der Vertrag kann über ein Reisebüro abgeschlossen werden. (Bemerkung: Eine F. bezieht sich immer auf genau einen Flug und genau einen Passagier).

Fluggesellschaft, Unternehmen, welches Passagiere zwischen verschiedenen Orten auf dem Luftweg befördert.

-skürzel, dreistellige Zeichenkette, die eine Fluggesellschaft eindeutig identifiziert

-

Flughafen, Start- und Zielpunkt von Flügen

-kürzel, dreistellige Zeichenkette, die einen Flughafen eindeutig identifiziert.

Flugklasse, Einteilung der Sitzplatzbereiche eines Flugzeugs nach Qualitätsstufen (First Class, Business Class, Economy Class).

Flugkunde, 1. Geschäftspartner einer Fluggesellschaft. Gegenstand der Geschäftsbeziehung ist die Beförderung auf dem Luftweg. 2. Geschäftspartner eines Reisebüros. Gegenstand der Geschäftsbeziehung ist der Kauf von Flugreisen.

Flugnummer, siehe Flug.

Flugpreis, siehe Flug.

Flugreise: Die Buchung einer F. ist ein Vertrag zwischen einem Flugkunden und einem Reisebüro. Eine F. besteht aus einem Hinflug und (optional) aus einem Rückflug. Beide beziehen sich jeweils auf eine (vom Reisebüro angebotene) Flugverbindung. Eine F. kann mehrere Passagiere umfassen.

Durch das Buchen einer F. erwirbt der Flugkunde das Recht, dass alle benannten Passagiere auf dem Hinflug und (optional) dem Rückflug befördert werden. Der Flugkunde bezahlt dafür einen Preis.
Bei der Buchung einer F. werden vom Reisebüro alle notwendigen Flugbuchungen bei den Fluggesellschaften vorgenommen.

-preis, Geldbetrag, der vom Flugkunden für die Flugreise an das Reisebüro zu bezahlen ist.
Reisenummer, achtstellige Zahl, die zusammen mit der Reisebüronummer eine Flugreise eindeutig identifiziert.

Homonyme Verwendung von Flug: Umgangssprachlich wird die hier definierte Flugreise oft auch einfach als Flug bezeichnet. Zur besseren Abgrenzung wird hier immer von Flugreise gesprochen. Siehe auch Flug.

Flugverbindung, Strecke und Abflugzeit, für welche ein Reisebüro Beförderungsleistungen verkauft. Eine F. bezieht sich auf einen Startort und einen Zielort und auf eine bestimmte Abflugzeit (Datum, Uhrzeit). Eine F. besteht aus einer oder mehreren Teilstrecken. Jede Teilstrecke bezieht sich auf genau einen Flug. Dabei können in einer F. auch Flüge verschiedener Fluggesellschaften kombiniert werden.

-snummer, vierstellige Zahl, die zusammen mit der Reisebüronummer und dem Datum eine Flugverbindung eindeutig identifiziert.
-spreis, tarifabhängiger Geldbetrag, der für eine Beförderung an das Reisebüro zu bezahlen ist.

Flugverfügbarkeit, Anzahl der freien Plätze eines Fluges in den verschiedenen Flugklassen.

Passagier, Person, die von einer Fluggesellschaft auf einem Flug befördert wird.

Preisschema, beschreibt die Auf- bzw. Abschläge, die für die Business und First Class bzw. bei Kindern und Kleinkindern berechnet werden. Anhand des einem Flug zugewiesenen P. werden die Flugpreise berechnet.

Reisebüro, Geschäftspartner von Fluggesellschaften. Das R. vertreibt die von Fluggesellschaften angebotenen Beförderungsleistungen.

-nummer, achtstellige Zahl, die ein Reisebüro eindeutig identifiziert.

Reisenummer, siehe Flugreise.

Teilstrecke einer Flugverbindung, Teil einer Flugverbindung. Eine T. bezieht sich auf einen Startort und einen Zielort und auf eine bestimmte Zeit (Datum, Uhrzeit). Die Beförderung auf der T. wird durch genau einen Flug erbracht.

Komponentenfindung in monolithischen betrieblichen Anwendungssystemen

Andreas Krammer, Johannes Maria Zaha

Lehrstuhl Wirtschaftsinformatik II, Universität Augsburg, Universitätsstrasse 16, 86135 Augsburg, E-mail: {Andreas.Krammer|Johannes.Maria.Zaha}@wiwi.uni-augsburg.de, Tel: +49(821)598-4431, FAX:-4432, URL: http://wi2.wiwi.uni-augsburg.de

Zusammenfassung: Die Transformation von existierenden monolithischen zu komponentenorientierten betrieblichen Anwendungssystemen erschließt die Vorteile der Komponentenorientierung, bei gleichzeitig weitreichender Wiederverwendung vorhandener Programmbausteine. In diesem Beitrag wird das komponentenorientierte Architekturparadigma als Lösungsansatz für Probleme dargestellt, die bei monolithischen Anwendungssystemen auftreten. Vorhandene Ansätze zur Identifikation von Komponenten bei der Entwicklung von Anwendungssystemen werden diskutiert. Schließlich wird ein Lösungsalgorithmus entwickelt, der zur effizienten und intersubjektiv nachvollziehbaren Identifikation von Komponenten in existierenden monolithischen betrieblichen Anwendungssystemen genutzt werden kann.

Schlüsselworte: Monolithische betriebliche Anwendungssysteme, Komponentenfindung, Design to Component, Fachkomponente, Komponenten-Anwendungs-Framework, Komponenten-System-Framework, Funktionsdekomposition

1 Einleitung

Das Ziel, Software der betrieblichen Anwendungsdomäne in handhabbaren, für sich stehenden Einheiten zu produzieren und mit geringem Aufwand zu betrieblichen Anwendungssystemen zu kombinieren, wobei dem Kompositeur Implementierungsdetails verborgen bleiben, wird seit langem verfolgt [McIl1968]. Zusammenfassen lässt sich dieses Ziel als Leitbild der kompositorischen, plug-and-play-artigen Wiederverwendung von Black-Box-Komponenten, die auf einem Softwaremarkt gehandelt werden [vgl. Turo2002, S. 1].

Im Rahmen der komponentenorientierten Software-Entwicklung lassen sich drei grundlegende Design-Prinzipien unterscheiden [WKU+1999]:

- Design for Component

  Als wesentliches Ziel des Design for Component kann die Neuentwicklung kleiner, funktional leicht definierbarer Software-Komponenten verstanden werden, welche später in eine neue Anwendung schnell und einfach integriert werden können.

- Design from Component

  Dieses grundlegende Design-Prinzip beschreibt die Komposition von eigenentwickelten oder am Software-Markt erworbenen Komponenten zu einem betrieblichen Anwendungssystem. Hierbei sind sowohl technische als auch fachliche Anpassungen im Rahmen eines vom Programmierer vorgesehenen Parametrisierungsraumes als auch die technische und fachliche Integration der Fachkomponenten in ein Anwendungssystem

durchzuführen [vgl. Turo2001, S. 44-47].

- Design to Component

  Im Rahmen des Design to Component werden existierende Altanwendungen einem grundlegende Redesign in architektonischer Hinsicht unterworfen. Durch die Identifikation und ggf. programmtechnische Umgestaltung des Altsystems werden eigenständige Komponenten geschaffen, sodass die Vorteile der Komponentenorientierung erschlossen werden können.

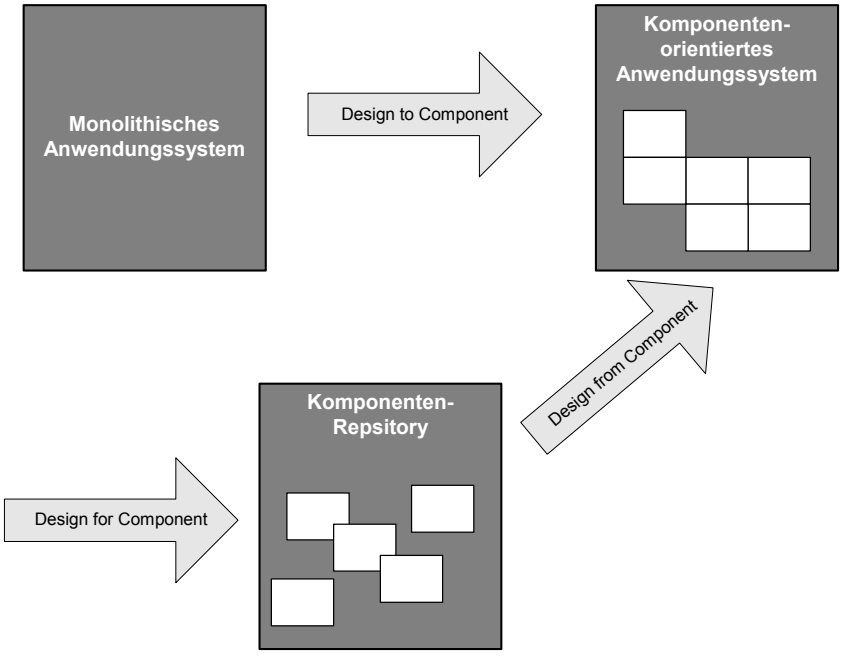

Bild 1: Design Prinzipien [vgl. HöWe2001, S. 5]

Bild 1 zeigt die grundlegenden Design-Prinzipien im Überblick. In diesem Beitrag wird auf das Design to Component fokussiert. Gegenstand der Untersuchung ist die Komponentenfindung in existierenden monolithischen Anwendungssystemen der betrieblichen Anwendungsdomäne.

Nach der Definition monolithischer Anwendungssysteme und einer Darstellung von Problemen, die mit einer derartigen Anwendungsarchitektur einhergehen, werden das Paradigma der komponentenorientierten Softwarearchitektur beschrieben und die Vorteile einer Transformation von monolithischen zu komponentenorientierten Anwendungssystemen aufgezeigt. Anschließend werden vorhandene Arbeiten zur Komponentenfindung ausgewertet und auf ihre Anwendbarkeit hin untersucht. Ergebnis der Untersuchung ist die Entwicklung eines neuen Ansatzes zur Komponentenfindung in existierenden monolithischen Anwendungssystemen auf Basis der Dekomposition der durch das Anwendungssystem angebotenen Funktionen und deren Gegenüberstellung zu Implementierungsartefakten des Anwendungssystems. Anschließend wird ein Vorschlag entwickelt, in welcher Weise die Implementierungsartefakte auf die Architekturbestandteile des komponentenorientierten Systems zu verteilen sind.

- Dieser Beitrag beschränkt sich auf die Umgestaltung eines existierenden monolithischen Anwendungssystems und bietet keine Vorschläge zur Umgestaltung des Funktionsumfangs des Gesamtanwendungssystems. Insbesondere wird durch die Identifikati-

on der Fachkomponenten keine Standardisierung derselben behandelt. Ferner beschränkt sich die vorliegende Arbeit bei der Identifikation von Fachkomponenten auf Aspekte, die der Funktionssicht zuzuordnen sind. Die Datensicht wird dabei lediglich implizit durch den von der Funktionalität verwalteten Ausschnitt des Datenmodells berücksichtigt.

2 Monolithische versus komponentenorientierte Anwendungssysteme

2.1 Monolithische Anwendungssysteme

Charakteristisch für frühe betriebliche Anwendungssysteme ist deren monolithische Umsetzung. Die Systemteile eines Monolithen sind teilweise nicht klar voneinander abgrenzbar, seine Systemteile sind so eng miteinander vermascht, dass Systemteile nur schwer herausgelöst oder ersetzt werden können. Ferner zeichnen sich monolithische Anwendungssysteme dadurch aus, dass sie sowohl Systemteile umfassen, die anwendungsbezogen sind, als auch solche, die anwendungsübergreifend verwendet werden können, z. B. Systemteile zur Datenverwaltung [vgl. Turo2001, S. 28].

Der monolithischen Umsetzung sind einige informationstechnische als auch wirtschaftliche Probleme immanent:

- Kostenintensive Wartbarkeit und Erweiterbarkeit

  Betriebliche Anwendungssysteme sind während ihres Lebenszyklus einer ständigen Weiterentwicklung unterworfen. Durch die enge Vermaschung der Systemteile ist bei Änderungen immer mit unbeabsichtigten Auswirkungen auf andere Systemteile zu rechnen. Diese können in der Regel nur durch intensive Testanstrengungen lokalisiert und beseitigt werden. Ferner werden hohe Anforderungen an die Entwickler gestellt, die sich aufgrund der engen Verzahnung der Systemteile einen Überblick über Abhängigkeiten weiter Systemteile verschaffen müssen. Daher ist mit langen, kostenintensiven Einarbeitungszeiten bei neuen Mitarbeitern zu rechnen.

- Eingeschränkte Vermarktungsmöglichkeiten

  Monolithische Systeme sind nur als Komplettlösung zu installieren und bieten dem Käufer nur eingeschränkte Erweiterungsmöglichkeiten. Ferner ergibt sich für den Nutzer eine enge Bindung und damit Abhängigkeit vom Hersteller des Systems. Daher können die Vermarktungsmöglichkeiten monolithische Systeme als sehr begrenzt angesehen werden.

- Eingeschränkte Skalierbarkeit

  Bei wachsenden Anforderungen an die Performance, ist eine Erhöhung der Verfügbarkeit nur durch den Einsatz eines identischen Replikats der Software möglich. Einzelne Teile der Software können nicht extrahiert und redundant angeboten werden. Dadurch sind die Kosten für eine Verbesserung der Performance des Gesamtsystems sehr hoch, obwohl nicht alle Teile davon kritische Ressourcen darstellen.

2.2 Komponentenorientierte Anwendungssysteme

Oben genannte Probleme können durch den Einsatz komponentenorientierter betrieblicher Anwendungssysteme weitgehend vermieden werden. Im Folgenden soll ein Überblick über

die Architektur komponentenorientierter betrieblicher Anwendungssysteme gegeben werden und die damit verbundenen Vorteile dargestellt werden. Hierbei wird die Komponentendefinition des Arbeitskreises 5.10.3 „Komponentenorientierte betriebliche Anwendungssysteme“ der Gesellschaft für Informatik verwendet [Turo2002]:

> Eine *Komponente* besteht aus verschiedenartigen (Software-)Artefakten. Sie ist wiederverwendbar, abgeschlossen und vermarktbar, stellt Dienste über wohldefinierte Schnittstellen zur Verfügung, verbirgt ihre Realisierung und kann in Kombination mit anderen Komponenten eingesetzt werden, die zur Zeit der Entwicklung nicht unbedingt vorhersehbar ist.
>
> Eine *Fachkomponente (FK)* ist eine Komponente, die eine bestimmte Menge von Diensten einer *betrieblichen* Anwendungsdomäne anbietet.

Damit Fachkomponenten in der im Leitbild definierten Art und Weise betrieben werden können, werden zusätzliche Systemteile benötigt, sog. Komponenten-Anwendungs-Frameworks und Komponenten-System-Frameworks [vgl. Turo2001, S. 35].

Unter einem *Komponenten-Anwendungs-Framework* wird ein Systemteil verstanden, der Fachkomponenten anwendungsdomänenbezogene (Standard-)Dienste bereitstellt und für diese eine Integrationsplattform darstellt.

Darüber hinaus werden von sog. *Komponenten-System-Frameworks* anwendungsinvariante, middlewarenahe Dienste zur Verfügung gestellt. Beispiele für Komponenten-System-Frameworks sind Datenbank-Management-Systeme (DBMS) oder Workflow-Management-Systeme (WFMS).

Bild 2 zeigt den strukturellen Aufbau komponentenorientierter betrieblicher Anwendungssysteme mit den oben identifizierten Systemteilen im Überblick.

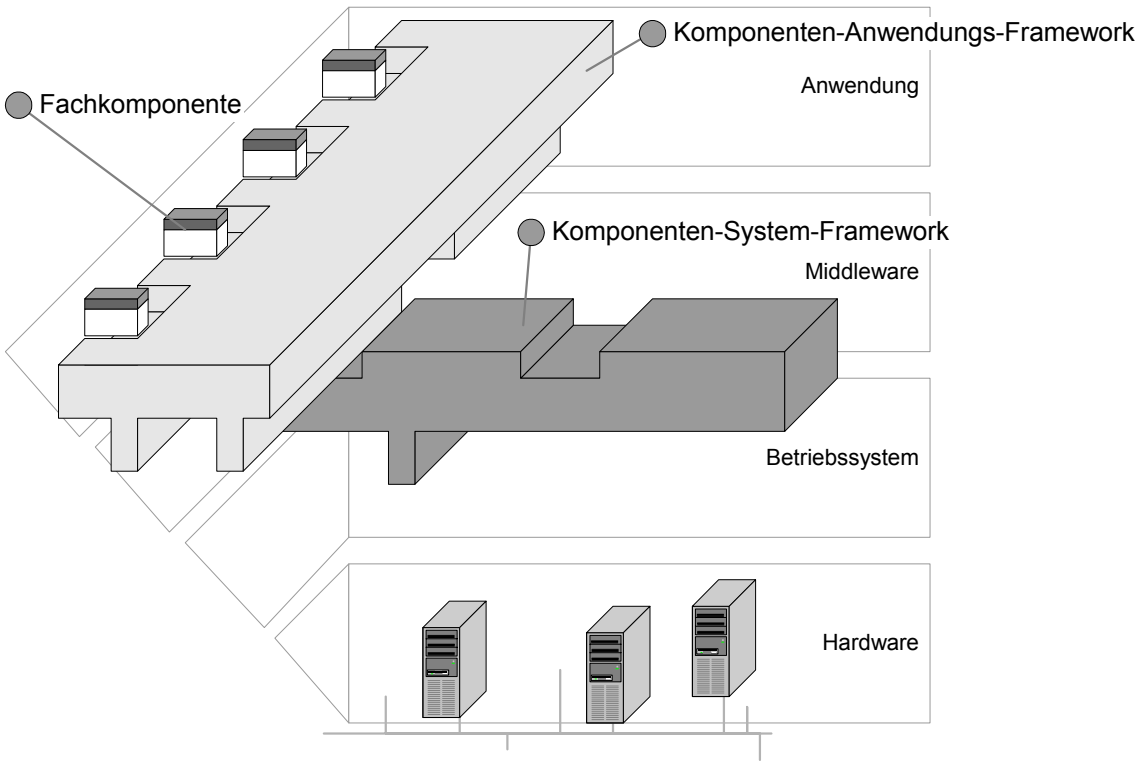

Bild 2: Generelle Architektur komponentenorientierter betrieblicher Anwendungssysteme [vgl. Turo2001, S. 36]

Die mit monolithischen Anwendungssystemen verbundenen grundlegenden Probleme treten bei komponentenorientierten Anwendungssystemen in der Regel nicht auf.

