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).

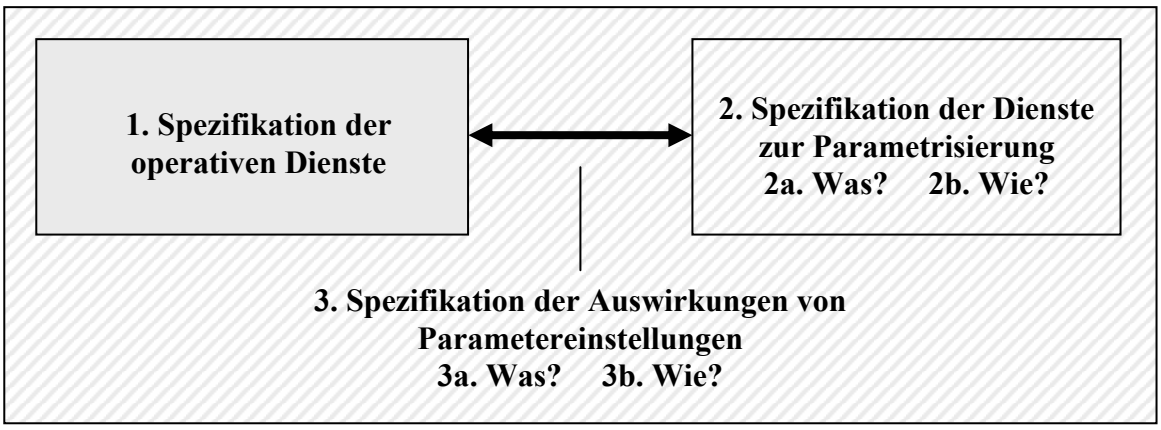

Bild 1: Aufgabenstellungen für die Spezifikation von Fachkomponenten

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.

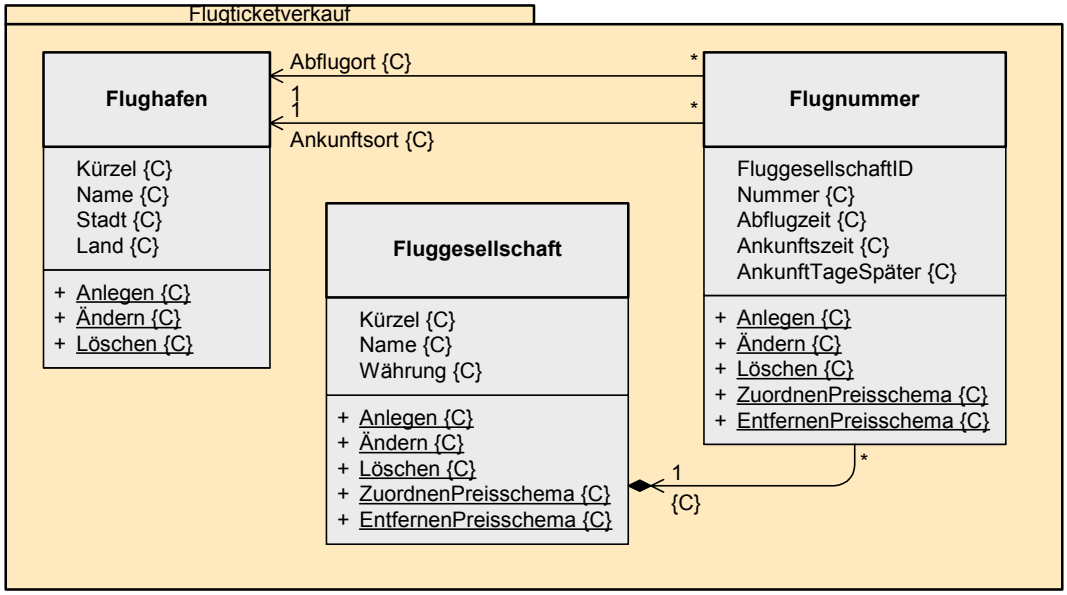

Bild 2: Ausschnitt aus dem UML-Klassendiagramm zur Fachkomponente *Flugticketverkauf*

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.