- Aufgrund der Abgeschlossenheit der einzelnen Fachkomponente beschränken sich Wartungsarbeiten immer auf eine bzw. wenige Fachkomponenten. Änderungen an der Implementierung einer Fachkomponente haben definitionsgemäß keine Auswirkungen auf andere Fachkomponenten.
- Die Weiterentwicklung von Anwendungssystemen findet nicht durch Änderungen an der Implementierung des Gesamtsystems statt, sondern durch Austausch einzelner Fachkomponenten. Werden die Fachkomponenten mit erweiterter Funktionalität auf dem Softwaremarkt bezogen, so ergeben sich geringere Aufwände als durch die Modifikation der Implementierung.
- Die (marktliche) Wiederverwendung von Fachkomponenten ist leichter als diejenige von kompletten Anwendungssystemen. Da erst durch die Komposition von Fachkomponenten ein auf die individuellen Bedürfnisse eines Unternehmens zugeschnittenes Anwendungssystem entsteht, ist der Markt für Bauteile, die in unterschiedlichen Konfigurationen eingesetzt werden können größer, als derjenige für starre Komplettlösungen. Erfahrungen aus anderen Industriezweigen, wie beispielsweise die Plattformstrategien in der Automobilindustrie, zeigen, dass durch die Komponentenorientierung eine Erhöhung der Stückzahl der wiederverwendbaren Bauteile möglich ist (Economies of Scale).
- In der Regel lassen sich durch die Komponentenorientierung auch Vorteile im Bereich der Skalierbarkeit erzielen. Einerseits können einzelne Komponenten bei Bedarf gegen Komponenten mit besserer Performance ausgetauscht werden. Andererseits wird die Verteilung der Anwendungskomponenten auf unterschiedliche Rechner möglich, was die Aufwände zur Skalierung des Gesamtsystems auf die Komponenten beschränkt, die den Engpass darstellen.

3 Transformation von monolithischen zu komponentenorientierten Anwendungssystemen

Unternehmen, die komplexe monolithische Anwendungssysteme im Einsatz haben und zukünftig die oben beschriebenen Vorteile komponentenorientierter Anwendungssysteme erschließen wollen, stehen im Rahmen der Transformation ihres Systems vor dem Problem der Komponentenfindung. Um den Transformationsprozess effizient zu gestalten, ist ein Lösungsalgorithmus zur Komponentenfindung notwendig, der zu einer intersubjektiv nachvollziehbaren Identifikation von Komponenten führt.

3.1 Vorhandene Arbeiten zur Komponentenfindung

Das Business Systems Planning [IBM1981] stellt eine Methode zur Entwicklung einer Informationssystem-Architektur dar [vgl. Hein1999, S. 349-359]. Die Strukturierung von Daten- und Informationssystemen erfolgt unter besonderer Berücksichtigung der Informationsnachfrage im Unternehmen.

Ausgehend von einer Definition der für das jeweilige Unternehmen relevanten Geschäftsprozesse sowie der Zusammenfassung logisch zusammengehöriger Datenobjekte zu Datenklassen, werden in einer zweidimensionalen Matrix Entstehungs- und Verwendungsbeziehungen zwischen Geschäftsprozessen und Datenklassen analysiert. Anschließend werden in der er-

stellten Matrix durch Clusterbildung Informationssysteme und Kommunikationsbeziehungen zwischen diesen identifiziert.

Das Business Systems Planning stellt einen Top-down-Ansatz zur Systemstrukturierung dar, der von betrieblichen Prozessen ausgehend die Neuentwicklung von Informationssystemen unterstützt. Durch Beschränkung auf die vom bestehenden Anwendungssystem unterstützten Geschäftsprozesse ist eine Anwendung des Ansatzes bei der Transformation von bestehenden monolithischen zu komponentenorientierten Anwendungssystemen denkbar. Die sich beim Business Systems Planning ergebenden (Teil-)Systeme sind jedoch für die Definition von Fachkomponenten zu grobgranular. Insbesondere ist keine Abgrenzung von Komponenten-Anwendungs-Framework, Komponenten-System-Framework und Fachkomponenten möglich.

Ein Ansatz, der explizit zur Komponentenfindung bei der Entwicklung komponentenorientierter betrieblicher Anwendungssysteme dient, wird in [FeTu2000] beschrieben.

Ausgehend von konzeptuellen Referenzmodellen in Form von erweiterten ereignisgesteuerten Prozessketten (eEPK) [Sche1994] für die betrachtete Anwendungsdomäne, wird für jedes Funktions- und Informationsobjekt in einem ersten Schritt eine Fachkomponente definiert. Während die Fachkomponenten zu Informationsobjekten auf der Basis von Entitäten in detaillierteren Datenmodellen weiter unterteilt werden, findet eine Verfeinerung der Fachkomponenten, die zu Funktionsobjekten gehören, nicht statt. In einem weiteren Schritt werden Fachkomponenten ähnlicher Funktionalität zu abstrakten Fachkomponenten verallgemeinert. Schließlich werden Vorschläge beschrieben, wie die identifizierten Fachkomponenten zu komplexen Fachkomponenten zusammengeführt werden, die einen gegebenen Geschäftsprozess unterstützen.

Vergleichbar zum Business Systems Planning, ist die Unternehmensmodellierung der Ausgangspunkt der beschriebenen Untersuchung. Daher erscheint dieser Ansatz vor allem für die Standardisierung und Neuentwicklung komponentenbasierter Anwendungssysteme geeignet. Um diesen Ansatz für die Transformation von monolithischen zu komponentenorientierten betrieblichen Anwendungssystemen zu nutzen, ist als Ausgangspunkt ein Prozessmodell zu erstellen, welches die durch das existierende Anwendungssystem unterstützten Geschäftsprozesse darstellt. Wie beim Business Systems Planning, ist auch bei diesem Ansatz eine Identifikation von Teilen des Komponenten-Anwendungs-Framework und des Komponenten-System-Framework nicht möglich. Dies liegt in der Beschränkung des Prozessmodells auf die für die Anwendungsdomäne spezifischen betrieblichen Funktionen begründet.

3.2 Komponentenfindung in monolithischen betrieblichen Anwendungssystemen

Im Folgenden wird ein Lösungsalgorithmus zur Komponentenfindung in existierenden monolithischen Anwendungssystemen der betrieblichen Anwendungsdomäne vorgestellt, der die mit oben angeführten Ansätzen verbundenen Probleme nicht aufweist. Darüber hinaus führt dieser Ansatz zu einer effizienten und intersubjektiv nachvollziehbaren Identifikation von Fachkomponenten.

Arbeitsschritt 1: Zerlegung in funktionaler Sicht

Ausgangspunkt der Komponentenfindung ist das betrachtete Anwendungssystem, das einer hierarchischen Zerlegung in Funktionen unterzogen wird. Diese wiederum werden zu Elementarfunktionen verfeinert.

Diese Zerlegung funktioniert analog zur Erstellung eines Funktionsdekompositionsdiagramms

(bzw. Funktionsbaums) [vgl. Sche1991, S. 65], das in seiner ursprünglichen Anwendung zur Modellierung der betrieblichen Funktionen eines Unternehmens eingesetzt wird. Dabei wird der zu modellierende Geschäftsprozess (verstanden als Bündelung von betrieblichen Funktionen), hierarchisch in Funktionen und diese wiederum in Teilfunktionen zerlegt. Auf unterster Ebene stehen Elementarfunktionen, die aus betriebswirtschaftlicher Sicht nicht mehr sinnvoll zerlegt werden können.

In diesem Beitrag wird diese Methode zur Modellierung der Funktionssicht eines existierenden monolithischen Anwendungssystems benutzt. Das Gesamtsystem wird dazu, wie in Bild 3 dargestellt, in die angebotenen Funktionen zerlegt, die wiederum aus Elementarfunktionen bestehen. Auf die Berücksichtigung von Teilfunktionen wird verzichtet.

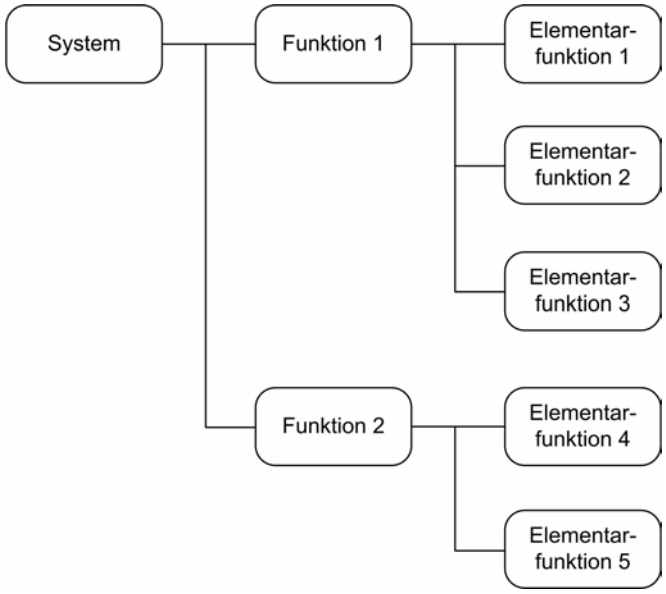

Bild 3: Funktionsdekompositionsdiagramm

Arbeitsschritt 2: Gegenüberstellung der implementierten Methoden bzw. Prozeduren

Bild 4 zeigt einen Ausschnitt aus einem Vertragsverwaltungssystem für den Versicherungsbereich. Im monolithischen Gesamtsystem lassen sich Funktionen zur Verwaltung von Lebensversicherungen (LV) und Haftpflichtversicherungen (HV) identifizieren. Diese werden jeweils in Elementarfunktionen untergliedert.

Dieser Zerlegung wird eine aus der Systemdokumentation bzw. dem Quellcode entnommene Auflistung der im Anwendungssystem implementierten Methoden bzw. Prozeduren gegenübergestellt. Dabei wird unterstellt, dass es sich um eine prozedural oder objektorientiert implementierte Software handelt. Weiterhin wird angenommen, dass bei der Entwicklung des Systems die grundlegenden Prinzipien des Software-Engineerings zugrunde gelegt wurden. Andernfalls müsste ein Refactoring [vgl. Fowl2000] der Software durchgeführt werden, um nach den im Folgenden beschriebenen Schritten ein sinnvolles Ergebnis zu erhalten.

Die im Altsystem implementierten Methoden bzw. Prozeduren werden den Elementarfunktionen zugeordnet. Hierbei kann eine Methode bzw. Prozedur in mehreren Elementarfunktionen genutzt werden und ist dann mit den entsprechenden Elementarfunktionen zu verbinden (siehe Bild 4, Methode/Prozedur updateTechLV(), cancelLV() und newCustomer()). Dadurch wird die streng hierarchische Gliederung des Funktionsdekompositionsdiagramms (bzw. Funktionsbaums) aufgegeben. Methoden bzw. Prozeduren, die keiner Elementarfunktion zuordenbar sind, werden neben dem Diagramm notiert (siehe Bild 4, Methode/Prozedur getDa-

taFromDB() und updateDataInDB()).

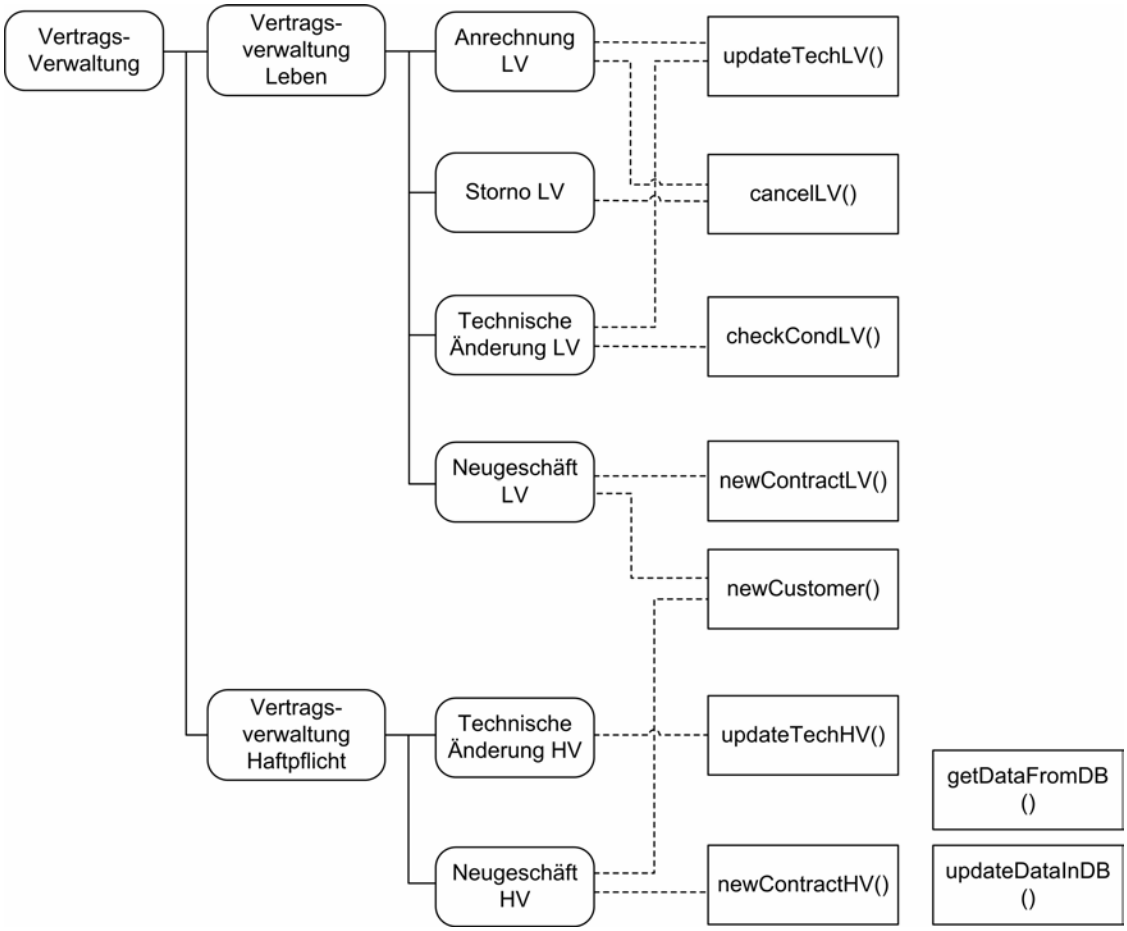

Bild 4: Funktionsdekompositionsdiagramm – Zuordnung Methoden/Prozeduren

Arbeitsschritt 3: Identifikation der Komponenten-Frameworks

Anhand der Verteilung der einzelnen Methoden bzw. Prozeduren kann eine Aussage darüber getroffen werden, ob diese dem Komponenten-System-Framework, dem Komponenten-Anwendungs-Framework oder den (zu definierenden) Fachkomponenten zuzuordnen sind.

Methoden bzw. Prozeduren, die mit keiner Elementarfunktion verbunden sind, bilden das Komponenten-System-Framework, da sie keinen Bezug zu den fachlichen Aspekten der Anwendung haben.

Werden Methoden bzw. Prozeduren von mehreren Elementarfunktionen benutzt, die unterschiedlichen Funktionen angehören, bilden diese das Komponenten-Anwendungsframework. Eine Integration der betreffenden Methoden/Prozeduren in mehrere Fachkomponenten ist aufgrund der dadurch entstehenden Redundanz nicht zu empfehlen.

Die Methode/Prozedur newCustomer() wird dem Komponenten-Anwendungs-Framework zugeordnet, da Sie in den Funktionen Vertragsverwaltung Leben und Vertragsverwaltung Haftpflicht genutzt wird.

Da ein Komponenten-Anwendungs-Framework als Systemteil definiert ist, der Fachkomponenten anwendungsdomänenbezogene (Standard-)Dienste (Elementarfunktionen) zur Verfügung stellt, wird der Methode bzw. Prozedur newCustomer() ein neuer Dienst zugeordnet (siehe Bild 5, Dienst Anlegen neuer Kunde).

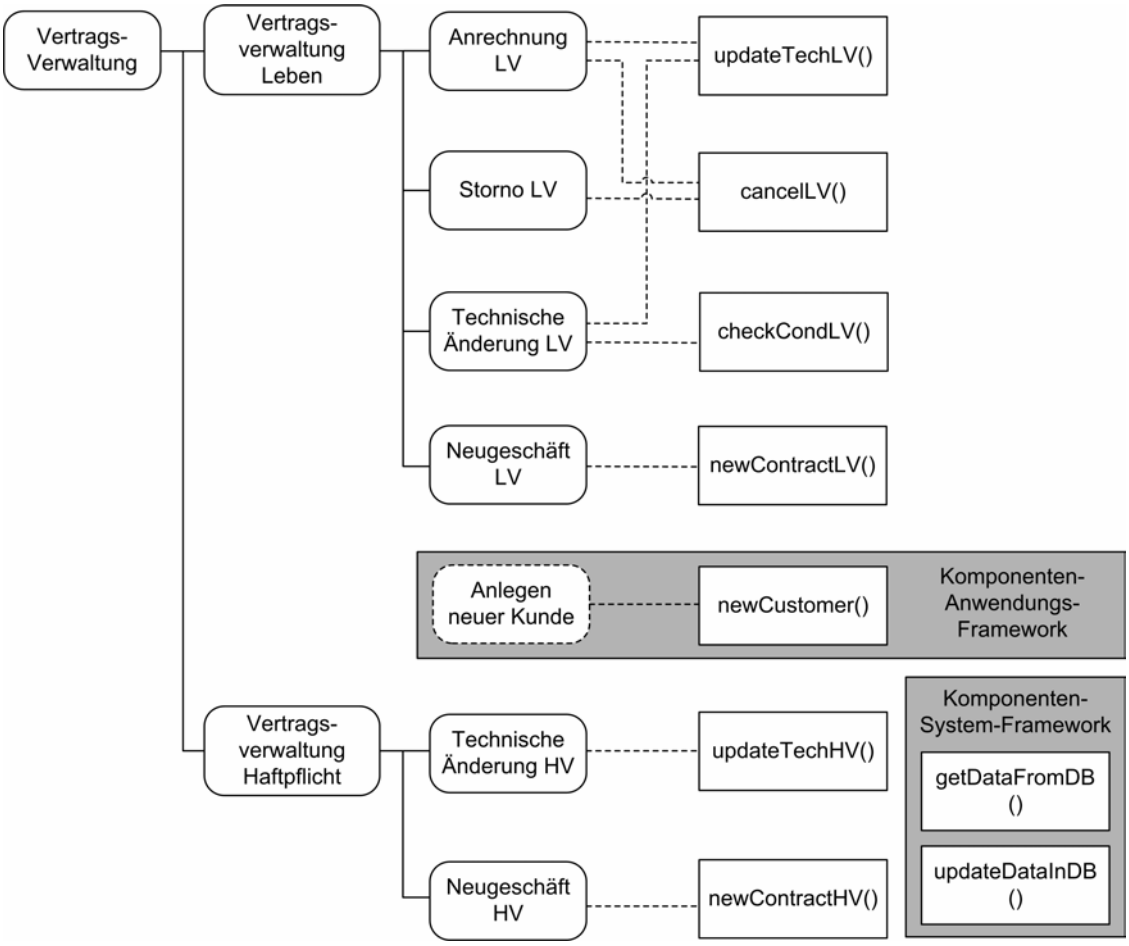

Bild 5: Funktionsdekompositionsdiagramm – Komponenten-Frameworks

Arbeitsschritt 4: Identifikation von Fachkomponenten

Nach oben wiedergegebener Definition kann eine Fachkomponente aus anderen Fachkomponenten aufgebaut sein.

Eine *elementare Fachkomponente (eFK)* hingegen ist definiert als Fachkomponente, die nicht weiter unterteilt ist. Es existiert keine andere (kleinere) FK, die eine Teilmenge der Funktionalität der eFK umfasst [vgl. FeTu2000, S. 162]. Elementare Fachkomponenten sind bzgl. der von ihnen angebotenen Dienste (entspricht Elementarfunktion) disjunkt [vgl. Turo2001, S. 83].

Bei der hier vorgenommenen Zerlegung eines monolithischen Anwendungssystems entstehen ausschließlich elementare Fachkomponenten. Fachliche Konflikte [vgl. Turo2001, S. 166] sind hierdurch ausgeschlossen.

Generell stellt sich die Frage nach der geeigneten Größe einer Fachkomponente und damit ihrer Granularität. Mit zunehmend feiner Granularität der Fachkomponenten erhöht sich der Kommunikations- und Koordinationsbedarf zwischen vielen Einzelkomponenten eines Systems. Gleichzeitig gewinnen folgende Vorteile an Bedeutung:

- Vereinfachung der Wiederverwendung der Komponenten, da die sich ergebenden Kombinationen nicht allgemeingültig sein müssen und gegebenenfalls nur in dem betrachteten monolithischen System Sinn machen, nicht aber in anderen Anwendungssze-

narien

- Leichtere Erweiterbarkeit um neue (elementare) Fachkomponenten, da die Fachkomponenten bezüglich der enthaltenen Dienste disjunkt sind und dadurch keine Konflikte durch mehrfach angebotener Dienste auftreten können
- Vergleichbarkeit mehrerer auf einem Komponentenmarkt gehandelter Komponenten bezüglich Funktionsumfang und Preis

Als Untergrenze für die Granularität wird die Bildung von minimalen Legacy-Komponenten vorgeschlagen.

> Eine *minimale Legacy-Komponente* setzt sich aus der minimalen Anzahl von Elementarfunktionen (bzw. diesen zugeordneten Methoden/Prozeduren) einer Funktion zusammen, wobei eine Methode/Prozedur in genau einer minimalen Legacy-Komponente implementiert ist.

Die Obergrenze der in einer Fachkomponente zusammengefassten Dienste (Elementarfunktionen) ist durch die Methoden bzw. Prozeduren vorgegeben, die innerhalb einer identifizierten Funktion liegen.

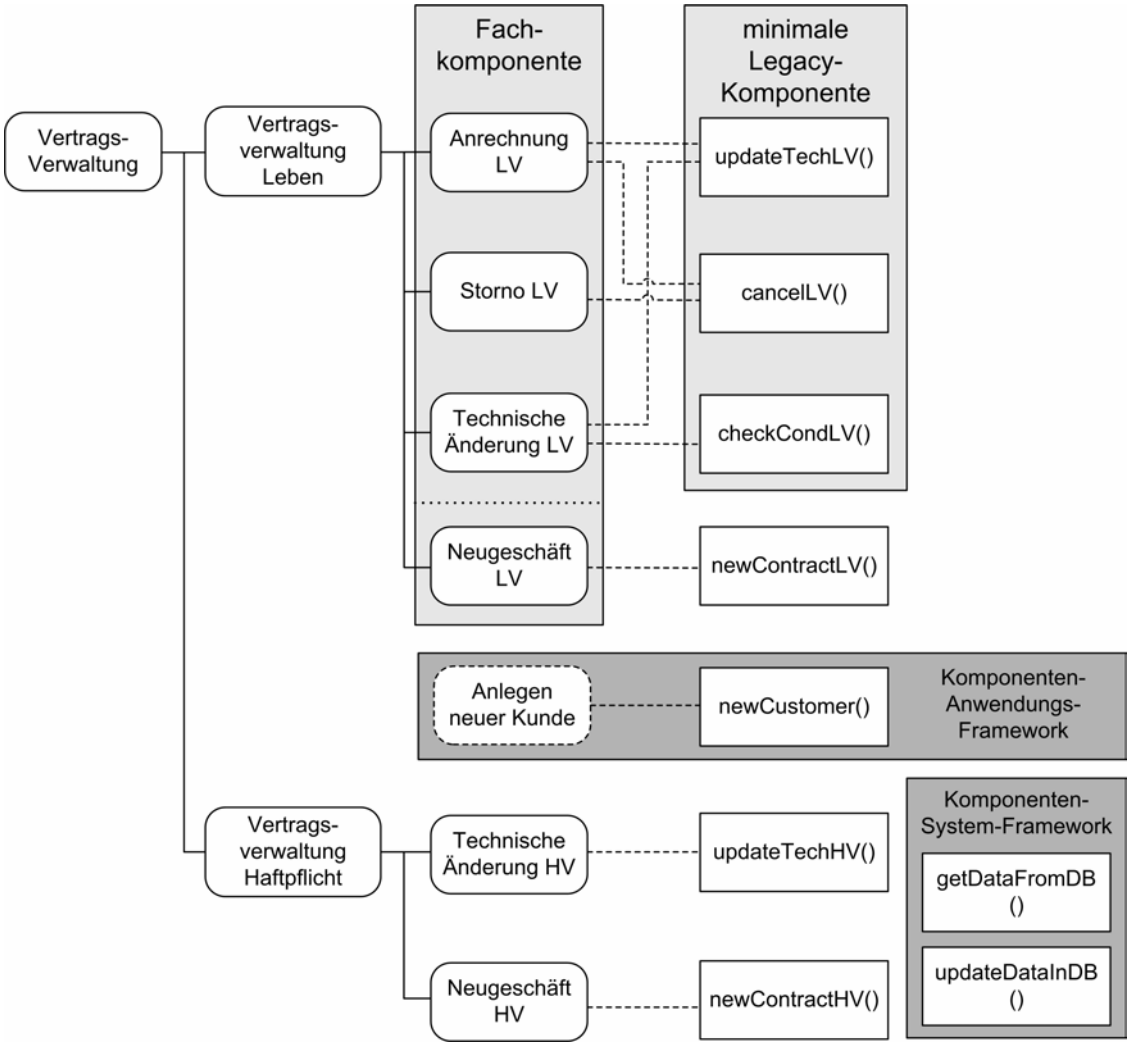

Bild 6: Funktionsdekompositionsdiagramm - Fachkomponenten

Damit ergibt sich ein erheblicher Handlungsspielraum bei der Bildung der Fachkomponenten. Einen Anhaltspunkt dafür können auf erster Ebene die Beziehungen zwischen den Diensten (Elementarfunktionen) und auf der darunter liegenden Ebene die Beziehungen zwischen Methoden bzw. Prozeduren geben. Haben zwei Elemente bezüglich des ausgetauschten Datenvolumens eine besonders große Bedeutung, ist es sinnvoll, diese in einer Fachkomponente zusammenzufassen. Dies zielt auf die Reduktion des Koordinations- und Kommunikationsbedarfs zwischen den Fachkomponenten ab.

In Bild 6 ist die Bildung einer minimalen Legacy-Komponente nach oben angegebener Definition gezeigt.

4 Zusammenfassung und Ausblick

Im Beitrag wurden die Probleme, die mit monolithischen Anwendungssystemen einhergehen, aufgezeigt und die Transformation zu einer komponentenorientierten Architektur motiviert. Ein Lösungsalgorithmus zur Komponentenfindung im Rahmen der Transformation wurde vorgestellt, der zu einer eindeutigen Aufteilung der vorhandenen Implementierungsartefakte auf die verschiedenen Architekturbestandteile komponentenorientierter Systeme führt und den Gestaltungsspielraum bei der Identifikation von Fachkomponenten einschränkt. Weiterhin wurden Kriterien identifiziert, die zu einem begründeten Abweichen vom dargestellten Vorgehen zur Komponentenfindung führen.

Grundsätzlich lässt sich der dargestellte Lösungsalgorithmus auf die Transformation von Client/Server-Systemen übertragen. Bei hinreichender Kapselung der Serverfunktionalität bleibt zu untersuchen, ob eine Betrachtung von Prozeduren bzw. Methoden notwendig ist, oder eine Berücksichtigung grobgranularerer Softwareartefakte zur Komponentenfindung ausreicht.

Die Anwendbarkeit der vorgeschlagenen Methode bleibt durch eine geeignete Fallstudie zu verifizieren, bei der auch die Beschränkung der Methode auf die Funktionssicht kritisch betrachtet werden sollte. Ferner sind die Einbeziehung des Funktionsumfangs am Markt erhältlicher Fachkomponenten in den Prozess der Komponentenfindung und die Aspekte der Standardisierung von Fachkomponenten in weiterführenden Forschungsarbeiten zu betrachten.

Literatur

[FeTu2000] *Fellner, K.; Turowski, K.*: Identifying Business Components Using Conceptual Models. In: *M. Khosrowpour (Hrsg.)*: Challenges of Information Technology Management in the 21st Century: 2000 Information Resources Management Association International Conference, Anchorage, Alaska, USA, May 21-24, 2000. Anchorage 2000, S. 161-165.

[Fowl2000] *Fowler, M.:* Refactoring: Wie Sie das Design vorhandener Software verbessern. Addison-Wesley, München 2000.

[Hein1999] *Heinrich, L. J.*: Informationsmanagement: Planung, Überwachung und Steuerung der Informationsinfrastruktur. 6. Aufl., Oldenbourg, München 1999.

[HöWe2001] *Höß, O.; Weisbecker, A.*: Komponentenbasierte Software für Produkte und Dienstleistungen (KoSPuD) - Ergebnisse und Erfahrungen eines praxisorientierten Verbundforschungsprojekts. In: *K. Turowski (Hrsg.)*: Tagungsband zum 3. Workshop komponentenorientierte betriebliche Anwendungssysteme (WKBA3). Frankfurt am Main 2001, S. 1-13.

[IBM1981] *IBM Corporation (Hrsg.)*: Business Systems Planning: Information Systems Planning Guide. Armonk 1981.

[McIl1968] *Mcllroy, M. D.*: Mass Produced Software Components. In: *P. Naur; B. Randell (Hrsg.)*: Software Engineering: Report on a Conference by the NATO Science Comittee. NATO Scientific Affairs Division, Brussels 1968, S. 138-150.

[Sche1991] *Scheer, A.-W.*: Architektur integrierter Informationssysteme. Springer, Berlin 1991.

[Sche1994] *Scheer, A.-W.*: Business Process Engineering: Reference Models for Industrial Enterprises. 2. Aufl., Springer, Berlin 1994.

[Turo2001] *Turowski, K.*: Fachkomponenten: Komponentenbasierte betriebliche Anwendungssysteme. Habilitationsschrift, Otto-von-Guericke-Universität Magdeburg. Magdeburg 2001.

[Turo2002] *Turowski, K. (Hrsg.)*: Vereinheitlichte Spezifikation von Fachkomponenten. Arbeitskreis 5.3.10 der Gesellschaft für Informatik, Augsburg 2002.

[WKU+1999] *Weisbecker, A.; Kunsmann, J.; Ullrich, A.; Schuster, E.*: Komponentenbasierte Software-Entwicklung für Produkte und Dienstleistungen. In: Information Management und Consulting 14 (1999), S. 19-23.

Eine Infrastruktur für den Austausch von Fachkomponenten

Holger Jaekel, Thorsten Teschke

OFFIS, Escherweg 2, 26121 Oldenburg, Deutschland, Tel.: (04 41) 97 22 - 1 25, Fax: - 1 02, E-Mail: {holger.jaekel | thorsten.teschke}@offis.de. URL: http://www.offis.de

Zusammenfassung. Neben der fachlichen und technischen Standardisierung ist für den Erfolg der komponentenbasierten Softwareentwicklung auch eine Unterstützung des Entwicklungsprozesses durch entsprechende Werkzeuge notwendig. In diesem Beitrag wird die im Projekt KOSOBAR entwickelte Infrastruktur zur Unterstützung der komponentenbasierten Softwareentwicklung vorgestellt. Mittels dieser Infrastruktur interagieren die an der Softwareentwicklung beteiligten Akteure durch Austausch standardisierter Komponentenbeschreibungen, deren Semantik durch einfache normsprachliche Sätze über einer standardisierten Terminologie definiert wird. Diese Beschreibungen werden von Suchdiensten verwendet, die auf einem Komponentenmarkt geeignete Komponenten auf Basis von Geschäftsprozessmodellen finden.

Schlüsselworte: Fachkomponente, Vorgehensmodell, Terminologie, Normsprache, Komponentensuche, Geschäftsprozessmodell, Behavioural Subtyping

1 Einleitung

Eine Voraussetzung für den Erfolg der komponentenbasierten Softwareentwicklung ist neben der technischen Standardisierung durch Komponentenmodelle wie z. B. Component Object Model (COM) und dessen Nachfolger DCOM und COM+, Enterprise JavaBeans (EJB) oder CORBA Component Model (CCM) und der fachlichen Standardisierung durch Interoperabilitätsbestrebungen wie z. B. ebXML oder OAGIS die Unterstützung des gesamten Entwicklungsprozesses durch geeignete Werkzeuge. Dazu gehören einerseits Werkzeuge zur Veröffentlichung standardisierter Komponentenbeschreibungen, andererseits sind Dienste für die Komponentensuche notwendig, die geeignete Komponenten auf einem Komponentenmarkt auf Basis fachlicher Anforderungen finden.

Ausgehend von dem Gedanken, Komponenten ähnlich wie große, betriebliche Standardsoftwareprodukte wie SAP R/3 durch strukturierte, semi-formale Beschreibungen (sogenannte *Softwarereferenzmodelle*) zu beschreiben [Rit98], ist in den Jahren 1999 bis 2002 das Projekt *KOSOBAR (Komponentenbasierte Softwareentwicklung auf Basis von Referenzmodellen)* in OFFIS durchgeführt worden. Ziel dieses Projekts ist die Entwicklung einer Infrastruktur für den Austausch von Fachkomponenten auf internetbasierten Komponentenmärkten gewesen. Dazu sind die an der Erstellung komponentenbasierter Software beteiligten Akteure und deren Interaktionen wie z. B. der Austausch von Komponenten oder Anforderungsdefinitionen in einem Vorgehensmodell für die komponentenbasierte Softwareentwicklung definiert worden. Ausgehend von diesem Vorgehensmodell wurde eine Infrastruktur zur Unterstützung komponentenbasierter Softwareentwicklungsprozesse entwickelt, die die Kommunikation der beteiligten Akteure auf der Grundlage standardisierter Komponentenbeschreibungen unterstützt. Diese Kommunikation betrifft insbesondere das Anbieten und die Suche von Fachkomponenten.