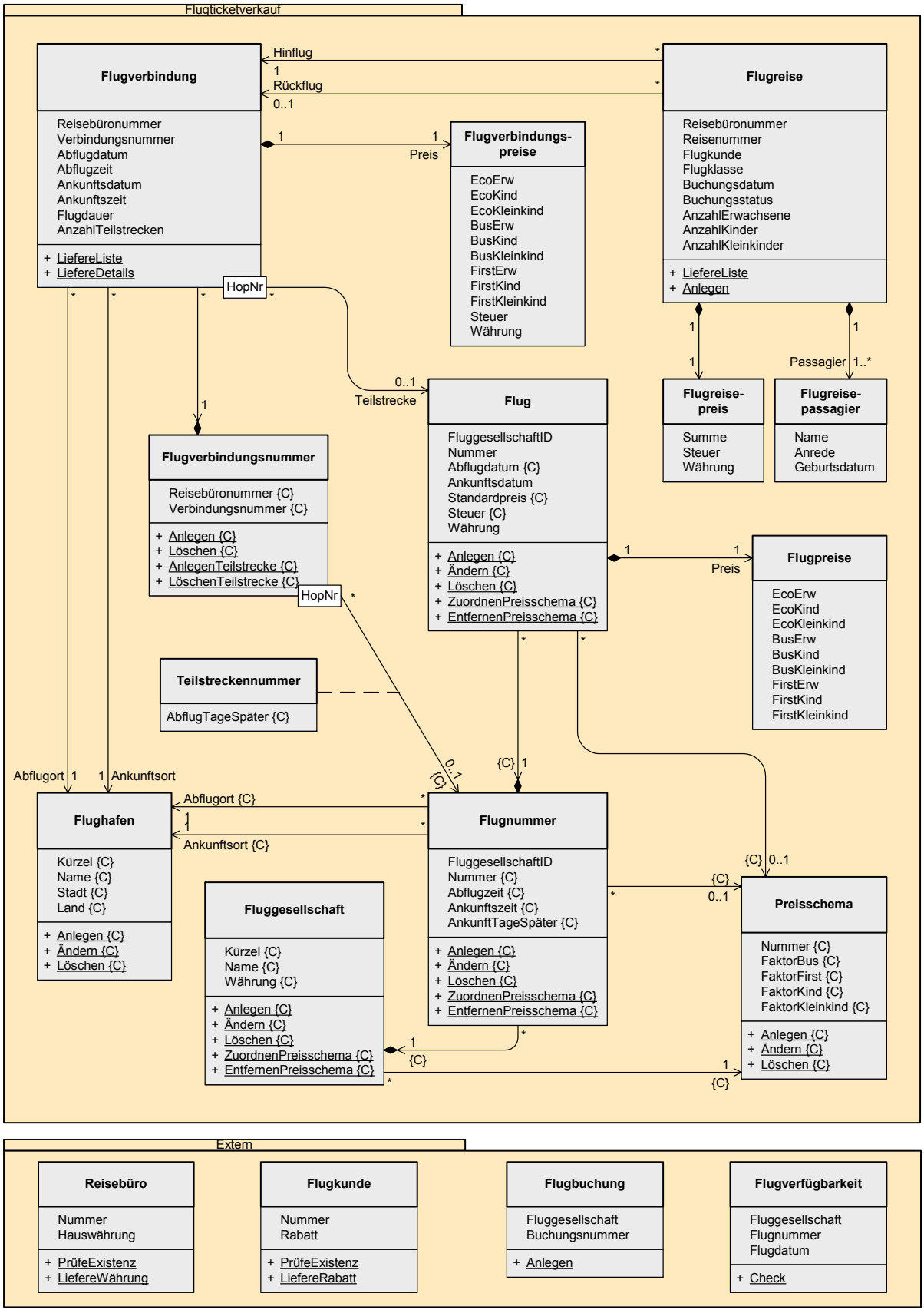

Bild B-1: UML-Modell der Fachkomponente *Flugticketverkauf*

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.