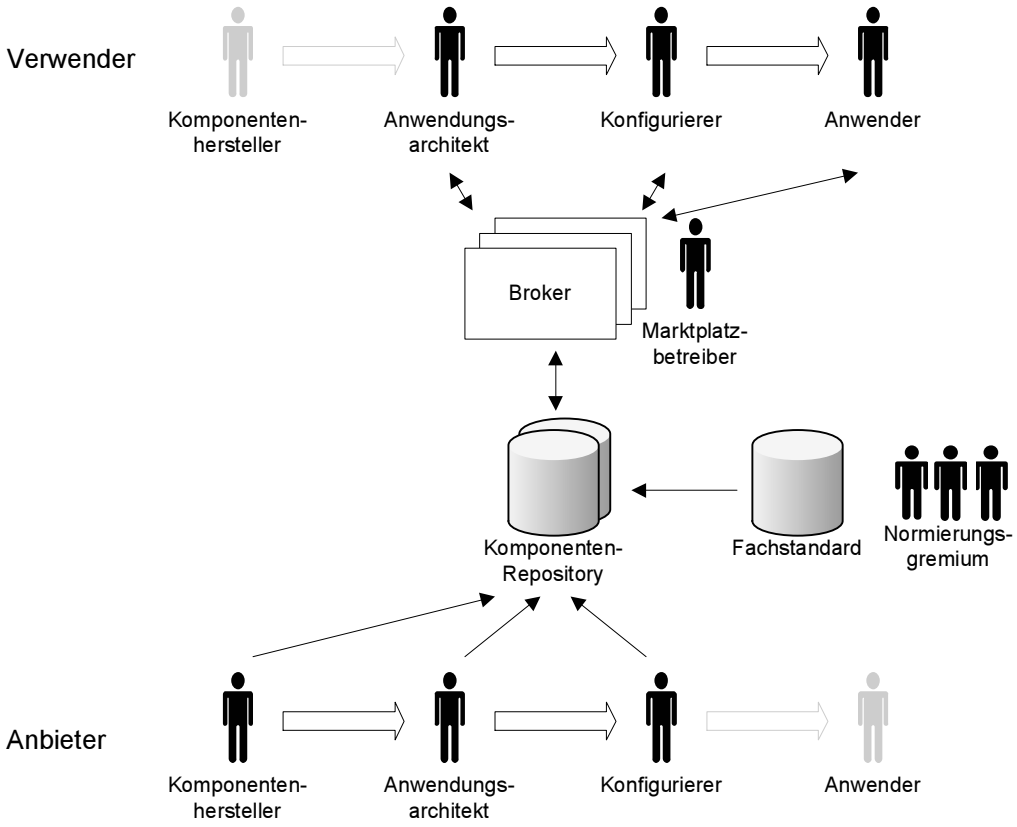

Abbildung 1: KOSOBAR-Vorgehensmodell

2 Vorgehensmodell und Architektur

Traditionelle (z. B. sequentielle, iterative oder evolutionäre) Vorgehensmodelle beschreiben in der Regel nur produktbezogene Aktivitäten wie Analyse, Entwurf, Implementierung, Test und Wartung, deren Ergebnisse unmittelbar in das Softwareprodukt eingehen. Demgegenüber werden prozessbezogene Aktivitäten wie z. B. die Koordination eines Softwareprojektes, die Kooperation zwischen den beteiligten Akteuren oder die Verwaltung des zu entwickelnden Produktes oft nicht in hinreichendem Maße berücksichtigt. Durch den Ansatz der komponentenbasierten Anwendungsentwicklung entsteht die Notwendigkeit, traditionelle Vorgehensweisen für die Entwicklung von Software zu überdenken, da dort Probleme wie die Vermarktung und der Erwerb von Komponenten nicht ausreichend berücksichtigt werden. Das im Projekt KOSOBAR entwickelte Vorgehensmodell [STR00] basiert auf der Annahme, dass sich künftig ein Komponentenmarkt etablieren wird, auf dem Anbieter umfassende Beschreibungen ihrer Komponenten in speziellen Repositories kostenlos zur Verfügung stellen. Broker ermöglichen Interessenten die Suche nach Komponenten auf Basis der in diesen Anbieter-Repositories abgelegten Beschreibungen. Abbildung 1 stellt das Vorgehensmodell anhand der beteiligten Akteure, ihrer Interaktionen sowie der zugrunde liegenden Architektur graphisch dar.

Während in anderen Vorgehensmodellen häufig nur die Rollen des Softwareherstellers und des Anwenders unterschieden werden, sehen wir die sechs Rollen Komponentenhersteller, Anwendungsarchitekt, Konfigurierer und Anwender sowie Marktplatzbetreiber und Normierungsgremium vor. Sie sind durch ihre jeweiligen charakteristischen Tätigkeiten (Entwicklung, Konstruktion, Konfiguration, Einsatz sowie Vermittlung und Normierung) gekennzeichnet. Die Rol-

le des *Komponentenherstellers* beinhaltet die Aufgabe der Entwicklung möglichst domänenunabhängiger Komponenten mit den Mitteln des klassischen Software Engineerings. Die Aufgabe des *Anwendungsarchitekten* besteht darin, aus verschiedenen Komponenten ein konfigurierbares Anwendungssoftwareprodukt zusammenzustellen. Die Grundlage eines solchen Anwendungssoftwareprodukts bilden in der Regel Frameworks, die durch Komposition mit geeigneten Komponenten erweitert werden müssen. Anwendungssoftwareprodukte können einerseits wieder als Komponente von einem Anwendungsarchitekten zur Entwicklung komplexerer Anwendungssoftwareprodukte weiterverwendet werden, andererseits können sie von einem *Konfigurierer* zu domänen-, branchen- oder unternehmenspezifischen Lösungen (vor-)konfiguriert werden. Indem der Konfigurierer den *Anwender* bei der Auswahl geeigneter Anwendungssoftwareprodukte und der Bewertung von Konfigurationsalternativen unterstützt, erbringt er zudem Beratungsleistungen gegenüber dem Anwender. Für die Auswahl von geeigneten Komponenten oder (vorkonfigurierten) Anwendungssoftwareprodukten nutzen die *Verwender* (Anwendungsarchitekten, Konfigurierer und Anwender) die Dienste eines *Brokers*, der von einem *Marktplatzbetreiber* unterhalten wird. Im Rahmen der Komponentensuche vergleicht ein Broker die von den *Anbietern* (Komponentenhersteller, Anwendungsarchitekten und Konfigurierer) in Komponenten-Repositories angebotenen Komponentenbeschreibungen mit den spezifizierten Anforderungen eines Verwenders. Zur Unterstützung der Interoperabilität von Komponenten sowie Effektivitätsverbesserungen bei der Komponentensuche beschreiben Anbieter die fachliche Semantik der von einer Komponente angebotenen Dienste unter Berücksichtigung von (domänenspezifischen) Fachstandards, die von *Normierungsgremien* spezifiziert werden. Die Rolle des Normierungsgremiums könnte einerseits von Organisationen wie z. B. die Open Applications Group (OAG) und die Organization for the Advancement of Structured Information Standards (OASIS) oder Unternehmen mit einer besonderen Kompetenz in einem Anwendungsbereich wie z. B. die DATEV eG eingenommen werden.

Ein Ziel des Projektes KOSOBAR ist es, die oben dargestellten Interaktionen zwischen den an der Erstellung von komponentenbasierten Anwendungssystemen beteiligten Akteuren informationstechnisch zu unterstützen. Dazu sind verschiedene Werkzeugen und Schnittstellen entwickelt worden, die im Folgenden überblicksartig vorgestellt werden sollen (siehe auch Abbildung 2). Die Aufgaben des Normierungsgremiums werden durch einen *Terminologie-Manager* zur Pflege domänenspezifischer Terminologien unterstützt, auf deren Grundlage Anbieter bzw. Verwender die fachliche Semantik von Komponenten bzw. Anforderungen beschreiben. Abschnitt 3 beschreibt den in KOSOBAR verfolgten Ansatz zum Aufbau solcher Terminologien und geht kurz auf den *Terminologie-Manager* ein. Zur Verwaltung von Komponentenbeschreibungen durch Anbieter dient der *CDL Component Manager*, der in der Beschreibungssprache *Component Description Language (CDL)* verfasste Komponentenbeschreibungen in einem UML-Repository ablegt. CDL und der *CDL Component Manager* werden in Abschnitt 4 vorgestellt. Die Suche nach Komponenten auf Komponentenmärkten wird durch Broker unterstützt, die auf in Form von Geschäftsprozessmodellen spezifizierte fachliche Anforderungen zurückgreifen. Der für die Komponentensuche verfolgte Ansatz wird in Abschnitt 5 beschrieben. Abschließend folgt eine Zusammenfassung in Abschnitt 6.

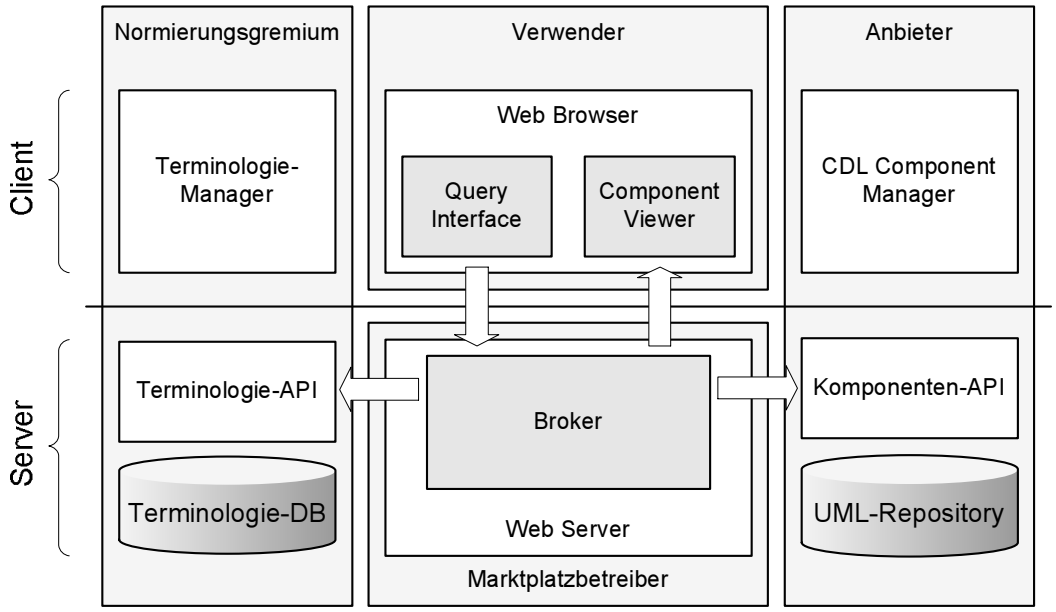

Abbildung 2: Architektur zur Unterstützung der komponentenbasierten Softwareentwicklung

3 Terminologische Semantik von Komponenten

3.1 Sprachliche Spezifikation der Semantik von Komponenten

In der Informatik wird die Semantik syntaktischer Konstrukte verbreitet formal (z. B. mit Hilfe der Prädikatenlogik) oder semi-formal (z. B. auf Basis der UML) definiert. So wird beispielsweise vorgeschlagen, die Semantik von Methoden durch prädikatenlogische Zusicherungen (Vor- und Nachbedingungen sowie Invarianten) zu spezifizieren (vgl. [Hoa69, Mey92, LW94]). Zwar sind derartige formale Spezifikationen präzise, sie weisen jedoch den Nachteil auf, nicht oder nur in eingeschränktem Maße für Fachexperten verständlich zu sein.

Alternativ zur formalen Spezifikation der Semantik von Methoden lässt sich die Bedeutung einer Methode im objektorientierten Paradigma auch als natürlichsprachliche Aussage verstehen, die bei jedem Aufruf der Methode geäußert wird. Der Bezeichner der Methode entspricht dabei einer *Aktion*, die durch ein *Prädikat* und ein *direktes Objekt* spezifiziert wird. Ihre Eingabe- und Ausgabeparameter werden durch ein entsprechende Anzahl *indirekter Objekte* in der Aussage repräsentiert [Tes02]. Als Akteur einer Methodenausführung betrachten wir die Bezeichnung der Benutzerrolle, die für die Ausführung der Methode erforderlich ist, und ordnen ihr die Stellung des *Subjekts* in der Aussage zu [JT02]. Dieses einfache syntaktische Muster lässt sich auf beliebige objektorientierte Methodendeklarationen anwenden. Das folgende Beispiel zeigt eine Methodendeklaration in OMG IDL-Notation:

```
interface SalesDepartment {
      Offer createOffer(in Request request);
}
```

Dabei wird in diesem Beispiel der Begriff „Offer“ (bzw. das deutsche Pendant „Angebot“) zweifach verwendet: zum einen im Methodennamen (dem Prädikat) und zum anderen als Bezeichner des Typs eines erzeugten Objekts (dem direkten Objekt). Damit wird explizit zwischen einer intensionalen Sicht (der Methodenname bezieht sich auf das abstrakte Konzept, das durch

den Begriff bezeichnet wird) und einer extensionalen Sicht (das Objekt bezieht sich auf eine Instanz des Konzepts) auf das „Angebot“ unterschieden. Wenn wir von der intensionalen Verwendung absehen, können wir den Aufruf der Methode `createOffer()` als die Äußerung „Ein Vertriebsmitarbeiter erzeugt ein Angebot mit einer Anfrage“ verstehen. Dabei geht die geforderte Benutzerrolle des Vertriebsmitarbeiters allerdings nicht aus der IDL-Spezifikation hervor. Außerdem haben wir die Lesbarkeit durch unbestimmte Artikel und die Präposition „mit“ erhöht.

3.2 Normsprachliche Spezifikation fachlicher Semantik

Natürliche Sprachen haben gegenüber formalen Sprachen den Vorteil, auch für Fachexperten verständlich zu sein. Sie weisen allerdings Schwächen wie Mehrdeutigkeiten, unzureichende Präzision und Missverständlichkeit auf, die auch als *Sprachdefekte* bezeichnet werden. *Kontrollierte Sprachen* wie z. B. *Attempto Controlled English (ACE)* [FS96] stellen einen Ansatz zur Behebung dieser Defekte natürlicher Sprachen dar. Sie bestehen aus einer vereinfachten Grammatik, die durch spezielle Muster für die Konstruktion von Sätzen (sogenannte *Satzbaupläne*) gegeben ist, und einem kontrollierten, d. h. vordefinierten und überwachten Vokabular, das für die Konstruktion von Sätzen zur Verfügung steht.

Normsprachen können als eine besondere Form kontrollierter Sprachen angesehen werden. Sie haben ihren Ursprung in der konstruktiven Wissenschaftstheorie und werden durch methodische Rekonstruktion der in einem Anwendungsbereich gesprochenen natürlichen (Fach-)Sprache entwickelt [Ort97]. Dabei bezeichnet der Begriff der Rekonstruktion die Klärung und eindeutige Definition der Bedeutung von Wörtern, ihrer Beziehungen untereinander (z. B. mereologische Strukturen, Abstraktionsbeziehungen) sowie von Regeln für ihren Gebrauch. In diesem Zusammenhang ist auch der Grundstock der Logik (Bindewörter „und“, „oder“) zu errichten ebenso wie die z. B. die Bedeutung der Abstraktion zu klären [Büt95].

Abhängig von der gewählten Einteilung in Wortarten gestatten Normsprachen grundsätzlich die Definition verschiedenster Satzbaupläne für die Konstruktion von Sätzen. So definiert *TAOS (Terminologiebasierter Ansatz für die objektorientierte Spezifikation)* [Sch97], ein Rahmenwerk für die Entwicklung objektorientierter Spezifikationen, eine umfassende Menge von Satzbauplänen für die Spezifikation objektorientierter Softwaresysteme. In unserer Arbeit zur fachlich ausgerichteten Beschreibung von Komponenten nutzen wir jedoch lediglich den Satzbauplan

$$[\ \mathtt{N}\ |\ \pi\ |\ \mathtt{P}\ |\ \mathtt{O}_1\ |\ ...|\ \mathtt{O}_n] \tag{1}$$

zur Beschreibung der Semantik von Diensten. Dabei bezeichnet `N` das Subjekt des Satzes (den sogenannten *Nominator*), π die Kopula „tut“, `P` das Prädikat, $\mathtt{O}_1$ das direkte Objekt und $\mathtt{O}_i(\mathtt{i} > 1)$ evtl. vorhandene indirekte Objekte. Das Prädikat `P` wird immer in seiner Infinitivform gebraucht. Die Verwendung von Adverbien und Adjektiven ist in diesem einfachen Satzbauplan nicht vorgesehen. Ein Beispiel für einen Satz, der sich mit diesem Satzbauplan konstruieren lässt, ist `Angestellter` π `hinzufügen Auftragsposition zu-Auftrag` (lies „[ein] Angestellter tut hinzufügen [eine] Auftragsposition zu [einem] Auftrag“). In diesem Beispiel wird das indirekte Objekt „Auftrag“ um die Präposition „zu“ ergänzt.

Die Fähigkeit, Sätze zu bilden, erfordert ein gewisses Vokabular (*Material*), das genutzt werden kann, um einem Satzbauplan (dem formalen Rahmen) „Leben einzuhauchen“. Normsprachen greifen auf ein kontrolliertes Vokabular zurück, dessen Worte wie oben beschrieben methodisch rekonstruiert wurden. Die Bedeutung einzelner Wörter ist dabei mittels expliziter

Definitionen wie z. B. „ein Auftrag ist eine rechtliche Vereinbarung zwischen einem Abnehmer und einem Anbieter einer Leistung . . . “ festgelegt, während semantische Beziehungen zwischen Wörtern auf normsprachliche Aussagen reduziert werden können. So gestattet der Satzbauplan

$$[\ N_1\ |\ \epsilon\ |\ N_2\ ] \tag{2}$$

auszudrücken, dass ein Kunde ein Geschäftspartner ist („`Kunde` ϵ `Geschäftspartner`“) oder dass Müller ein Kunde ist („`Müller` ϵ `Kunde`“). Während die erste Aussage als *allgemeine Aussage* bezeichnet wird, die für die Entwicklung eines Schemas herangezogen werden kann, stellt die zweite Aussage eine *singuläre Aussage* dar, die als Beispiel einer zulässigen Ausprägung eines Schemas dienen kann [Ort97]. ϵ repräsentiert dabei die Kopula „ist ein“, die Generalisierungsbeziehungen zwischen Konzepten beschreibt. Ganzes-Teile-Beziehungen werden durch den Satzbauplan

$$[\ N_1\ |\ \nu\ |\ N_2\ ] \tag{3}$$

formuliert, wobei das Symbol ν für die Kopula „hat“ steht. Als Beispiel mag die Aussage „`Auftrag` ν `Auftragsposition`“ dienen.

Die Satzbaupläne 2 und 3 ermöglichen die Rekonstruktion der statischen Struktur eines Anwendungsbereichs, d. h. die Definition der relevanten Konzepte eines Anwendungsbereiches sowie deren semantische Beziehungen untereinander. Satzbauplan 1 gestattet die Beschreibung des Verhaltens eines Systems innerhalb dieses Anwendungsbereichs, d. h. die beobachtbaren Interaktionen zwischen Instanzen der Konzepte [Ort97].

3.3 Eine Terminologie für die fachliche Beschreibung von Komponenten

In den vergangenen Jahren wird zur Bezeichnung semantischer Modelle von Themenbereichen vielfach der Begriff der *Ontologie* herangezogen. Ursprünglich aus der Philosophie stammend bezeichnet der Begriff „Ontologie“ die *Lehre vom Sein*. In der Informatik wird er jedoch uneinheitlich unter zwei unterschiedlichen Bedeutungen verwendet. Zum einen bezeichnet er verbreitet ein Vokabular für die Repräsentation von Wissen und die dieser Repräsentation zugrunde liegende Konzeptualisierung, zum anderen bezieht er sich auf Wissen über einen Anwendungsbereich selbst [CJB99]. In unserer Arbeit zur Terminologieverwaltung berücksichtigen wir beide Bedeutungen:

- Auf der *Konzeptualisierungsebene* spezifizieren wir ein allgemeines, domänenunabhängiges Vokabular für die Repräsentation von (normsprachlichen) Aussagen.
- Basierend auf der Konzeptualisierungsebene definiert die *Wissensebene* Wissen über einen betrachteten Anwendungsbereich.

3.4 Konzeptualisierungsebene

Die Konzeptualisierungsebene leitet sich aus den drei in Abschnitt 3.2 vorgestellten Satzbauplänen ab. Abbildung 3 zeigt ein UML Klassendiagramm, das die konzeptuelle Struktur unserer Terminologie spezifiziert. Bezüglich der betrachteten Wortarten unterscheiden wir ausgehend von einer generellen Klasse abstrakter *Wörter* zunächst *Partikel* (Strukturwörter) und *Prädikatoren* (Fachwörter, die einem Gegenstand zu- oder abgesprochen werden). Bei den Partikeln

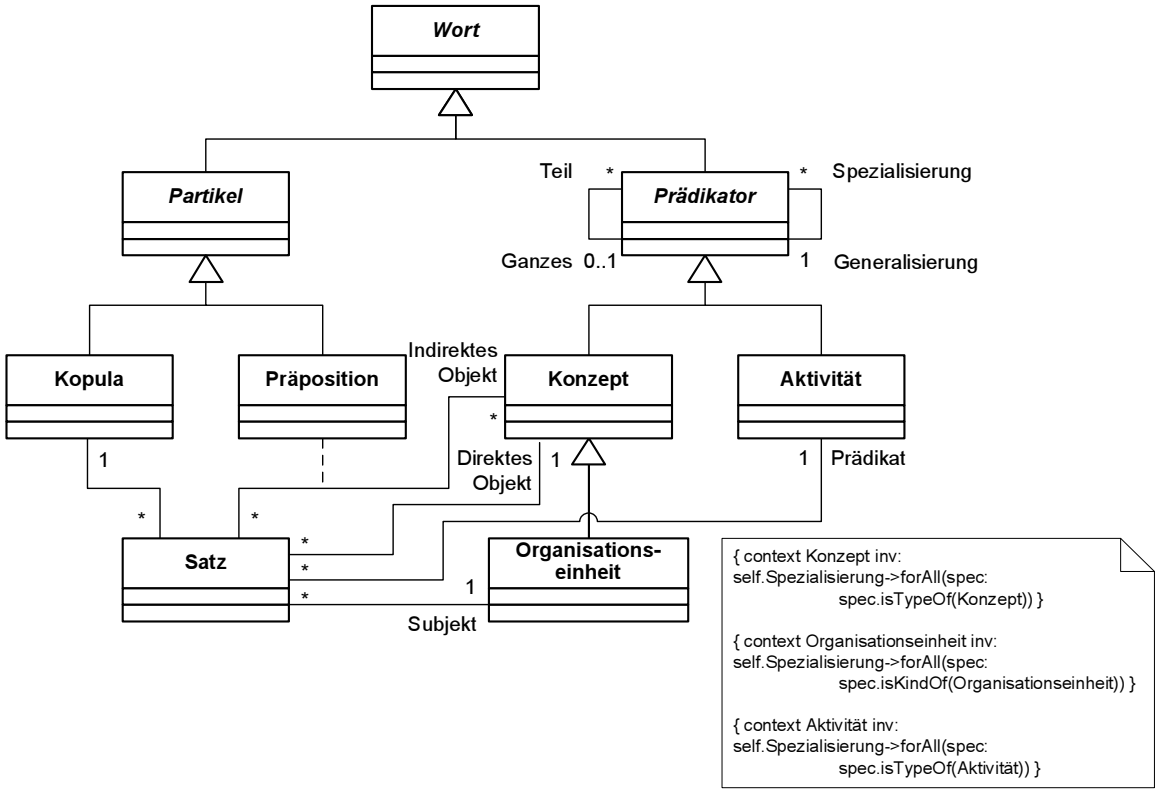

Abbildung 3: Konzeptualisierung der Terminologie

betrachten wir *Kopulae* wie z. B. π („tut“) und *Präpositionen* wie z. B. „zu“ oder „mit“. Auf Seiten der abstrakten Klasse der Prädikatoren unterscheiden wir zwischen *Aktivitäten* wie z. B. „erzeugen“ oder „modifizieren“ und *Konzepten* wie z. B. „Auftrag“, auf die Aktivitäten einwirken können. *Organisationseinheiten* wie z. B. „Angestellter“ sind spezielle Arten von Konzepten; sie können nicht nur Gegenstand der Ausführung von Aktivitäten sein, sondern diese auch selbst ausführen. Prädikatoren können in einer Generalisierungshierarchie angeordnet werden, wobei allerdings eine Instanz eines Prädikators nur durch Instanzen derselben Klasse oder einer Unterklasse spezialisiert werden kann (vgl. hierzu auch die OCL-Ausdrücke in Abbildung 3). Dies bedeutet insbesondere, dass eine Organisationseinheit nicht durch ein Konzept spezialisiert werden darf. Neben Generalisierungsbeziehungen gestattet die Konzeptualisierungsebene auch die Definition von Ganzes-Teile-Beziehungen zwischen Prädikatoren.

Sätze sind syntaktische Strukturen für die Repräsentation von Aussagen. Sie entsprechen den syntaktischen Konventionen, die durch den in Abschnitt 3.2 genannten Satzbauplan 1 definiert werden. Folglich besteht ein Satz aus einer Organisationseinheit in der Subjektstellung, einer Kopula und einer Aktivität in der Stellung des Prädikats sowie Konzepten als direkte und indirekte Objekte. Indirekte Objekte sind mit dem Prädikat über Präpositionen verbunden.

3.5 Wissensebene

Die Wissensebene definiert Taxonomien von Konzepten und Aktivitäten, die für die Konstruktion von Sätzen innerhalb eines Anwendungsbereichs genutzt werden können. Diese Taxonomien definieren z. B., dass „Auftrag“ eine „Transaktion“ ist und dass „Produktionsauftrag“ und „Kundenauftrag“ Spezialisierungen ($\leq$) von „Auftrag“ sind. Auf ähnliche Weise können „er-

zeugen“, „ändern“ und „löschen“ als Spezialisierungen der Aktivität „modifizieren“ definiert werden. Neben diesen Taxonomien definiert die Wissensebene auch semantische Restriktionen bzgl. der Kombination von Konzepten und Aktivitäten in Sätzen. Diese Restriktionen verbieten z. B. die Konstruktion von Sätzen wie „Kunde (tut) erzeugen Angestellter“. Sie sind durch sogenannte *Kernsätze*, die Beispiele „korrekter“ Sätze geben, sowie eine Regel für die Ableitung neuer Sätze aus diesen Kernsätzen definiert: Ein neuer Satz darf nur dann abgeleitet werden, wenn er einen Kernsatz spezialisiert und gleichzeitig nicht gegen die durch speziellere Kernsätze spezifizierten Restriktionen verstößt. Die allgemeine Idee, die der Spezialisierungsbeziehung zwischen Sätzen zugrunde liegt, ist zu verlangen, dass alle Satzbestandteile (Subjekt, Prädikat sowie direkte und indirekte Objekte) des generelleren Satzes durch entsprechende Satzbestandteile des spezielleren Satzes spezialisiert werden. In der praktischen Anwendung dieses Ansatzes schränken wir die Spezialisierungsbeziehung jedoch auf die Betrachtung von Subjekt, Prädikat und direktem Objekt ein:

Definition 1 (Spezialisierungsbeziehung) *Seien $S_i = (S_{S_i}, P_{S_i}, O_{S_i})$, $i \in \{1, 2\}$ Sätze, wobei S_{S_i} das Subjekt, P_{S_i} das Prädikat und O_{S_i} das direkte Objekt des Satzes S_i repräsentieren. Dann gilt:*

$$S_1 \text{ ist eine Spezialisierung von } S_2 \ (S_1 \leq S_2)$$
$$\iff \quad S_{S_1} \leq S_{S_2} \wedge P_{S_1} \leq P_{S_2} \wedge O_{S_1} \leq O_{S_2},$$

wobei sich die Spezialisierungsbeziehungen zwischen Subjekten, Prädikaten und direkten Objekten aus der Aktivitäts- bzw. der Konzepttaxonomie ergeben.

Damit ein abgeleiteter Satz gültig ist, ist es nicht ausreichend, einen Kernsatz zu spezialisieren. Wir verlangen zusätzlich, dass der abgeleitete Satz auch die Resktriktionen speziellerer Kernsätze respektiert:

Definition 2 (Gültige Spezialisierung) *Sei CS die Menge der Kernsätze und $S = (S_S, P_S, O_S)$ ein Satz. Dann gilt:*

$$S \text{ ist ein gültiger Satz bzgl. } CS$$
$$\begin{aligned} \iff \quad & \exists S_C \in CS : \ S \leq S_C \\ \wedge \quad & \forall S'_C \in CS, \ S'_C \leq S_C, S'_C \neq S : \\ & \quad \neg(S_S \leq S_{S'_C} \vee P_S \leq P_{S'_C} \vee O_S \leq O_{S'_C}), \end{aligned}$$

d. h. der abgeleitete Satz ist nicht teilweise spezieller als ein Kernsatz S'_C, der spezieller als der Kernsatz S_C ist.

Ein neuer Satz stellt eine gültige Spezialisierung eines Kernsatzes dar, genau dann wenn es keinen spezielleren Kernsatz gibt, der die Benutzung des Subjekts, Prädikats oder direkten Objekts weiter einschränkt. Ein Beispiel soll diese Gültigkeitsregel erläutern. Nehmen wir an, dass der Satz „Organisationseinheit (tut) tun etwas“ der einzige Kernsatz ist. Vorausgesetzt, dass die Konzept- und Aktivitätstaxonomien entsprechend definiert sind, ließen sich dann die Sätze „Angestellter (tut) planen Produktionsauftrag“ und „Angestellter (tut) planen Rechnung“ ableiten. Um auszudrücken, dass wir nur die Planung von Aufträgen, nicht aber von Rechnungen gestattet wollen, können wir den obigen Kernsatz durch einen weiteren Kernsatz „Organisationseinheit (tut) planen Auftrag“ spezialisieren. Nun stellt „Angestellter (tut) planen Rechnung“ keine gültige Spezialisierung des ersten Kernsatzes dar, da $\neg((Organisationseinheit \leq$

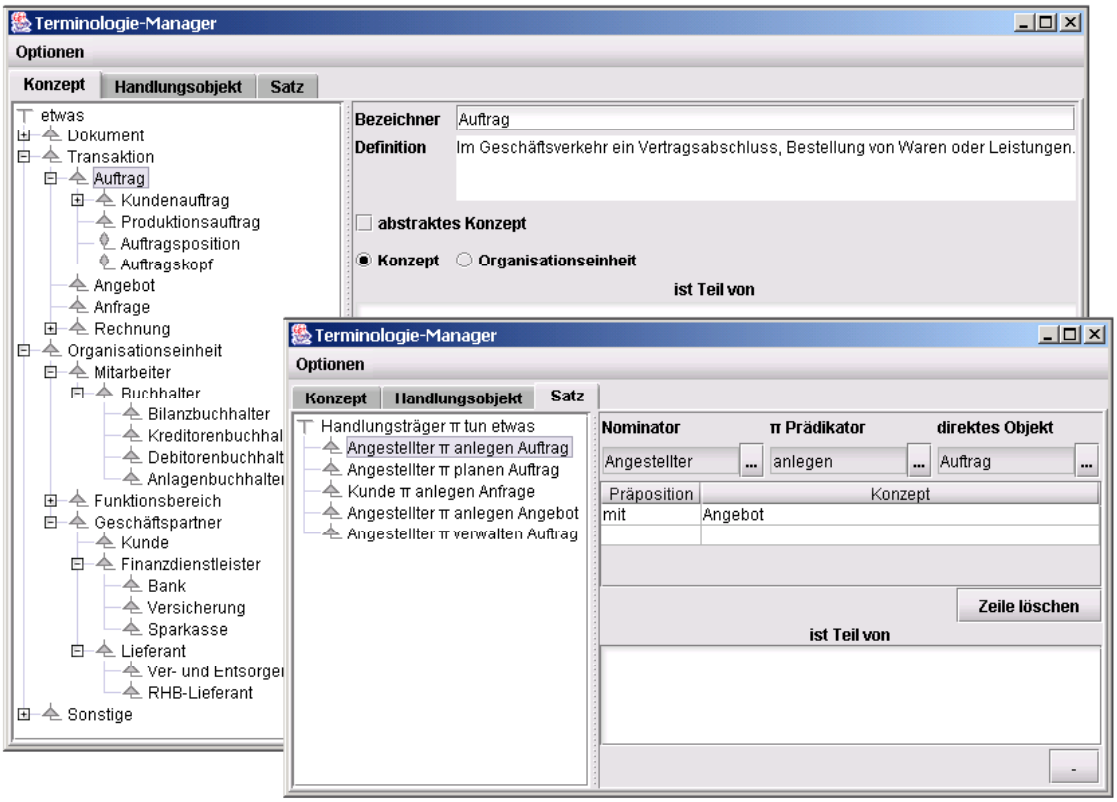

Abbildung 4: *Terminologie-Manager* zur Verwaltung von Fachterminologien

$Organisationseinheit) \vee (planen \leq planen) \vee (Rechnung \leq Auftrag)) = false$ (vgl. den zweiten Teil von Definition 2). Es ist offensichtlich, dass dieser Satz ebensowenig eine gültige Spezialisierung des zweiten Kernsatzes darstellt.

3.6 Werkzeugunterstützung

Zur Unterstützung der Aufgaben eines Normierungsgremiums haben wir mit dem *Terminologie-Manager* ein Werkzeug für die Verwaltung von Prädikatoren einerseits sowie von Kernsätzen andererseits prototypisch entwickelt (siehe Abbildung 4). Daneben ist ein Algorithmus zur Berechnung gültiger Spezialisierungen von Kernsätzen gemäß Definition 2 entwickelt und in Form eines Dialogs zur Ableitung von Sätzen implementiert worden. Dieser Dialog unterstützt die Aufgaben von Anbietern bzw. Verwendern, die in ihren Komponentenbeschreibungen bzw. Geschäftsprozessmodellen Bezug auf eine Fachterminologie nehmen.

4 Komponentenbeschreibungen mit CDL

4.1 Das CDL-Komponentenmodell

Voraussetzung für die Definition einer Komponentenbeschreibungssprache ist die Existenz eines (abstrakten) Komponentenmodells. Das Komponentenmodell der CDL stellt eine einheitliche Abstraktion von den Komponentenmodellen der Komponententechnologien COM, EJB und

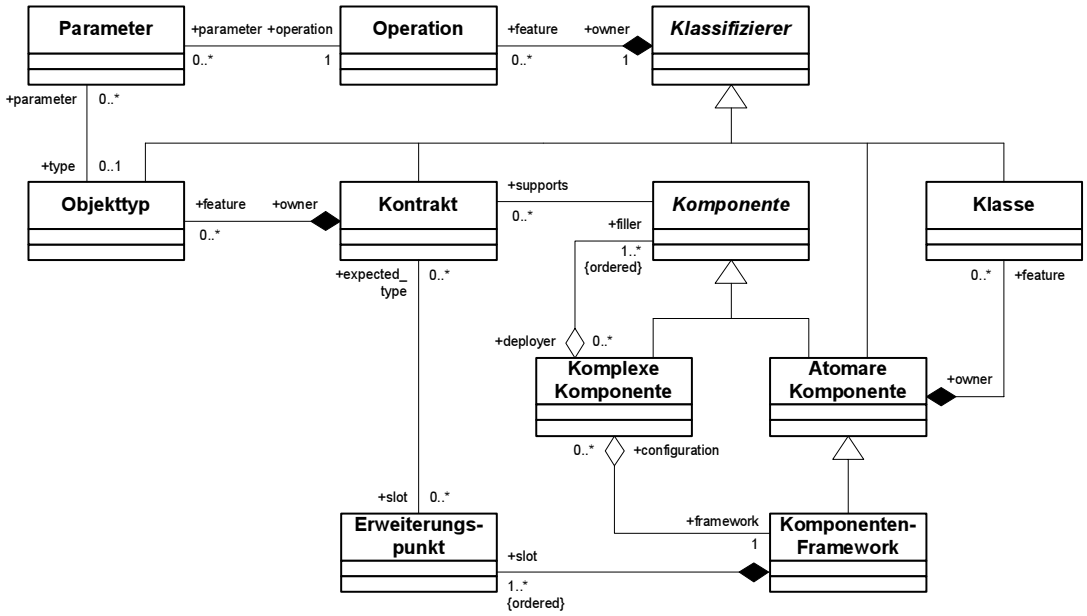

Abbildung 5: CDL-Komponentenmodell

CCM dar (vgl. auch [TR01]). Es ist in Abbildung 5 in Form eines Klassendiagramms graphisch dargestellt.

Zentrales Element des CDL-Komponentenmodells ist der abstrakte Begriff der Komponente, von der sich die spezielleren Varianten atomare Komponente und komplexe Komponente ableiten. Atomare Komponenten können eine *direkte Schnittstelle* und – über die möglicherweise von ihr realisierten Klassen von Geschäftsobjekten – *indirekte Schnittstellen* anbieten, da sowohl atomare Komponenten als auch Klassen Klassifizierer darstellen und somit eine Menge von Operationen als Schnittstelle exportieren. Instanzen einer Klasse (die Geschäftsobjekte) werden über die direkte Schnittstelle einer atomaren Komponente erzeugt und lassen sich über ihre eigene Schnittstelle (die Teil der indirekten Schnittstelle der atomaren Komponente ist) ansprechen. Komponenten-Frameworks sind spezielle atomare Komponenten, die sich dadurch auszeichnen, dass sie Erweiterungspunkte („hot spots“) definieren, über die sie im Sinne einer Komposition mit anderen Komponenten verbunden werden können (atomare Komponenten, die keine Komponenten-Frameworks sind, werden – falls eine explizite Unterscheidung erforderlich ist – auch als *geschlossene atomare Komponenten* bezeichnet). Komponenten-Frameworks stellen damit die Basis für die Erstellung komplexer Komponenten, d. h. durch Komposition mehrerer Komponenten hervorgegangene Komponenten, dar. Eine komplexe Komponente besteht aus einem Komponenten-Framework und einer Menge von Komponenten, mit denen die Erweiterungspunkte des Komponenten-Frameworks belegt werden. Anders als eine atomare Komponente definiert eine komplexe Komponente keine eigene (direkte und/oder indirekte) Schnittstelle, sondern exportiert die Schnittstellen des verwendeten Komponenten-Frameworks.

Die Belegung der Erweiterungspunkte eines Komponenten-Frameworks wird durch Typisierung mit Kontrakten kontrolliert. Kontrakte repräsentieren ein zu den Komponenten auf Seiten der Implementierung analoges Konzept auf der Vertragsebene: Ähnlich wie atomare Komponenten (und indirekt auch komplexe Komponenten) eine direkte Schnittstelle sowie (über Klassen) indirekte Schnittstellen definieren, spezifizieren Kontrakte neben einer direkten Schnittstelle mittels sogenannter Objekttypen indirekte Schnittstellen. Objekttypen repräsentieren da-

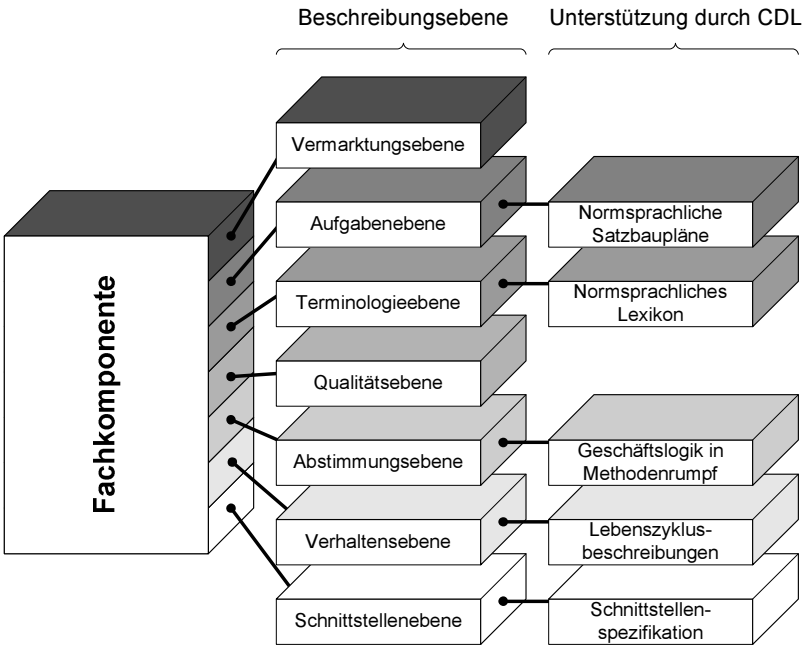

Abbildung 6: Berücksichtigung der Beschreibungsebenen nach [ABC+02] durch CDL

bei Typen von Geschäftsobjekten, die durch Klassen einer atomaren Komponente realisiert und neben Basisdatentypen wie `int`, `float` und `String` als Typen der Parameter von Operationen verwendet werden können.[1] Eine Komponente stellt eine gültige Belegung eines Erweiterungspunktes dar, wenn sie den vom Erweiterungspunkt vorgegebenen Kontrakt unterstützt, d. h. wenn sie mindestens die im Kontrakt geforderte direkte Schnittstelle anbietet und darüber hinaus noch Klassen realisiert, die die Objekttypen des Kontrakts implementieren. Dabei kann eine Klasse mehrere Objekttypen gleichzeitig implementieren.

4.2 CDL – Component Description Language

Die Entwicklung der CDL wurde durch die Anforderung motiviert, fachlich orientierte Beschreibungen bei hinreichend formaler Fundierung zu ermöglichen. Eine zentrale Annahme unserer Arbeit zur Beschreibung und Suche von Komponenten besteht darin, dass Komponenten bzw. ihre Geschäftsobjekte ihre Dienste nicht gleichförmig zur Verfügung stellen (*non-uniform service availability* [Nie95]), sondern oft eine spezifische Geschäftslogik implementieren. So muss ein Auftrag z. B. freigegeben werden, bevor er ausgeführt werden kann. Aus diesem Grund bestand ein Anspruch an die CDL darin, neben strukturellen Merkmalen von Komponenten auch deren Verhalten beschreiben zu können. Im Memorandum zur vereinheitlichen Spezifikation von Fachkomponenten [ABC+02] wurden sieben Ebenen für die Beschreibung von Fachkomponenten identifiziert. CDL berücksichtigt die fünf in Abbildung 6 dargestellten Beschreibungsebenen. Weitere, durchaus relevante Informationen wie z. B. zu Hersteller, erforderlicher Plattform, Lizenzierung, Quality of Service oder Performance, die auf der Vermarktungsebene bzw. Qualitätsebene einzuordnen sind, werden durch CDL nicht näher betrachtet.

[1] Aus praktischen Erwägungen lassen wir in CDL neben Objekttypen auch den mengenwertigen Typ `Set<·>`, der in Anlehnung an C++-Templates mit einem Elementtyp parametrisiert werden kann, und Klassen für die Typisierung von Parametern bei der Spezifikation von Komponenten zu.

Wie bereits aus dem CDL-Komponentenmodell hervorgeht, unterscheidet CDL zwischen Komponenten- und Kontraktbeschreibungen, die wie folgt charakterisiert werden können. Komponentenbeschreibungen definieren eine obere Grenze der Menge von Interaktionssequenzen, an der die Komponente (evtl. mittels ihrer Geschäftsobjekte) teilnehmen kann. Diese Semantik ist aus Sicherheitsbetrachtungen heraus interessant: Eine Komponente geht nicht mehr Interaktionen ein, als durch ihre Beschreibung zugesichert wird. Demgegenüber spezifizieren Kontraktbeschreibungen eine untere Grenze der Menge erwarteter Interaktionssequenzen und damit die erwartete Lebendigkeit einer Komponente. Dieses Semantik stellt sicher, dass eine Komponente, die einen bestimmten Kontrakt unterstützt, auch mindestens die erforderlichen Dienste anbietet. Da CDL jedoch explizit Unterspezifikationen von Komponenten und Kontrakten unterstützt (z. B. durch die Möglichkeit, nicht-deterministische Entscheidungen zu spezifizieren), darf die vorangegangene Charakterisierung von Komponenten- und Kontraktbeschreibungen nicht strikt (z. B. im Sinne einer Garantie des Komponentenherstellers) interpretiert werden. Sie stellt stattdessen lediglich eine Art Richtschnur dar, an der sich „gute“ Beschreibungen ausrichten.

Komponentenbeschreibungen werden durch das Schlüsselwort `component` eingeleitet. Sie können Beschreibungen einer beliebigen Anzahl von Klassen enthalten, die die durch die Komponente implementierten Typen von Geschäftsobjekten spezifizieren und durch das Schlüsselwort `class` gekennzeichnet sind (siehe Listing 1). Die direkte Schnittstelle einer Komponente besteht aus den Operationen, die die Komponente direkt, d. h. nicht über ihre Geschäftsobjekte, anbietet, während die indirekte Schnittstelle einer Komponente aus ihren Klassen und den von ihnen angebotenen Operationen besteht. Parameter von Operationen sind gerichtet und mit Objekttypen, Klassen, Basisdatentypen wie `int`, `float` und `String` oder dem mengenwertigen Typ `Set<·>` typisiert. Die Richtung eines Parameters wird durch die Schlüsselworte `in` für Eingabeparameter, `out` für Ausgabeparameter und `inout` für Ein- und Ausgabeparameter gekennzeichnet. Die Beschreibung von Komponenten-Frameworks und komplexen Komponenten wird durch spezielle Schlüsselworte für Erweiterungspunkte und die Komposition von Komponenten ermöglicht. Die CDL-Syntax für die Beschreibung von Kontrakten unterscheidet sich kaum von der für die Beschreibung geschlossener atomarer Komponenten: lediglich die für die Beschreibung von Komponenten verwendeten Schlüsselwörter `component` und `class` müssen bei der Beschreibung von Kontrakten durch die Schlüsselwörter `contract` und `objecttype` ersetzt werden.

Neben diesen syntaktischen Elementen zur Beschreibung struktureller Eigenschaften von Komponenten (und Kontrakten) bietet CDL auch Beschreibungselemente für die Spezifikation des Verhaltens einer Komponente und ihrer Klassen. Zu diesem Zweck greifen wir auf den Grundgedanken zurück, der der Kommunikation im π-Kalkül zugrunde liegt. Das Senden eines Operationsaufrufs wird durch den Bezeichner der Operation, gefolgt von einem „!“ gekennzeichnet, die Bereitschaft zum Empfang eines solchen Aufrufs durch ein dem Bezeichner folgendes „?“. Eine Operation wird ausgeführt, wenn das Senden eines Operationsaufrufs durch eine Komponente mit der Empfangsbereitschaft einer entsprechenden Operation einer Zielkomponente zusammenfällt. Dieser dualen Betrachtung des Aufrufs von Operationen folgend basiert die Beschreibung des Verhaltens von Komponenten und Klassen mit der CDL auf der Idee, die Spezifikation des *aktiven Verhaltens*, d. h. die ausgelösten Operationsaufrufe, von der Spezifikation des *passiven Verhaltens*, d. h. der Bereitschaft zum Empfang solcher Aufrufe, zu trennen. Dazu spezifizieren wir das aktive Verhalten in Operationsrümpfen, während das passive Verhalten in speziellen Verhaltensbeschreibungen dokumentiert wird.

Listing 1: Komponente `Vertrieb`

```
component Vertrieb {
   class Auftrag {
      plan() [Vertriebsangestellter terminplanen Auftrag] {
         return; };
      execute() [Vertriebsangestellter ausführen Auftrag] {
         return; };
      updateTerms() [Vertriebsangestellter ändern Auftrag] {
         return; };
      cancel() [Vertriebsangestellter stornieren Auftrag] {
         return; };
       ... /* Beschreibung weiterer Operationen der Klasse Auftrag */
      created()     =   initialized();
      initialized() =   plan?().planned()
                      + cancel?().cancelled();
      planned()     =   execute?().executed()
                      + updateTerms?(terms).initialized()
                      + cancel?().cancelled();
      executed()    =   ();
      cancelled     =   ();
   };
   class Angebot {
      ... /* Beschreibung der Klasse Angebot */
   };
   createOrder(in offer: Vertrieb.Angebot, out order: Vertrieb.Auftrag)
      [Vertriebsangestellter anlegen Auftrag mit-Angebot] {
         new (Vertrieb.Auftrag order);
         return(order);
   };
   createOffer(out offer: Vertrieb.Angebot)
      [Vertriebsangestellter anlegen Angebot] {
         new (Vertrieb.Angebot order);
         return(offer);
   };
   created()     =    initialized();
   initialized() =    createOrder?(offer, order).initialized()
                    + createOffer?(offer).initialized();
};
```

Für die Beschreibung des aktiven Verhaltens im Rumpf einer Operation stellt CDL Konstrukte für den Aufruf von Operationen, die Erzeugung von Referenzen auf neue bzw. bereits existierende Geschäftsobjekte, (nicht-deterministische) Entscheidungen, Wiederholungsblöcke und die Rückgabe von Parameterwerten zur Verfügung. Die Beschreibung des aktiven Verhaltens einer Komponente in Operationsrümpfen hat primär illustrierenden Charakter. Sie wird im praktischen Einsatz oftmals als starke Unterspezifikation des tatsächlichen Verhaltens angegeben werden und eignet sich insbesondere dazu, um verwendete Algorithmen zu dokumentieren und Abhängigkeiten von anderen Komponenten explizit darzustellen.

Die Verhaltensbeschreibungen zur Spezifizierung des passiven Verhaltens einer Komponente bzw. einer Klasse basieren auf *Protokollmaschinen* (*Protocol State Machines*), einer Variante der UML State Machines, die für die Spezifikation der Lebenszyklen von Objekten eingesetzt wird [OMG01]. Zustände einer solchen Protokollmaschine beschreiben dabei sinnvolle Ab-

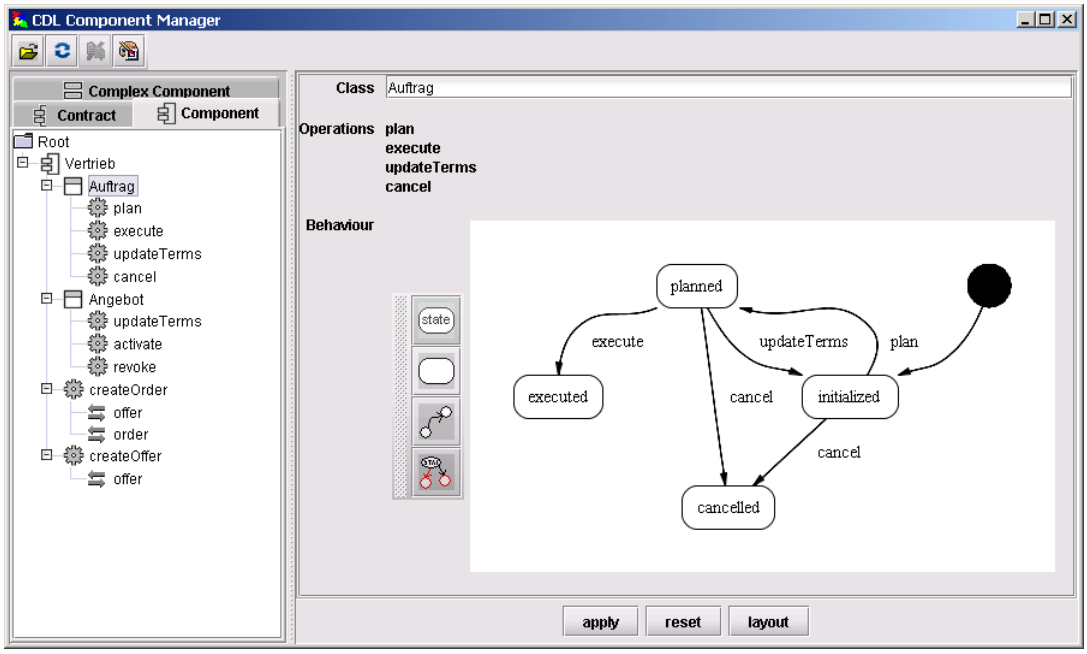

Abbildung 7: *CDL Component Manager* zur Verwaltung von Komponentenbeschreibungen

straktionen der tatsächlichen Zustände, in denen sich eine Komponente oder ein Geschäftsobjekt befinden kann, während Transitionen die Operationsaufrufe darstellen, die eine Komponente oder Geschäftsobjekt von einem Zustand in einen Folgezustand überführen. Beschreibungen des Verhaltens, das in einem Zustand möglich ist, dürfen nur die Bereitschaft zum Empfang von Operationsaufrufen, d. h. das passive Verhalten der Komponente oder des Geschäftsobjekts spezifizieren.

Als illustrierendes Beispiel für Komponentenbeschreibungen in CDL zeigt Listing 1 eine Komponente namens `Vertrieb`, die einfache Dienste zur Verwaltung von Angeboten und Aufträgen anbietet. Die fachliche Semantik dieser Dienste ist gemäß Abschnitt 3 mittels normsprachlicher Sätze definiert (siehe eckige Klammern im Listing). Die Komponente bietet über die beiden Klassen `Auftrag` und `Angebot` zwei indirekte Schnittstellen an. Die Klasse `Auftrag` bietet in ihrer Schnittstelle Dienste zum Einplanen, Ausführen, Ändern und Stornieren des Auftrages an. Die anschließend spezifizierte Verhaltensbeschreibung ist als Protokollmaschine in Abbildung 7 im rechten Teil dargestellt. Die direkte Schnittstelle der Komponente `Vertrieb` beschränkt sich auf das Erzeugen der Geschäftsobjekte. Ihre Verhaltensbeschreibung erlaubt die beliebige Erzeugung von Aufträgen und Angeboten.

4.3 Werkzeugunterstützung

Zur Unterstützung der Aufgaben eines Marktplatzbetreibers oder Komponentenherstellers haben wir mit dem *CDL Component Manager* ein Werkzeug für die Verwaltung von Komponentenbeschreibungen prototypisch entwickelt (siehe Abbildung 7). Der *CDL Component Manager* gestattet die CDL-konforme Beschreibung von Komponenten und Kontrakten und stellt Verhaltensbescheibungen graphisch als editierbare UML State Machines dar.

Auf technischer Ebene setzen wir für die Verwaltung von CDL-Komponentenbeschreibungen ein MOF-basiertes Metadaten-Repository [HT01] ein, auf das wir über ein speziel-

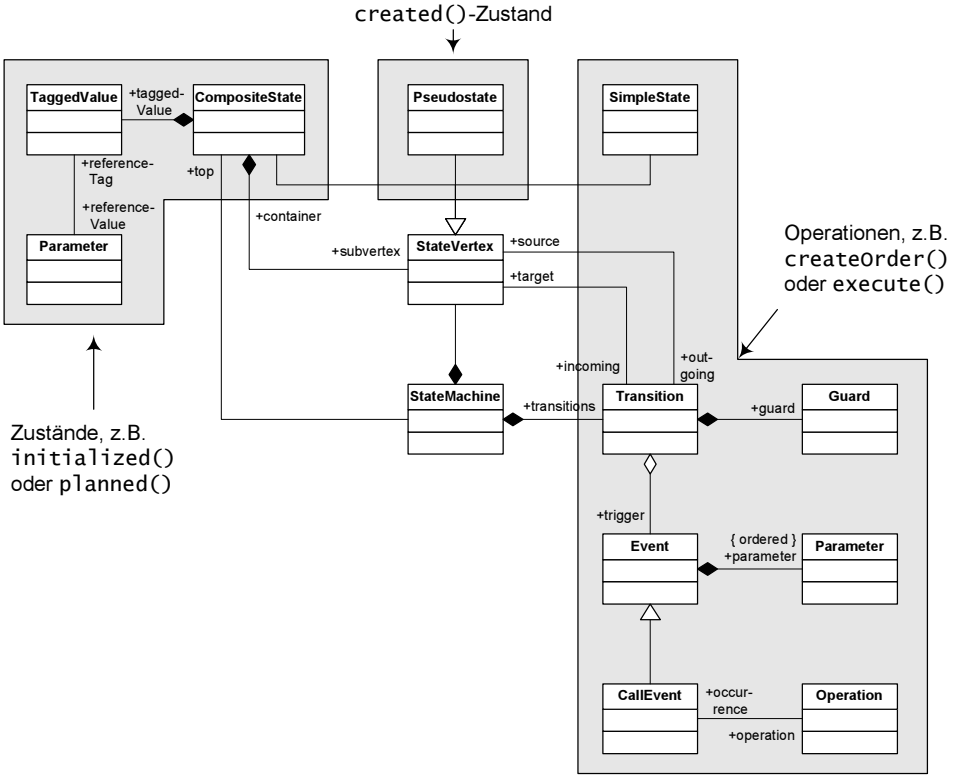

Abbildung 8: Abbildung der CDL-Verhaltensbeschreibungen auf die UML

les Komponenten-API zugreifen (vgl. auch Abbildung 2). Zu diesem Zweck haben wir die Sprachelemente der CDL auf das Metamodell der UML abgebildet, das die konzeptionelle Struktur unseres Komponenten-Repositories definiert. Während die Abbildung der strukturellen CDL-Konstrukte wie z. B. Komponenten, Klassen und Operationen durch Einsatz der UML-Erweiterungsmechanismen *Tagged Values* und *Stereotypen* relativ direkt vorgenommen werden kann, gestaltet sich die Abbildung der verhaltensbeschreibenden Elemente aufwändiger. Wir legen CDL-Verhaltensbeschreibungen als *StateMachines* in dem UML-Repository ab und repräsentieren die Zustände einer CDL-Verhaltensbeschreibung als *CompositeStates* einer solchen *StateMachine*. Die Verknüpfung von CDL-Zuständen wird dann über *Transitionen* zwischen *CompositeStates* modelliert. Sequenzen von Operationen, Alternativen und bedingte Ausführungen können durch Transitionen zwischen internen Zuständen innerhalb eines *CompositeState* abgebildet werden. Für die Repräsentation solcher interner Zustände verwenden wir *SimpleStates*, die einem *CompositeState* untergeordnet werden können. Die in Abbildung 8 skizzierte Abbildung von CDL-Verhaltensbeschreibungen auf das UML-Metamodell kann automatisiert durch einen Parser für CDL-Beschreibungen vorgenommen werden.

5 Geschäftsprozessorientierte Komponentensuche

5.1 Komponentensuche auf Basis von Geschäftsprozessmodellen

Seit den ersten Erfolgen beim Handel mit Komponenten für graphische Benutzungsoberflächen (z. B. *ActiveX-Controls* oder *JavaBeans*) über das Internet sind eine Reihe elektronischer

Marktplätze entstanden, über die Komponentenhersteller die von ihnen entwickelten System- und Fachkomponenten vertreiben können.[2] Ein wichtiger Erfolgsfaktor solcher Komponentenmärkte im Speziellen und der komponentenbasierten Softwareentwicklung durch Wiederverwendung im Allgemeinen ist die Fähigkeit, das Problem der gezielten Suche nach Komponenten zu lösen. Die Suche nach geeigneten Komponenten wird auf aktuellen Komponentenmärkten lediglich durch relativ simple Suchverfahren wie Stichwortsuche in Volltextbeschreibungen von Komponenten, hierarchische Klassifikation von Komponenten oder Feldsuche unterstützt, die kaum entscheidende Charakeristika von Software berücksichtigen (vgl. z. B. [Com02, Fla02, Sun02, Sof02, SAP02, Nom02]). Als Vorteil dieser Suchverfahren darf gelten, dass sie für einen Interessenten einfach zu verstehen und benutzen sind. Ihr Nachteil allerdings ist, dass ein Interessent nur sehr eingeschränkte Möglichkeiten hat, seine (fachlichen) Anforderungen an eine Komponente in den Suchvorgang einzubringen. Dem Interessenten fällt daher die Aufgabe zu, aus einer u. U. umfangreichen Menge gefundener Komponenten diejenige auszuwählen, die seine Anforderungen am besten erfüllt. Da Beschreibungen verfügbarer Komponenten in der Realität aktueller Komponentenmärkte jedoch insbesondere im Hinblick auf die angebotene Funktionalität zumeist uneinheitlich, unvollständig und missverständlich sind, wird der „manuelle“ Abgleich der Anforderungen mit den Leistungen gefundener Komponenten bzw. der direkte Vergleich gefundener Komponenten erheblich erschwert, wenn nicht gar faktisch verhindert. In Anbetracht des erwarteten Zuwachses an angebotenen Komponenten lässt sich zusammenfassend feststellen, dass die Kombination unpräziser Suchverfahren mit nur eingeschränkt aussagekräftigen Komponentenbeschreibungen (zukünftig) zu erheblichen Problemen führen bzw. die Entwicklung von Komponentenmärkten behindern wird.

Bei der Betrachtung der Spezifikation fachlicher Anforderungen einerseits und der Charakteristika von (Fach-)Komponenten andererseits lässt sich folgendes feststellen:

1. Seit den 90er Jahren hat sich bei der Realisierung von Softwareprojekten ein Wandel von einer bis dato primär daten- und funktionsorientierten Sicht hin zu einer verstärkt prozessorientierten Ausrichtung vollzogen. Anforderungen werden daher vielfach in Form von Geschäftsprozessmodellen spezifiziert (z. B. mittels ereignisgesteuerter Prozessketten (EPKs) [KNS92] oder Aktivitätsdiagrammen [OMG01]). Die in diesen Modellen erfassten Informationen finden derzeit keine ausreichende Berücksichtigung bei der Komponentensuche und damit bei der komponentenbasierten Softwareentwicklung.

2. Komponenten implementieren häufig eine bestimmte Geschäftslogik, die die Verfügbarkeit von Diensten und damit die Einsetzbarkeit der Komponente einschränkt (vgl. auch Abschnitt 4.2). Ähnlich wie bei der Einführung von Standardsoftware sind infolgedessen die Geschäftsprozesse eines Wiederverwenders ggf. an die eingesetzten Komponenten anzupassen.

Die Notwendigkeit der Anpassung von Geschäftsprozessen an die eingesetzte Software erscheint inakzeptabel, da gerade die Komponentensoftware den Anspruch erhebt, durch ihre Flexibilität und die Vielfalt verfügbarer Komponenten die Anpassung der Software an spezifische Anforderungen zu ermöglichen [Has02]. Eine stärkere Berücksichtigung der in Form von Geschäftsprozessmodellen spezifizierten fachlichen Anforderungen bei der Komponentensuche kann dazu beitragen, dieses Problem zu entschärfen.

[2] Eine Auswahl aktueller Komponentenmärkte findet sich in [Tec02].

In unserer Arbeit zur geschäftsprozessorientierten Komponentensuche konzentrieren wir uns daher auf fachliche Anforderungen, die in Geschäftsprozessmodellen spezifiziert sind. Dieser Ansatz zur Komponentensuche kann aufgrund seiner starken Differenzierung als Ergänzung einfacherer Suchverfahren wie der Klassifikation von Komponenten (z. B. zur Eingrenzung der Anwendungsdomäne) oder der Feldsuche (z. B. zur Wahl der Komponententechnologie) verstanden werden.

5.2 Komponentensuche als Subtyping-Problem

Komponentensuche setzt sich mit der Frage auseinander, ob eine Komponente in einem spezifischem Kontext (wieder-)verwendet werden kann, der zur Zeit ihrer Entwicklung nicht in allen Details bekannt gewesen ist. Im objektorientierten Paradigma ist der Begriff der (Wieder-)Verwendbarkeit eng mit der Frage nach der *Substituierbarkeit* von Klassen bzw. ihren Objekten verknüpft. In beiden Fällen ist für eine gegebene Schnittstelle eine geeignete Implementierung zu finden. Während bei Wiederverwendungsproblemen im Allgemeinen noch keine solche Implementierung vorhanden ist, geht es bei der Frage der Substitutierbarkeit darum, eine vorhandene Implementierung durch eine andere zu ersetzen. Dabei wird verlangt, dass die Ersetzung dieser Implementierung für einen Nutzer der Schnittstelle nicht feststellbar ist. Der Begriff der Substituierbarkeit wird gewöhnlich durch das Konzept des *Subtypings* formal gefasst, d. h. durch die Feststellung von Subtyp-Beziehungen zwischen Klassen bzw. den durch sie implementierten Typ. WEGNER und ZDONIK beschreiben das *Prinzip der Substituierbarkeit* wie folgt:

> »*An instance of a subtype can always be used in any context in which an instance of a supertype was expected.*« [WZ88]

Ein *Typ* kann dabei als Sammlung von Objekten betrachtet werden, denen gewisse gemeinsame, extern beobachtbare Eigenschaften innewohnen [Ame91]. Beim Subtyping wird allgemein die Fragestellung betrachtet, ob ein Typ die Eigenschaften eines anderen Typs aufweist (die er durch weitere, nicht konkurrierende Eigenschaften ergänzen darf). Ist dies der Fall, so wird der erstgenannte Typ als *Subtyp* des zweiten Typs bezeichnet, der dann einen *Supertyp* des ersten Typs darstellt. Die Eigenschaften eines Supertyps können also als Anforderungen betrachtet werden, die der Subtyp durch entsprechende Leistungen erfüllen muss. Eine solche Anforderung besteht z. B. darin, dass ein Subtyp alle Methoden des Supertyps anbietet (bzw. geeignete, z. B. durch ko- und kontravariante Veränderungen der Eingabe- und Ausgabeparametertypen erzeugte Varianten davon [Szy98]). Allerdings berücksichtigt dieser auf die Betrachtung von Signaturen ausgerichtete Begriff des Subtypings nicht das Verhalten eines Typs. So kann ein Typ eine Geschäftslogik definieren, die die zulässigen Sequenzen von Methodenaufrufen und damit die Wiederverwendbarkeit des Typs einschränkt. Daneben können zwei Dienste trotz gleicher Signatur ein abweichendes, im Extremfall sogar konträres Verhalten aufweisen. Da diese Einflüsse dazu führen können, dass eine bei bloßer Betrachtung von Signaturen bestehende Subtypeigenschaft bei zusätzlicher Berücksichtigung des Verhaltens verletzt wird, ergänzen das *verhaltensorientierte* bzw. das *zustandsbasierte Behavioural Subtyping* [Weh02] den Vergleich von Signaturen um die zusätzliche Betrachtung des Verhaltens von Typen.

Übertragen auf das Problem der geschäftsprozessorientierten Komponentensuche kann ein Geschäftsprozess, der in einem Unternehmen ausgeführt werden soll, vergleichbar mit einem Supertyp als eine Spezifikation verhaltensmäßiger Anforderungen betrachtet werden. Ziel der

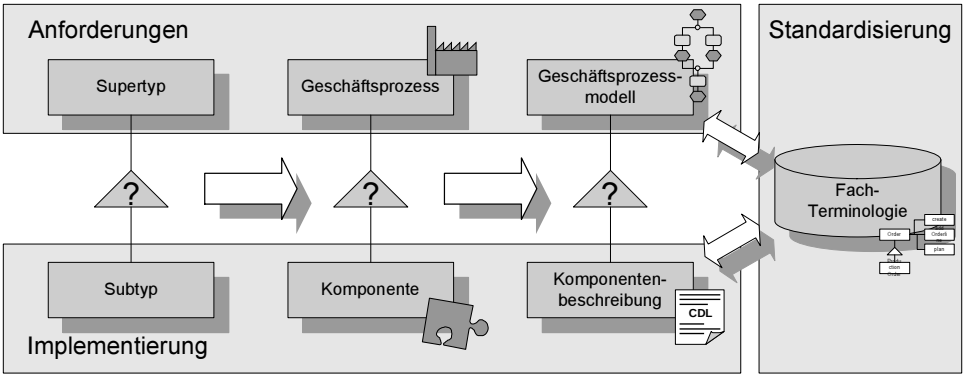

Abbildung 9: Geschäftsprozessorientierte Komponentensuche als Subtyping-Problem

geschäftsprozessorientierten Komponentensuche ist es nun, Komponenten zu finden, die vergleichbar mit einem Subtyp zumindest Teile dieser Anforderungen durch ihre Leistungen erfüllen. Grob gesagt erfüllt eine Komponente die Anforderungen eines Geschäftsprozesses, wenn sie

1. Dienste anbietet, die die Ausführung der einzelnen Aktivitäten des Geschäftsprozesses geeignet unterstützen,
2. den Aufruf dieser Dienste in der geforderten Reihenfolge ermöglicht und
3. die Verzweigungsstruktur des Geschäftsprozesses respektiert.

Auf einer höheren Abstraktionsebene schließlich beschreiben Geschäftsprozessmodelle Ablaufvarianten der in einem Unternehmen auszuführenden Geschäftsprozesse, während Komponentenbeschreibungen die von Komponenten angebotene Funktionalität spezifizieren, die für die Ausführung von (Teilen von) Geschäftsprozessen zur Verfügung steht. Um die fachliche Zuordnung von Dienste einer Komponente zu betrieblichen Aktivitäten eines Geschäftsprozesses zu ermöglichen, sehen wir die Verwendung einer standardisierten Fachterminologie gemäß Abschnitt 3 für die Spezifikation der fachlichen Semantik von Aktivitäten und Methoden vor. Abbildung 9 fasst unsere Interpretation der geschäftsprozessorientierten Komponentensuche als Subtyping-Problem graphisch zusammen.

5.3 Phasenmodell der geschäftsprozessorientierten Komponentensuche

Der Ansatz der semantischen Komponentensuche auf Basis von Geschäftsprozessmodellen soll im Folgenden anhand eines einfachen Phasenmodells erläutert werden. Dabei betrachten wir ein einfaches Beispiel zur Veranschaulichung des Ansatzes. Abbildung 10 zeigt Auszüge eines Geschäftsprozessmodells, der bereits vorgestellten CDL-Komponentenbeschreibung der Komponente `Vertrieb` sowie einer Terminologie.

1. *Angebotsphase*: *Anbieter* veröffentlichen CDL-Beschreibungen von Komponenten, die auf dem Marktplatz angeboten werden sollen, in einem Komponenten-Repository. Dabei ordnen Sie den einzelnen Diensten der Komponenten Sätze aus einer Terminologie nach Abschnitt 3 zu.

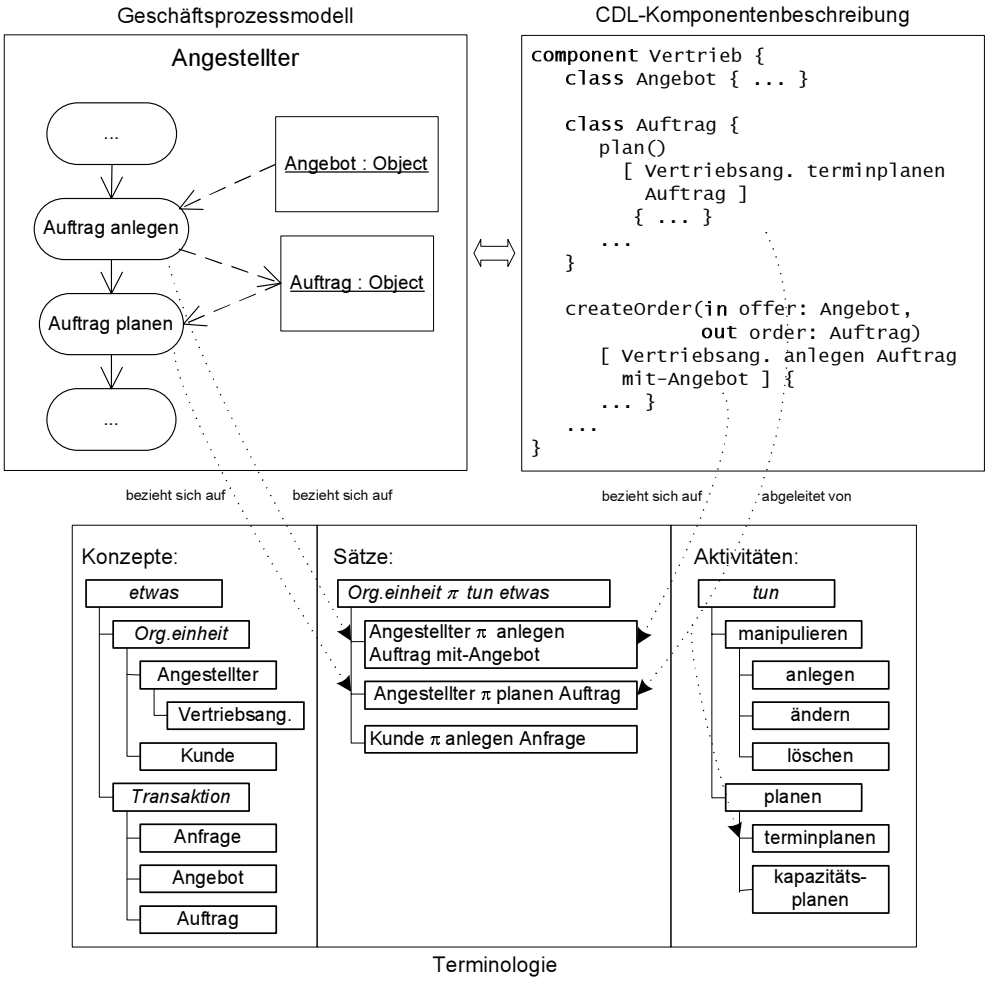

Abbildung 10: Komponentensuche auf Basis von Geschäftsprozessmodellen

In Abbildung 10 wird der Operation `createOrder` der Komponente `Vertrieb` die Semantik „Vertriebsangestellter (tut) anlegen Auftrag mit-Angebot“ zugeordnet (in eckigen Klammern). Dieser Satz stellt eine gültige Spezialisierung des in der Beispiel-Terminologie definierten Kernsatzes „Angestellter (tut) anlegen Auftrag mit-Angebot“ dar (vgl. Definition 2). Analog ist der Operation `plan` der enthaltenen Klasse `Auftrag` die Semantik „Vertriebsangestellter (tut) terminplanen Auftrag“ zugeordnet. Dieser Satz repräsentiert eine Spezialisierung des Satzes „Angestellter (tut) planen Auftrag“.

2. *Anforderungsphase*: *Verwender*, die am Erwerb von Komponenten interessiert sind, spezifizieren ihre Anforderungen gegenüber einem Broker in Form von Geschäftsprozessmodellen. In unserer Arbeit zur Komponentensuche konzentrieren wir uns auf ablauforganisatorische Aspekte, während Anforderungen an die Aufbauorganisation oder den Datenfluss etc. unberücksichtigt bleiben. Die Semantik der einzelnen Prozessschritte definieren wir analog zur Semantik von Diensten über Sätze aus einer Terminologie. Dies lässt sich damit begründen, dass sich die Schritte eines Geschäftsprozessmodells ebenfalls als sprachliche Handlungen mit Subjekt, Prädikat und direktem sowie indirekten Objekten verstehen lassen (vgl. [Tes02]).

   Die Aktivitäten des in Form eines UML Aktivitätsdiagramms spezifizierten Geschäftpro-

zessmodells aus Abbildung 10 sind direkt über Kernsätze aus der Terminologie definiert.

3. *Suchphase*: Für den Vergleich von Geschäftsprozessmodellen mit Komponentenbeschreibungen auf der *Protokollebene* (partielle Ordnung zwischen Diensten) greifen wir direkt auf den Ansatz des verhaltensorientierten Behavioural Subtypings zurück und untersuchen eine Subtyping-Relation zwischen Protokollautomaten. Wir verwenden dabei eine abgewandelte Form der Verfeinerungsrelation *Weak Simulation* [Mil80]. Eine genauere Einführung in den verfolgten Ansatz bietet [Tes01].

   Der Vergleich auf der *Semantikebene* (Bedeutung der Dienste) basiert auf der Idee, die fachliche Semantik von Schritten eines Geschäftsprozessmodells einerseits und Operationen von Komponenten andererseits miteinander zu vergleichen. In Anlehnung an das zustandsbasierte Behavioural Subtyping, bei dem die Semantik von Operationen über Vor- und Nachbedingungen beschrieben wird, verlangen wir für eine Übereinstimmung von Prozessschritten und Operationen auf der Semantikebene, dass der Satz, der die fachliche Semantik der Operation beschreibt, spezieller ist als der Satz, der die fachliche Semantik des Prozessschrittes spezifiziert. Dabei gehen jedoch abweichend von Definition 1 und 2 lediglich das Prädikat und das direkte Objekt in den Vergleich ein.

   In unserem Beispiel entspricht die Operation `createOrder` der Komponente dem initialen Prozessschritt des Geschäftsprozessmodells, da ihre Semantik – gemäß Definition 2 unter Berücksichtigung der erwähnten Einschränkung auf Prädikate und direkte Objekte – eine gültige Spezialisierung des Satzes „Angestellter (tut) anlegen Auftrag mit-Angebot“ (der fachlichen Semantik des Prozessschrittes) darstellt. Auf ähnliche Weise ist die Operation `plan` der Klasse `Auftrag` eine gültige Spezialisierung des Prozessschrittes „Angestellter (tut) planen Auftrag“, da die Aktivität „terminplanen“ als Spezialisierung der geforderten Aktivität „planen“ definiert ist.

4. *Präsentations- und Evaluierungsphase*: Für die Präsentation gefundener Komponenten bieten sich grundsätzlich zwei Ausrichtungen an. Zum einen können aufgrund des verfolgten normsprachlichen Ansatzes für die Beschreibung der fachlichen Semantik von Komponenten auf einfache Weise textuelle Repräsentationen der gefundenen Komponenten hergestellt werden, die auch von einem Fachexperten bewertet werden können. Zum anderen lässt sich das Verhalten von Komponenten und ihren Klassen in Form von Automatenmodellen graphisch veranschaulichen. Hierbei ist es möglich, die Übereinstimmung von angefordertem Geschäftsprozessmodell und Komponentenverhalten geeignet hervorzuheben.

5.4 Werkzeugunterstützung

Teile des im Projekt KOSOBAR verfolgten Ansatzes zur geschäftsprozessorientierten Komponentensuche sind im Rahmen einer Diplomarbeit in Form eines internetbasierten Brokers umgesetzt worden (siehe [Kei01]). Dabei beschränkte sich die Komponentensuche allerdings auf die Untersuchung von Behavioural-Subtyping-Beziehungen auf der Protokollebene. Aktuell wird diese Umsetzung im Rahmen eines Dissertationsvorhabens überarbeitet und um die Berücksichtigung der Semantikebene beim Vergleich von Geschäftsprozessmodellen und Komponentenbeschreibungen ergänzt.

6 Zusammenfassung

Neben der fachlichen und technischen Standardisierung ist für den Erfolg der komponentenbasierten Softwareentwicklung auch eine Unterstützung des Entwicklungsprozesses durch entsprechende Werkzeuge erforderlich. Im OFFIS-Projekt KOSOBAR ist auf der Grundlage eines Vorgehensmodells für die komponentenbasierte Softwareentwicklung eine Infrastruktur für die Suche und den Austausch von Fachkomponenten entwickelt worden. Das Vorgehensmodell berücksichtigt dabei insb. Entwicklungsprozesse, bei denen Akteure in verschiedenen Rollen über internetbasierte Komponentenmärkte interagieren. Die Infrastruktur unterstützt die beteiligten Akteure durch Werkzeuge zur Verwaltung von (domänenspezifischen) Terminologien, zur Beschreibung und Verwaltung von Komponentenbeschreibungen, zur Veröffentlichung dieser Beschreibungen in Komponenten-Repositories sowie durch Broker zur geschäftsprozessorientierten Komponentensuche. Die Beschreibung und Suche nach Fachkomponenten basiert auf dem Gedanken, die fachliche Semantik von Komponenten und Suchanfragen in Form normsprachlicher Sätze über einer standardisierten Terminologie zu definieren. Die im Projekt *KOSOBAR* verfolgten Ansätze im Bereich der Beschreibung von Fachkomponenten sind in Form von Workshop-Beiträgen [JT01, JT02] in die Arbeit des Arbeitskreises 5.10.3 *Komponentenorientierte betriebliche Anwendungssysteme* zur Vereinheitlichung der Spezifikation von Fachkomponenten eingeflossen.

Eine Evaluierung der vorgestellten Konzepte wird im Rahmen des in Abschnitt 5.4 angesprochenen Dissertationsvorhabens durchgeführt. Dazu werden Fachkomponenten (EJBs) für die kommunale Verwaltung im Kfz-Zulassungswesen mittels CDL unter Benutzung einer geeigneten Fachterminologie beschrieben und auf der Grundlage von Modellen kommunaler Verwaltungsprozesse gesucht.

Literatur

[ABC+02] J. Ackermann, F. Brinkop, S. Conrad, P. Fettke, A. Frick, E. Glistau, H. Jaekel, O. Kotlar, P. Loos, H. Mrech, E. Ortner, U. Raape, S. Overhage, S. Sahm, A. Schmietendorf, T. Teschke und K. Turowski. Vereinheitlichte Spezifikation von Fachkomponenten, Februar 2002. Memorandum des Arbeitskreises 5.10.3 Komponentenorientierte betriebliche Anwendungssysteme.

[Ame91] P. America. Designing an Object-Oriented Programming Language with Behavioural Subtyping. In J. W. de Bakker, W. P. de Roever und G. Rozenberg, Hrsg., *Foundations of Object-Oriented Languages, Proceedings of REX School/Workshop, Noordwijkerhout, Niederlande, 28. Mai - 1. Juni 1990*, Jgg. 489 von *Lecture Notes in Computer Science*, Seiten 60–90. Springer-Verlag, 1991.

[Büt95] W. Büttemeyer. *Wissenschaftstheorie für Informatiker*. Spektrum Akademischer Verlag, 1995.

[CJB99] B. Chandrasekaran, J. R. Josephson und V. R. Benjamins. What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, 14(1):20–26, 1999.

[Com02] ComponentSource, ComponentSource: The Definitive Source of Software Components. Elektronische Quelle: *http://www.componentsource.com/*, zuletzt besucht im Juni 2002.

[Fla02] Flashline.com, Flashline – Transforming Software Development. Elektronische Quelle: *http://www.flashline.com/*, zuletzt besucht im Juni 2002.

[FS96] N. E. Fuchs und R. Schwitter. Attempto Controlled English (ACE). In *The First International Workshop on Controlled Language Applications (CLAW 96)*, Katholieke Universiteit Leuven, 1996.

[Has02] W. Hasselbring. Component-Based Software Engineering. In S.K. Chang, Hrsg., *Handbook of Software Engineering and Knowledge Engineering*, Jgg. 2. World Scientific Publishing, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[HT01] A. Harren und H. Tapken. TODAY Open Repository: An Extensible, MOF-Based Metadata Repository System. Technischer Bericht, OFFIS, Oldenburg, 2001.

[JT01] H. Jaekel und T. Teschke. Prozessorientierte Beschreibung von Fachkomponenten. In K. Turowski, Hrsg., *Modellierung und Spezifikation von Fachkomponenten: 2. Workshop*, Seiten 105–112, Bamberg, 2001.

[JT02] H. Jaekel und T. Teschke. Berücksichtigung von Berechtigungskonzepten bei der Spezifikation von Fachkomponenten. In K. Turowski, Hrsg., *Modellierung und Spezifikation von Fachkomponenten: 3. Workshop*, Seiten 69–85, Nürnberg, 2002.

[Kei01] M. Keilers. Ein internetbasierter Komponentensuchdienst auf Basis von Verhaltensbeschreibungen. Diplomarbeit, Carl von Ossietzky Universität Oldenburg, Fachbereich Informatik, 2001.

[KNS92] G. Keller, M. Nüttgens und A.-W. Scheer. Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)“. Technischer Bericht 89, Institut für Wirtschaftsinformatik, Universität des Saarlandes, 1992.

[LW94] B. Liskov und J. Wing. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[Mey92] B. Meyer. Applying “Design By Contract”. *IEEE Computer*, 25(10):40–51, 1992.

[Mil80] R. Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science 92*. Springer-Verlag, 1980.

[Nie95] O. Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz und D. Tsichritzis, Hrsg., *Object-Oriented Software Composition*, Seiten 99–121. Prentice Hall, 1995.

[Nom02] Nomina, Software-Marktplatz: Die ISIS Software- und Firmendatenbanken. Elektronische Quelle: *http://www.software-marktplatz.de/*, zuletzt besucht im Juni 2002.

[OMG01] Object Management Group. *Unified Modeling Language Specification, Version 1.4*, 2001.

[Ort97] E. Ortner. *Methodenneutraler Fachentwurf*. Wirtschaftsinformatik. B. G. Teubner Verlag, Stuttgart, Leipzig, 1997.

[Rit98] J. Ritter. PROSECCO – Eine Methode zur modellbasierten Anwendungssystemgestaltung. In *Proceedings of Business Information Systems '98*, Posen, Polen, 1998.

[SAP02] SAP, SAP – Software Partner Directory. Elektronische Quelle: *http://www.mysap.com/partners/software/directory/*, zuletzt besucht im Juni 2002.

[Sch97] B. Schienmann. *Objektorientierter Fachentwurf (in German)*, Jgg. 20 von *Teubner-Texte zur Informatik*. B.G. Teubner Verlag, Stuttgart, Leipzig, 1997.

[Sof02] SoftSelect, SoftSelect Matching Plattform. Elektronische Quelle: *http://www.softselect.de/*, zuletzt besucht im Juni 2002.

[STR00] C. Sandmann, T. Teschke und J. Ritter. Ein Vorgehensmodell für die komponentenbasierte Anwendungsentwicklung. In B. Britzelmaier und S. Geberl, Hrsg., *Information als Erfolgsfaktor*, Seiten 49–58, Stuttgart, 2000. Teubner.

[Sun02] Sun Microsystems, SUN Solutions Marketplace. Elektronische Quelle: *http://industry.java.sun.com/solutions/*, zuletzt besucht im Juni 2002.

[Szy98] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[Tec02] Technische Universität Chemnitz, Informationssysteme & Management, Component Markets – An Overview. Elektronische Quelle: *http://www.tu-chemnitz.de/wirtschaft/wi2/projects/components/*, zuletzt besucht im Mai 2002.

[Tes01] T. Teschke. Using Business Process Knowledge for Software Component Retrieval. In *Proceedings of 11th Annual BIT Conference*, Manchester, 2001.

[Tes02] T. Teschke. Ontological Intermediation between Business Process Models and Software Components. In H.-M. Haav und A. Kalja, Hrsg., *Databases and Information Systems, Proceedings of Fifth International Baltic Conference Baltic DB&IS'2002*, Jgg. 1, Tallinn / Estland, 2002.

[TR01] T. Teschke und J. Ritter. Towards a Foundation of Component-Oriented Software Reference Models. In G. Butler und S. Jarzabek, Hrsg., *Generative and Component-Based Software Engineering*, Lecture Notes in Computer Science (LNCS) 2177, Seiten 70–84, Berlin, 2001. Springer.

[Weh02] H. Wehrheim. *Behavioural Subtyping in Object-Oriented Specification Formalisms*. Dissertation, Carl von Ossietzky Universität Oldenburg, Fachbereich Informatik, 2002.

[WZ88] P. Wegner und S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing und K. Nygaard, Hrsg., *Proceedings ECOOP '88*, Lecture Notes in Computer Science 322, Seiten 55–77, Oslo, Norwegen, 1988. Springer-Verlag.

Herausgeber:

Prof. Dr. Klaus Turowski
Lehrstuhl für Betriebswirtschaftslehre, insbesondere Wirtschaftsinformatik II
Universität Augsburg
Universitätsstraße 16, 86135 Augsburg
Phone: +49(821)598-4431; Fax : -4432
E-Mail: klaus.turowski@wiwi.uni-augsburg.de
URL: http://wi2.wiwi.uni-augsburg.de

ISSN 1619-9006